Resource-agnostic programming of microgrids

Raphael `kena' Poss — University of Amsterdam, the Netherlands

HPPC'10, Ischia, September 2010

apple core Universiteit van Amsterdam

donderdag 2 september 2010

I will talk today about our work on fine grained concurrency. The background guide for this work is the idea that concurrency management is hard and should be left out of the hands of the programmer. For example, the risk of deadlock and race conditions under manual composition of complex concurrent code should be avoided. However we are chiefly interested in granularity issues: the overheads associated with synchronization and concurrency creation must be properly amortized with computations.

Overview

- The Apple-CORE project is developing a novel many-core chip — the Microgrid — with the following attributes:
  - concurrency primitives in the core's ISA
    = *low overheads* for concurrency management
  - many fine grain threads per core to *tolerate extremes of latency*
  - binary code executes unchanged
    — whether in a single thread slot or over many threads on many cores, i.e. the code is *resource agnostic*
- This presentation will demonstrate that these attributes allow us to *predict performance* of code whatever the target on the Microgrid — which can then be *dynamically configured*

donderdag 2 september 2010

- Our approach to these issues is a hardware/software co-design in the form of a novel many-core chip and a programming model.
- On the hardware side, we implement concurrency primitives in the core ISA, for minimal overheads of scheduling and synchronization. We allow many threads per core, down to a few instructions per thread; what is important is that many threads allow to tolerate very large latencies.
- The ISA extensions are designed so that binary code is independent from the specific resources used for the scheduling; either running code sequentially within one hardware thread context or over many thread contexts on many cores. This is what we call resource agnosticism.
- I will show in this presentation that our design decisions allow to predict performance of computations when executed in different Microgrid configurations. This is a key finding that allow us to match efficiently resource allocation to application demands.

Fine grained hardware concurrency

H/W support for 256 threads

- Schedule unit: instruction or processor cycle

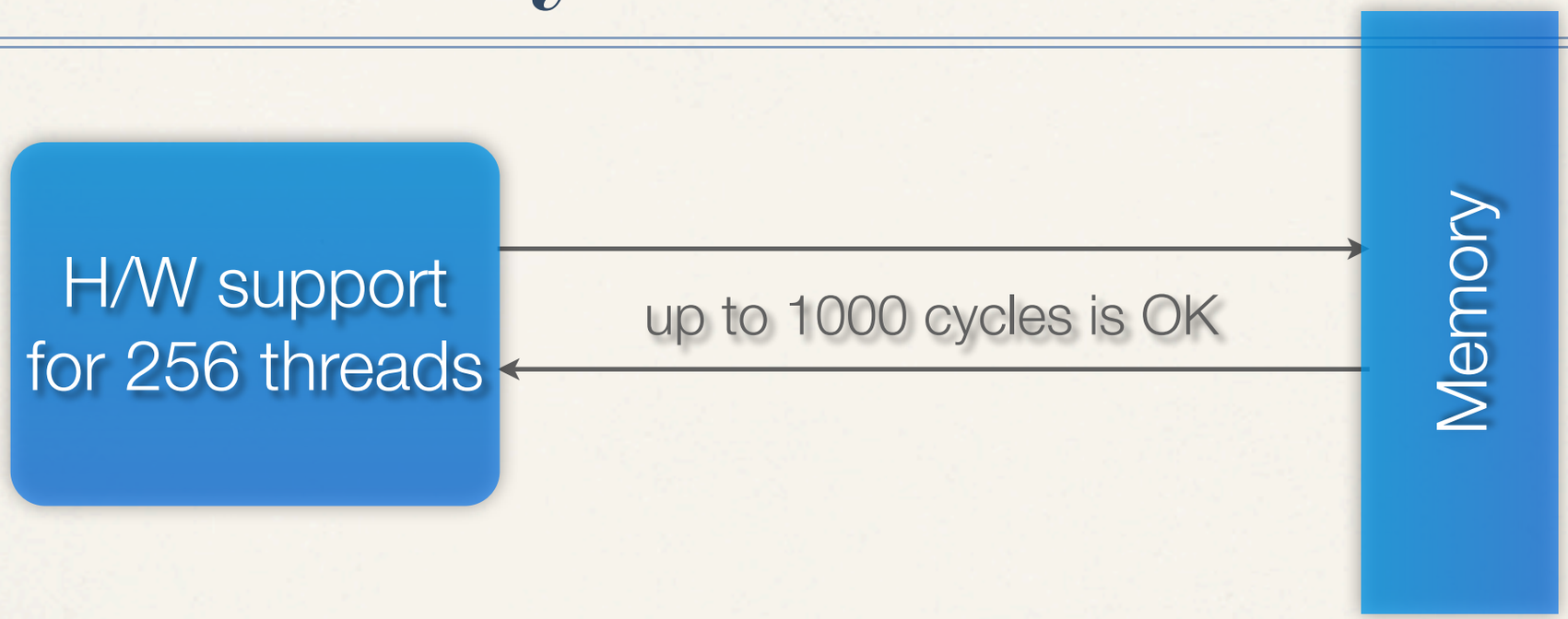
FPU, FPGA, encryption etc.

- Thread creation in one pipeline cycle
- Blocking threads: long latency operations cause thread to suspend and switch to next thread
- Can context switch on every cycle to keep the pipeline full - apparent 1 cycle latencies as long as there are sufficient threads ready

donderdag 2 september 2010

Fine grained in our context means that the schedule unit is the single instruction or processor cycle. We are able to allocate one new thread context per cycle, and when threads suspend on long latency operations control can be transferred to the next thread without pipeline stalls or bubbles. We achieve this by synchronizing through registers, with blocking reads and wake on write. With enough threads active, latencies up to thousands of cycles can be tolerated; i.e. appear to run in one cycle in each thread's time scale. This allows to tolerate memory access latencies as well as latencies to accelerators, FPUs etc which can be shared between cores.

Fine grained hardware concurrency



- Schedule unit: instruction or processor cycle

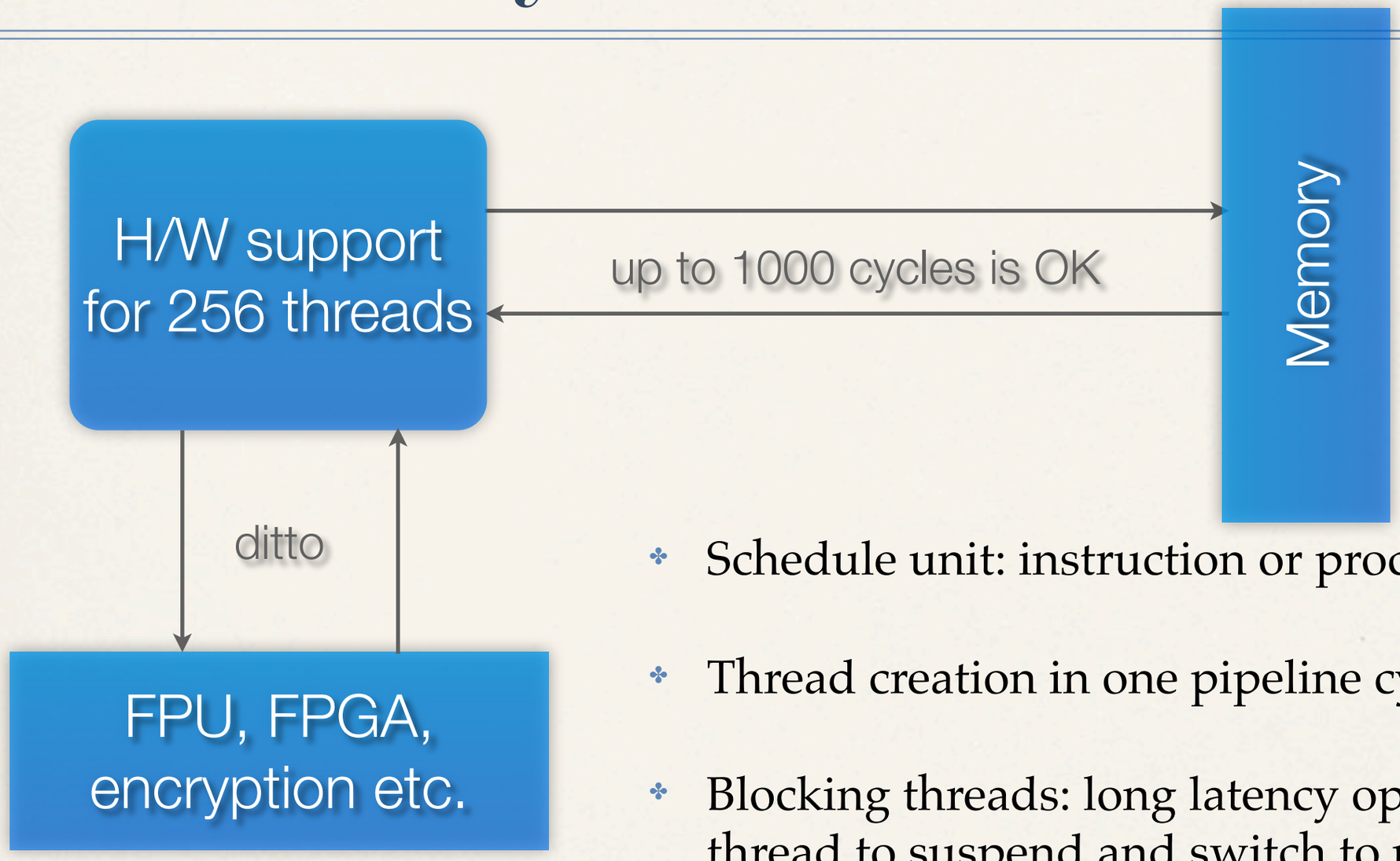
FPU, FPGA, encryption etc.

- Thread creation in one pipeline cycle
- Blocking threads: long latency operations cause thread to suspend and switch to next thread
- Can context switch on every cycle to keep the pipeline full - apparent 1 cycle latencies as long as there are sufficient threads ready

donderdag 2 september 2010

Fine grained in our context means that the schedule unit is the single instruction or processor cycle. We are able to allocate one new thread context per cycle, and when threads suspend on long latency operations control can be transferred to the next thread without pipeline stalls or bubbles. We achieve this by synchronizing through registers, with blocking reads and wake on write. With enough threads active, latencies up to thousands of cycles can be tolerated; i.e. appear to run in one cycle in each thread's time scale. This allows to tolerate memory access latencies as well as latencies to accelerators, FPUs etc which can be shared between cores.

Fine grained hardware concurrency

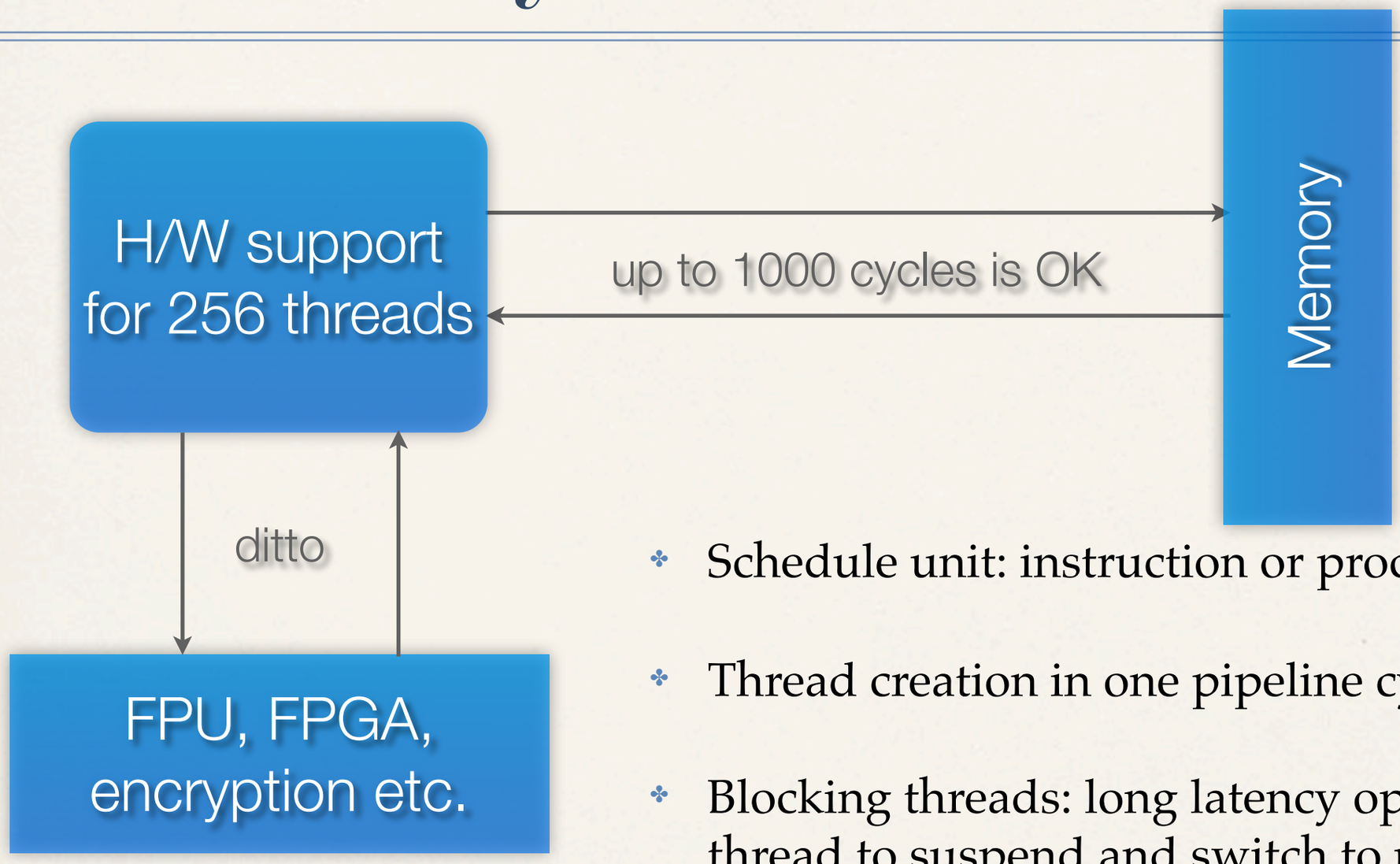


- Schedule unit: instruction or processor cycle
- Thread creation in one pipeline cycle
- Blocking threads: long latency operations cause thread to suspend and switch to next thread
- Can context switch on every cycle to keep the pipeline full - apparent 1 cycle latencies as long as there are sufficient threads ready

donderdag 2 september 2010

Fine grained in our context means that the schedule unit is the single instruction or processor cycle. We are able to allocate one new thread context per cycle, and when threads suspend on long latency operations control can be transferred to the next thread without pipeline stalls or bubbles. We achieve this by synchronizing through registers, with blocking reads and wake on write. With enough threads active, latencies up to thousands of cycles can be tolerated; i.e. appear to run in one cycle in each thread's time scale. This allows to tolerate memory access latencies as well as latencies to accelerators, FPUs etc which can be shared between cores.

Fine grained hardware concurrency

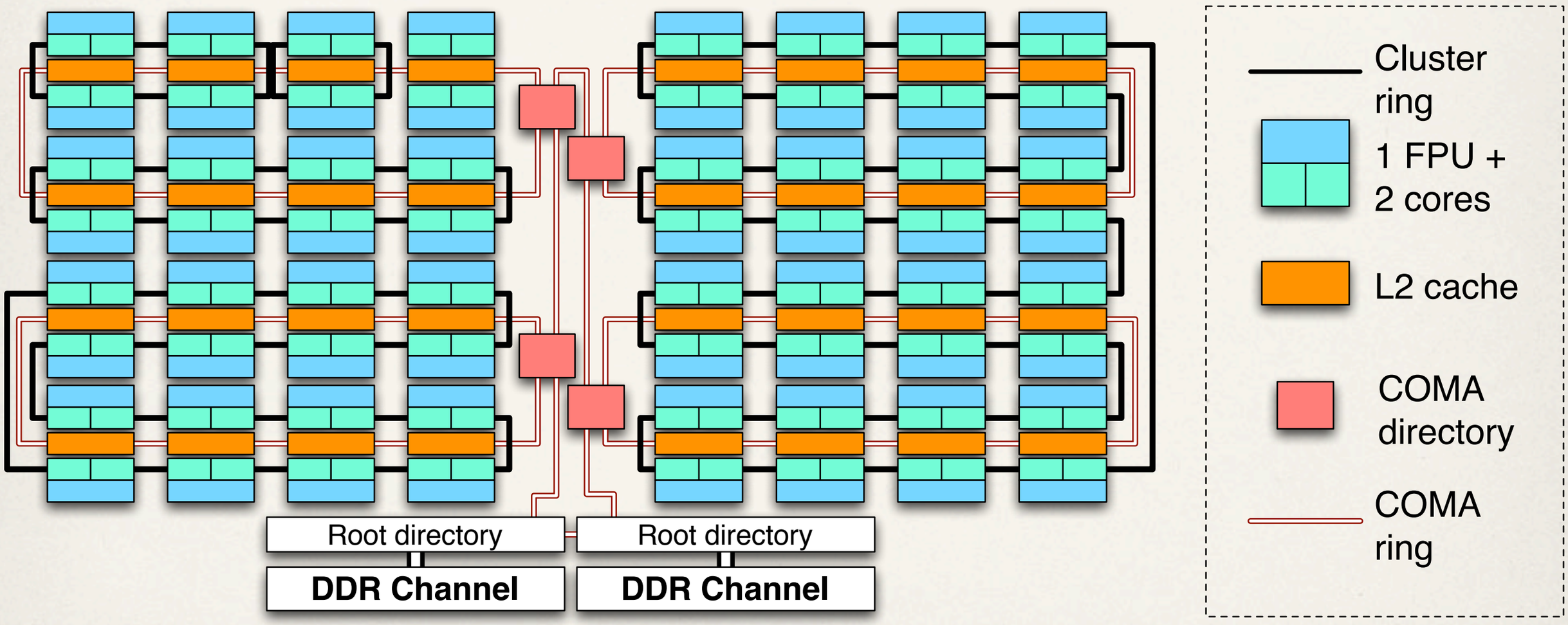


- Schedule unit: instruction or processor cycle
- Thread creation in one pipeline cycle
- Blocking threads: long latency operations cause thread to suspend and switch to next thread
- Can context switch on every cycle to keep the pipeline full - apparent 1 cycle latencies as long as there are sufficient threads ready

donderdag 2 september 2010

Fine grained in our context means that the schedule unit is the single instruction or processor cycle. We are able to allocate one new thread context per cycle, and when threads suspend on long latency operations control can be transferred to the next thread without pipeline stalls or bubbles. We achieve this by synchronizing through registers, with blocking reads and wake on write. With enough threads active, latencies up to thousands of cycles can be tolerated; i.e. appear to run in one cycle in each thread's time scale. This allows to tolerate memory access latencies as well as latencies to accelerators, FPUs etc which can be shared between cores.

Functional overview of the Microgrid



The cluster is the processor; the chip is heterogeneous and general purpose

donderdag 2 september 2010

Here you can see an overall diagram of a Microgrid. This is a specific configuration used for benchmarking; other configurations are possible. In this configuration there are 128 cores sharing 64 FPUs.

