UNIVERSITY OF CAMBRIDGE

Configurable memory systems for embedded many-core processors

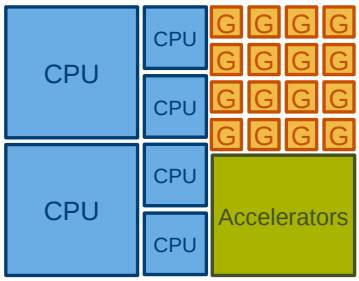
Daniel Bates, Alex Chadwick and Robert Mullins

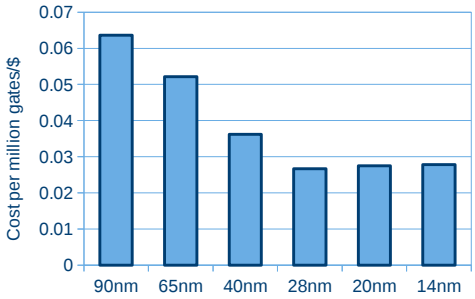
HIP3ES 18/01/2016

Motivation: ever-changing tradeoffs

Heterogeneity won't be optimal forever

- Rising complexity
- Rising transistor costs





Source: *Feature dimension reduction slowdown*, by Handel Jones

UNIVERSITY OF CAMBRIDGE

The tradeoffs in computer architecture are forever changing.

The current trend (particularly in mobile) is heterogeneity. Many different pieces of logic are included on-chip, which are all good for different purposes. This is a good direction to head at the moment – there is a need for performance and energy efficiency which can't be satisfied by continually building more-complex cores.

However, tradeoffs are continuing to change, and we can't keep going in this direction forever. As our designs increase in complexity, they become more expensive to design and validate. Optimising programs to use a varying array of accelerators is challenging. As transistors get increasingly unreliable, each different component will need to be made fault tolerant individually.

We can also no longer rely on an ever-increasing number of transistors, since transistors are now getting more expensive as they get smaller. Instead, we must make better use of the transistors we have.

We're exploring what comes next: homogeneity. We propose having small, identical cores for energy efficiency, and lots of them (hundreds). This brings with it its own problems: what should these cores look like and how can we use so many?

Making homogeneity efficient

- Specialise in software, not hardware
- We want resources to be
  - **A**ccessible
  - **B**ypassable
  - **C**omposable
- Today's focus is on the memory system

UNIVERSITY OF CAMBRIDGE

We don't want to lose the benefits of heterogeneity when we move to a homogeneous system – we still need to be able to specialise to improve performance and reduce energy consumption. We must now do this in software, however.

We use an ABC philosophy when choosing appropriate designs.

Accessible: if resources elsewhere on the chip can be accessed easily, the local component can be made smaller, simpler, and lower-energy.

Bypassable: reduce overheads by not using logic if it isn't needed.

Composable: if there are many more components on chip, they are going to be smaller. Allow them to be grouped together at run time to make up for this. Also, since our components are accessible, we want to be able to simplify software by having a uniform interface, regardless of what is being accessed.

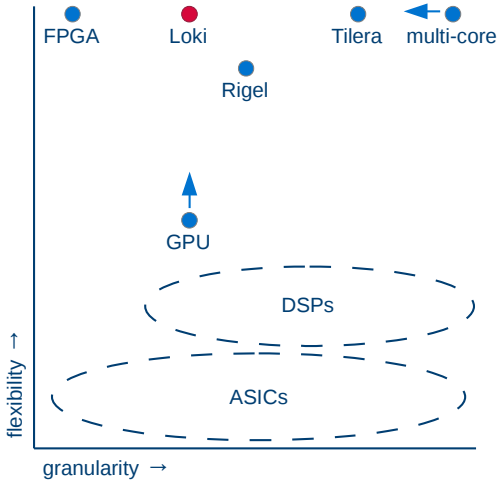
Resources include cores, memory banks, register files, even memory/network bandwidth. For this work, we focus on the memory system.

Related work

Many projects in this field:

- Elm
- Raw/Tilera
- Rigel
- TRIPS
- ...

None have the same focus on flexibility and cooperation between cores.



UNIVERSITY OF CAMBRIDGE

The ideas I'm going to talk about apply to most architectures, but I'm going to use one particular architecture as a concrete example. The reasons for this will become clear at the end of the talk.

