MicroGrid Architecture Reference

Mike Lankamp

5th June 2015

Contents

Notice

Chapter 1

Introduction

This document is a technical reference to the Microgrid architecture, developed by the Computer Systems Architecture group at the University of Amsterdam. This document is meant for everyone who will work with, work on, or is interested in the architecture.

The document begins with a high-level overview of the multicore processor and how it interfaces with the rest of the system. Subsequent chapters will focus in on the components on the processor such as the cores themselves, the memory network, the delegation and link networks and the I/O interface. Going further, the components and processes on each core are described that together give the microgrid its unique performance properties.

Chapter 2

Overview

The Microgrid is a manycore system-on-chip that has been designed to achieve performance through multi-threading, latency hiding and scalability of hardware and software. The core architecture is a result of more than a decade of research, originating in 1996 as a latency-tolerant processor architecture called DRISC [?]. This architecture has since been adapted and improved into a full system-on-chip with network, suited memory protocol and I/O interfaces.

The architecture is built on several concepts [?] which will be explained below.

2.1 Families of threads

Programs written for the Microgrid are concurrent compositions of families of threads where a family of threads is an ordered set of identical threads. Each family is a unit of work that can have certain properties which determine how and where it is run on the microgrid.

A family is created by a thread and can have arguments and produce results communicated from and to this thread. Family termination defines a synchronization points with respect to results, whereas the arguments can be calculated after the family has been created to obtain maximum concurrency. Within a family, threads can synchronize with each other on local items of data. These forms of synchronized communication allow a family to encapsulate both regularity and locality. Analogies to the family in the sequential programming model are loops and function calls, which both capture regularity and locality in that model. All threads within a family are allowed to create families of their own, thus allowing microgrid programs to capture concurrency at many levels by creating a hierarchy of families, a *concurrency tree*, as illustrated in figure **??**.

This concurrency tree will evolve dynamically in an application and can capture concurrency at all levels, e.g. at the task level and, due to the threads' blocking nature, even at the instruction level. With a family being an ordered set of identical threads, and each thread within a family knowing its own index in that order, both homogeneous and heterogeneous computation can be defined. The latter can be achieved by using the index value to control the statically defined actions of a thread (e.g. by branching to different control paths on the index). The index values are defined by a sequence specified by a start index, a constant difference between successive index values and an optional maximum index value.

Each thread is a sequence of operations defined on a collection of registers, which form the thread's context

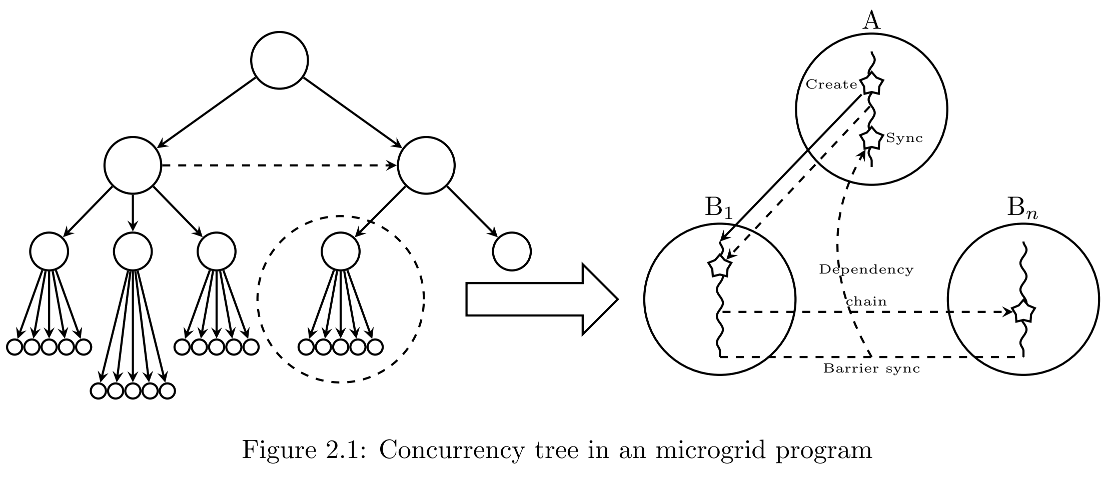

Figure 2.1: Concurrency tree in an microgrid program

in a synchronizing memory. Dependencies between threads in the same family are defined on this context. The index value of a thread is also part of this context and is set for each thread automatically.

2.1.1 Index sequence

When a program creates a family, it specifies the start, limit and step of the index sequence of the family as signed integers. These indices work exactly as in a for-loop in C; a positive step defines one thread per iteration from *start* up to *limit* and a negative step defines one thread per iteration from *start* down to *limit*. In both cases, the limit is excluded and a limit that lies below or above the start index, respectively, causes no threads to be created. Following are some examples of traditional C loops and their start, limit, step equivalents.

C loop	Start	Limit	Step
`for (i = 0; i < 10; i++)`	0	10	1
`for (i = 14; i > 3; i-=2)`	14	3	-2

2.1.2 Block size

To avoid rampant resource usage when creating families with a large number of threads, a *block* size can be specified during family creation. This value specifies the upper limit for the number of threads created on a single execution core. By properly choosing a value during family creation, a program can allocate few resources to a family, leaving more resources available for other, perhaps more important families. Without specifying a block size, every execution core will attempt to fill up all its resources with threads from the family.

2.1.3 Family synchronization

A thread can synchronize on a family of threads, meaning the thread suspends, if necessary, until the family has terminated. This means that all threads of the target family have terminated and any side-effects of those threads are made consistent across the microgrid. Threads in a family terminate by executing a special kind of thread termination instruction (see section **??**). When a family has terminated, all of its dynamic synchronizing state is lost and all of the *shared memory state* (see section **??**) that it has modified becomes defined.

2.1.4 Breaking a family

Any thread may *break* its own family (and that family only). When a break on a family has been issued by one of its threads, the creation of new threads ceases and all currently active threads are allows to finish. This mechanism allows families to be halted before its index sequence has been iterated over and is useful for implementing loops with dynamic exit conditions as a family of threads.

2.2 Synchronizing Registers

Every thread has a context of registers which are special in that they are *synchronizing registers*. These registers, besides providing the temporary storage for threads to store their intermediate results, proviede the mechanism by which threads in a family can synchronize with each other, their creating thread and themselves. Each thread is allocated a context of synchronizing registers when it is created; when the thread terminates, its context is discarded. Each synchronizing register provides dataflow-like synchronization with blocking reads and non-blocking writes. Dependencies between operations in the same or different threads are enforced by this synchronizing memory; an operation cannot proceed unless its operands are defined, i.e. have been written as a result of executing other operations, including loads from shared memory.

Synchronization between threads in the same families is effected through these synchronizing registers by overlapping contexts for communicating threads, causing the sharing of synchronizing registers, creating a communication channel. However, a constraint is imposed on this mechanism, which arises from the combined requirements of locality, speed of operation and deadlock freedom. A thread may only have a dependency on values produced by one other thread: its predecessor in the family's index sequence. This constraint ensures that dependencies between operations in a family of threads can be represented as an acyclic graph, which in turn ensures freedom from communication deadlock in the model. These acyclic dependency graphs are initialized by the creating thread, and values generated by the last thread are available to the creating thread after the family's termination.