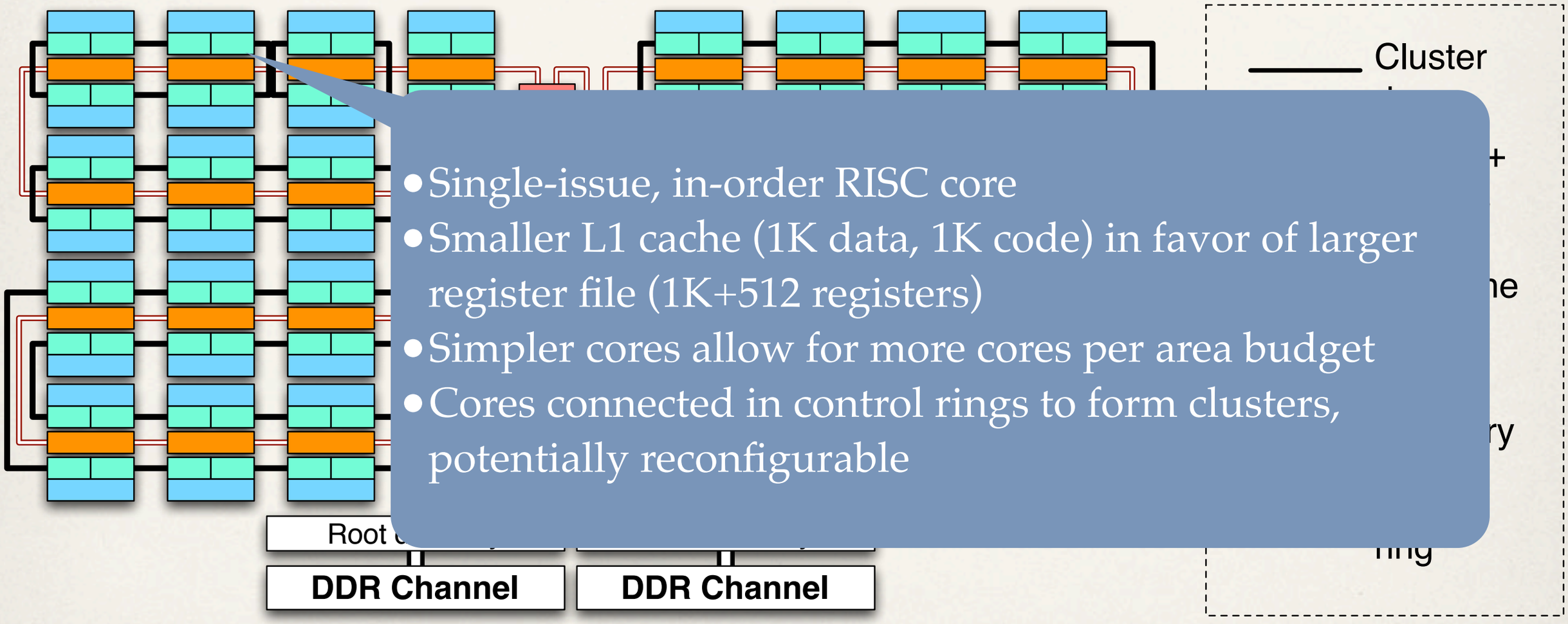
Each core is a simple RISC core at 1GHz without register renaming, branch prediction, multiple issue or SIMD. We also reduce the L1 cache size in favor of a larger register file. The smaller footprint on chip allows to build more cores on the same die area.

The cores are connected in narrow control rings for concurrency management and synchronization, drawn in black in this diagram and may be reconfigurable.

The on-chip memory network is based on a cache-only distributed protocol. In this Microgrid we have 4 rings of 8 L2 caches of 32KB each. The COMA rings are separate from the control rings, and support up to 64GB/s.The top level directories are connected to standard DDR3-1600 interfaces. Cache lines are migrated on use and stay at the point of last use.

The chip supports a single shared address space; no readdressing happens on chip to avoid overheads. VMM happens at the chip boundary.

Functional overview of the Microgrid



- Single-issue, in-order RISC core
- Smaller L1 cache (1K data, 1K code) in favor of larger register file (1K+512 registers)
- Simpler cores allow for more cores per area budget
- Cores connected in control rings to form clusters, potentially reconfigurable

The cluster is the processor; the chip is heterogeneous and general purpose

donderdag 2 september 2010

Here you can see an overall diagram of a Microgrid. This is a specific configuration used for benchmarking; other configurations are possible. In this configuration there are 128 cores sharing 64 FPUs.

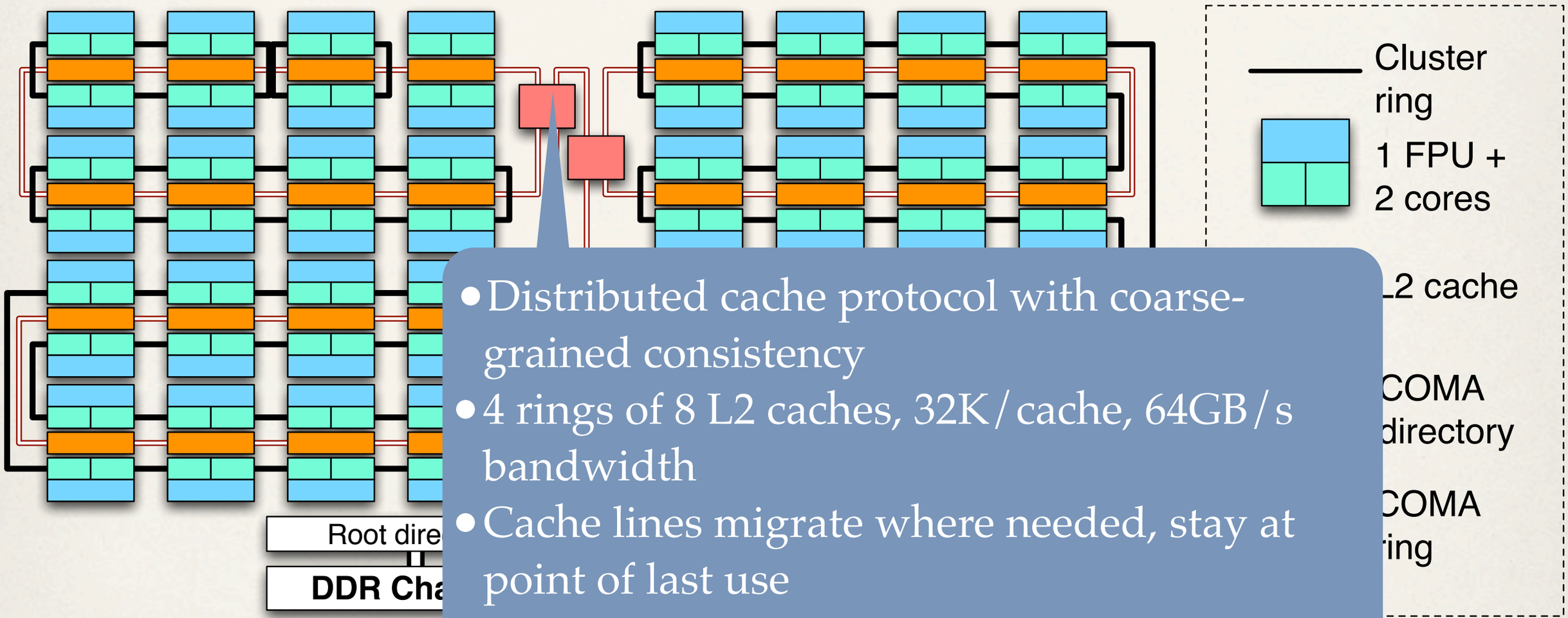
Each core is a simple RISC core at 1GHz without register renaming, branch prediction, multiple issue or SIMD. We also reduce the L1 cache size in favor of a larger register file. The smaller footprint on chip allows to build more cores on the same die area.

The cores are connected in narrow control rings for concurrency management and synchronization, drawn in black in this diagram and may be reconfigurable.

The on-chip memory network is based on a cache-only distributed protocol. In this Microgrid we have 4 rings of 8 L2 caches of 32KB each. The COMA rings are separate from the control rings, and support up to 64GB/s.The top level directories are connected to standard DDR3-1600 interfaces. Cache lines are migrated on use and stay at the point of last use.

The chip supports a single shared address space; no readdressing happens on chip to avoid overheads. VMM happens at the chip boundary.

Functional overview of the Microgrid



- Distributed cache protocol with coarse-grained consistency
- 4 rings of 8 L2 caches, 32K/cache, 64GB/s bandwidth
- Cache lines migrate where needed, stay at point of last use
- COMA rings separate from control rings

The cluster is the processor; the chip is heterogeneous and general purpose

donderdag 2 september 2010

Here you can see an overall diagram of a Microgrid. This is a specific configuration used for benchmarking; other configurations are possible. In this configuration there are 128 cores sharing 64 FPUs.

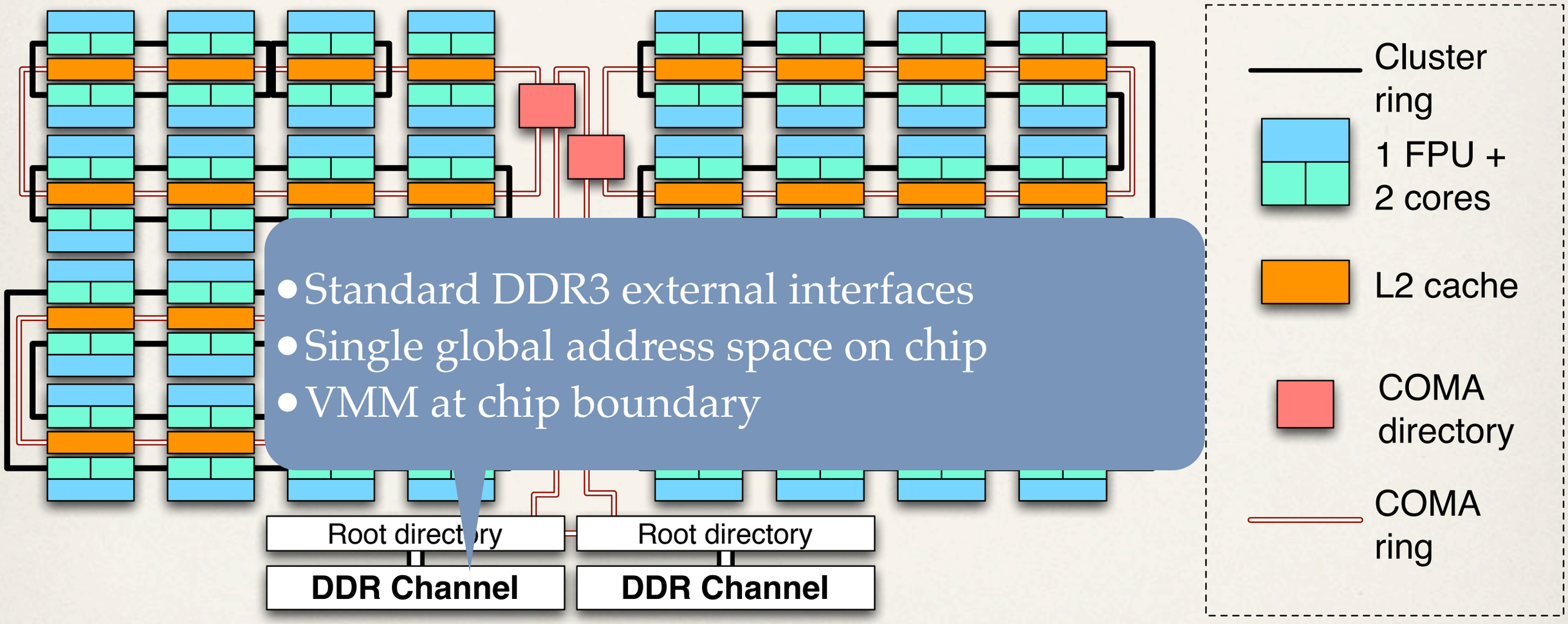
Each core is a simple RISC core at 1GHz without register renaming, branch prediction, multiple issue or SIMD. We also reduce the L1 cache size in favor of a larger register file. The smaller footprint on chip allows to build more cores on the same die area.

The cores are connected in narrow control rings for concurrency management and synchronization, drawn in black in this diagram and may be reconfigurable.

The on-chip memory network is based on a cache-only distributed protocol. In this Microgrid we have 4 rings of 8 L2 caches of 32KB each. The COMA rings are separate from the control rings, and support up to 64GB/s.The top level directories are connected to standard DDR3-1600 interfaces. Cache lines are migrated on use and stay at the point of last use.

The chip supports a single shared address space; no readdressing happens on chip to avoid overheads. VMM happens at the chip boundary.

Functional overview of the Microgrid



The cluster is the processor; the chip is heterogeneous and general purpose

donderdag 2 september 2010

Here you can see an overall diagram of a Microgrid. This is a specific configuration used for benchmarking; other configurations are possible. In this configuration there are 128 cores sharing 64 FPUs.

Each core is a simple RISC core at 1GHz without register renaming, branch prediction, multiple issue or SIMD. We also reduce the L1 cache size in favor of a larger register file. The smaller footprint on chip allows to build more cores on the same die area.

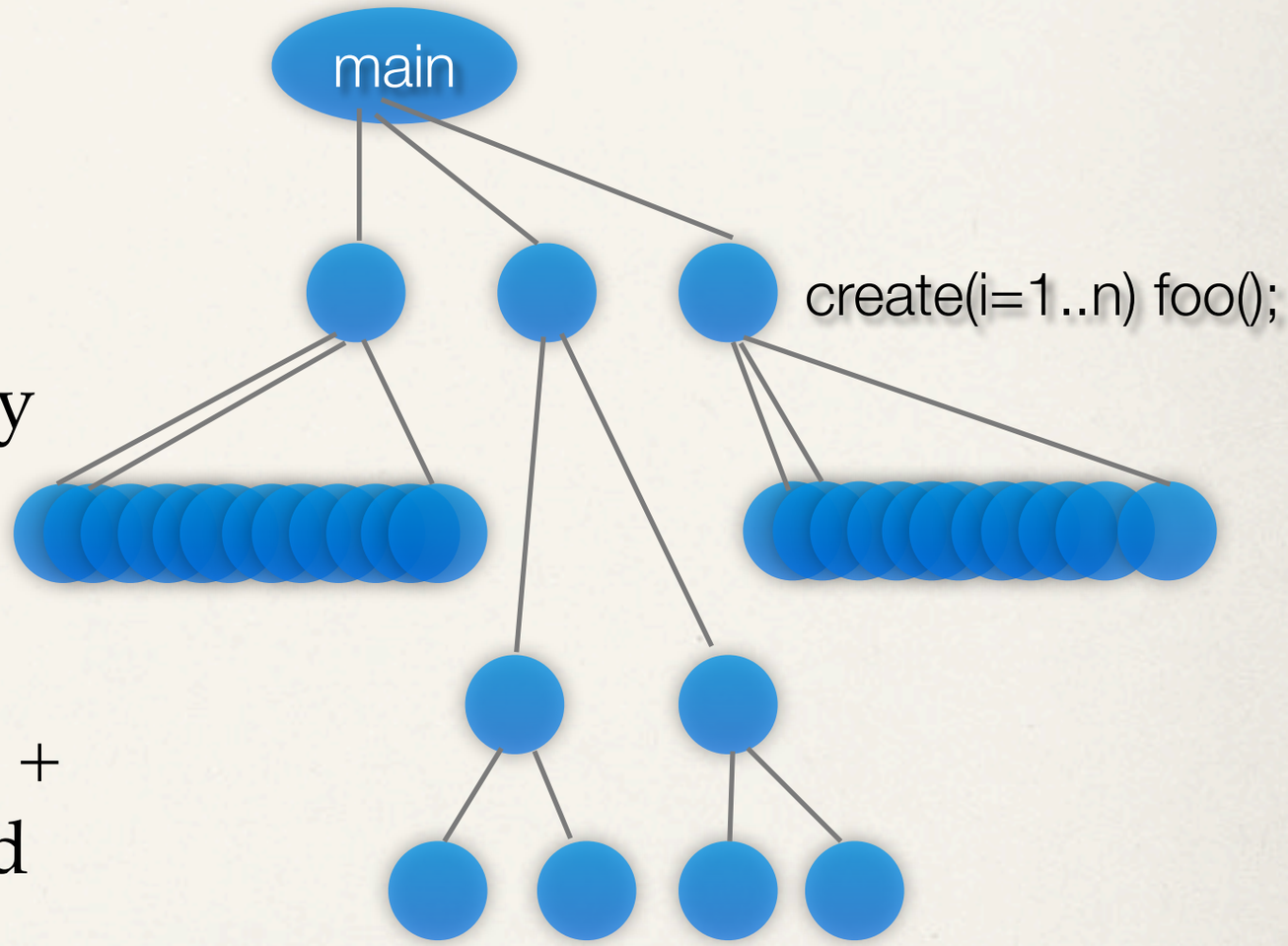
The cores are connected in narrow control rings for concurrency management and synchronization, drawn in black in this diagram and may be reconfigurable.

The on-chip memory network is based on a cache-only distributed protocol. In this Microgrid we have 4 rings of 8 L2 caches of 32KB each. The COMA rings are separate from the control rings, and support up to 64GB/s.The top level directories are connected to standard DDR3-1600 interfaces. Cache lines are migrated on use and stay at the point of last use.

The chip supports a single shared address space; no readdressing happens on chip to avoid overheads. VMM happens at the chip boundary.

SVP execution model

- Dataflow concurrency, with clustering of related tasks in *families*
- Programs express all concurrency available; on-chip hardware concurrency OS automatically spreads concurrency over available cores / thread contexts + sequences threads when required
- Dependencies are restricted to avoid deadlock on composition



donderdag 2 september 2010

To program the Microgrid we use an abstract programming model called SVP. SVP stands for the Self-Adaptive Virtual Processor. SVP embodies dataflow concurrency, where the availability of dependencies allow further progresss of execution. Related tasks are clustered in families of threads with the same initial PC and different indices.

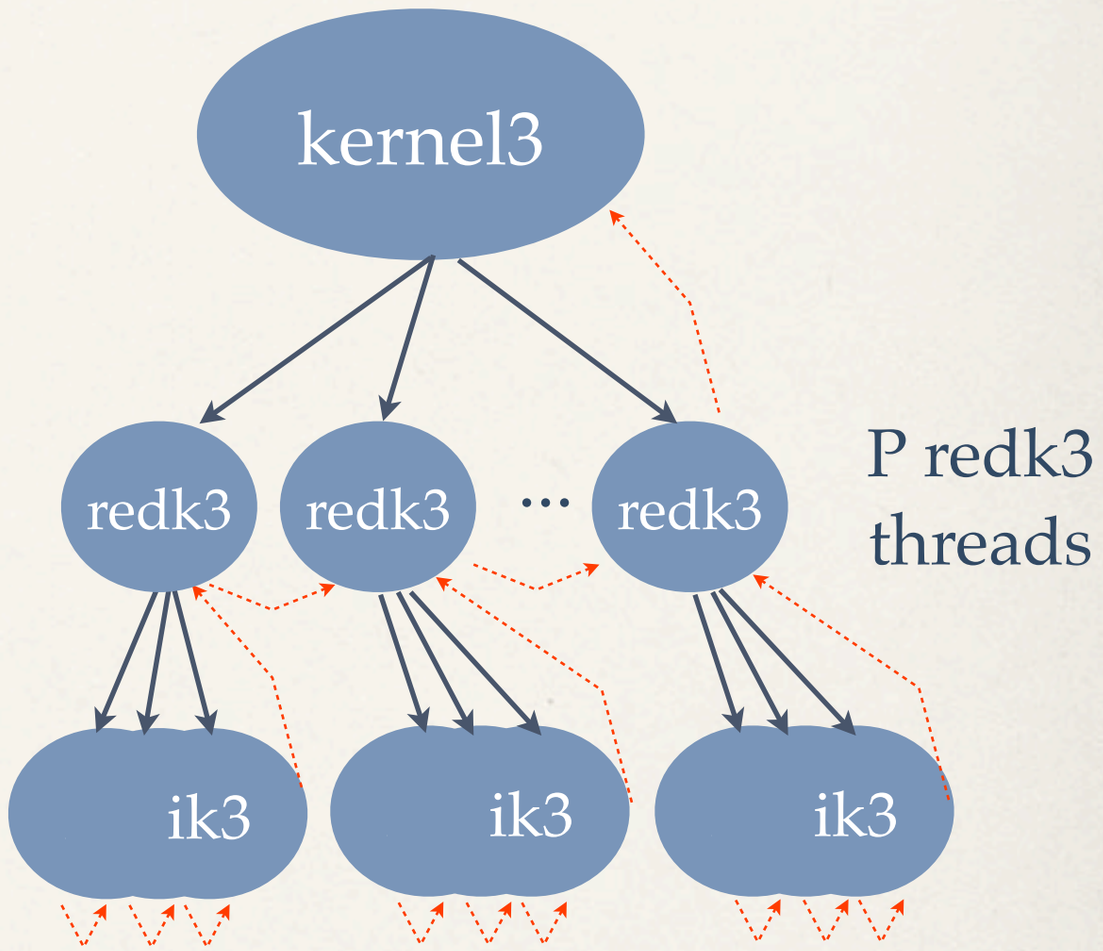
The purpose of SVP is to annotate in programs all the concurrency available in algorithms. This information is compiled down to the ISA extensions. The on-chip hardware OS then uses this information to spread concurrency over cores and thread contexts, and possibly eliminates extra concurrency by sequential scheduling.

By restricting programs to use only forward only dependency patterns we guarantee that this sequential scheduling and composition are always possible.

SVP intermediate language: μTC

SVP currently captured in μTC, extends C99:

```
thread kernel3(shared double Q,
               int N, double Z[N], double X[N])
{
   int P = get_ncores();
   create(DEFAULT; 0; P)
      redk3(Qr = 0, Z, X, N/P);
   sync();
   Q = Qr;
}
thread redk3(shared double Q,
             double*Z, double *X, int span) {
   index ri;
   create(LOCAL; ri * span; (ri+1) * span)
         ik3(Qr = 0, Z, X);
   sync();
   Q += Qr;
}
thread ik3(shared double Q,
           double*Z, double *X) {
   index i;
   Q += Z[i]*X[i];
}
```



μTC designed as target-neutral IR for compilers

Work ongoing to translate from SAC and sequential C

donderdag 2 september 2010

SVP is currently captured in a set of extensions to C99, that we call μTC. μTC is not intended to be used by programmers; in the context of Apple-CORE we target μTC from the array language SAC and sequential C through a parallelizing compiler.