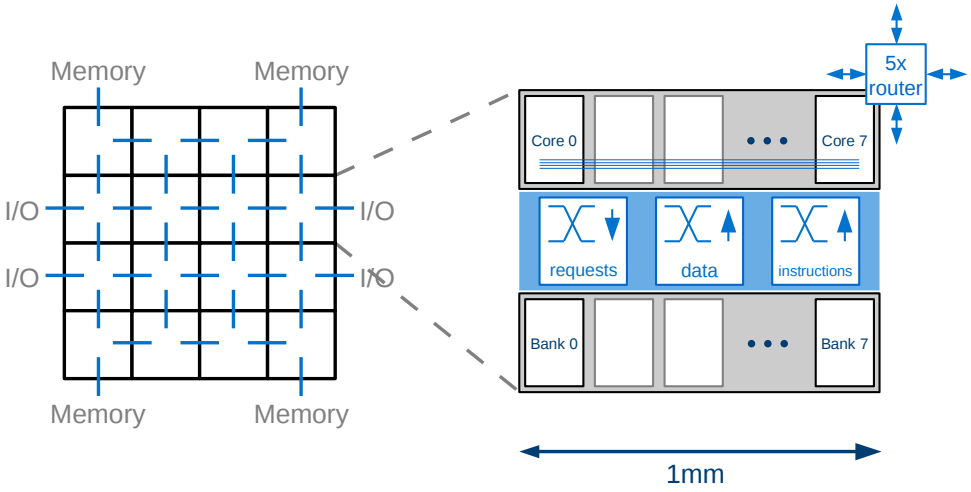
There are several dimensions missing from this plot... performance, energy consumption, cost, etc.

It's meant to give a flavour of where this work (Loki) is in the design-space: a fine-grained but general-purpose architecture. Not an accelerator.

You can see us as being somewhere between an FPGA and a multi-core processor because we have lots of identical units repeated across the chip, but ours are considerably smaller than those of typical multi-cores'. We also have similarities with a GPU. We've seen that GPUs are getting more general-purpose and that multi-cores are increasing the number of cores they have – Loki could be a potential convergence point.

Key feature: almost everything on chip communicates over an efficient on-chip network. This gives us access to other components, allows us to bypass them and go straight to others (e.g. skip cache and access off-chip memory controller), and software is responsible for composing resources in an application-specific way.

Loki architecture: chip and tile



UNIVERSITY OF CAMBRIDGE

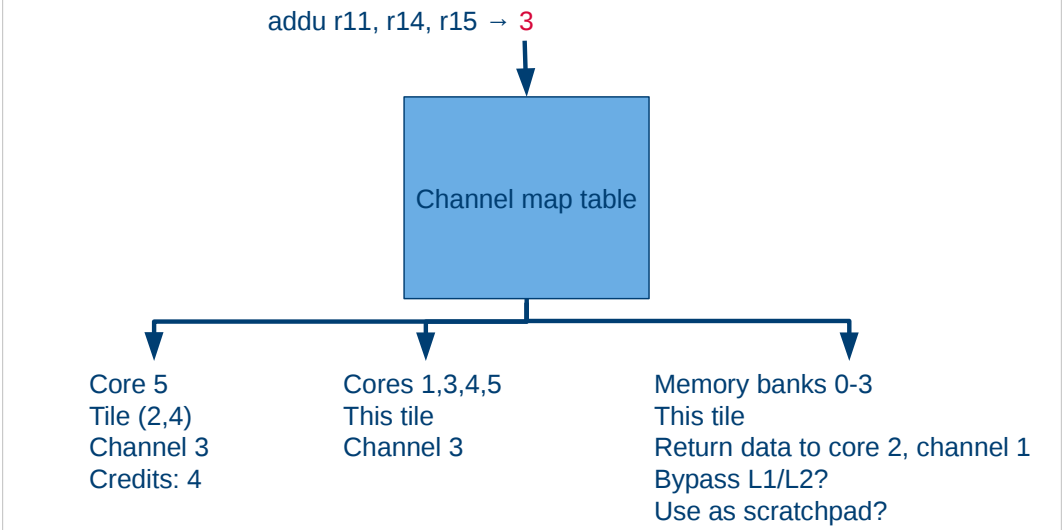
Here's how we've arranged our many cores.

We have a hierarchical structure, with identical tiles stamped out across the chip, each connected to their nearest neighbours. Each tile is about 1mm² (in a 40nm process) and contains 8 cores and 8 shared L1 cache banks.

There is also a lot of interconnect. Fast, low-energy, flexible interconnect is key to allowing cores to co-operate effectively. The on-chip network extends to the chip interface, including components like memory controllers and I/O.