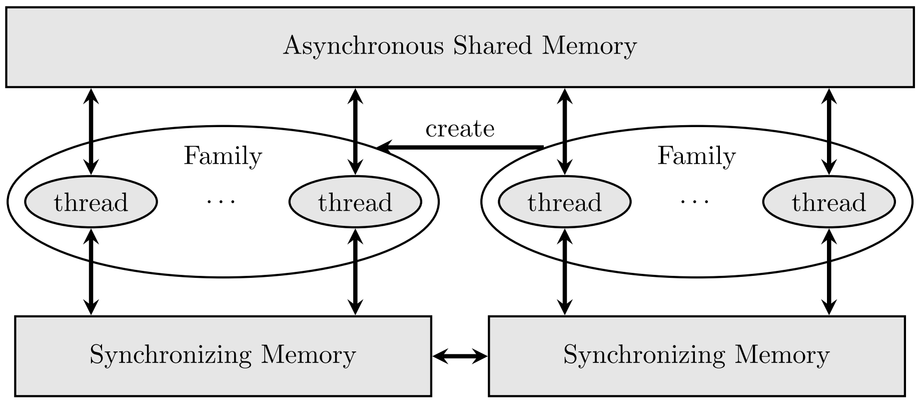

Figure 2.2: Shared memory from the point of view of an microgrid program

2.3 Shared memory

To store or pass large amounts of data, the microgrid offers a persistent shared memory which can be read and written by all threads (see figure **??**). This memory, however, is weakly consistent and is only defined using bulk synchronization on family creation and termination. At any other time, because the microgrid is a potentially asynchronous concurrent system, there can always be locations in the shared memory whose state cannot be unambigiously determined. Values written by threads in a family are guaranteed to be well defined only on termination of that family. Thus, arguments passed via shared memory must be defined before the family is created and the family's results are only defined when the family signals termination.

2.4 Places

A place on the microgrid is a collection of one or more cores where families can be executed. When a thread creates a family, it can specify the place where to create that family. Creating families on different places has several uses: first, it allows programs to avoid resource deadlock by creating a unit of work on a different set of resources. Second, since a place also holds one or more mutually exclusive states, families can be created on a place in a mutually exclusive manner—this guarantees that of all families created on the same place, only one will run at any time, allowing it to operate on shared data. The act of creating a family on a different places is called "delegation".

2.5 Network

Figure **??** shows an example layout of a microgrid. A microgrid consists of a set of cores, each supporting an extended RISC ISA to implement hardware support for creating and managing families. The cores in a microgrid communicate with each other via two networks. A global point-to-point packet routed network (the "delegation network") and a single static network that links all together in a single chain (the "link network"). Several cores are connected to an L2 cache via a bus, and the L2 caches are connected via a hierarchical ring network to one or more external memory channels. I/O occurs via dedicated cores that have an I/O controller attached to them that communicates with the I/O channel.

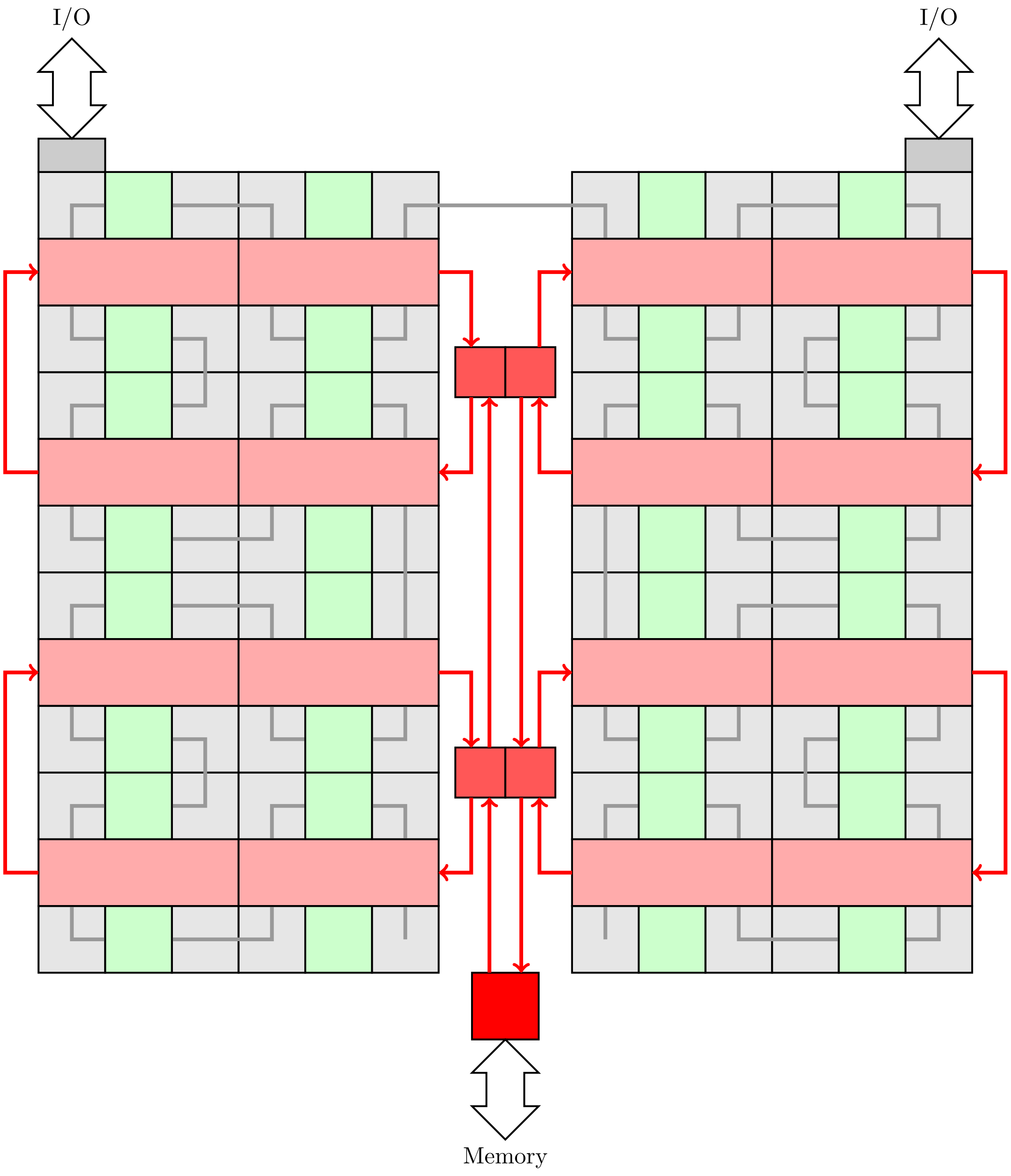

Figure 2.3: Overview of an example microgrid consisting of 64 cores, 32 FPUs, 16 L2 caches, 2 external I/O busses and 1 external memory bus. Not shown is the mesh network that connects all cores together. The cores are also connected via an optimized link network that connects them via a Moore curve, optimized for neighbour-to-neighbour communication. Components are not to scale.

Chapter 3

Dataflow Scheduling

A familiar concept from traditional architecture is that of instruction latencies. This is the time it takes an instruction to produce its result. While an integer addition might be able to complete without stalls in the pipeline, a multiply or division most likely will not and will take anywhere from 2 to a few dozen cycles. These latencies, however, are always the same and can be known by the compiler. As a result, the compiler can schedule the instructions as best as possible such that, on an out-of-order pipeline, the pipeline stalls for the least amount of time.

These latencies, however, are still low compared to latencies of a memory read that misses the cache. A read that misses all on-chip caches and needs to read from external memory can take several hundred, or even a thousand cycles. Because the actual latency is so large, varies hugely and depends on run-time conditions, the compiler can often not schedule enough instruction to tolerate the latency. To refer to the these two distinct classes of instructions, we denote the former class of instructions *short-latency* instructions and the latter *long-latency* instructions. Note that long-latency instructions can still complete within a cycle (e.g., a load that hits the L1 cache).

A microgrid core has been designed to use multithreading to hide the latency of long-latency instructions in a single thread. It accomplishes this by flushing the pipeline and switching to another thread when the pipeline detects that an operand that is required by the current instruction is not available.

3.1 Register states

Since operands are registers, all registers on a microgrid core are extended with state bits that identify whether the register is *full* or *empty*. Normal operations such as integer additions write the register and set the state bits to *full*. Long-latency operations such memory loads set the state bits to *empty* when the operation is issued. When the operation completes, the register is written and its state bits set to *full*. The state transition diagram is shown in figure **??**.