However I give an idea of what this provides:

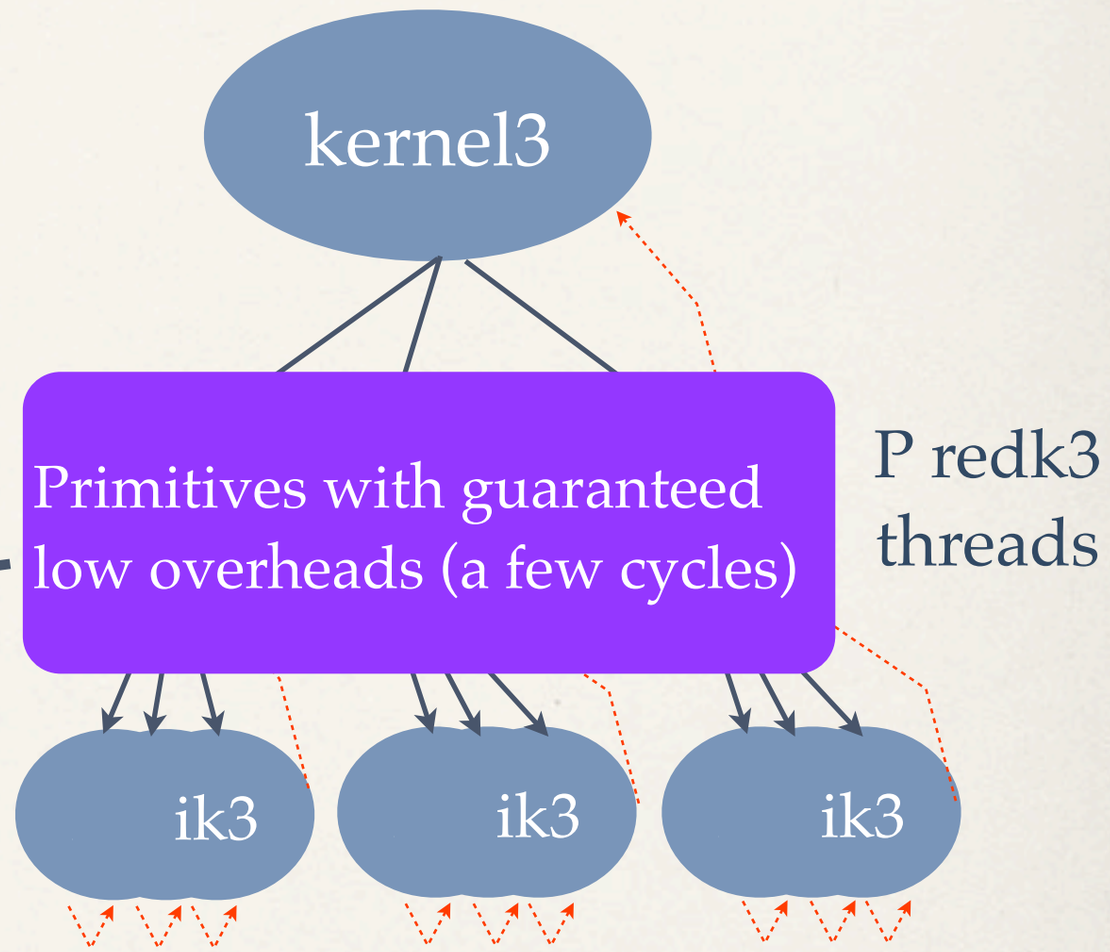
This example implements an inner vector product using a parallel reduction. You see here 3 thread functions, with the kernel entry point at the top. This creates a family using the "default" place, which means the entire cluster. The family is defined to contain P threads where P is the number of cores in the cluster. The family runs this function in each thread, and each thread is identified by "ri". Each redk3 thread further creates one family of N/P threads running function ik3. The keyword LOCAL here hints that the concurrency should be kept local relative to "redk3"; intuitively this means on the same core if redk3 is spread over multiple cores.

An important point here is that these primitives are guaranteed to have extremely low overheads. For example the index is prepopulated in a register when a thread starts. The placement operations and the create instructions are also resolved within a few cycles.

SVP intermediate language: µTC

SVP currently captured in μTC, extends C99:

```
thread kernel3(shared double Q,
               int N, double Z[N], double X[N])
{
   int P = get_ncores();
   create(DEFAULT; 0; P)
      redk3(Qr = 0, Z, X, N/P);
   sync();
   Q = Qr;
}
thread redk3(shared double Q,
             double*Z, double *X, int span) {
   index ri;
   create(LOCAL; ri * span; (ri+1) * span)
         ik3(Qr = 0, Z, X);
   sync();
   Q += Qr;
}
thread ik3(shared double Q,
           double*Z, double *X) {
   index i;
   Q += Z[i]*X[i];
}
```



μTC designed as target-neutral IR for compilers

Work ongoing to translate from SAC and sequential C

donderdag 2 september 2010

SVP is currently captured in a set of extensions to C99, that we call μTC. μTC is not intended to be used by programmers; in the context of Apple-CORE we target μTC from the array language SAC and sequential C through a parallelizing compiler.

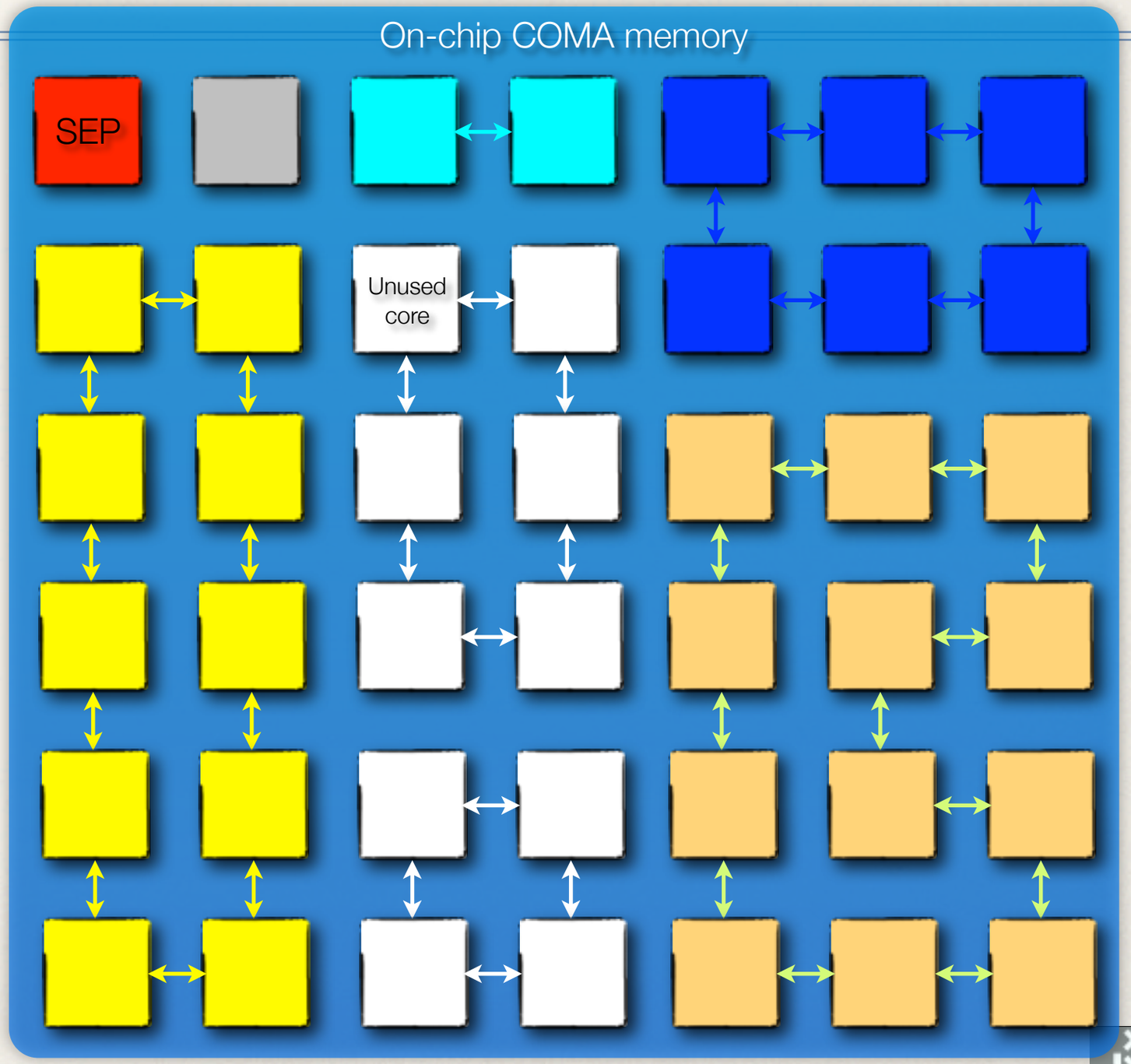
However I give an idea of what this provides:

This example implements an inner vector product using a parallel reduction. You see here 3 thread functions, with the kernel entry point at the top. This creates a family using the "default" place, which means the entire cluster. The family is defined to contain P threads where P is the number of cores in the cluster. The family runs this function in each thread, and each thread is identified by "ri". Each redk3 thread further creates one family of N/P threads running function ik3. The keyword LOCAL here hints that the concurrency should be kept local relative to "redk3"; intuitively this means on the same core if redk3 is spread over multiple cores.

An important point here is that these primitives are guaranteed to have extremely low overheads. For example the index is prepopulated in a register when a thread starts. The placement operations and the create instructions are also resolved within a few cycles.

Resources are allocated dynamically

- Grid of DRISC cores with clusters fixed or configured dynamically
- Software components mapped to separate clusters
- **SEP**: resource manager
- Families delegated to clusters using a **place**
- named place = cluster
  local = same core
  default = same cluster



donderdag 2 september 2010

On a larger scale, we assume that entire applications are clustered into components. To match this, we have multiple clusters of cores on chip, either statically or dynamically configured. When execution reaches a composition point, the following protocol occurs.

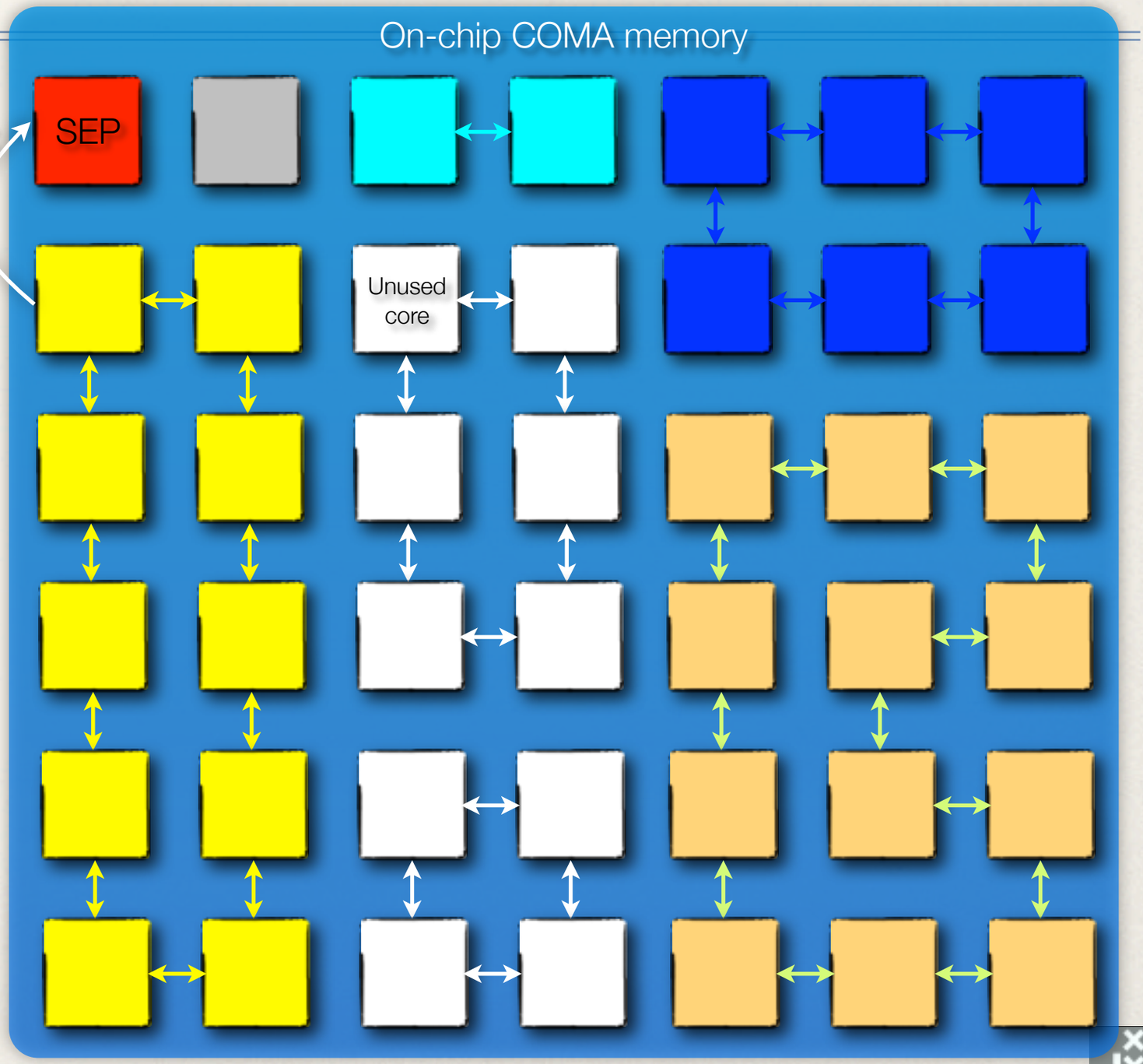
Assuming a component requires execution of another component. First it requests a resource to the local resource manager. The resource is configured on chip, and a placement identifier is returned .The component uses this, until the computation is complete. Then the resource is released.

Within an allocation, the component can further use the LOCAL and DEFAULT pseudo-places that restrict concurrency within the allocated resource.

This system is hierachically defined; in principle allows for high utilization of resources. From this point, we are left with the selection issue: how to size resources depending on application demand? This is where we propose a performance model.

Resources are allocated dynamically

- Grid of DRISC cores with clusters fixed or configured dynamically
- Software components mapped to separate clusters
- **SEP**: resource manager
- Families delegated to clusters using a **place**
- named place = cluster
  local = same core
  default = same cluster



donderdag 2 september 2010

On a larger scale, we assume that entire applications are clustered into components. To match this, we have multiple clusters of cores on chip, either statically or dynamically configured. When execution reaches a composition point, the following protocol occurs.

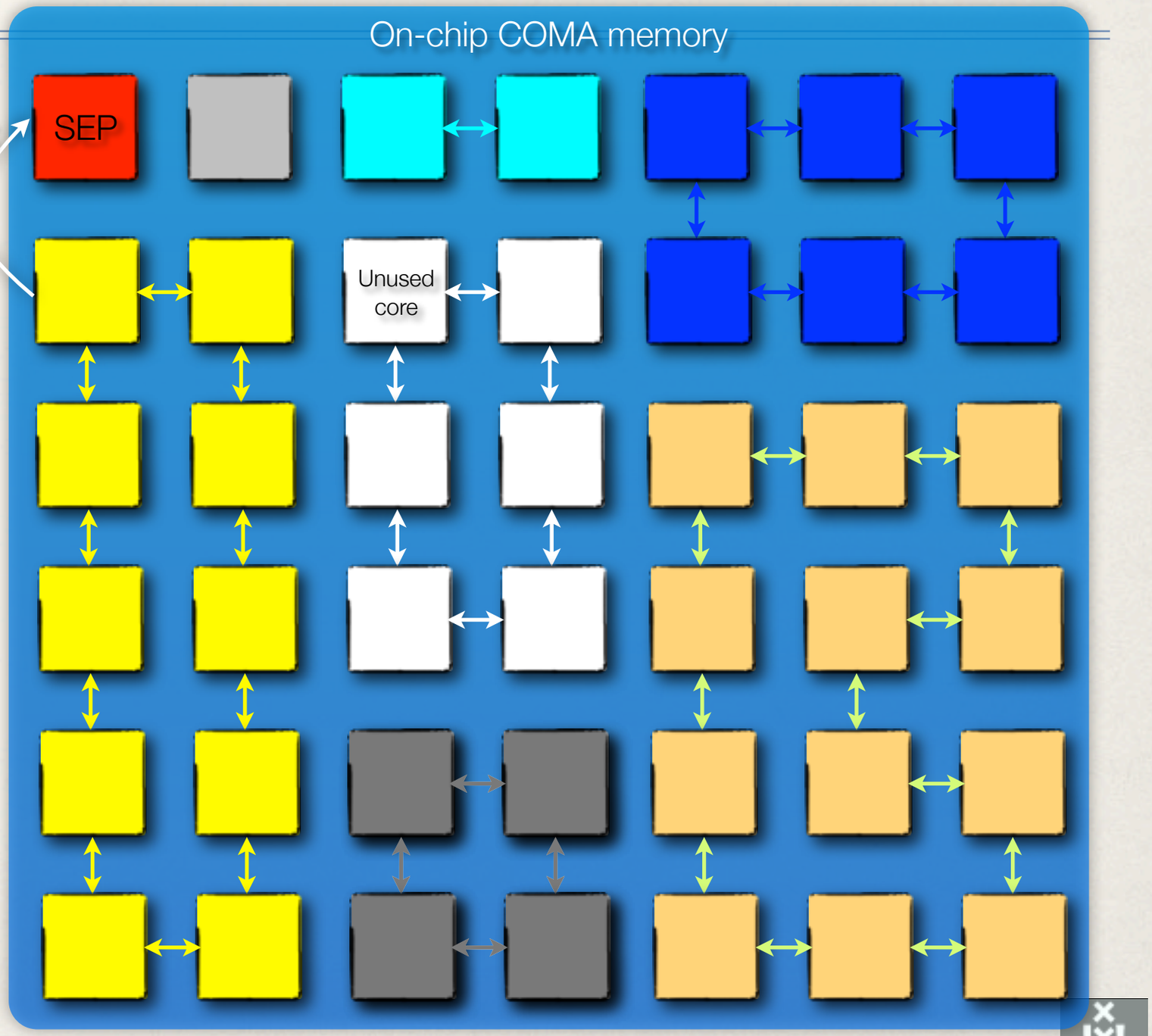
Assuming a component requires execution of another component. First it requests a resource to the local resource manager. The resource is configured on chip, and a placement identifier is returned .The component uses this, until the computation is complete. Then the resource is released.

Within an allocation, the component can further use the LOCAL and DEFAULT pseudo-places that restrict concurrency within the allocated resource.

This system is hierachically defined; in principle allows for high utilization of resources. From this point, we are left with the selection issue: how to size resources depending on application demand? This is where we propose a performance model.

Resources are allocated dynamically

- Grid of DRISC cores with clusters fixed or configured dynamically
- Software components mapped to separate clusters
- **SEP**: resource manager
- Families delegated to clusters using a **place**
- named place = cluster
  local = same core
  default = same cluster



donderdag 2 september 2010

On a larger scale, we assume that entire applications are clustered into components. To match this, we have multiple clusters of cores on chip, either statically or dynamically configured. When execution reaches a composition point, the following protocol occurs.