We make use of end-to-end flow control to avoid any possibility of deadlock. If a message is put onto the network, it is guaranteed that it will eventually be removed.

Channel map table



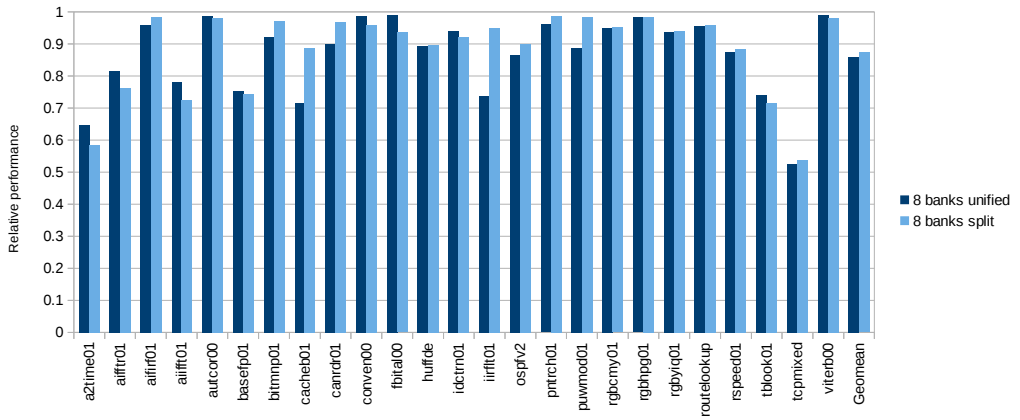
Here is the component which manages the network connections. It is called the channel map table, and each core has one. It describes which component or components to access, and how to access them. This is one of the key components for reconfiguration.

Almost all Loki instructions include the option to send their result over the network as well as storing it in the local register file. The channel map table converts this logical address into a physical address.

Data can be sent to an arbitrary set of cores on the local tile, a single core on a remote tile, or a memory bank on the local tile. Memory access has the most configuration options: banks can be composed into larger virtual groups, with cache lines distributed evenly across them; they can be bypassed entirely to allow access to lower levels of the memory hierarchy; and they can be treated as either cache banks or scratchpads. Scratchpads bypass the tag check mechanism when it is not required – previous work has shown that this can save about 10% of total chip energy.

The channel map table can be updated with a single instruction in a single clock cycle – it is basically a mini register file.

L1 configuration – 8 banks



Unified cache penalty: 3%
Split cache penalty: 2%

UNIVERSITY OF CAMBRIDGE

Here we compare performance when all 8 cache banks are shared between instructions and data, and when they are split evenly with 4 banks for instructions and 4 banks for data. We're using the EEMBC benchmark suite.

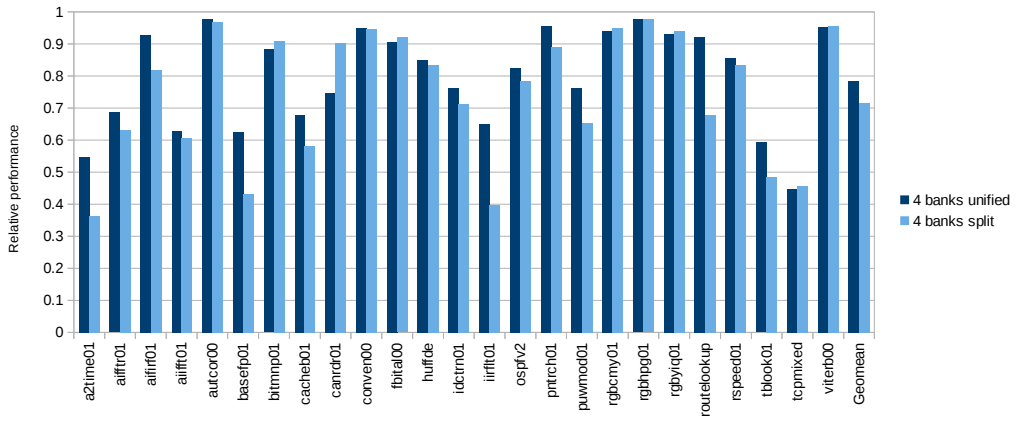
For many benchmarks, the configuration doesn't really matter – either there is plenty of cache available in either case, or neither cache is sufficient.

For the remaining benchmarks, there is no single best configuration – some prefer the unified cache and some prefer the split cache. iirflt (infinite impulse response filter) shows the most variation, with a 20% performance drop when using a unified cache.

A configurable system is able to choose the best configuration in each case. On average, we lose a couple of percent performance by fixing the cache configuration for all benchmarks.