3.2 Thread control stream

As described above, if the pipeline executes an instruction that does not have its operands available, the pipeline (before the offending instruction) is flushed of the thread's instructions and execution is switched to another thread. This flush creates several bubbles in the pipeline for every missed data dependency. The number of bubbles depends on the number of stages between the start of the pipeline and the place where the registers have been read and the decision is made to flush or not based on their state. In a traditional 6-stage pipeline, this can be around 3 cycles. To avoid this cost, a secondary instruction stream is read by the pipeline. This secondary stream consists of two bits per instruction, indicating one of several thread control operations to perform after reading it:

- Do nothing; continue executing from this thread as normal.
- Switch thread; if another thread is available for execution, switch to it after fetching this instruction.
- End thread; after fetching this instruction, end the thread. Note that this implies a switch.

Having the first two options available, the compiler can 'tag' instructions with the *switch* value in the secondary instruction stream (also called the *control stream*). With this, the compiler tells the hardware that this instruction uses, or can use, the result of a long-latency operation. The pipeline, upon seeing the *switch* bit set after fetching the instruction, will immediately switch to another thread without waiting to see if the

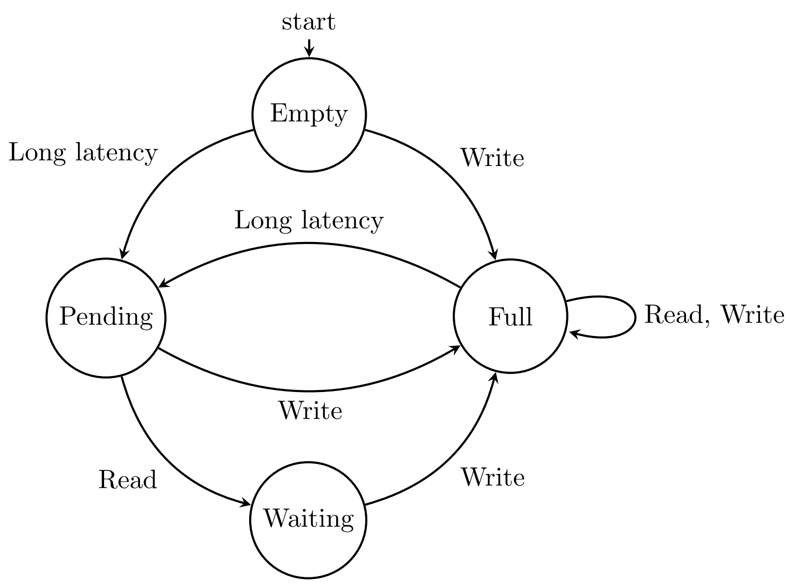

Figure 3.1: Register state transitions.

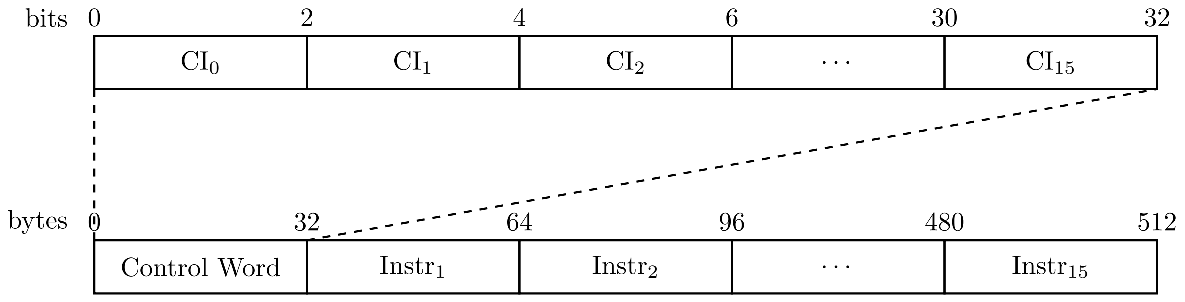

Figure 3.2: Cache line with control word. Shown is the control word on an architecture with 32-bit instructions and a 512-bit cache-line. Control Instruction CI_i is associated with instruction Instr_i. CI_0 is associated with the control word itself and is unused.

operands are available or not. This way, the pipeline before the instruction does not contain intruction from the current thread and the flush does not create bubbles.

The control stream is interleaved with the instruction stream. Since each control instruction is 2 bits wide, on a typical 32-bit RISC architecture, a single instruction word can contain 16 control instructions, enough to match 16 regular instructions. 16 32-bit instructions fit exactly in a 64-bit cache-line. Thus, if we sacrifice the first regular instruction and store a *control word* containing 16 control words in its place, we can store 15 instructions and their control instructions in a single 64-byte cache-line. This is illustrated in figure **??**. Having both streams in a single cache-line ensure that only a single cache-line has to be loaded for a thread, no matter its program counter. Note that this means that the size of an I-cache line must be at least the size of the instructions covered by a single control word. In the example above, a 32-byte I-Cache line can be created by only using half of the control word (thus covering only 7 instructions).

The cost of this optimization is additional logic in the assembler and hardware to recognize that the first instruction in any cache-line is a special control word that should not be executed. This matters, for instance, with labels, branches or return-from-procedure. If the target is the beginning of a cache-line, the label or PC must be updated to point to the instruction after the control word.

Alternatives As an alternative, the “end thread” control instruction can also be implemented as a normal instruction to bring the number of bits per instructions in the control stream down to 1. The advantage of this approach would be that a single control word can now cover twice the number of instructions. This means that a single I-cache line can be twice the size. On a machine with 32-bit instructions, this is 128 bytes instead of 64 bytes. The disadvantage is that very small threads that operate on arrays (i.e., array addition) will grow by one instruction, which is a significant amount on those levels of granularity. In the current design, 128-byte cache-line were deemed too big for general purpose as this point in time, so “sacrificing” them for not having the downside mentioned above seemed a sane path.

Another alternative is to switch on every cycle (interleaved multi-threading) and remove the *switch* control instruction. This, again, has the benefit of enabling larger I-cache lines. However, the cost of this is two-fold. First, every time a thread switch occurs, the family table and thread table must be read for the chosen thread's information that's required in the pipeline. At the end of the pipeline, additional logic required for switching out the old thread also requires accesses to the thread table and thread queues. This actions cost energy, which can be avoided since the compiler can statically determine if a switch is unnecessary. Secondly, when a thread switch occurs, the old thread (assuming it does not suspend) cannot be re-executed by the pipeline until several cycles later. The thread needs to first traverse the pipeline and then one or two thread queues before it is available to the pipeline again. If there are not enough threads to tolerate this rescheduling latency, the pipeline will have to execute nops until a thread becomes available. In the worst case, only two threads are available. Given a six-stage pipeline and two thread queues, the reschedule latency is 8 cycles, meaning performance is now 25% of what it could have been had the pipeline not switched blindly on every cycle.

Chapter 4

Instruction Set

4.1 Instruction list

The following instructions are the recommended instructions that can be added to a base RISC-like instruction set. They comprise the main functionality of the microgrid extensions to a RISC core.

4.1.1 Allocate

The `allocate` instruction allocates a single family context the cores on a place. A family context consists of a single family table entry, one thread table entry and enough registers to accomodate one thread's worth of registers assuming worst-case register usage. This way, after the `allocate` instruction has completed succesfully, the program can be sure that any family it places there will not suffer resource deadlock.

The instruction has one input operand containing the identifier (see section **??**) of the place where the context should be allocated, one input operand that contains flags and one output operand that will receive the family identifier (see section **??**) of the allocated context, or 0 if the allocation fails.

There are two variants of this instruction which the compiler can choose from based on static information at the point of compilation. The three variants are:

- Suspend (/S). This variant will not return failure. If not enough resources are available to satisfy the desired allocate, the allocate will suspend until it can satisfy the desired allocate. Note that if the *exact* flag (see above) is not given, this will only suspend if no context can be allocated on the first core.
- Exclusive (/X). This variant will try to allocate the special *exclusive* context on the first core in the specified place. A program can use this allocate to execute a family mutually exclusive with other threads that might run the same code exclusively on this place. Note that this variant implies the suspend behavior described above.