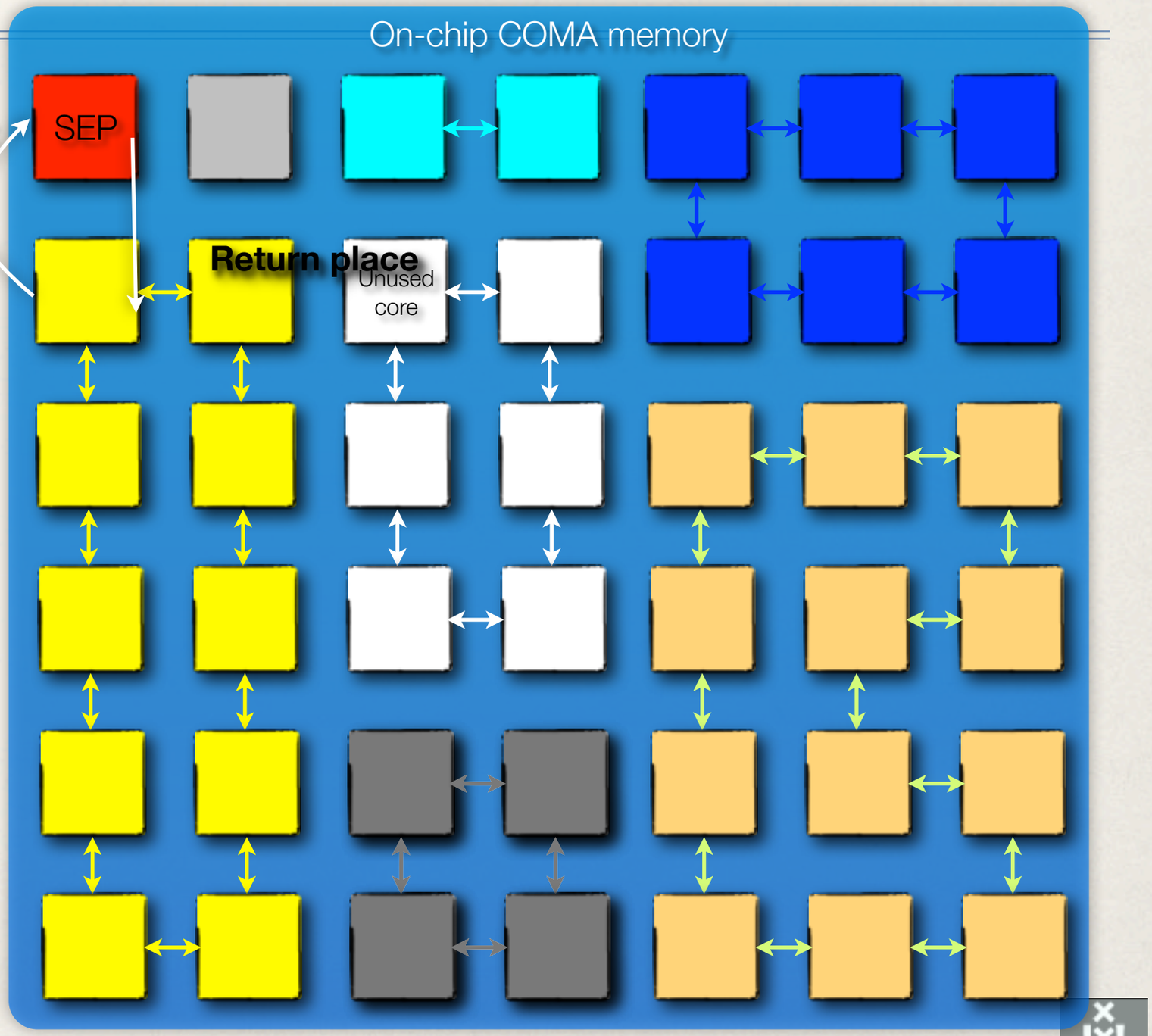
Assuming a component requires execution of another component. First it requests a resource to the local resource manager. The resource is configured on chip, and a placement identifier is returned .The component uses this, until the computation is complete. Then the resource is released.

Within an allocation, the component can further use the LOCAL and DEFAULT pseudo-places that restrict concurrency within the allocated resource.

This system is hierachically defined; in principle allows for high utilization of resources. From this point, we are left with the selection issue: how to size resources depending on application demand? This is where we propose a performance model.

Resources are allocated dynamically

- Grid of DRISC cores with clusters fixed or configured dynamically
- Software components mapped to separate clusters
- **SEP**: resource manager
- Families delegated to clusters using a **place**
- named place = cluster
  local = same core
  default = same cluster



donderdag 2 september 2010

On a larger scale, we assume that entire applications are clustered into components. To match this, we have multiple clusters of cores on chip, either statically or dynamically configured. When execution reaches a composition point, the following protocol occurs.

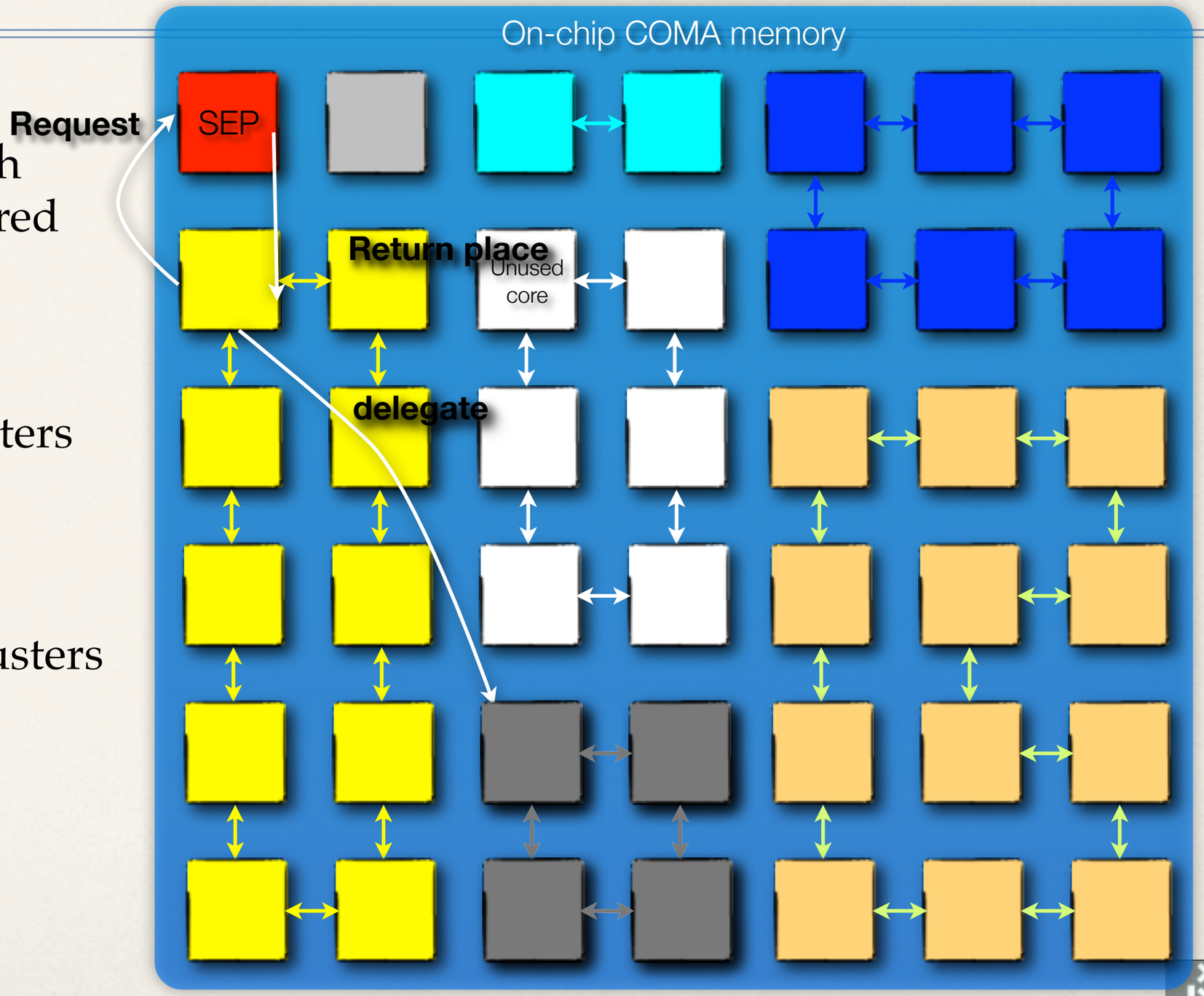
Assuming a component requires execution of another component. First it requests a resource to the local resource manager. The resource is configured on chip, and a placement identifier is returned .The component uses this, until the computation is complete. Then the resource is released.

Within an allocation, the component can further use the LOCAL and DEFAULT pseudo-places that restrict concurrency within the allocated resource.

This system is hierachically defined; in principle allows for high utilization of resources. From this point, we are left with the selection issue: how to size resources depending on application demand? This is where we propose a performance model.

Resources are allocated dynamically

- Grid of DRISC cores with clusters fixed or configured dynamically
- Software components mapped to separate clusters
- **SEP**: resource manager
- Families delegated to clusters using a **place**
- named place = cluster
  local = same core
  default = same cluster



donderdag 2 september 2010

On a larger scale, we assume that entire applications are clustered into components. To match this, we have multiple clusters of cores on chip, either statically or dynamically configured. When execution reaches a composition point, the following protocol occurs.

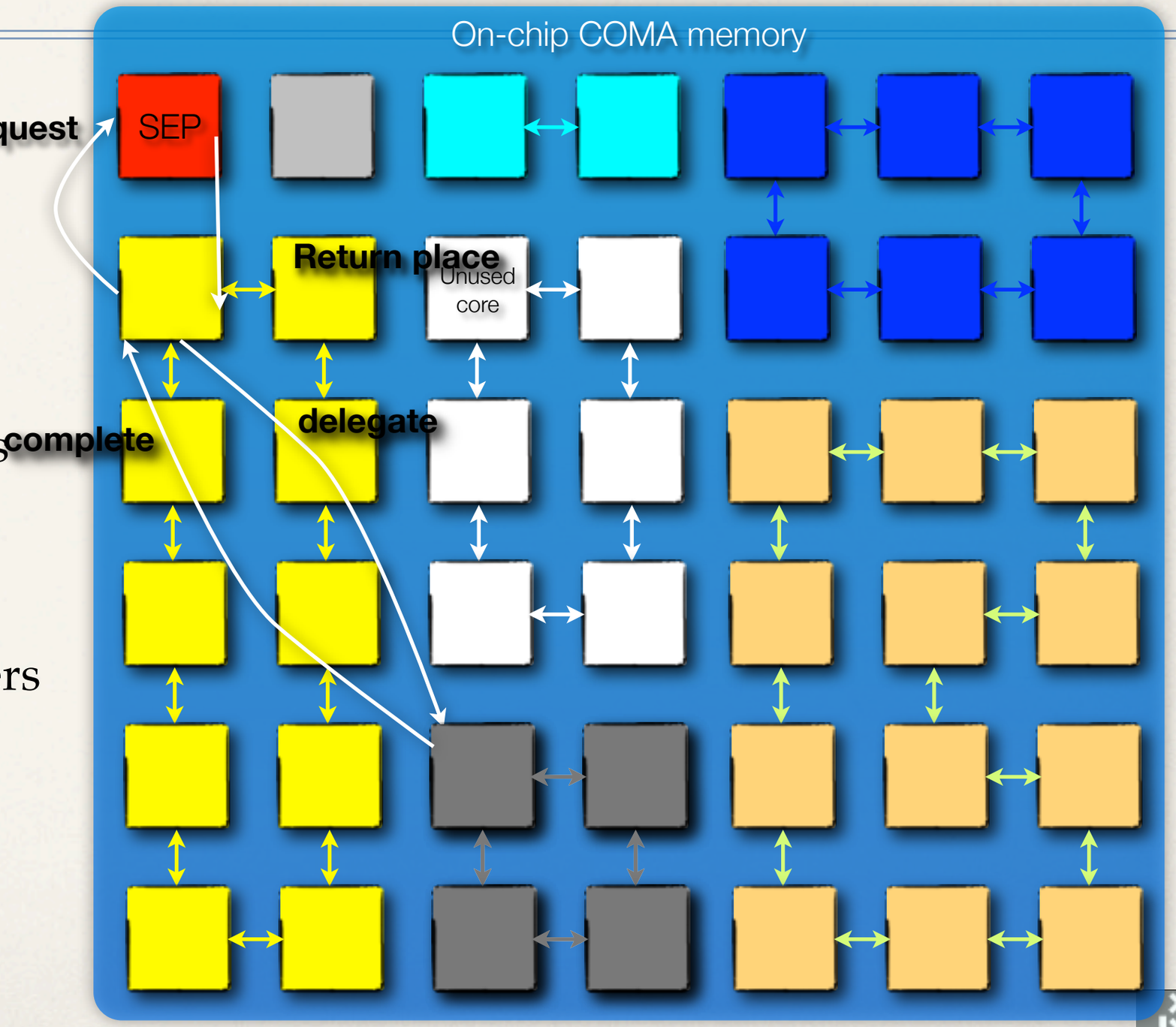
Assuming a component requires execution of another component. First it requests a resource to the local resource manager. The resource is configured on chip, and a placement identifier is returned .The component uses this, until the computation is complete. Then the resource is released.

Within an allocation, the component can further use the LOCAL and DEFAULT pseudo-places that restrict concurrency within the allocated resource.

This system is hierachically defined; in principle allows for high utilization of resources. From this point, we are left with the selection issue: how to size resources depending on application demand? This is where we propose a performance model.

Resources are allocated dynamically

- Grid of DRISC cores with clusters fixed or configured dynamically
- Software components mapped to separate clusters
- **SEP**: resource manager
- Families delegated to clusters using a **place**
- named place = cluster
  local = same core
  default = same cluster



donderdag 2 september 2010

On a larger scale, we assume that entire applications are clustered into components. To match this, we have multiple clusters of cores on chip, either statically or dynamically configured. When execution reaches a composition point, the following protocol occurs.

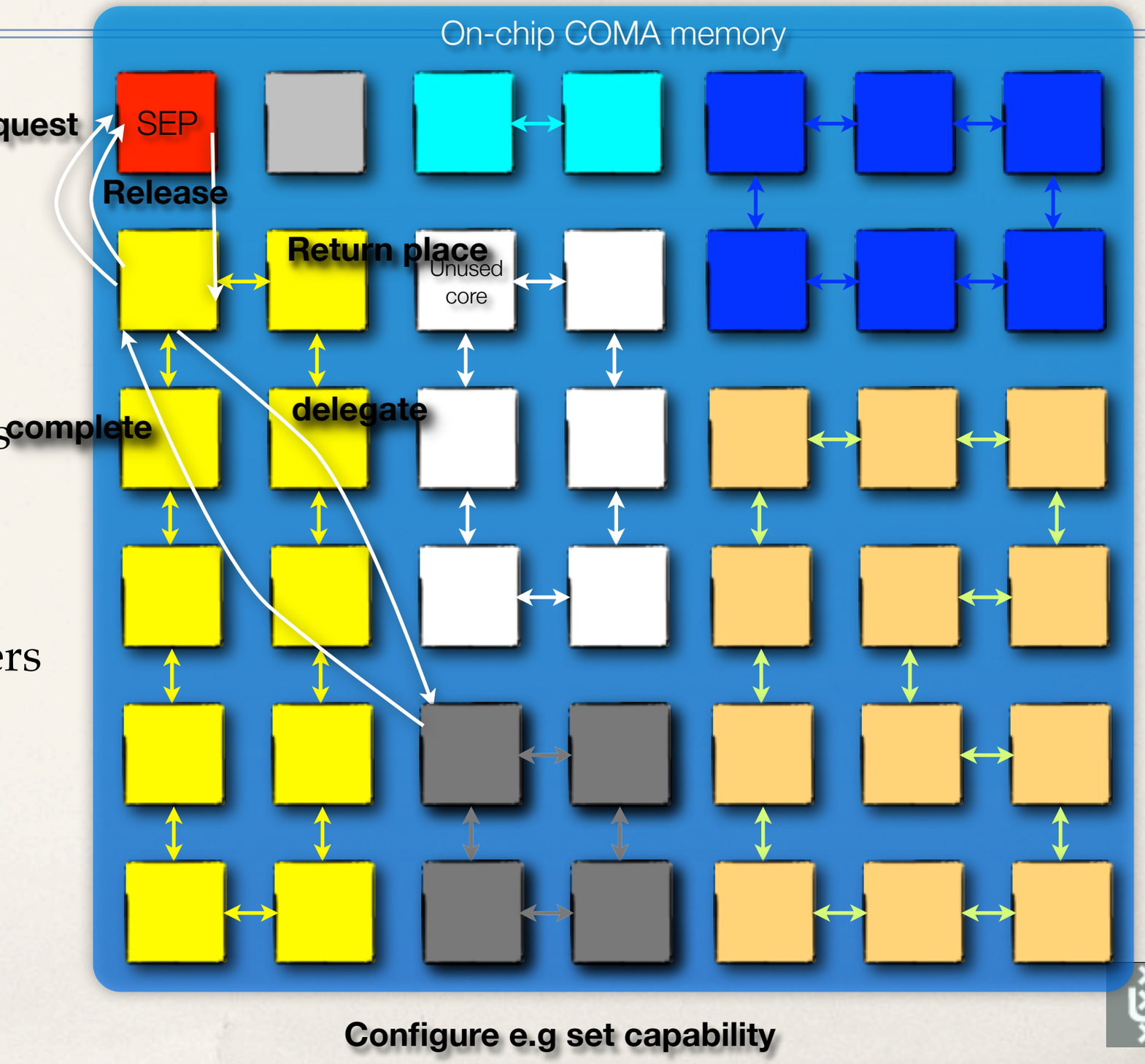
Assuming a component requires execution of another component. First it requests a resource to the local resource manager. The resource is configured on chip, and a placement identifier is returned .The component uses this, until the computation is complete. Then the resource is released.

Within an allocation, the component can further use the LOCAL and DEFAULT pseudo-places that restrict concurrency within the allocated resource.

This system is hierachically defined; in principle allows for high utilization of resources. From this point, we are left with the selection issue: how to size resources depending on application demand? This is where we propose a performance model.

Resources are allocated dynamically

- Grid of DRISC cores with clusters fixed or configured dynamically
- Software components mapped to separate clusters
- **SEP**: resource manager
- Families delegated to clusters using a **place**
- named place = cluster
  local = same core
  default = same cluster



donderdag 2 september 2010

On a larger scale, we assume that entire applications are clustered into components. To match this, we have multiple clusters of cores on chip, either statically or dynamically configured. When execution reaches a composition point, the following protocol occurs.

Assuming a component requires execution of another component. First it requests a resource to the local resource manager. The resource is configured on chip, and a placement identifier is returned .The component uses this, until the computation is complete. Then the resource is released.

Within an allocation, the component can further use the LOCAL and DEFAULT pseudo-places that restrict concurrency within the allocated resource.

This system is hierachically defined; in principle allows for high utilization of resources. From this point, we are left with the selection issue: how to size resources depending on application demand? This is where we propose a performance model.

Performance envelope

- *Arithmetic intensity:* number of FP operations relative to other operations
  - Data-parallel computations: **AI1 = FPops ÷ ops issued**
    (i.e. all insns. appear to run in 1 cycle with latency hiding)

    peak FLOP/s = P × AI1
    with P = number of cores × frequency, we use 1GHz cores
  - With thread-to-thread dependencies: **AI1' = FPops ÷ (ops issued + latency of dependent operations)**
    (i.e. dependent operations must be executed in sequence, latency cannot be hidden)
  - With memory communication: **AI2 = FPops ÷ bytes communicated** to/from memory

    peak FLOP/s = IO × AI2
    with IO = max bandwidth of ext. interface (25Gb/s) or internal ring (4-64Gb/s)
- Performance **constrained** by either AI1 if $P \times AI1 \leq IO \times AI2$
  or AI2 if $P \times AI1 \geq IO \times AI2$
- AI1 and AI2 are **properties of the (compiled) program code**, *independent from target architecture parameters*
  can be extracted automatically by compiler / assembler after thread generating hot spots are identified

donderdag 2 september 2010

Here we propose the definition of arithmetic intensity. The ratio of FP operations relative to other operations.

The first arithmetic intensity is the ratio of FP operations to the number of operations issued. Assuming that latency can be hidden, the peak performance achievable from optimal pipeline use is the number of cores times AI1.