These results are for when a single program has access to all memory banks on a tile. However, there are 7 other cores on that tile which might be competing for resources. We'll next perform the same experiment, but with half as much cache available.

L1 configuration – 4 banks



Unified cache penalty: 3% → 1%
Split cache penalty: 2% → 10%

UNIVERSITY OF CAMBRIDGE

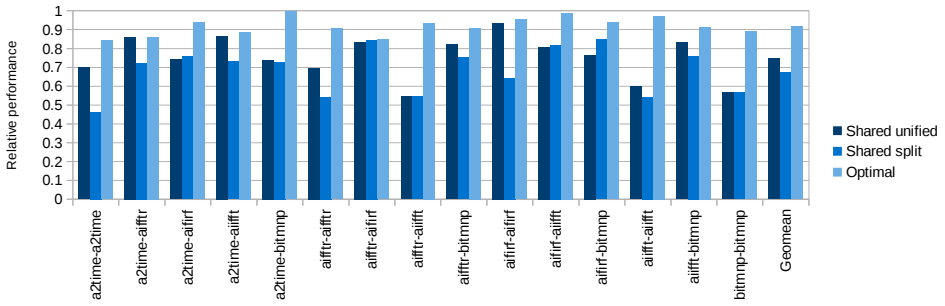
Of course, on average, performance is lower when less cache is available. Perhaps less expected is that the variability of each benchmark increases – choosing the best configuration is more important when there are fewer resources available. We can now lose up to 10% performance on average if we are not able to adjust the configuration for each benchmark.

Again, iirflt is the most variable benchmark, but interestingly, this time it performs better with a unified cache. With 8 banks, the unified cache was 20% worse, but with 4, it's almost 70% better.

So we can't just choose a single configuration and scale it up and down – the optimal configuration depends on both the benchmark and the resources available to it.

We expect the difference between configurable and fixed caches to increase as resources become more constrained. This could be due to more programs competing for the same memory banks, or larger applications with larger working sets.

L1 configuration – competing applications



Unified cache penalty: 19%
Split cache penalty: 27%

70% fewer cache misses
Less data movement
Less off-chip memory access

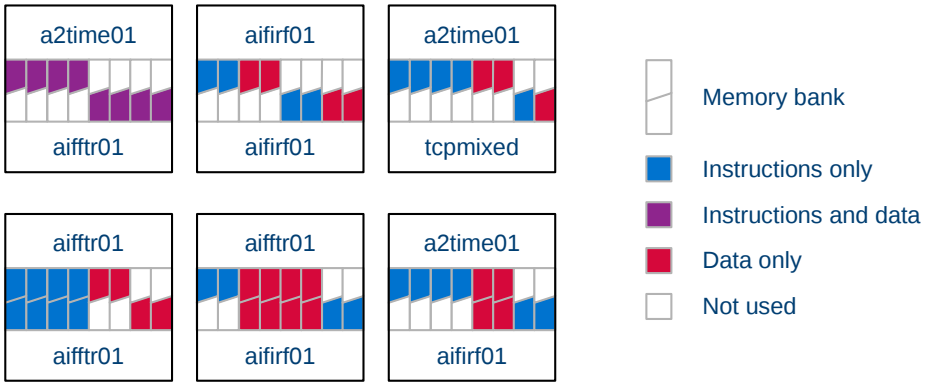
UNIVERSITY OF CAMBRIDGE

(Figures are relative to the benchmarks running in isolation, with all 8 banks, and assume both benchmarks have equal priority. In practice, there are a range of optimal configurations, depending on the applications' relative priorities.)

The performance implications of choosing the best cache configuration are higher than ever. If our architecture didn't support configuration, performance would drop 20-30% on average. This situation will only get worse as more benchmarks run at once, or as applications get larger and put more pressure on the cache system.

As well as the performance benefits of configuration, there are also energy benefits. When we configure the cache to suit the programs running, we can reduce the miss rate by over 70% on average (a few of the benchmarks are over 95%). This means there is less data movement, less replacement of cache data, and in particular, less access to the more-expensive lower levels of the memory hierarchy.