As mentioned above, the `allocate` instruction also takes flags as input. These flags modify the behavior of allocate based on runtime constraints. Currently, the following flags are defined:

- Exact. By setting this flag, the instruction will try to allocate a context on *exactly* as many cores as specified in the place identifier. If this variant is not used, it will allocate *up to* the specified number of cores. Since a family can run on any number of cores, not using this variant provides the program with automatic flexibility in the case of high load. However, sometimes there are algorithmic constraint or optimizations such as cache-usage that require exactly as many cores used as specified. Note that if the *suspend* variant of allocate is not used, the allocate will fail if no context on every core could be allocated.
- Single. By setting this flag, the allocate process will only allocate a context to run the entire family on the first core in the specified place. However, the family's place identifier is still set to the specified place. This allows child families of the created family to use a default place identifier (which inherits the parent's place identifier), and thus distribute across the entire place.
- Load balance. By setting this flag, the allocate process will find the *least busy* core in the specified place and create the family on that single core. This flag is conceptually equivalent to a "single" allocate on the least busy core in the place. The least busy core is found by choosing the core with the least number of used family table entries. Note that this flag, when used with the "exclusive" variant mentioned above, has the same effect as the "single" flag; i.e., it does not perform load balancing.

The `allocate` instruction is a long-latency instruction that clears its output register upon issuing. A reference to the register is passed along with the message to the allocate process. Once the allocate process has completed (succesfully or not), the family identifier, or 0, is written back into the register.

4.1.2 Set Start/Limit/Step

The `setstart`, `setlimit` and `setstep` instructions setup the family index sequence (see section **??**) of a successfully allocated family. Each of the three instructions has the same format, where they have two operands: one input operand containing the family identifier of the family whose index sequence to set up, and one input operand containing the value to write to the start, limit, or step field, respectively. These instructions can only be issued between the “allocate” and “create” instructions mentioned above and below, respectively. They are, however, idempotent, and can be issued multiple times to overwrite the previously set values.

To allow for efficient creation of functional concurrency, the defaults of the start, limit and step and 0, 1 and 1, respectively, creating a family of one thread. Thus, a function can be started concurrently with an `allocate`, `create` and one instruction per argument in between.

4.1.3 SetBlock

The `setblock` instruction sets the block size of a successfully allocated family. The block size is explained in section **??**. This instruction has one input operand containing the family identifier of the family whose block size to set, and one input operand containing the new value of the block size. If this instruction is not executed, the family retains its default block size, 0, which is replaced with the thread table size upon family creation. This instruction can only be issued between the “allocate” and “create” instructions mentioned above and below, respectively. It is, however, idempotent, and can be issued multiple times to overwrite the previously set value.

4.1.4 Create

The `create` instruction sets the program counter of the family and signals its creation at the same time. As such, this instruction has two operands: one input operand containing the family identifier of the family to create, one input operand containing the program counter of the threads in the family and one output operand that will receive the family identifier of the family (i.e., a copy of in the input operand) when the family has been created and its arguments can be sent. This instruction both writes the program counter to the allocated context(s) and signals the family creation process (see section **??**) that it can begin creating the threads of this family.

The `create` instruction is a long-latency instruction that clears its output register upon issuing. A reference to the register is passed along with the message to the create process. Once the create process has completed; meaning the registers have been allocated and the threads can be created and the arguments written, the family identifier is written back into the register.

4.1.5 Put Global/Shared

The `putg` and `puts` instructions write a register value into the globals or input of the dependency chain of a created family, respectively. Both instructions have three operands: one input operand containing the family identifier of the family whose global or first dependent to write, one input operand identifying which global or shared to write and one input operand containing the value to write. These instructions can only be issued after the family has been created with the “create” instruction described above.

The globals and initial dependencies start off as empty when the family is created, so when a thread attempts to read them, it will suspend on them. Thus, there is no time constraint for writing these arguments after the family has been created. However, after writing a global or shared, it should not be written again (with a different value) as this might produce non-deterministic behavior: the parent thread cannot know which value the child thread(s) use, this depends entirely on timing.

4.1.6 Synchronize

The `sync` instruction signals a created family that the issuing thread wants to synchronize (see section **??**) on its termination. The instruction has an input operand, containing the family identifier of the family it wishes to synchronize upon and an output operand, which will be written when the specified family has synchronized. This instruction sends a message to the target family, notifying it of its desire to synchronize on it, along with a reference to the output register. If, upon arrival of the message, the family has already completed, a message is sent back that will write an undetermined value to the output register. Otherwise, the reference to the register is stored in the family context, so that the message can be sent when the family completes. This instruction can only be issued after the family has been created with the “create” instruction described above and should be issued only once for the specified family.

Note that issuing this instruction will *not* suspend the thread until the specified family has completed. It only notifies the specified family and sets up a reference to the output register. The thread should read this instruction’s output register after issuing this instruction to actually suspend until the family has terminated.

The `sync` instruction is a long-latency instruction that clears its output register upon issuing. Once the specified family has completed, meaning all threads have terminated and all memory writes are consistent, the output register is written.

4.1.7 Get Shared

The `gets` instruction reads the output of the dependency chain of a terminated family. The instruction has three operands: one input operand containing the family identifier of the family whose last dependent to read, one input operand identifying which shared to read and one output register that will receive the read value. This instruction can only be issued after the family has been synchronized upon with the “sync” instruction described above.

The `gets` instruction is a long-latency instruction that clears its output register upon issuing, since the act of retrieving the output of the dependency chain will most likely several cycles, depending on the distance of the family from the issuing thread. Once the value is retrieved, the output register is written.

4.1.8 Detach

The `detach` instruction signals a created family that the issuing thread is no longer interested in it. This means that once the specified family has terminated, it can be cleaned up as no further requests will be made to it. This instruction has only a single input operand containing the family identifier of the family it wishes to detach from. `detach` is typically the last instruction for a family to be executed, usually after the `sync` and optionally `gets` instructions described above. Failing to issue this instruction will cause a resource leak that could lead to deadlock or severely reduced performance once the hardware’s physical resources have been exhausted.

Note that this instruction can be issued immediately after the `create` instruction described above. In such an event, the thread can no longer synchronize on the family’s completion or read its output dependents.

Chapter 5

Thread Table

The thread table is a conceptual table on every core, where every rows contains the information for a thread that has been created on that core. In practice, its different fields are used by different components and therefore the thread table will most likely be split up into distinct parts and located near the core components that use them. This document describes the thread table in its conceptual form. Its contents is listed in table **??**. The rest of this chapter describes, in detail, the rationale and uses of the fields.

5.1 Registers

This section covers *int_locals*, *int_dependents*, *int_shareds*, *flt_locals*, *flt_dependents* and *flt_shareds*.

The address of the local, dependent and shared registers of each thread is different for each of them. Since the location of a thread within the block of allocated register does not have to be the same (modulo the block size) due to out-of-order cleanup and thread entry reuse for independent families, there is no relation that can describe the mapping of a thread's index to the location of its locals. Therefore, the address of the locals must be explicitely stored in the thread table. Similarly, any relation describing the location of the shareds and/or dependents from the thread index would be significantly complicated. It is easier to take advantage of the sequential allocation mechanism of threads and set the address of the dependents to the address of the shareds of the last allocated thread. Although the shareds of new threads (those allocated to a not-yet-used thread entry) always lie above the locals, when threads are reused, the location of these shareds no longer maintains this relationship, meaning the address of a thread's shareds also has to be stored explicitely.

5.2 Dependencies

This section covers *killed*, *prev_cleaned*, *next_sequence* and *pending_writes*.

When a thread terminates, it may not yet be able to be cleaned up or reused. The events that prevent this from happening are called the thread's dependencies.