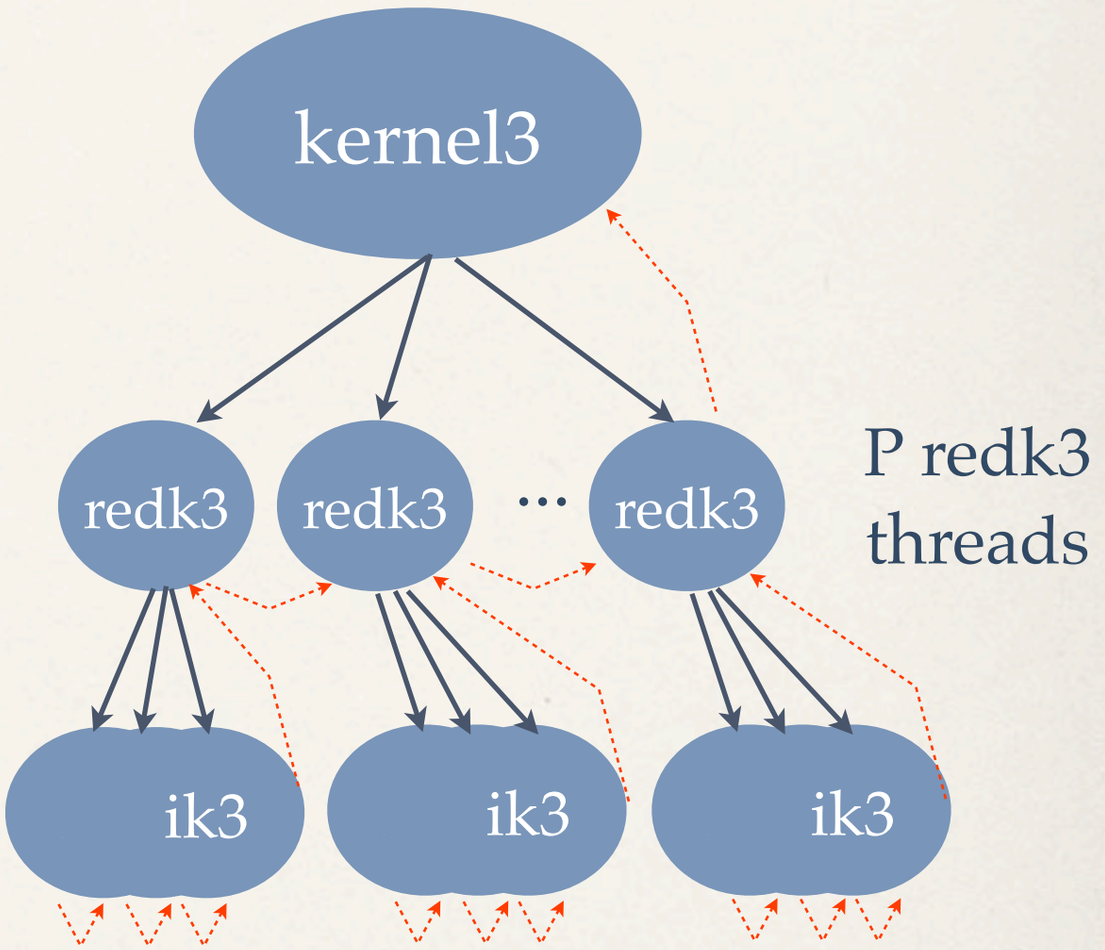
When thread-to-thread dependencies force a sequential schedule for some instructions, we derive AI1' by adding the forced latencies to the denominator.

The second arithmetic intensity is the ratio of FP operations to the number of bytes exchanged with memory. The peak performance achievable from optimal use of the memory network is IO times AI2, where IO is either the internal or external bandwidth, whichever is lower.

We are able to predict performance as follows: the peak performance reachable by a given algorithm is given either by AI1 (if the code is compute-bound) or AI2 (if the code is memory-bound). It is interesting to note that these are essentially properties of the compiled program code.

Example code: Inner product

```
thread kernel3(shared double Q,
               int N, double Z[N], double X[N])
{
   int P = get_ncores();
   create(DEFAULT; 0; P)
      redk3(Qr = 0, Z, X, N/P);
   sync();
   Q = Qr;
}
thread redk3(shared double Q,
             double*Z, double *X, int span) {
   index ri;
   create(LOCAL; ri * span; (ri+1) * span)
         ik3(Qr = 0, Z, X);
   sync();
   Q += Qr;
}
thread ik3(shared double Q,
           double*Z, double *X) {
   index i;
   Q += Z[i]*X[i];
}
```



P redk3 threads

N ik3 threads, N/P per core

communication / reduction of Q

donderdag 2 september 2010

I come back to the example from earlier.

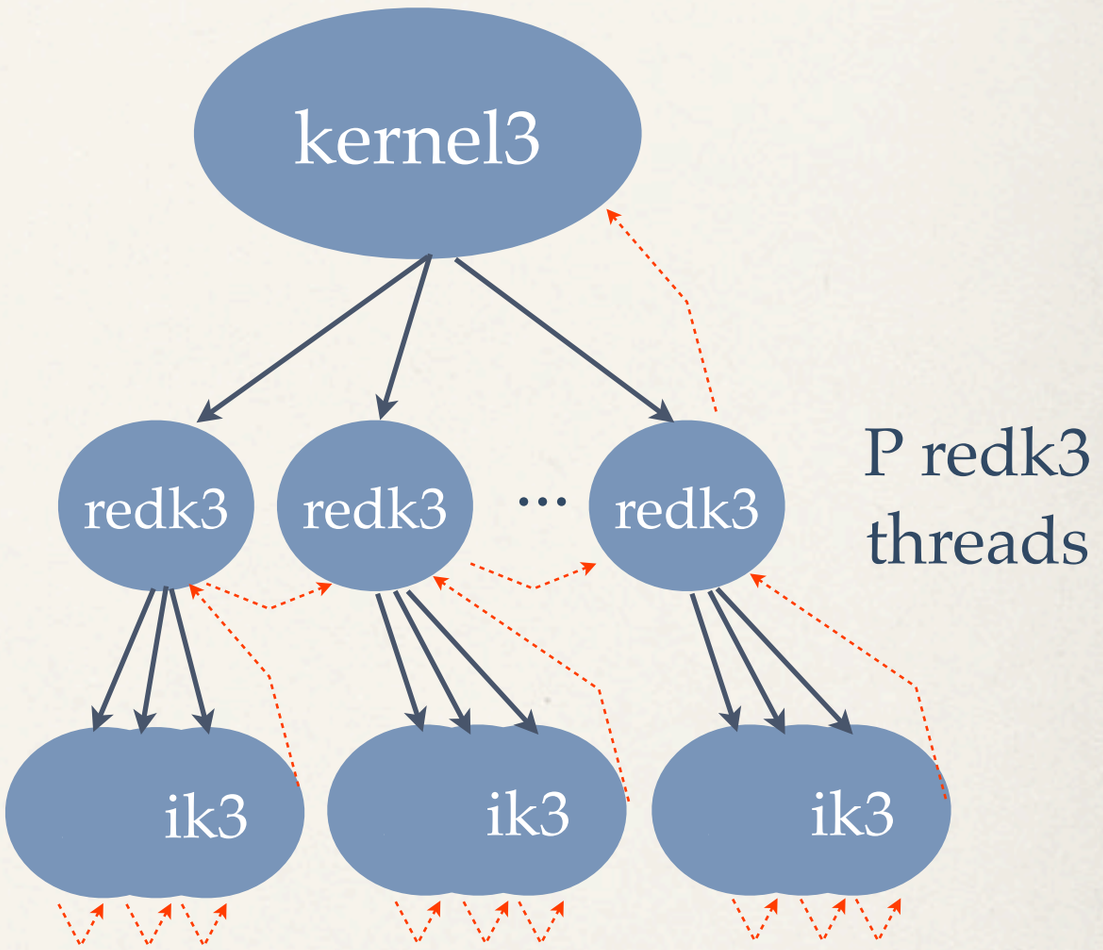
First you can see that on the dependent computations the common prefix is independent and therefore will benefit from latency hiding.

Then we note that execution is dominated by the inner computation. This is compiled to 7 instructions, including 2 FP ops and 2 memory operations. This gives AI1 and AI2 as follows. However the 2nd FP operation is dependent, which lowers AI1' as follows.

Example code: Inner product

```
thread kernel3(shared double Q,
               int N, double Z[N], double X[N])
{
   int P = get_ncores();
   create(DEFAULT; 0; P)
      redk3(Qr = 0, Z, X, N/P);
   sync();
   Q = Qr;
}
thread redk3(shared double Q,
             double*Z, double *X, int span) {
   index ri;
   create(LOCAL; ri * span; (ri+1) * span)
         ik3(Qr = 0, Z, X);
   sync();
   Q += Qr;
}
thread ik3(shared double Q,
           double*Z, double *X) {
   index i;
   Q += Z[i]*X[i];
}
```

independent prefix



P redk3 threads

N ik3 threads, N/P per core

communication / reduction of Q

donderdag 2 september 2010

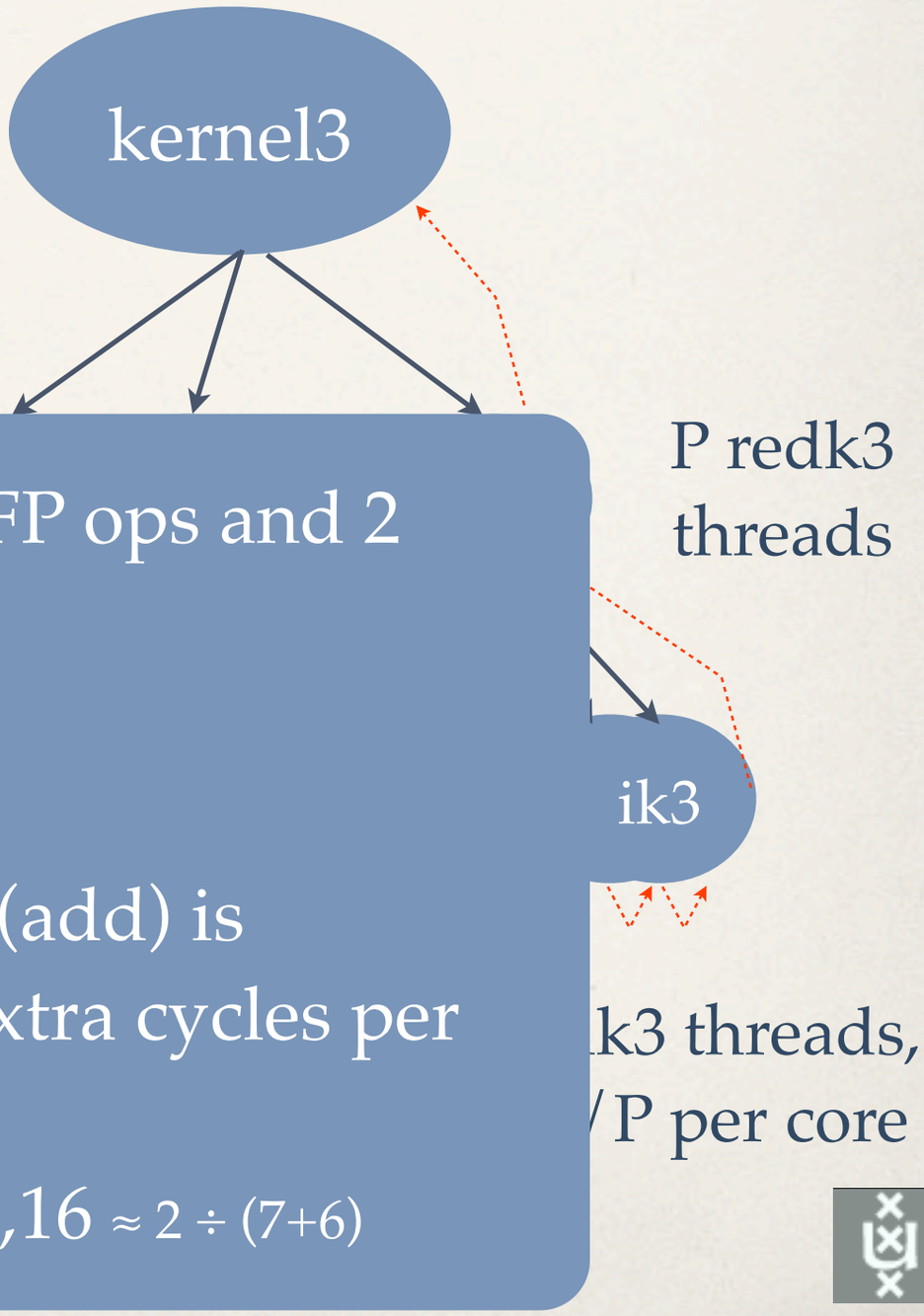
I come back to the example from earlier.

First you can see that on the dependent computations the common prefix is independent and therefore will benefit from latency hiding.

Then we note that execution is dominated by the inner computation. This is compiled to 7 instructions, including 2 FP ops and 2 memory operations. This gives AI1 and AI2 as follows. However the 2nd FP operation is dependent, which lowers AI1' as follows.

Example code: Inner product

```
thread kernel3(shared double Q,
               int N, double Z[N], double X[N])
{
   int P = get_ncores();
   create(DEFAULT; 0; P)
      redk3(Qr = 0, Z, X, N/P);
   sync();
   Q = Qr;
}
thread redk3(shared double
             double*Z, doub
   index ri;
   create(LOCAL; ri * span;
         ik3(Qr = 0, Z, X);
   sync();
   Q += Qr;
}
thread ik3(shared double Q,
           double*Z, double
   index i;
   Q += Z[i]*X[i];
}
```



- 7 instructions, incl 2 FP ops and 2 memops:
  - $AI1 = 2 \div 7 \approx 0{,}29$
  - $AI2 = 2 \div 16 = 0{,}125$
- However, 2nd FP op (add) is dependent, thus 6-11 extra cycles per reduction:

$2 \div (7+11) \approx 0{,}11 \leq AI1' \leq 0{,}16 \approx 2 \div (7+6)$

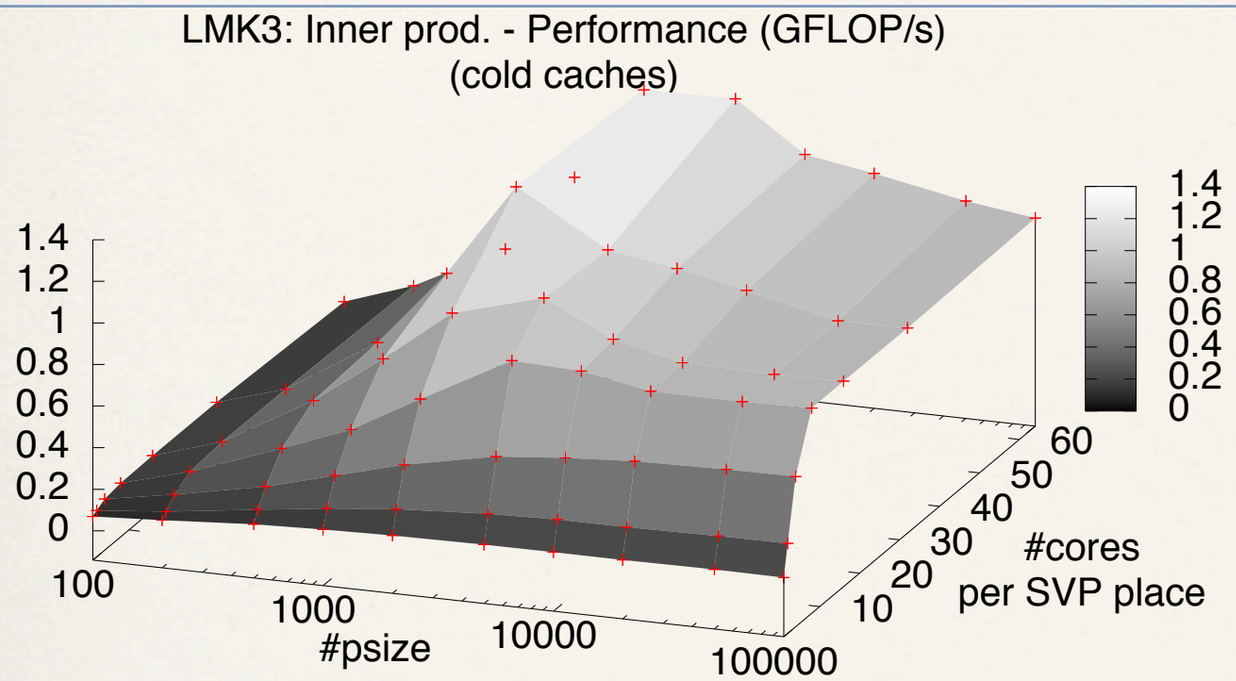
donderdag 2 september 2010

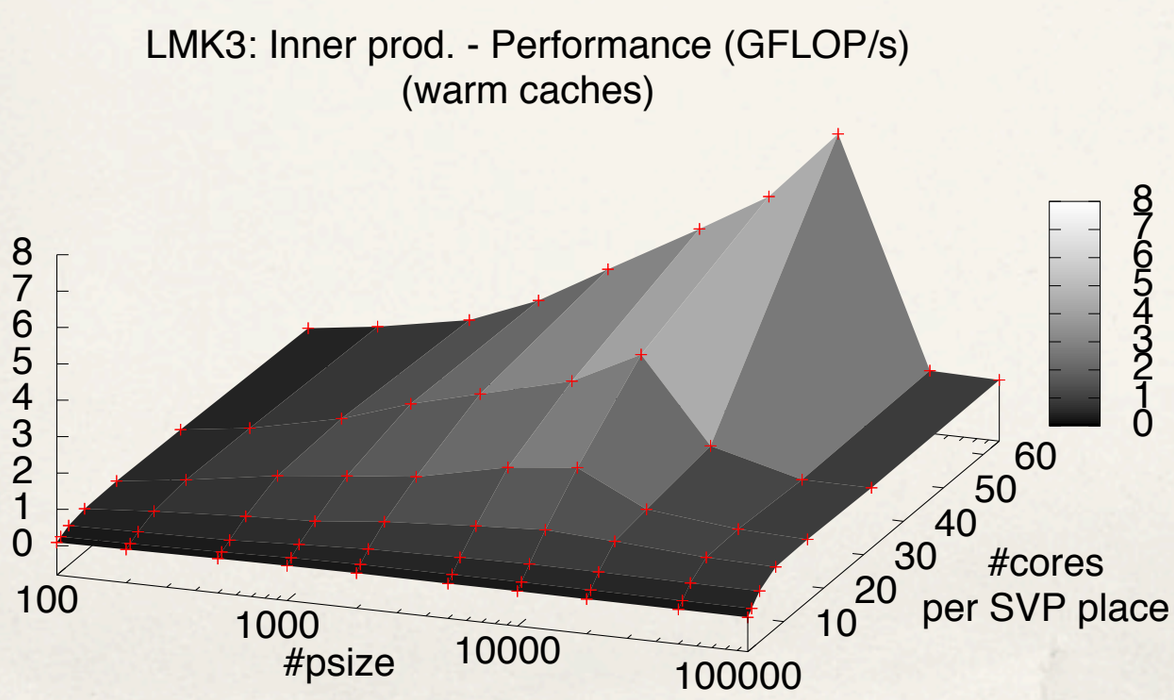
I come back to the example from earlier.

First you can see that on the dependent computations the common prefix is independent and therefore will benefit from latency hiding.

Then we note that execution is dominated by the inner computation. This is compiled to 7 instructions, including 2 FP ops and 2 memory operations. This gives AI1 and AI2 as follows. However the 2nd FP operation is dependent, which lowers AI1' as follows.

Example code: Inner product





- When P=1 we observe 0,12-0,15 GFLOP/s, as predicted by AI1
- When data fits in caches, performance scales linearly — with enough threads
- When N large, cache evictions can reduce effective on-chip bandwidth down to 4GB/s, i.e 0,5 GFLOP/s (AI2 = 1/8)
  We observe up to 0,85 GFLOP/s
- When N small, the outer reduction dominates + not enough threads to tolerate latency

donderdag 2 september 2010

Here you see the actual measured performance of this code. On the X axis you have N, the problem size in iterations. On the Z axis you have the number of cores; we have run the kernel in different grid configurations. The Y axis shows the performance in GFLOP/s.

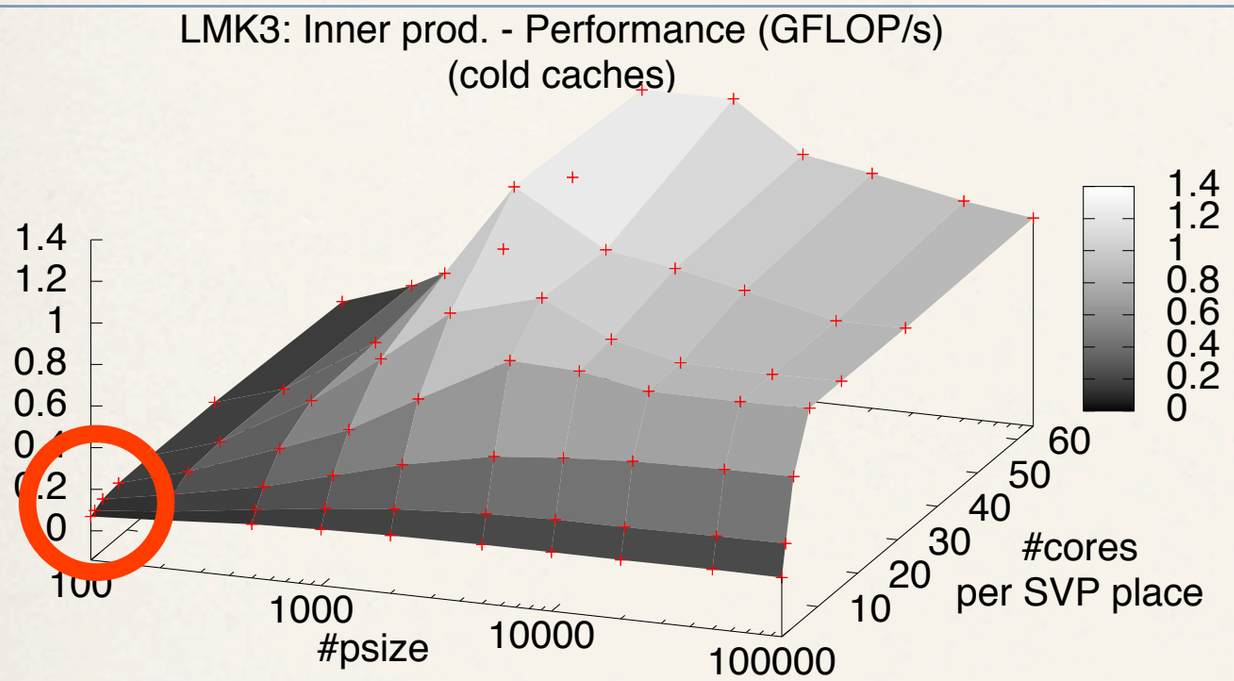
At P=1 we observe between 0,12 and 0,15 GFLOP/s, as predicted by AI1. From this point he program scales linearly as long as the data fits in caches and there are enough threads to tolerate latency.