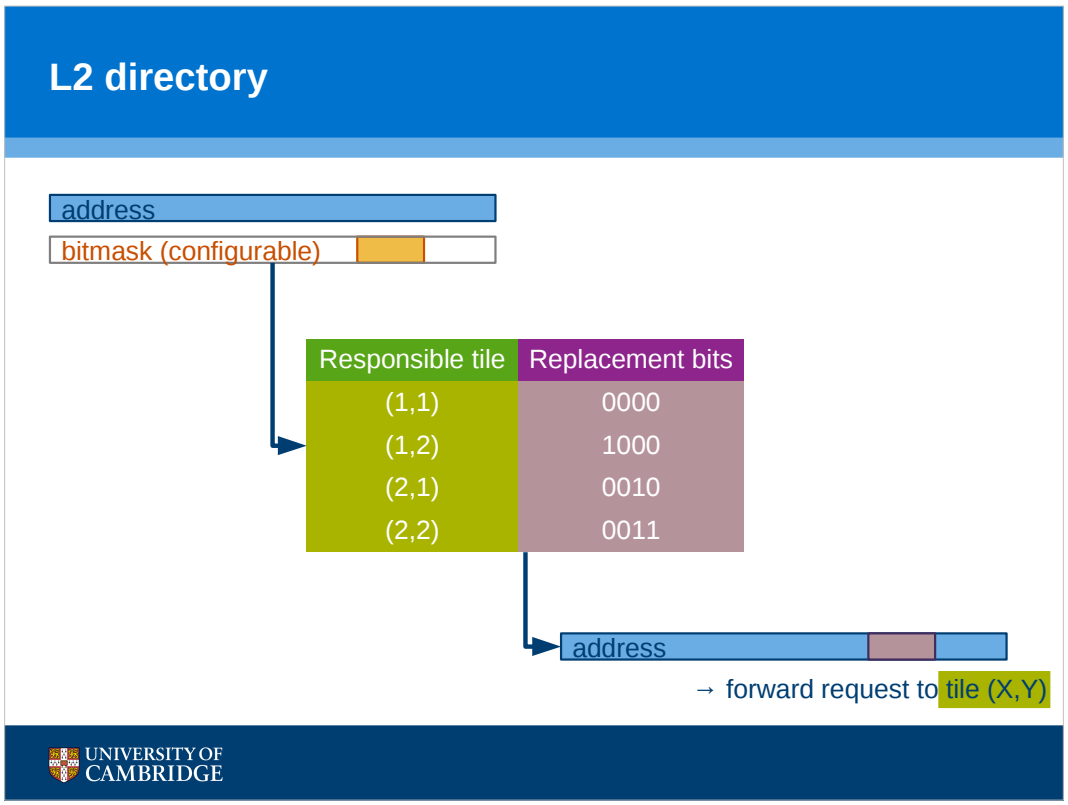
L1 configuration – competing applications



We've only been looking at single programs running in isolation so far; the configuration space when there are multiple benchmarks running simultaneously is much more complex.

Here are a few different optimal cache configurations for different pairs of programs. We can see that almost every possible decision is best some of the time. The top three pairs prefer to completely isolate themselves from each other, and some prefer unified caches, some prefer split, some share the cache fairly, and some allocate more memory to one of the benchmarks.

The bottom three pairs of programs all work best when sharing some portion of their cache. This gives them access to potentially more cache space, but they may experience a higher latency if the other program is already using a particular bank. Again, we see different configurations performing best – some share their instruction caches, some share their data caches, some share banks fairly, and some allocate more banks to one program than the other.



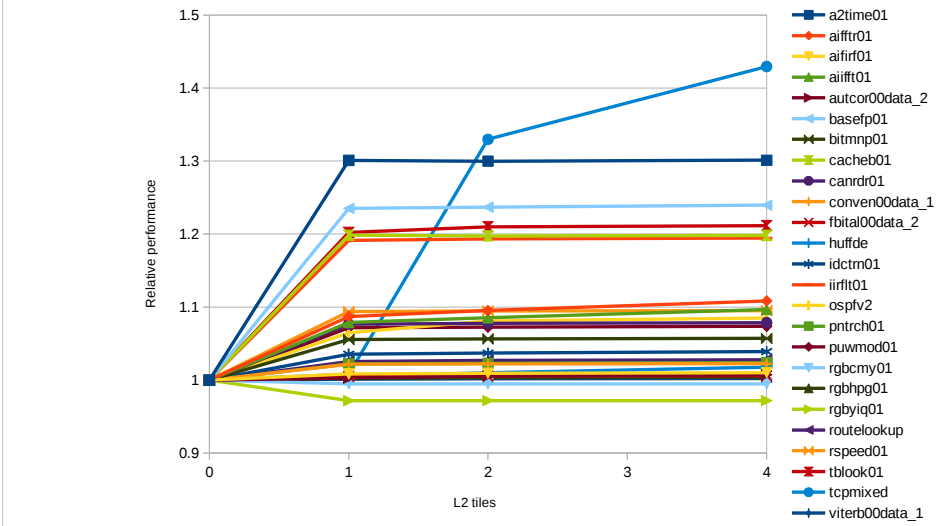
We've looked at the L1 cache, so let's now look at the L2. Each Loki tile has a directory which is consulted whenever there is an L1 cache miss. The directory tells which tile on the chip is responsible for caching a particular memory address, or points directly to an off-chip memory controller. The directory can also optionally replace some bits of the address to implement a simple virtual memory system.