First, all of a thread's writes must be acknowledged by the data cache. If this dependency did not exist, the thread could have been cleaned up or reused when its pending writes are acknowledged by the memory system. The data cache, will adjust the pending_writes count under the assumption that the thread might have executed a memory write barrier instruction and may want to know when its writes have completed. However, if the thread has been reused, the data cache can now potentially wrongly acknowledge writes made by the new thread. This, in turn, can make that thread assume that its writes have been acknowledged while they have not, leading to a violation of the memory consistency model. Thus, a thread cannot be cleaned up or reused until all of its pending writes have completed, i.e., until the pending_writes counter is zero.

Note that there is no technical upper bound on the number of pending writes a thread can have. In the event that issuing a memory write would overflow the counter, the memory write will have to stall. Thus, the number of bits for the pending_writes field has to be chosen to minimize the number of stalls, which is dependent on the implementation, notably the rate of memory writes as issued by the thread and the latency of memory write completions.

The second dependency of a thread is that of *sequential cleanup*; in a family with a dependency (i.e., at least one shared), a thread cannot be cleaned up until its predecessor in the index sequence has been cleaned up. This dependency arises from the implementation of communication of shareds between threads. The dependents of thread $i + 1$ are mapped onto the shareds of thread i. Given that a thread may not read all shareds that its predecessors writes, thread $i + 1$ should not be cleaned up and reused before thread i has terminated, guaranteeing that it will write no more shareds. The prev_cleaned flag indicates whether the previous thread in the index sequence has been cleaned up. To enable a thread to notify the next thread in index sequence that it

Name	Purpose	Bits
pc	Current program counter	62
int_locals	Offset in the integer register file of the locals	$\lceil \log_2 \#\text{Registers} \rceil$
int_dependents	Offset in the integer register file of the dependents	$\lceil \log_2 \#\text{Registers} \rceil$
int_shareds	Offset in the integer register file of the shareds	$\lceil \log_2 \#\text{Registers} \rceil$
flt_locals	Offset in the FP register file of the locals	$\lceil \log_2 \#\text{Registers} \rceil$
flt_dependents	Offset in the FP register file of the dependents	$\lceil \log_2 \#\text{Registers} \rceil$
flt_shareds	Offset in the FP register file of the shareds	$\lceil \log_2 \#\text{Registers} \rceil$
killed	Has the thread terminated execution?	1
prev_cleaned_up	Has the thread's predecessor been cleaned up?	1
write_barrier	Has the thread suspended on pending_writes?	1
pending_writes	Number of unacknowledged writes	N
cid	Cache-line containing the thread's instructions	$\lceil \log_2 \#\text{I-Cache lines} \rceil$
family	Family index of thread's family	$\lceil \log_2 \#\text{Families} \rceil$
next_sequence	Thread index of the thread's successor	$\lceil \log_2 \#\text{Threads} \rceil$
next	Next thread in the linked list	$\lceil \log_2 \#\text{Threads} \rceil$

Table 5.1: Contents of a thread table entry

has been cleaned up, every thread has a next_sequence field. If this field is invalid at the point where the thread is cleaned up (i.e., the next thread has not yet been allocated), the family's *prev_cleaned* field is set instead.

The third dependency, "killed", simply means that the thread has terminated. This flag is necessary because a thread can terminate before the other two dependencies have been met, so its status is encoded in this flag as another 'dependency'.

Combined, a thread can be cleaned up (pushed onto the cleanup queue) when the logical AND of "killed", "prev_cleaned" and "pending_writes = 0" is true.

5.3 Scheduler

This section covers *pc* and *cid*.

Since every thread can follow independent execution paths, each needs its own program counter (PC). When a thread on the active list is switched to by the pipeline, its program counter is read from the thread table. This program counter is used to read the instruction data from the instruction cache. Since threads on the active list are guaranteed to have their cache-line present in the cache-line, the associative lookup can be avoided by storing the cache-line index (CID) in the thread table as well. The pipeline uses this field to read the cache-line from the i-cache and execute the thread. The pipeline will increment the PC locally at the beginning of the pipeline while executing the thread. When the pipeline has switched to another thread, the next program counter of the thread is written back to the thread table at the end of the pipeline.

5.4 Management

This section covers *family* and *next*.

The family field identifies the family entry (on the same core) of the family that the thread belongs to. This is used when cleaning up, reusing or scheduling a thread, since family information is required at those points.

The next field allows linked lists of thread entries to be formed. Linked lists of threads are used when threads that were waiting on a single register are woken up and when threads that were waiting on a single i-cache line are woken up (see sec **??**).

5.5 Memory Write Barrier

This section covers *write_barrier*.

This single-bit field identifies whether the thread is waiting for the "pending_writes = 0" condition. This is separate from the dependency listed in the section above. When pending_writes reaches zero and write_barrier is set, the thread should be rescheduled. Checking the pending_writes field and setting the memory_barrier field is typically an atomic operation for a "memory write barrier" instruction. The D-Cache atomically decrements the pending_writes field and checks the memory_barrier field when a memory write completes. When the former hits zero and the latter is set, the thread is woken up.

Chapter 6

Family Table

The family table is a conceptual table on every core, where every rows contains the information for a family that has been created on that core. In practice, its different fields are used by different components and therefore the family table will most likely be split up into distinct parts and located near the core components that use them. This document describes the family table in its conceptual form. Its contents is listed in table **??**. The rest of this chapter describes, in detail, the rationale and uses of the fields.

6.1 Registers

This section covers *int_count_globals*, *int_count_shareds*, *int_count_locals*, *int_base*, *int_shareds*, *flt_count_globals*, *flt_count_shareds*, *flt_count_locals*, *flt_base* and *flt_shareds*.

Every thread in a family has the same number of globals, shared and local registers in its context. These counts are stored in the family table in the *int_count_** and *flt_count_** fields. During the family create process, these fields are loaded from the first cache-line of the thread, where they are stored in the second word of the cache-line (the first word contains the control word, as described in section **??**).

Using these fields, together with the index sequence that has been set up before the create, the number of registers required to run a number of threads concurrently is calculated. These registers are allocated and the base address to the consecutive block of registers stored in the *int_base* and *flt_base* fields.

When a family has shareds, a newly allocated thread must map its dependent registers to the shared registers of the previous thread in the index sequence, i.e., the last allocated thread in that family. To enable this, the family entry stores the address of these shareds in the *int_shareds* and *flt_shareds* fields. Thus, when creating a new thread, these fields can be copied to the thread table as the address of the thread's dependents, and the thread's shareds address copied into these fields.

6.2 Multi-core

This section covers *place_size*, *num_cores*, *first_lfid*, *has_link* and *link*.

A family is created on a place, a consecutive set of cores. However, under certain circumstances, the family uses fewer cores than given to it in the place identifier. Since the place size is a property of a family that is inherited when threads in that family create additional families with the "default" place identifier, the place size is stored for each family in the *place_size* field.

The actual number of cores that the family is distributed over is stored in the family's *num_cores* field. This value is guaranteed to be less than the place size, and also a power of two. This field is used to determine the distribution of threads when the family is created. Note that only the number of cores is stored in the family table. The address of the first core does not need to be stored as it can be inferred from the place size. Given that the address of the first core in a place is always a multiple of the place size (see section **??**), the address of the first core in a family on any core can always be determined by aligning the address of the core in question down to the highest multiple of the family's place size. Given that the place size is always a power of two, this operation is a simple shift and mask.

When a family is distributed across several cores, the family allocation protocol will allocate an entry on every core that the family will run on. Since this entry may not have the same index in the family table on every core, each family entry identifies the index in the family table of the family's corresponding entry on the next core with the *link* field. Since all communication (except break, which will be discussed shortly) strictly moves into one direction on this chain (starting at the first core), the end of the chain must be somehow identified. The *has_link* field serves this purpose. If set, that core is the last core in the chain onto which that family is distributed.