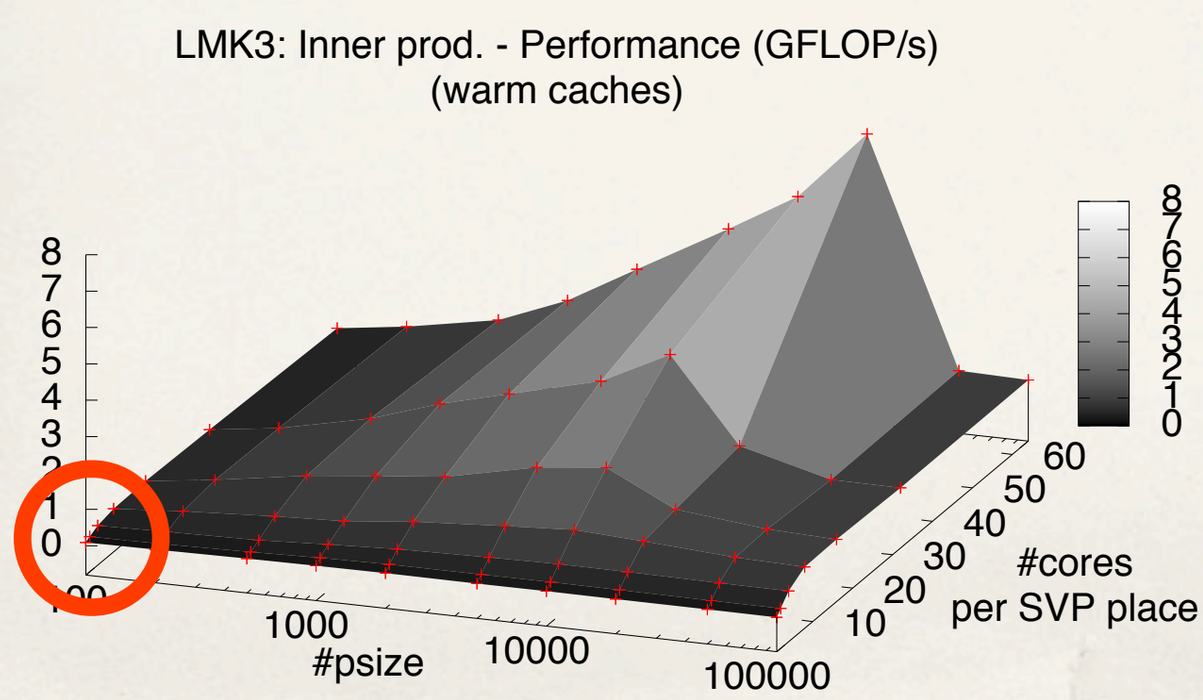
We predict further the behavior as follows. When N becomes larger, cache evictions are mixed with loads. In the worst case each load could require as much as two cache lines transfers, i.e. the effective ring bandwidth can be reduced to 4GB/s. With AI2 = 1/8 this gives an envelope of 0,5 GFLOP/s. We observe up to 0,85 GFLOP/s with this code.

When N is small, there are not enough threads to tolerate latency or the outer reduction dominates. This lowers the envelope and prevents full linear scaling.

With this model, we can optimize mapping by choosing to not execute with places larger than 8 cores for large problem sizes.

Example code: Inner product





- When P=1 we observe 0,12-0,15 GFLOP/s, as predicted by AI1
- When data fits in caches, performance scales linearly — with enough threads
- When N large, cache evictions can reduce effective on-chip bandwidth down to 4GB/s, i.e 0,5 GFLOP/s (AI2 = 1/8)
  We observe up to 0,85 GFLOP/s
- When N small, the outer reduction dominates + not enough threads to tolerate latency

donderdag 2 september 2010

Here you see the actual measured performance of this code. On the X axis you have N, the problem size in iterations. On the Z axis you have the number of cores; we have run the kernel in different grid configurations. The Y axis shows the performance in GFLOP/s.

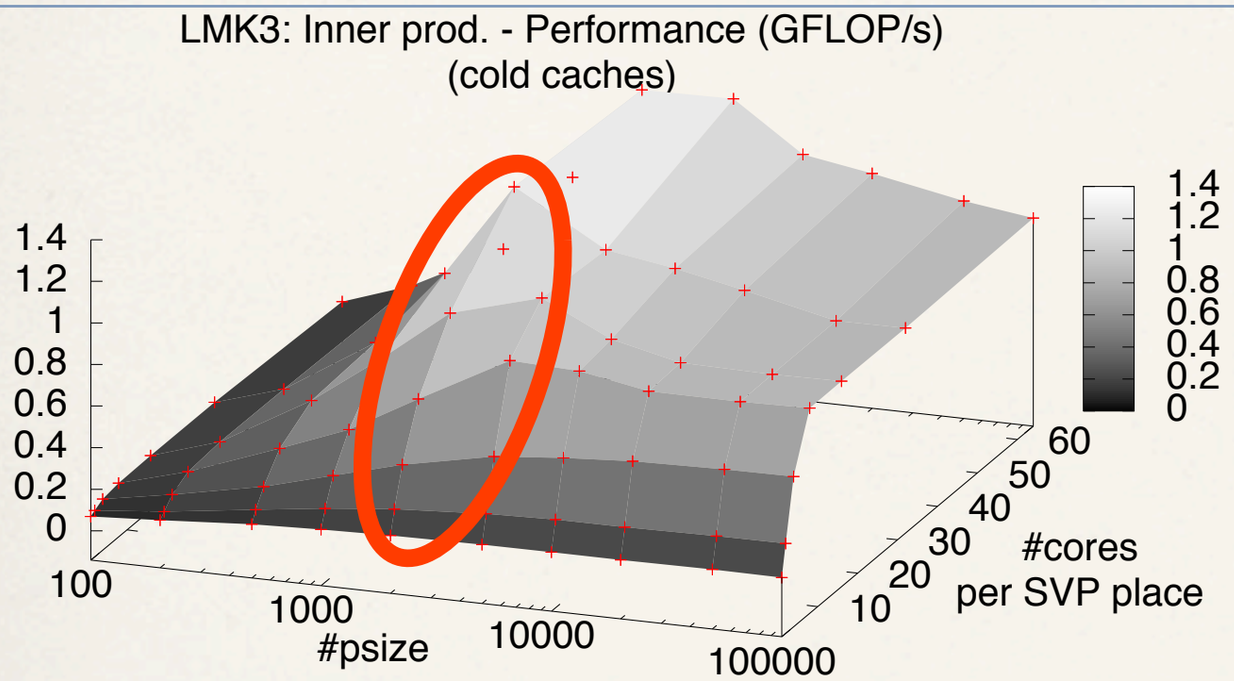
At P=1 we observe between 0,12 and 0,15 GFLOP/s, as predicted by AI1. From this point he program scales linearly as long as the data fits in caches and there are enough threads to tolerate latency.

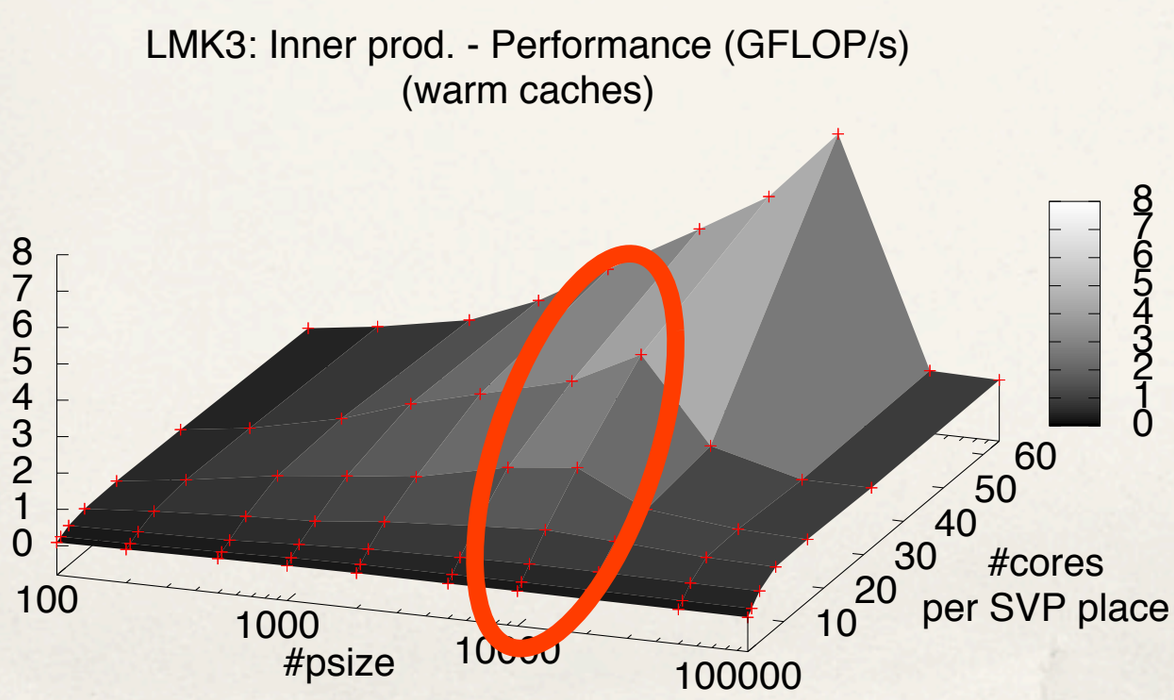
We predict further the behavior as follows. When N becomes larger, cache evictions are mixed with loads. In the worst case each load could require as much as two cache lines transfers, i.e. the effective ring bandwidth can be reduced to 4GB/s. With AI2 = 1/8 this gives an envelope of 0,5 GFLOP/s. We observe up to 0,85 GFLOP/s with this code.

When N is small, there are not enough threads to tolerate latency or the outer reduction dominates. This lowers the envelope and prevents full linear scaling.

With this model, we can optimize mapping by choosing to not execute with places larger than 8 cores for large problem sizes.

Example code: Inner product





- When P=1 we observe 0,12-0,15 GFLOP/s, as predicted by AI1
- When data fits in caches, performance scales linearly — with enough threads
- When N large, cache evictions can reduce effective on-chip bandwidth down to 4GB/s, i.e 0,5 GFLOP/s (AI2 = 1/8)
  We observe up to 0,85 GFLOP/s
- When N small, the outer reduction dominates + not enough threads to tolerate latency

donderdag 2 september 2010

Here you see the actual measured performance of this code. On the X axis you have N, the problem size in iterations. On the Z axis you have the number of cores; we have run the kernel in different grid configurations. The Y axis shows the performance in GFLOP/s.

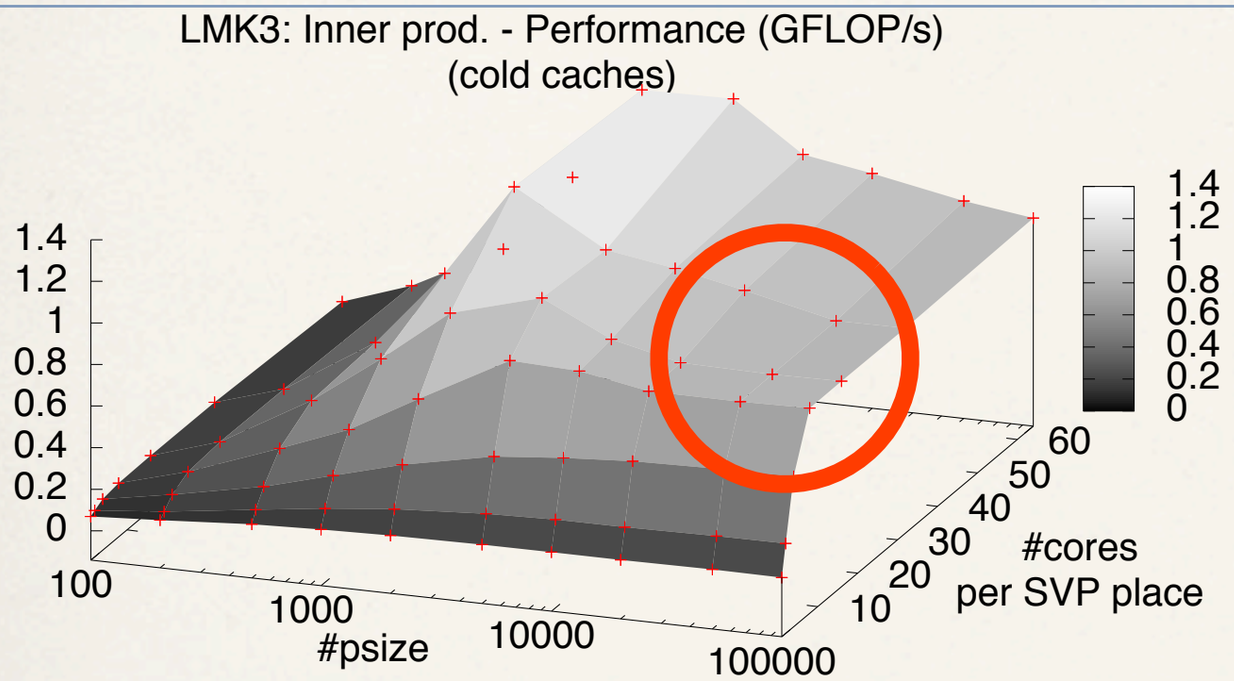
At P=1 we observe between 0,12 and 0,15 GFLOP/s, as predicted by AI1. From this point he program scales linearly as long as the data fits in caches and there are enough threads to tolerate latency.

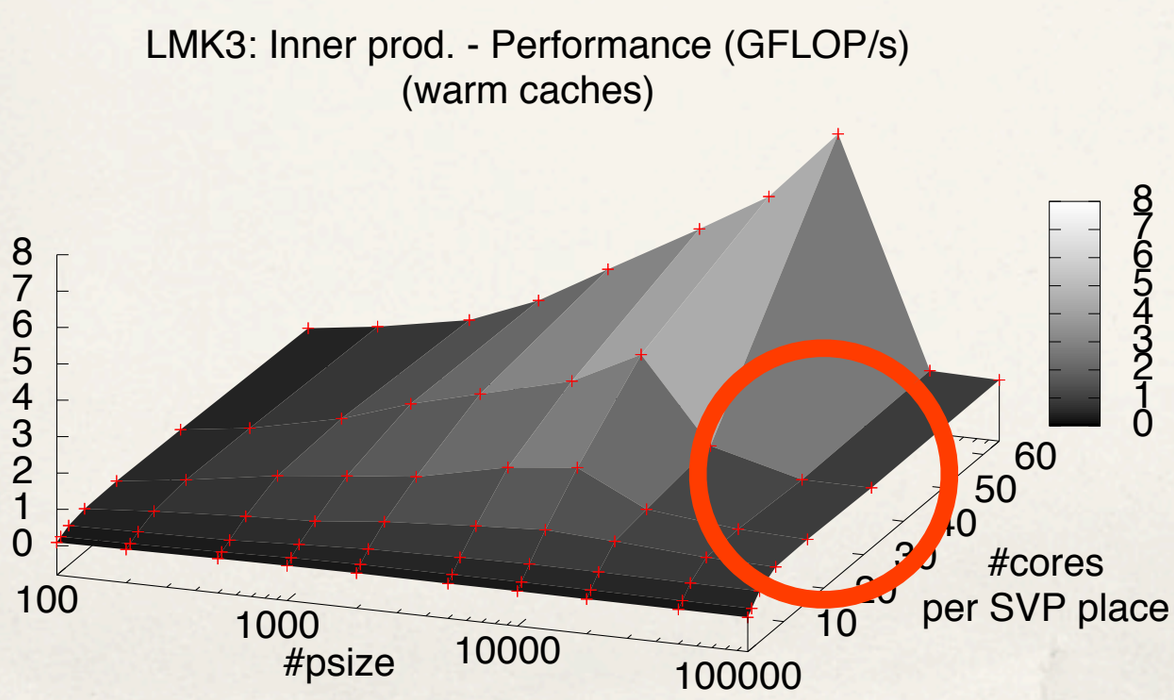
We predict further the behavior as follows. When N becomes larger, cache evictions are mixed with loads. In the worst case each load could require as much as two cache lines transfers, i.e. the effective ring bandwidth can be reduced to 4GB/s. With AI2 = 1/8 this gives an envelope of 0,5 GFLOP/s. We observe up to 0,85 GFLOP/s with this code.

When N is small, there are not enough threads to tolerate latency or the outer reduction dominates. This lowers the envelope and prevents full linear scaling.

With this model, we can optimize mapping by choosing to not execute with places larger than 8 cores for large problem sizes.

Example code: Inner product





- When P=1 we observe 0,12-0,15 GFLOP/s, as predicted by AI1
- When data fits in caches, performance scales linearly — with enough threads
- When N large, cache evictions can reduce effective on-chip bandwidth down to 4GB/s, i.e 0,5 GFLOP/s (AI2 = 1/8)
  We observe up to 0,85 GFLOP/s
- When N small, the outer reduction dominates + not enough threads to tolerate latency

donderdag 2 september 2010

Here you see the actual measured performance of this code. On the X axis you have N, the problem size in iterations. On the Z axis you have the number of cores; we have run the kernel in different grid configurations. The Y axis shows the performance in GFLOP/s.

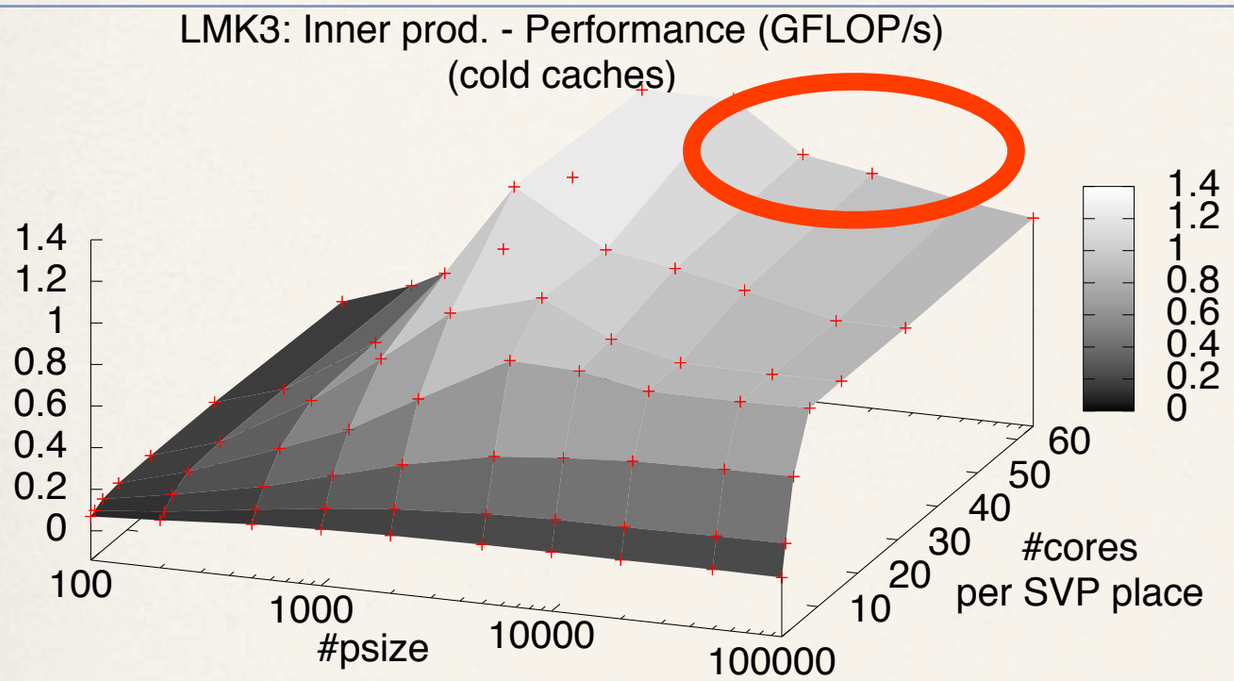
At P=1 we observe between 0,12 and 0,15 GFLOP/s, as predicted by AI1. From this point he program scales linearly as long as the data fits in caches and there are enough threads to tolerate latency.