At the target tile, all memory banks are accessed in parallel to form an 8-way set-associative cache.

The directory takes two instructions to update – one with the entry to update, and one with the data to store.

We now have many of the same options as we had with the L1 cache. We can change the amount of cache available, we can choose whether items are cached together or separately, etc.

L2 configuration



UNIVERSITY OF CAMBRIDGE

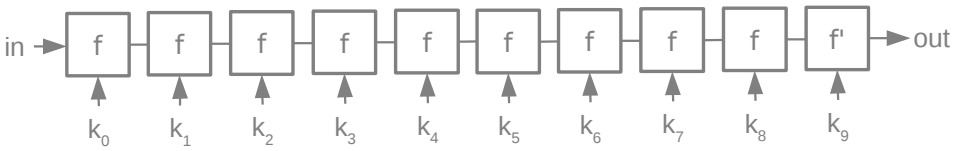
Here's how performance of the benchmarks scales as the number of L2 cache tiles increases.

As might be expected, different benchmarks have different sensitivities to the amount of cache available, but most show a modest improvement. It might be useful to use a tile to provide extra L2 cache if the tile is otherwise unused, but it probably isn't worth taking the cache away from a running program unless this one is high priority.

There are a couple of interesting cases here. Some of the benchmarks show a slight drop in performance when given an L2 cache. This is because they don't use any amount of cache very well, so adding an extra level to the memory hierarchy just increases latency. With a configurable memory system, we are not forced into using superfluous caches.

The TCP benchmark only shows a benefit once two tiles are available, and then goes on to have the largest improvement of all. Some benchmarks need their whole working set to be cached before a cache is of any use to them. If we are able to allocate cache in a fine-grained way, we can provide exactly the required amount, and maximise the resources available for other purposes.

Case study – AES counter mode



Implementation	Cycles/byte	Speed
1 core	64	7MB/s
8 cores	10	46MB/s

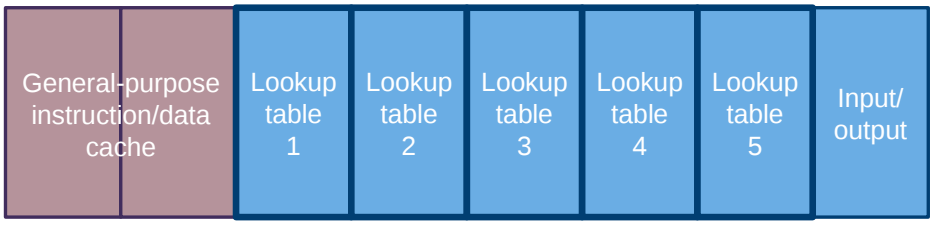
UNIVERSITY OF CAMBRIDGE

We're now going to look at a case study of AES encryption, and explore more of Loki's configuration options in an application-specific way.

AES takes a block of data, and applies the same function to it again and again, but with a different encryption key each time. The final version of the function is also very slightly different.

Optimised code with an 8-bank unified cache can process one byte every 64 clock cycles. We can parallelise the application by working on different blocks of data simultaneously, and we get a 6.4x speedup on 8 cores.

Case study – AES counter mode



Implementation	Cycles/byte	Speed
1 core	64	7MB/s
8 cores	10	46MB/s
1 core + custom L1	35	13MB/s
8 cores + custom L1	6.5	69MB/s