Name	Purpose	Bits
$\log_2$ place_size	Number of cores that this family wanted to run on	$\lceil\log_2\lceil\log_2(\#\text{Cores})\rceil\rceil$
$\log_2$ num_cores	Number of cores that this family is running on	$\lceil\log_2\lceil\log_2(\#\text{Cores})\rceil\rceil$
capability	Capability of the family	N
pc	Program counter for new threads	62
block_size	Maximum number of threads to have allocated on this core	$\lceil\log_2(\#\text{Threads})\rceil$
start	Start/Current index	64
step	Index step	64
limit	End index / Number of threads left to allocate	64
first_lfid	Index of this family on the first core in the place	$\lceil\log_2(\#\text{Families})\rceil$
has_link	Is there a family on the next core (link valid)?	1
link	Index of the associated family on the next core	$\lceil\log_2(\#\text{Families})\rceil$
prev_cleaned_up	Has the last allocated thread been cleaned up?	1
sync_done	Has the family completed for synchronization?	1
sync_waiting	Is there a thread waiting for sync (sync_pid and sync_reg valid)?	1
sync_pid	Address of the core containing the synchronizing thread	$\lceil\log_2(\#\text{Cores})\rceil$
sync_reg	Register offset on the sync_pid core that should receive the sync_code	$\lceil\log_2(\#\text{Registers})\rceil$
alloc_last	Index of the last allocated thread in the family	$\lceil\log_2(\#\text{Threads})\rceil$
alloc_done	Is the family done allocating threads?	1
prev_synced	Has the associated family on the previous core synchronized?	1
detached	Has the parent thread detached from the family?	1
alloc_count	Number of threads allocated to the family	$\lceil\log_2(\#\text{Threads})\rceil$
pending_reads	Number of pending reads made by threads in this family	$\lceil\log_2(\#\text{Registers})\rceil$
int_count_globals	Number of globals in the threads's integer context	5
int_count_shareds	Number of shareds in the threads's integer context	5
int_count_locals	Number of locals in the threads's integer context	5
int_base	Register file offset of the family's allocated integer registers	$\lceil\log_2(\#\text{Registers})\rceil$
int_shareds	Register file offset of the alloc_last's integer shareds	$\lceil\log_2(\#\text{Registers})\rceil$
flt_count_globals	Number of globals in the threads's FP context	5
flt_count_shareds	Number of shareds in the threads's FP context	5
flt_count_locals	Number of locals in the threads's FP context	5
flt_base	Register file offset of the family's allocated FP registers	$\lceil\log_2(\#\text{Registers})\rceil$
flt_shareds	Register file offset of the alloc_last's FP shareds	$\lceil\log_2(\#\text{Registers})\rceil$

Table 6.1: Contents of a family table entry

As mentioned, all communication is strictly down the chain of cores, except for break. When a thread in a family issues the `break` instruction, a message needs to be sent to the first core of the family, referring to the family entry on that core. Given that only the core address can be inferred from the core from which the break is issued, the index in the family table is explicitely stored in the family table on each core, in the *first_lfid* field[1].

6.3 Security

This section covers *capability*.

To prevent unauthorized access to family table entries, each family entry is assigned a (pseudo)random value that is generated when it is allocated. This value is returned to the thread that issued the original allocate, as part of the Family ID. Every request from programs that operates on a family is passed a Family ID. If the capability in the Family ID does not match the capability in the family entry, the request must not be fulfilled. Depending on the implementation, a security violation may be raised as well.

6.4 Thread Allocation

This section covers *pc*, *start*, *step*, *limit*, *block_size*, *alloc_count*, *alloc_last* and *prev_cleaned_up*.

Once a family has been allocated, setup, created and initialized, its threads have to be allocated. This is done by a special hardware process as described in section **??**. When this process has to initialize a thread for execution, it uses the fields described in this section as follows:

The *pc* field in the family table is copied to the *pc* field in the thread table, thereby pointing the new thread to its first instruction. After thread creation, the I-Cache will use the thread's *pc* to fetch its instructions from memory.

The *start* field indicates the index of the next thread. Upon thread creation, this value is written to the first local integer register of the thread, if any, and incremented by the *step* value. For distributed families, the family creation process will adjust the *start* field to the index of the first thread that is to be created on the core (see section **??**).

The *limit* field is setup by the parent thread as part of the family creation process. This value is used in the family creation process to determine the number of threads to create in the family. At this point, the *limit* field holds the number of threads that still have to be created on the core. Whenever a thread is created in a family, the *limit* field is decremented. When it has reached 0, no more threads are created on this core. Just like the *start* field, this field is set by the family creation logic to reflect the number of threads that are to be created on the core in question.

The *block_size* field is setup by the parent thread and indicates the maximum number of threads that can run concurrently on a single core (i.e., the maximum number of thread entries in use by a family). The thread creation process will keep allocating additional thread entries for the thread of a family on a core until either all threads in the family on that core have been created, or until the block size has been reached. This field defaults (and can be set) to 0, which will be replaced with the thread table size during family creation.

The *alloc_count* field keeps track of the number of thread entries allocated to this family. It is compared with the block size on thread creation, as described above. Towards the end of a family, when all threads in a family on the core have been created, when threads complete, their entries are freed. When a thread entry is freed, the *alloc_count* field in the corresponding family's entry is decremented by one. The family entry cannot be freed until this counter has reached 0, indicating that all threads in the family have been freed.

The *alloc_last* field points to the most recently allocated thread of the family. When a thread is created, and the *alloc_count* field is not zero (i.e., the new thread is not the first one), the *next_sequence* field in the thread entry indicated by the *alloc_last* field is set to the index of the new thread's entry. This way, a chain of the threads in a family is constructed.

The use of the *next_sequence*, as described is section **??**, is to proper cleanup semantics for families with shareds. However, it is possible that when a thread completes and it wants to notify the next thread in index sequence, this thread does not yet exist. In this case, the "cleanup" signal is buffered in the family table in the *prev_cleaned_up* field. Upon thread creation, this flag is copied to the new thread's *prev_cleaned* flag, which would otherwise have been set by the cleanup process of the previous thread.

6.5 Dependencies

This section covers *alloc_done*, *detached* and *pending_reads*.

[1]Alternatively, this field can be removed if a software solution can be adopted where the parent sends the family identifier as a global to the family, which the break operation can then use. This would then also allow for breaks from outside the family.

To know when a family entry can be freed, several conditions must be met. These conditions are called the family's dependencies.

First, a dependency which is mentioned in the previous section instead of this one, is the *alloc_count* field. A family entry cannot be freed until all its threads have been freed, i.e., when the *alloc_count* field is zero.

Secondly, a family must wait until all of its threads have been created, as indicated by the *alloc_done* flag. Without this condition, a family could be cleaned up as soon as it is created, as all other dependencies are fullfilled at that point. This flag is either set by the thread creation process, when the *limit* field has reached 0, or set by the family creation process, when the *limit* field starts out at 0 (i.e., the core has no threads to run).

The third dependency is that all memory reads must be completed. Either ill-written programs or programs with speculative memory loads can cause memory loads to still be pending when a thread terminates. These memory loads still have pending references to the family through the register file. As such, these loads must complete before the family entry (and its allocated registers) can be freed. The *pending_reads* field is incremented for every load that is sent to memory and decremented for every load that completes.

The last dependency is that the parent thread must have detached from the family, as indicated by the *detached* flag. Detaching from a family means that the thread can no longer reliably interact with the family. A thread can detach from a family at any point after executing the final family create operation. It can also detach without synchronizing on the family.

6.6 Synchronization

This section covers *prev_synced*, *sync_done*, *sync_waiting*, *sync_pid* and *sync_reg*.

In order to use the results of the thread of a family, a thread can *synchronize* on the completion of the family. To this end, it specifies a family to synchronize on and a register which is cleared when executing the `sync` instruction and written when the synchronization has completed.

The sync_done flag is used to signal the synchronizing state of a family. When set, the family is considered 'done' and any synchronization made on the family will complete.