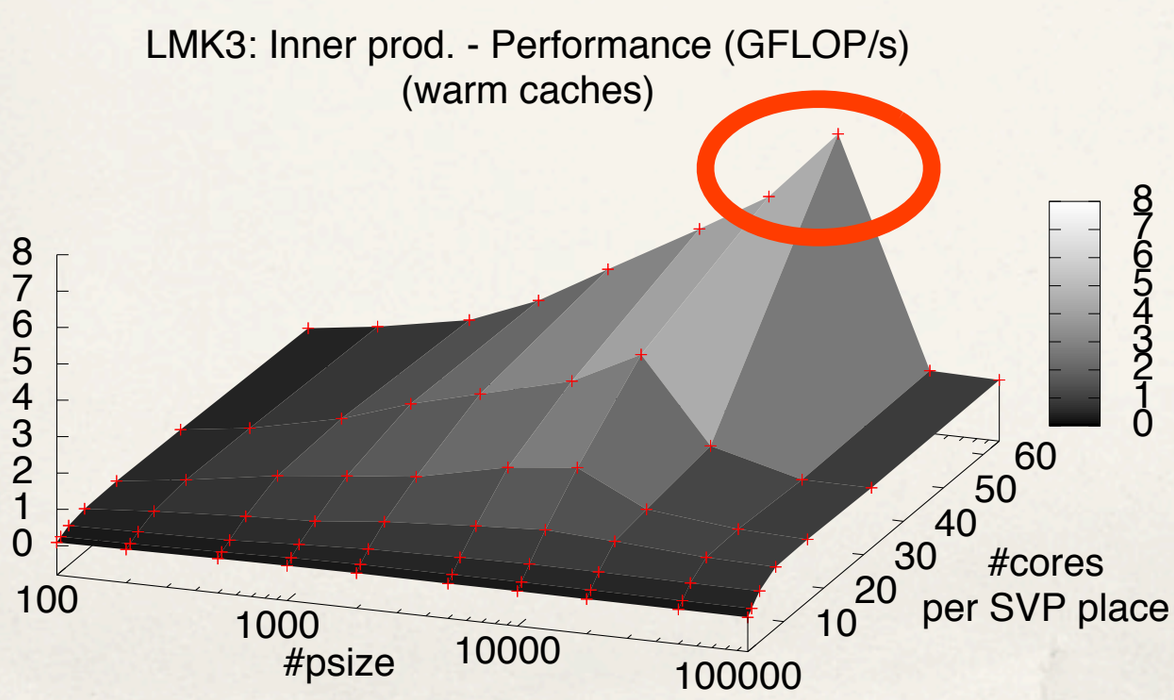
We predict further the behavior as follows. When N becomes larger, cache evictions are mixed with loads. In the worst case each load could require as much as two cache lines transfers, i.e. the effective ring bandwidth can be reduced to 4GB/s. With AI2 = 1/8 this gives an envelope of 0,5 GFLOP/s. We observe up to 0,85 GFLOP/s with this code.

When N is small, there are not enough threads to tolerate latency or the outer reduction dominates. This lowers the envelope and prevents full linear scaling.

With this model, we can optimize mapping by choosing to not execute with places larger than 8 cores for large problem sizes.

Example code: Inner product





- When P=1 we observe 0,12-0,15 GFLOP/s, as predicted by AI1
- When data fits in caches, performance scales linearly — with enough threads
- When N large, cache evictions can reduce effective on-chip bandwidth down to 4GB/s, i.e 0,5 GFLOP/s (AI2 = 1/8)
  We observe up to 0,85 GFLOP/s
- When N small, the outer reduction dominates + not enough threads to tolerate latency

donderdag 2 september 2010

Here you see the actual measured performance of this code. On the X axis you have N, the problem size in iterations. On the Z axis you have the number of cores; we have run the kernel in different grid configurations. The Y axis shows the performance in GFLOP/s.

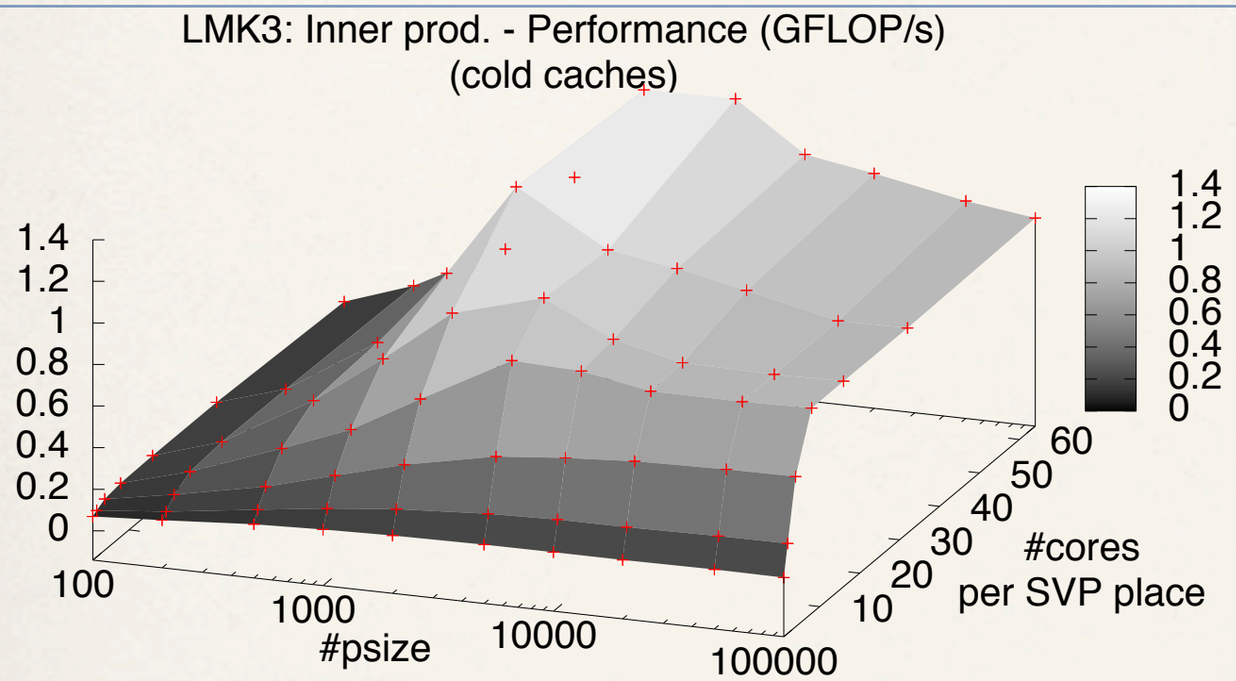
At P=1 we observe between 0,12 and 0,15 GFLOP/s, as predicted by AI1. From this point he program scales linearly as long as the data fits in caches and there are enough threads to tolerate latency.

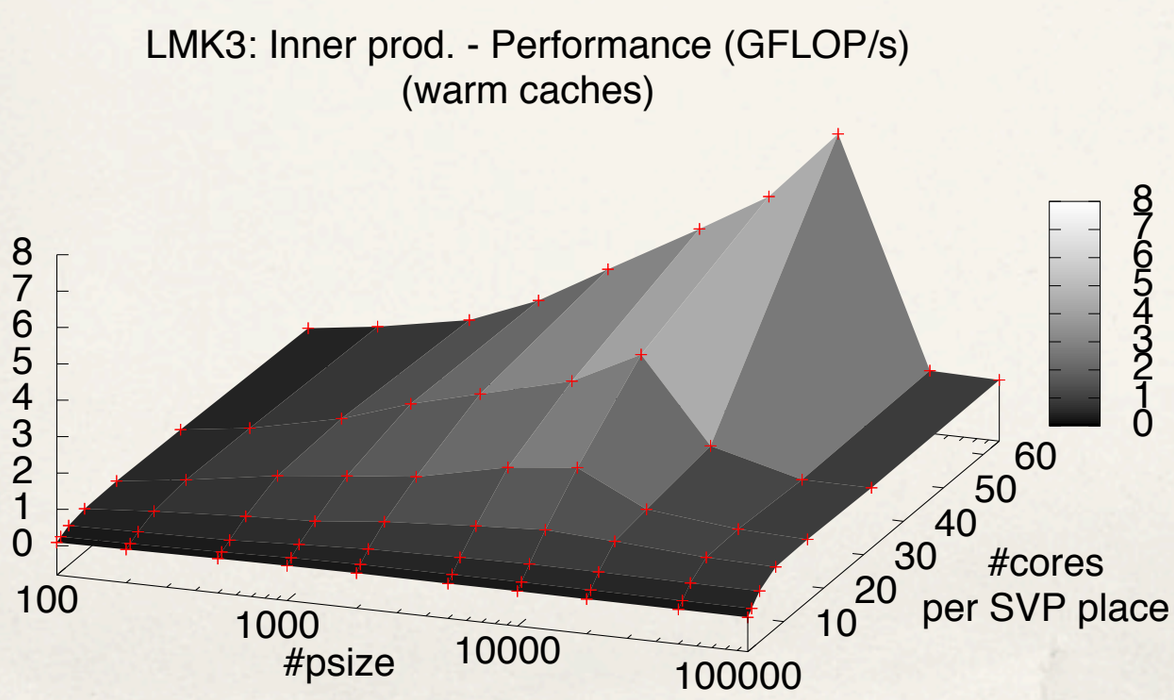
We predict further the behavior as follows. When N becomes larger, cache evictions are mixed with loads. In the worst case each load could require as much as two cache lines transfers, i.e. the effective ring bandwidth can be reduced to 4GB/s. With AI2 = 1/8 this gives an envelope of 0,5 GFLOP/s. We observe up to 0,85 GFLOP/s with this code.

When N is small, there are not enough threads to tolerate latency or the outer reduction dominates. This lowers the envelope and prevents full linear scaling.

With this model, we can optimize mapping by choosing to not execute with places larger than 8 cores for large problem sizes.

Example code: Inner product





- When P=1 we observe 0,12-0,15 GFLOP/s, as predicted by AI1
- When data fits in caches, performance scales linearly — with enough threads
- When N large, cache evictions can reduce effective on-chip bandwidth down to 4GB/s, i.e 0,5 GFLOP/s (AI2 = 1/8)
  We observe up to 0,85 GFLOP/s
- When N small, the outer reduction dominates + not enough threads to tolerate latency

donderdag 2 september 2010

Here you see the actual measured performance of this code. On the X axis you have N, the problem size in iterations. On the Z axis you have the number of cores; we have run the kernel in different grid configurations. The Y axis shows the performance in GFLOP/s.

At P=1 we observe between 0,12 and 0,15 GFLOP/s, as predicted by AI1. From this point he program scales linearly as long as the data fits in caches and there are enough threads to tolerate latency.

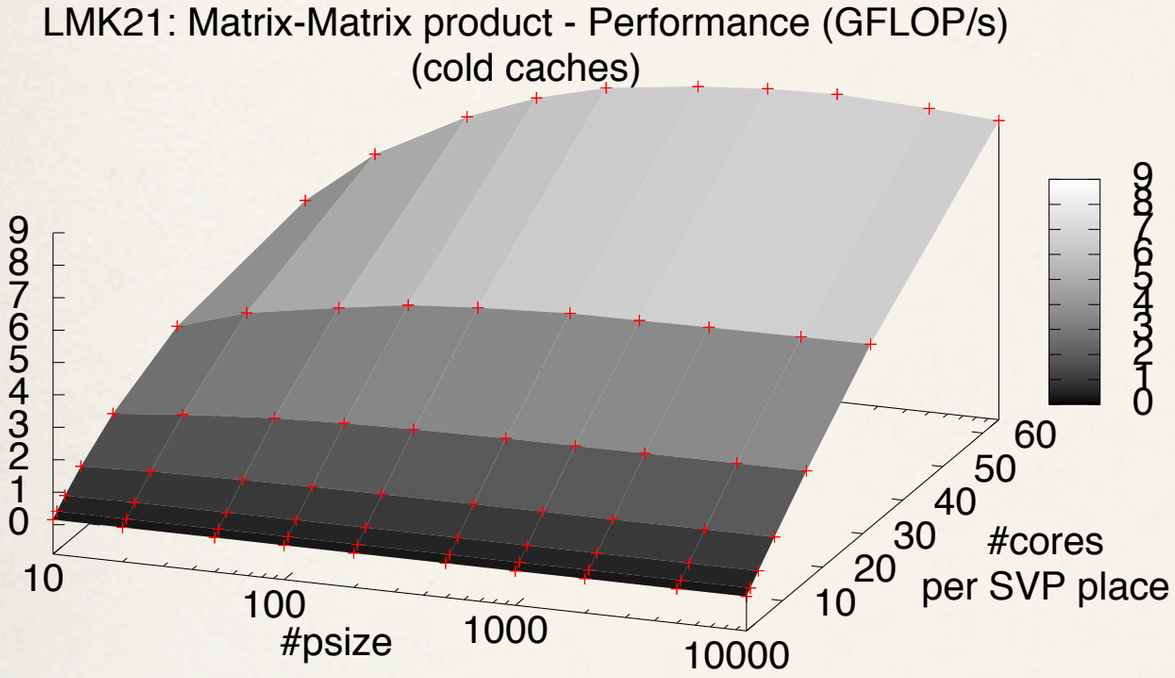
We predict further the behavior as follows. When N becomes larger, cache evictions are mixed with loads. In the worst case each load could require as much as two cache lines transfers, i.e. the effective ring bandwidth can be reduced to 4GB/s. With AI2 = 1/8 this gives an envelope of 0,5 GFLOP/s. We observe up to 0,85 GFLOP/s with this code.

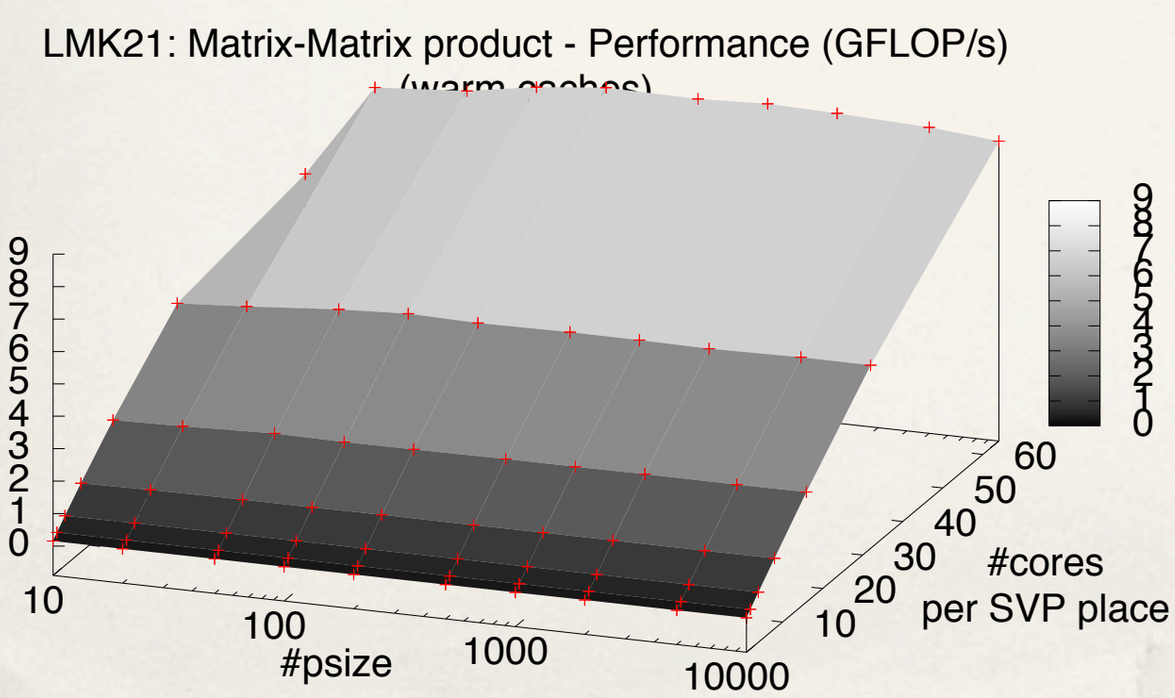
When N is small, there are not enough threads to tolerate latency or the outer reduction dominates. This lowers the envelope and prevents full linear scaling.

With this model, we can optimize mapping by choosing to not execute with places larger than 8 cores for large problem sizes.

Matrix-matrix multiply*

* simplified: 25xN





- AI2 = 3,1 thus max envelope 75 GFLOP/s
- However MM based on IP thus AI1 ≤ 0,16 i.e. envelope bound to 9,8 GFLOP/s (program is FPU-bound)
  We observe up to 8,7 GFLOP/s.
- Small rows in this code limit eviction-related bandwidth reduction at large N

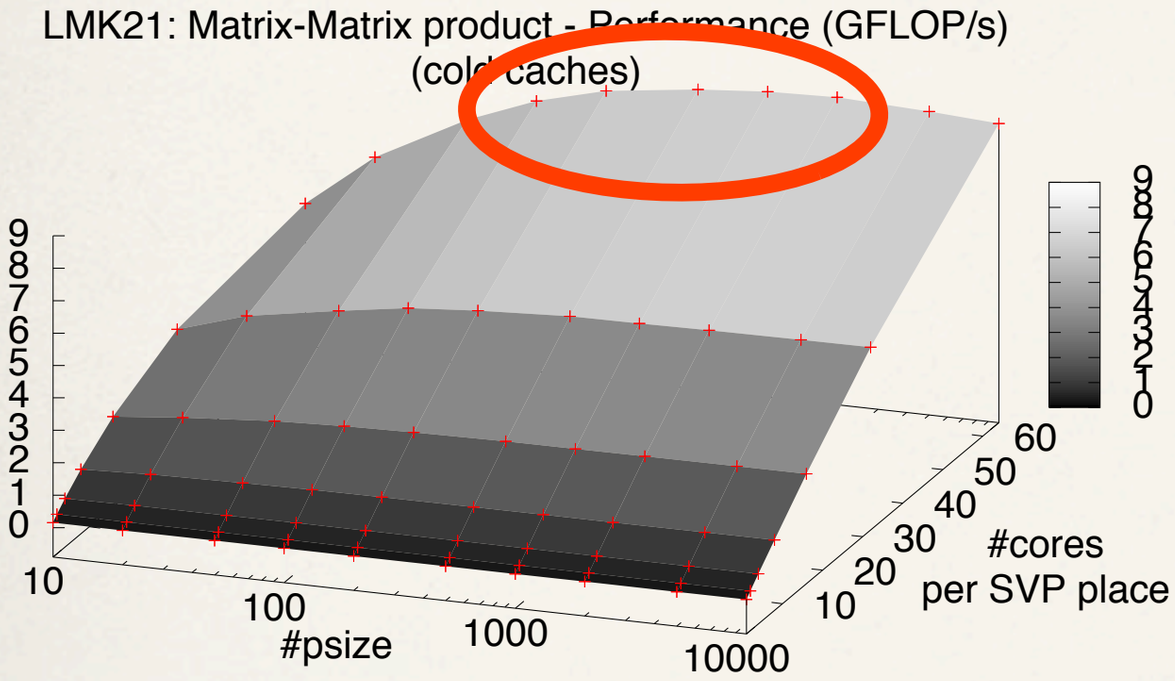
donderdag 2 september 2010

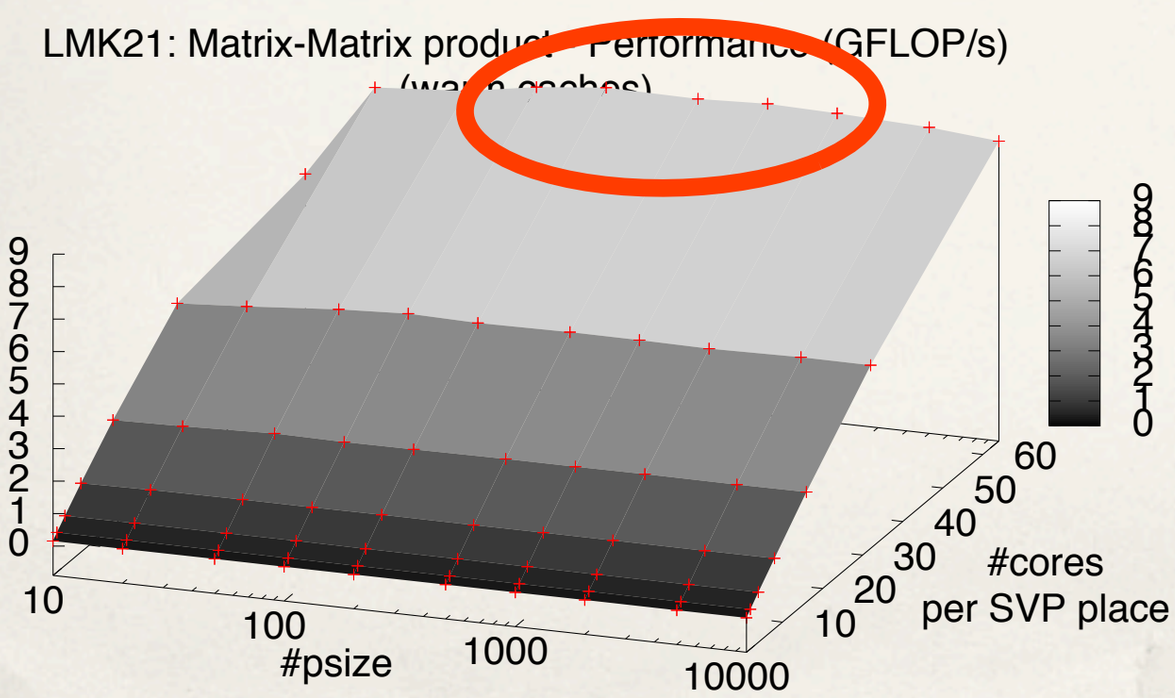
Here I take another example, matrix-matrix multiply. In this code AI2 = 3,1, so the code is essentially compute-bound. It is based on IP from above, thus AI1 bounds the envelope to a bit less than 10GFLOP/s. We observe nearly 9 GFLOP/s. Because the rows are small, we achieve good locality and reduce the number of evictions.