1.85x

1.51x

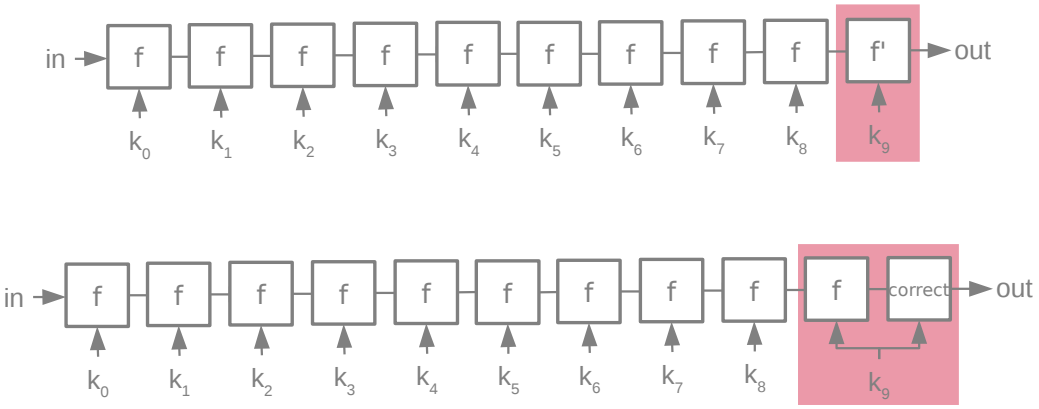
UNIVERSITY OF CAMBRIDGE

Here's an optimised cache configuration. We have two banks which serve instructions and hold general-purpose data like the stack and global variables. The final bank is dedicated to the input/output data.

The middle 5 bolded banks hold various lookup tables. These banks are accessed in scratchpad mode – the data is initialised at the start of the program and is then read-only. By using scratchpads, we bypass any unnecessary tag checks. We know that all tag checks are unnecessary here because the data fits entirely in the banks. Memory addresses are also simpler – they start at 0 instead of some offset provided by the memory allocator, so the code can be simplified.

The effects of simpler code and better caching behaviour combine to almost double performance for a single core, and give a 50% improvement for 8 cores.

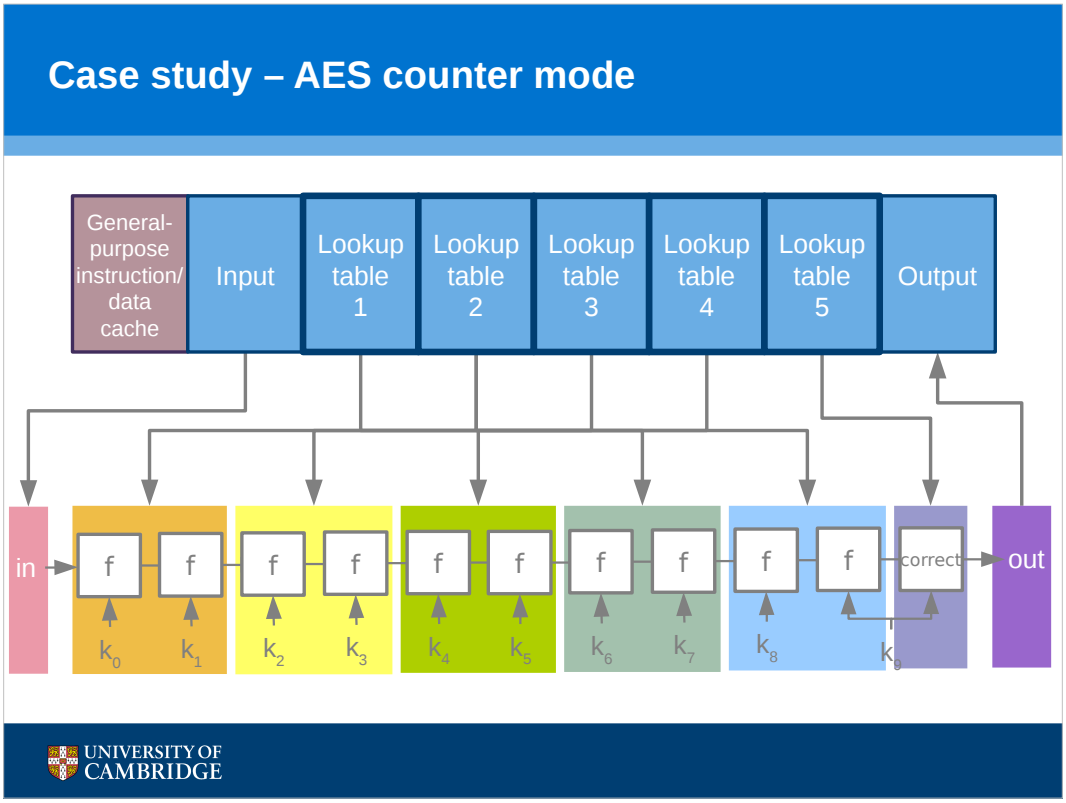
Case study – AES counter mode



UNIVERSITY OF CAMBRIDGE

If we're aware of how our memory system is performing, we might consider making minor changes to our programs to allow further memory system improvements. This isn't an unusual concept – cache blocking is often used to improve cache performance.