When a thread executes a synchronization, a message is sent to the family with the core ID and register address of the specified register. Upon arrival, the sync_done flag is checked. If set, the family is already done, and a message is sent back which will write to the specified register. Otherwise, the sync_waiting flag is set and the core ID and register address are written into the family's sync_pid and sync_reg fields, respectively. When the sync_done flag is set, the sync_waiting flag is checked and if set, a message is sent to the core specified by sync_pid to write to the register specified by sync_reg. Note that, because the sync_pid and sync_reg fields are overwritten, if multiple syncs are executed on the same family, at least one of them will complete. The others may or may not complete, depending on the state of the family at the time of their arrival.

Chapter 7

Thread Creation

This chapter describes the process and hardware involved with creating threads in the form of families of threads. The thread creation process starts when the software decides, at some point, to create a family of threads. The steps to create a family of threads are:

1. Obtain a place identifier to identify the place where the family should be created.
2. Allocate a context on the cores on the place.
3. Set up the allocated contexts with the index sequence and program counter.
4. "Create" the family; the threads in family can now be created.

The sections below describe these actions in detail.

7.1 Place identifier

7.1.1 Overview

First, the program must decide *where* to create the family. A microgrid consists of many cores, each of which can be the target of a create. The program constructs a place identifier, which is the address of the core to create to, and the number of cores to create the family on. This place identifier is obtained through an unspecified means (e.g., a software library responsible for allocating cores to applications). Figure **??** shows the contents of the place identifier. The place identifier is meant to fit in a single register. Its contents are:

- **PID/Size**. These bits identify both the address and size of the place to create at. The addressing scheme is implementation-dependent but should model a linear chain of consecutive cores.
- **Capability**. These bits, the number of which depends on the implementation, make up a value which must be matched with the destination core's configured security value. This value is used to ensure that only an authorized application can perform delegated creates to certain cores.

A special place identifier value, of all 0s, is used to signal a "default" place. The default place is the same place as the thread that issues the allocate. I.e., when the place identifier is 0, the place identifier of the thread is used.

7.1.2 PID/Size

The address and size of a place on a grid of 2^A cores are encoded in $A+1$ bits in the place identifier. Specifically, the encoding is defined as follows:

```
Encode(PID,Size) = PID * 2 + Size
```

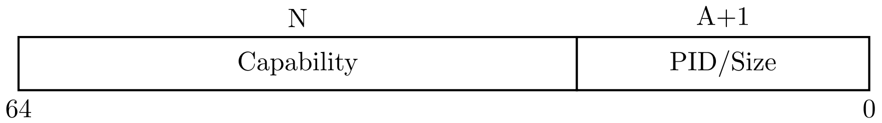

Figure 7.1: Contents of a place identifier. From left-to-right: capability, exact, place address and size.

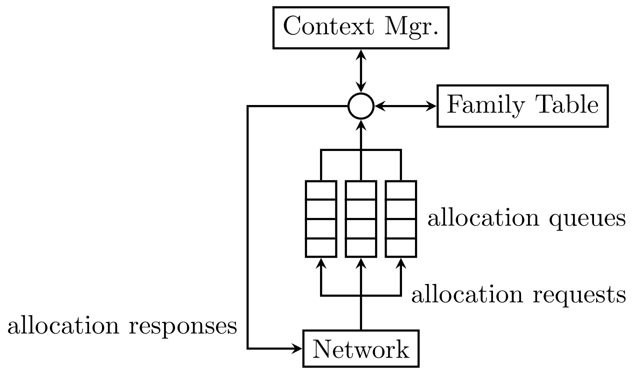

Figure 7.2: Overview of the components involved with the family allocation process.

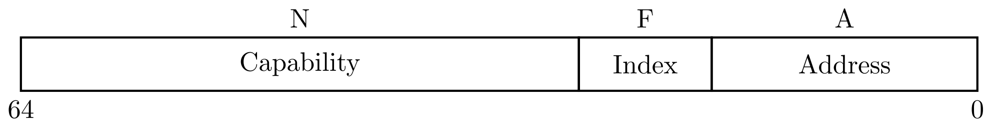

Figure 7.3: Contents of a family identifier. From left-to-right: capability, family index, core address.

This encoding requires that `Size` is a power of two and that `PID` is a multiple of `Size`. The former requirement ensures that the hardware can calculate thread distributions with simple shifts instead of complex divisions. The latter requirement enables this encoding, which allows for efficient distribution of families of threads over a place by software without implementation-dependent knowledge or library calls.

The PID/Size field is decoded as follows: the index (starting from the least significant bit) of the first '1' bit indicates the ($\log_2$ of the) size of the place. This bit is then cleared and the PID/Size field shifted right by 1 to obtain the place address.

Listed below are some simple operations to create a sub-place identifier from an existing place identifier (`PID`). The ($\log_2$ of the) size of the input place (`Size`) can be found with a single bit scan instruction.

- Lower half of the place: `PID - Size/2`
- Upper half of the place: `PID + Size/2`

Note that none of these operations require knowledge of A, the number of bits that the `PID/Size` is encoded in.

7.2 Family allocation

After a program has composed a place identifier, this is given to the `Allocate` instruction. This instruction decodes the place identifier and sends an allocation request to the target core, which might be the same as the core the instruction is executed on. Eventually, the network controller on the target core handles the allocation request. The result of an allocation request is an allocation response, which writes a *family ID* into the register specified by the `Allocate` instruction. A family ID, as illustrated in figure **??**, consists of the allocated family's index in the family table, the family's capability and the core address. Thus, the family ID provides global access to the allocated family. The capability exists to only allow access to the family entry to whoever holds the correct family ID.

Note that the number of bits for the family index and capability may depend on the core address. This allows for a heterogeneous system where certain cores have differently sized family tables. As a result, assuming that not every core knows about the architectural properties of other cores on the system, the FID should only be decoded on the core indicated by the core address. Other cores should only look at the core address part of the FID to determine whether to unpack the FID and act on it locally or send a message with the FID *as-is* to the specified core.

Figure **??** shows an overview of the components on a core involved with family allocation.

7.2.1 Allocation queue

Once an allocation request arrives on a target core, it is simply placed in one of three queues. The three queues represent different kinds of allocation requests (implemented with three different instructions, or instruction

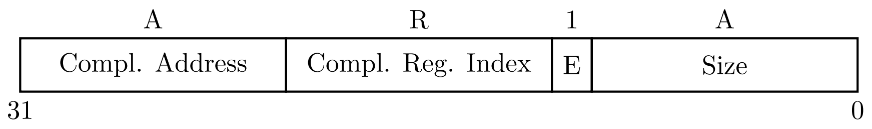

Figure 7.4: Contents of an entry in the allocation queue. From left-to-right: completion core address, completion register index, exact and place size.

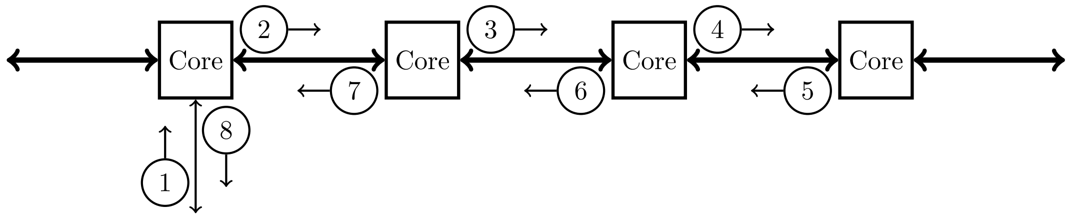

Figure 7.5: Place allocation process.

modifiers, see chapter ??), neither of which should stall the other kinds. The three kinds are:

- **Suspend**. If no context can be allocated, wait until a context is available.
- **Non-Suspend**. If no context can be allocated, return a failure code.
- **Exclusive**. Allocate the exclusive context or wait until it is available.