Here placement can select a number of cores depending on the requested throughput, up to the maximum place size.

Matrix-matrix multiply*

* simplified: 25xN





- AI2 = 3,1 thus max envelope 75 GFLOP/s
- However MM based on IP thus AI1 ≤ 0,16 i.e. envelope bound to 9,8 GFLOP/s (program is FPU-bound)
  We observe up to 8,7 GFLOP/s.
- Small rows in this code limit eviction-related bandwidth reduction at large N

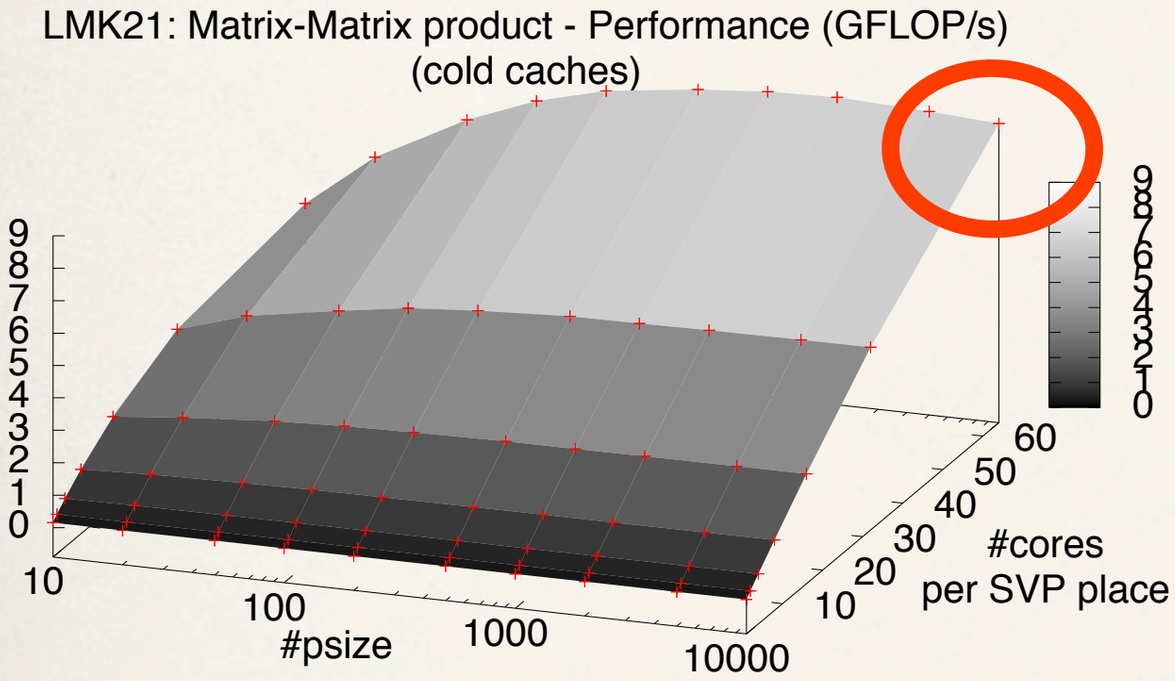
donderdag 2 september 2010

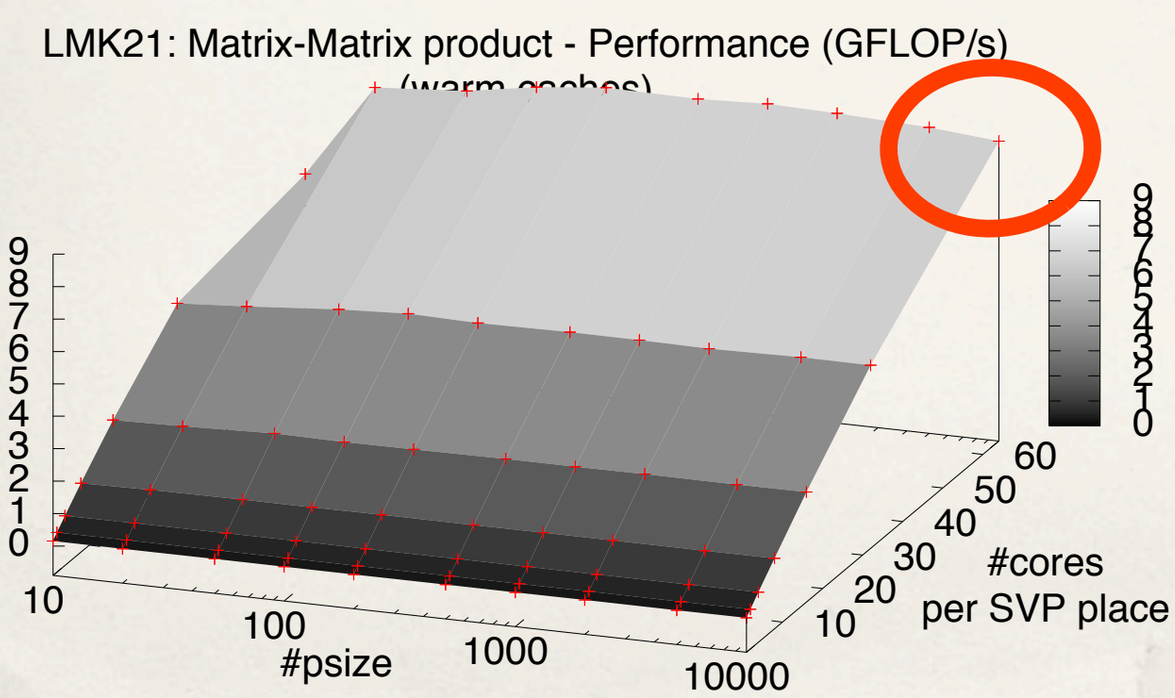
Here I take another example, matrix-matrix multiply. In this code AI2 = 3,1, so the code is essentially compute-bound. It is based on IP from above, thus AI1 bounds the envelope to a bit less than 10GFLOP/s. We observe nearly 9 GFLOP/s. Because the rows are small, we achieve good locality and reduce the number of evictions.

Here placement can select a number of cores depending on the requested throughput, up to the maximum place size.

Matrix-matrix multiply*

* simplified: 25xN





- AI2 = 3,1 thus max envelope 75 GFLOP/s
- However MM based on IP thus AI1 ≤ 0,16 i.e. envelope bound to 9,8 GFLOP/s (program is FPU-bound)
  We observe up to 8,7 GFLOP/s.
- Small rows in this code limit eviction-related bandwidth reduction at large N

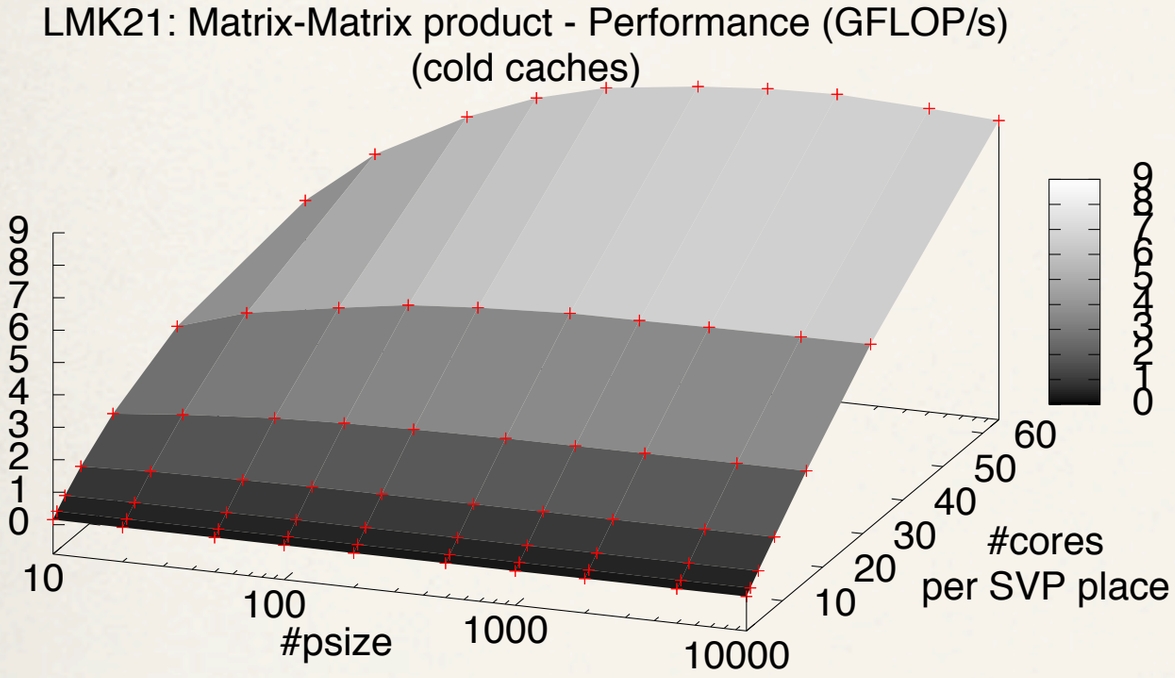
donderdag 2 september 2010

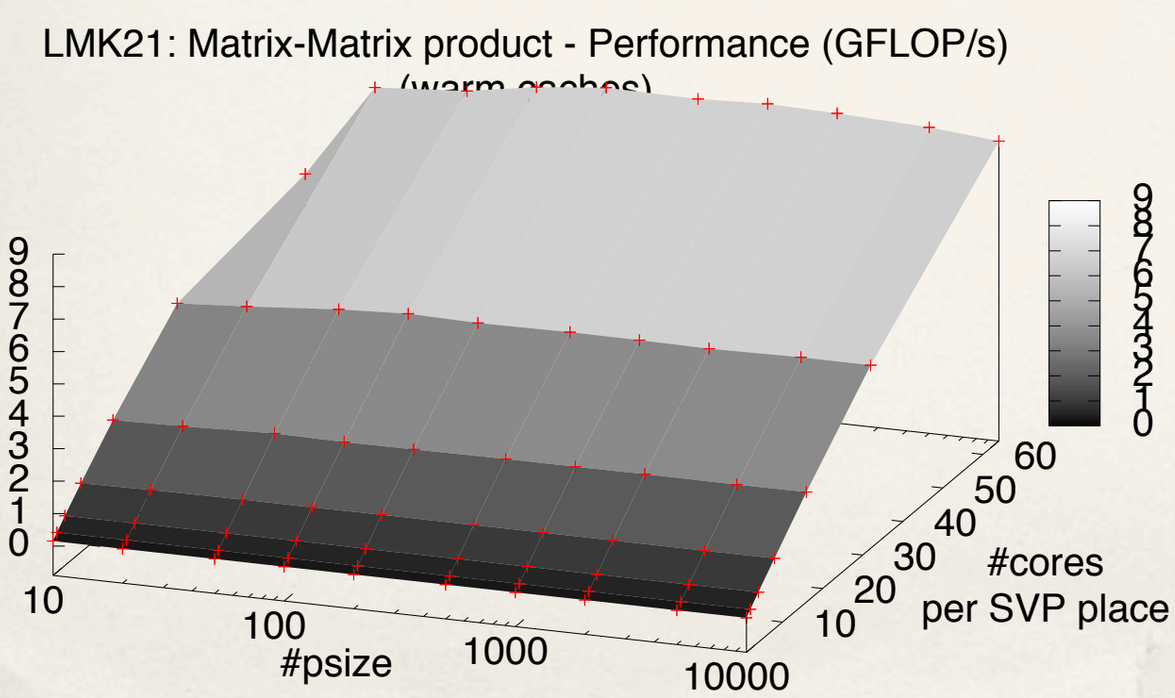
Here I take another example, matrix-matrix multiply. In this code AI2 = 3,1, so the code is essentially compute-bound. It is based on IP from above, thus AI1 bounds the envelope to a bit less than 10GFLOP/s. We observe nearly 9 GFLOP/s. Because the rows are small, we achieve good locality and reduce the number of evictions.

Here placement can select a number of cores depending on the requested throughput, up to the maximum place size.

Matrix-matrix multiply*

* simplified: 25xN





- AI2 = 3,1 thus max envelope 75 GFLOP/s
- However MM based on IP thus AI1 ≤ 0,16 i.e. envelope bound to 9,8 GFLOP/s (program is FPU-bound)
  We observe up to 8,7 GFLOP/s.
- Small rows in this code limit eviction-related bandwidth reduction at large N

donderdag 2 september 2010

Here I take another example, matrix-matrix multiply. In this code AI2 = 3,1, so the code is essentially compute-bound. It is based on IP from above, thus AI1 bounds the envelope to a bit less than 10GFLOP/s. We observe nearly 9 GFLOP/s. Because the rows are small, we achieve good locality and reduce the number of evictions.

Here placement can select a number of cores depending on the requested throughput, up to the maximum place size.

Other examples — Actual performance vs. envelope

Program	AI1	AI2	Restricted by	Max envelope	Observed
DNRM2	0,14-0,22	0,375	AI1	0,14-0,22	0,12-0,22
ESF	0,48	0,5	AI1	P x 0,48	P x 0,43
ESF (cache bound)	0,48	0,5	AI2	2-6,15 (IO=4-12,3G/s)	2,7
FFT	0,33	0,21	AI1	P x 0,33	P x 0,23
FFT (cache bound)	0,33	0,21	AI2	0,84-2,6 (IO=4-12,3G/s)	2,24

donderdag 2 september 2010

This table summarize results across a few other examples.

The first program is a pure sequential implementation of the BLAS function DNRM2. This code is dominated by a sequential reduction so it does not scale beyond 2 cores. However we observe consistent performance according to the AI1 estimator across all problem sizes.

The second program is the equation of state fragment from the Livermore loops benchmark. It has a high ratio of FP operations to memory operations. When the problem is compute-bound we observe up to 90% of the performance expected from AI1. When it is memory bound it fits within the performance window expected from AI2.

The 3rd program is the 1D FFT. This program has a communication complexity logarithmic with the problem size, and thus becomes memory bound more quickly. When it is compute-bound we observe up to 70% of the AI1 performance. When it becomes memory bound it also fits within the performance window expected from AI2.

Again, based on this synthetic model, resources can be efficiently allocated to tailor throughput requirements.

Conclusions

- Fine grained *hardware multithreading* on the Microgrid: asynchronous long latency instructions can *overlap*, maximizing *pipeline efficiency*
- *Naive* program implementations in SVP *expose* concurrency to the hardware, where it is mapped and scheduled automatically
- Code is *compiled once*, hardware adapts execution to granularity and layout at run time
- We can *predict performance* accurately based on program code and architecture parameters — allowing to scale resource usage to demand

donderdag 2 september 2010

Here is a summary of our approach in a few words.

First we use fine grained hardware multithreading; we let long latency instructions overlap to maximize pipeline efficiency and let all instructions appear as if they take only one cycle to execute

Then we write program code using naive expressions of concurrency, and let the hardware exploit this information efficiently and automatically. This code is compiled once, and the on-chip hardware adapts the execution to the granularity and layout at run time.

Within this framework, we can predict the performance ahead of time and scale resources based on requirements.

Thank you.

Example code: Inner product

```
kernel3:
  allocate 8, $l0      # (8: PLACE_DEFAULT)
  getcores $l1         # get P
  setlimit $l0, $l1
  cred $l0, rk3
  divqu $g0, $l1, $l1 # l1 = N / P
  putg $g1, $l0, 0     # send X
  putg $g2, $l0, 1     # send Z
  putg $l1, $l0, 2     # send span
  fputs $df0, $l0, 0   # send Q
  sync $l0             # wait for comp.
  fgets $l0, 0, $lf0   # read Qr back
  fmov $lf0, $sf0      # return
  end
```

```
ik3:
  s8addq $l0,0,$l0 # l0 = i * 8
  addq $g1,$l0,$l1 # l1 = Z+i
  addq $g0,$l0,$l0 # l0 = X+i
  ldt $lf1,0($l0)  # lf1 = X[i]
  ldt $lf0,0($l1)  # lf0 = Z[i]
  mult $lf1,$lf0,$lf0 # lf0 = X[i]*Z[i]
  addt $lf0,$df0,$sf0 # Q += X[i]*Z[i]
  end
rk3:
  mulq $g2,$l0,$l0 # l0 = ri * span
  allocate 12, $l1 # (12: PLACE_LOCAL)
  addq $l0,$g2,$l2 # l2 = (ri+1) * span
  setstart $l1, $l0
  setlimit $l1, $l2
  cred $l1, ik3
  fclr $lf0
  fputs $lf0, $l1, 0 # send Qr
  putg $g0, $l1, 0   # send X
  putg $g1, $l1, 1   # send Z
  sync $l1           # wait for comp.
  fgets $l1, 0, $lf0    # read Qr back
  addt $df0, $lf0, $sf0 # Q += Qr
  end
```

Example code: Inner product

```
kernel3:
  allocate 8, $l0       # (8: PLACE_DEFAULT)
  getcores $l1          # get P
  setlimit $l0, $l1
  cred $l0, rk3
  divqu $g0, $l1, $l1 # l1 = N / P
  putg $g1, $l0, 0      # send X
  putg $g2, $l0, 1      # send Z
  putg $l1, $l0, 2      # send span
  fputs $df0, $l0, 0    # send Q
  sync $l0              # wait for comp.
  fgets $l0, 0, $lf0    # read Qr back
  fmov $lf0, $sf0       # return
  end
```

independent prefix

```
ik3:
  s8addq $l0,0,$l0 # l0 = i * 8
  addq $g1,$l0,$l1 # l1 = Z+i
  addq $g0,$l0,$l0 # l0 = X+i
  ldt $lf1,0($l0)  # lf1 = X[i]
  ldt $lf0,0($l1)  # lf0 = Z[i]
  mult $lf1,$lf0,$lf0 # lf0 = X[i]*Z[i]
  addt $lf0,$df0,$sf0 # Q += X[i]*Z[i]
  end
rk3:
  mulq $g2,$l0,$l0 # l0 = ri * span
  allocate 12, $l1 # (12: PLACE_LOCAL)
  addq $l0,$g2,$l2 # l2 = (ri+1) * span
  setstart $l1, $l0
  setlimit $l1, $l2
  cred $l1, ik3
  fclr $lf0
  fputs $lf0, $l1, 0 # send Qr
  putg $g0, $l1, 0   # send X
  putg $g1, $l1, 1   # send Z
  sync $l1           # wait for comp.
  fgets $l1, 0, $lf0    # read Qr back
  addt $df0, $lf0, $sf0 # Q += Qr
  end
```