Here, we replace the slightly different final function with a function which is identical to all the previous ones, plus a simple correction.



Then, rather than giving each core identical code, we can distribute the software pipeline across all the cores neatly. Giving each core a smaller piece of the program improves caching behaviour and means we can reduce the general-purpose instruction/data cache down to one memory bank.

One core is dedicated to providing input data, and since we have a spare memory bank, we can now dedicate it to storing that input data.

The next five cores all perform two rounds of the function each. Each core only needs to access two of the encryption keys, so they can be saved in local registers, and we need to access the cache less often.

We then have our simple correction function and the output core (now with a memory bank dedicated to output data).

When in a steady state, there are no cache misses at all, so performance is completely predictable.

(We have 2xf on core 6 and correction on core 7 instead of 1xf on core 6 and 1xf' on core 7 because it reduces code size. 4 different types of core vs 5.)

Case study – AES counter mode

Implementation	Area	Power	Cycles/byte	Speed
1 core	0.5mm²	0.01W	64	7MB/s
8 cores	1mm²	0.06W	10	46MB/s
1 core + custom L1	0.5mm²	0.01W	35	13MB/s
8 cores + custom L1	1mm²	0.06W	6.5	69MB/s
Custom tile	1mm²	0.06W	5.1	88MB/s
16 custom tiles	16mm²	1W	0.32	1400MB/s

Intel Core i7-980X	60mm²	34W	1.3	2500MB/s
ARM9TDMI	6.6mm²	0.12W	45	3.3MB/s
ATI Radeon HD5650	104mm²	19W	1.9	340MB/s
NVIDIA GeForce 8800 GTS	484mm²	135W	17.1	95MB/s

We get a further 30% improvement by customising the whole tile in this way. The customised tile can then be repeated across the chip.

Here are a few other published figures. We have a couple of CPUs and a couple of GPUs. None get close to Loki's performance per Watt or performance per unit area. The Intel processor has very good performance because it has specialised instructions specifically for computing AES. Loki has performed all of its specialisation in software.

Summary

- Different configurations are best in different situations
  - Variance increases as resources are constrained further
- Configuration requires fine-grained allocation of resources
  - **A**ccessible, **B**ypassable, **C**omposable
- Intelligent resource allocation reduces data movement

UNIVERSITY OF CAMBRIDGE

There is no one-size-fits-all configuration. If we are to make the best use of our resources, we will need a different configuration in each situation.

Configuration requires fine-grained allocation of resources. For this, hardware structures must be small. This has the side effect of making them cheap to access.

We should be able to bypass any features which we don't need. For example, Loki doesn't need to have any cache beyond the L1, and even that can be bypassed. Scratchpad mode bypasses the tag check mechanism in cases where it isn't needed.

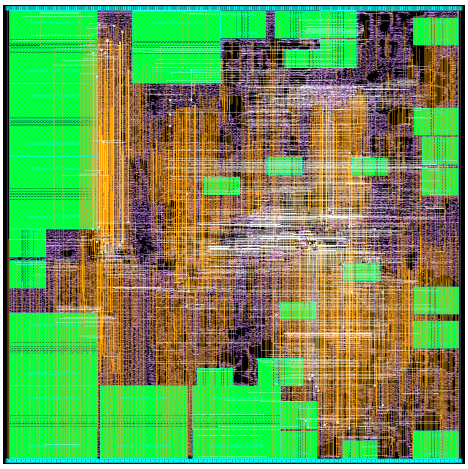
Fine-grained allocation also allows us to provide applications with “just enough” resources to function well, leaving as many resources as possible unused (for energy savings) or available for other purposes.

If we choose a good memory configuration, we greatly reduce the amount of data movement required. This improves performance and reduces energy consumption.

Advertisement

We are taping out a 128 core Loki test chip and plan to share development boards, tools, etc. with the community.

Please contact me if you are interested!



UNIVERSITY OF CAMBRIDGE

This is why I've been specifically mentioning Loki. As well as the memory system, Loki allows cores to co-operate in many interesting ways – see our previous work.

UNIVERSITY OF CAMBRIDGE

Configurable memory systems for embedded many-core processors

Daniel Bates, Alex Chadwick and Robert Mullins

Daniel.Bates@cl.cam.ac.uk

www.cl.cam.ac.uk/~rdm34/loki/

HIP3ES 18/01/2016