In case of the Non-Suspend queue, if the context manager indicates that not enough resources are available for a single context, the allocation request cannot continue. In this case, an allocation response with a failure code (a family ID of all 0s) is returned. Thus, if the program issued an Non-Suspend `Allocate` instruction, it should always test the returned family ID from the instruction.

Figure ?? shows what information is stored on the allocation queue for a single allocation. The `Size` and `Exact` fields are taken from the place identifier and describe how many cores should be allocated for this family. The two completion fields make up a globally unique register identifier to identify where the family ID should be written to when the allocation succeeds (or fails, for Non-Suspend allocates).

A hardware process is responsible for handling the suspended allocation requests on these queues. It does this by checking the context manager if there is a free context, and allocating it if there is. If no context is available, an entry is removed from the Non-Suspend queue and `failure` written back to its completion register.

7.2.2 Place allocation

The next step, after allocating a free context depends on the place size as indicated in the place identifier. If only a single core was to be allocated, the allocation process constructs the family ID for the allocation family entry and sends a network message to write this family ID to the completion register.

However, if the place spans multiple (consecutive) cores, a message is sent to the next core (over the link network for performance, see section ??) with the allocation details. Once received at the next core, the request is queued in the suspend or non-suspend allocation queue as well and handled by the allocation process.

As long as a context can be allocated, this process continues until a context has been allocated on the last core in the place. At this point, a `Commit` message is sent back (over the link network) which sets up the `link` field in each core's family entry to point to the corresponding family entry on the next core. Once this message returns to the beginning of the place, a network message is sent to write the family ID back to the completion register.

If, however, at some point, a context could not be allocated, and the exact flag is not set, an `Unwind` message is sent backwards, freeing up allocated contexts. When enough contexts have been freed to make the number of remaining allocated contexts a power of two, the unwind message continues as a commit message. This protocol effectively allocates a smaller place than requested (if the `Exact` flag is not set).

This process is illustrated in figure ??. Message 1 indicates the first allocation request coming in from the network. Messages 2–4 are allocation requests sent across the place over the link network. After the last core has allocated an entry in response to message 4, it returns a commit message which moves back over the link (messages 5–7). Once it has arrived back at the first core, an allocation response is sent back over the network.

7.2.3 Context management

The context manager maintains a count of free resources on the core, consisting of register blocks, thread entries and family entries. A context consists of one of each, and is sufficient to execute a family, one thread at a time. There are three kinds of contexts:

- **Free**. These contexts can be allocated or reserved for non-exclusive allocates.
- **Reserved**. A conceptual group whose contexts have been reserved for later use. Family entries never exist in this group, as they are either allocated or not. However, once a family entry is allocated, a thread entry and register block are moved to the reserved group. Note that it is strictly speaking not necessary to maintain an actual count of these contexts, assuming the allocation protocol works correctly.
- **Exclusive**. This is a single context that exists to guarantee that a core will always have a single context available for executing exclusive families. Non-exclusive allocates cannot use this context.

Only when all three counters of the desired context kind are non-zero can the allocation succeed. When this is the case, all counters are decremented by one, in essence allocating enough resources for the new family to be able to run at least one thread at a time. A family entry is allocated as well, and initialized with a (pseudo-)random security value ("capability").

7.3 Family setup

After the application has obtained a valid family ID from the allocation process, it can optionally override the index sequence and block size, whose values default values (after the family has been allocated) are set to create a family of one thread, as follows: start = 0, limit = 1, step = 1, block = 0.

7.3.1 Index sequence

The family's index sequence, determining how many threads are created, and which index values are given to those threads, is set with the `setstart`, `setlimit` and `setstep` instructions. These instructions take two arguments: the family ID and the new value of the specified family entry's start, limit or step field, respectively. Upon executing one of these instructions, the pipeline sends a network message to the destination core, which might be same core. On the destination core, the network logic handles the message and writes the value into the family entry.

The values of the start, limit and step fields are interpreted based on the value of the step field, as follows:

step = 0:	(Invalid)
step > 0:	`for (int i = start; i < limit; i += step)`
step < 0:	`for (int i = start; i > limit; i += step)`

That is, the family index is equivalent with the specified for loops, where one thread is created for each iteration, in sequence. Note that the fields, and their comparisons are *signed* comparisons on integers with machine-word size.

7.3.2 Block size

The block size for the new family is set with the `setblock` instruction (see section **??**). The block size determines an upper bound on the number of thread entries allocated on a single core for the family, i.e., the size of the family's block of threads. This value may be further constrained by the hardware based on the availability of registers, as described in section **??**. The block size exists to prevent an explosion of resource usage of the non-leaf nodes in the concurrency tree, by restricting the resources that those nodes use.

A block size of 0, which is the default when a family entry is allocated, indicates the maximum possible block size, i.e., the thread table size.

7.4 Family creation

Once a family has been allocated and initialized, the program can issue a `create` instruction. This instruction takes the family ID of the setup family and a pointer into memory identifying the beginning of the family's threads' executable code.

Upon executing this instruction, the pipeline checks if there are pending writes for this thread. If so, the pipeline pretends a memory write barrier has been executed and suspends the thread on the pending writes (see

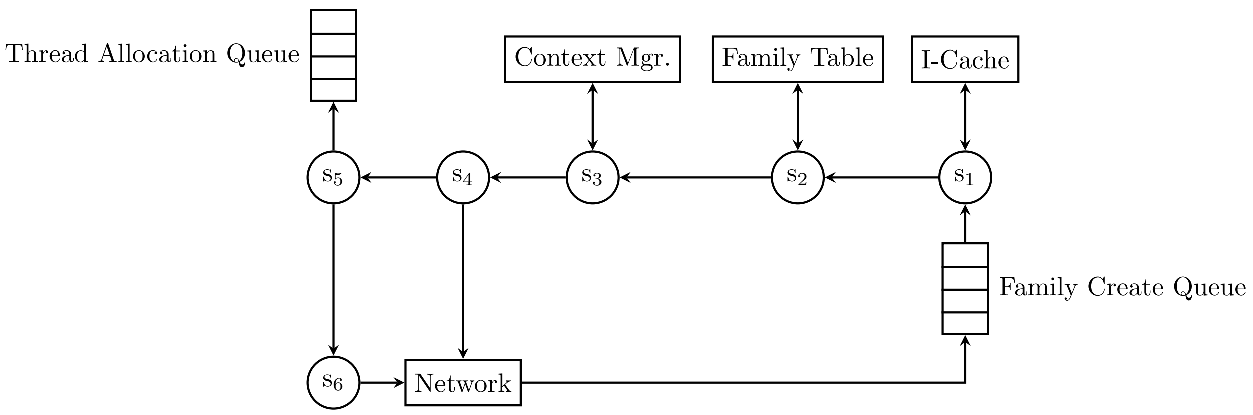

Figure 7.6: Family creation process.

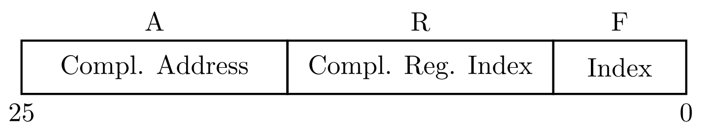

Figure 7.7: Contents of an entry in the create queue. From left-to-right: completion core address, completion register index, family index.

section ??). When the data cache acknowledges the last pending write, the thread is reschedule and the create instruction re-executed. This in effect turns family creation into an implicit write barrier, allowing earlier-made writes by the thread to be committed to memory before creating the family, which might need that data in memory.

If the create is executed and there are no pending writes, a network message with the create information is sent to the specified core. The network controller on the target core will write the code pointer and buffer the create information on that core's create queue. The buffered information is shown in figure ??.

The create queue is handled by a seperate process which implement a multi-stage, multi-cycle operation. This process can be implemented by a state machine or pipeline. Figure ?? shows this process in a pipelined form and how it interacts with the various components on the core. The stages of the process are explained next.

7.4.1 Register counts

The first stage of the create process reads the entry from the front of the family allocation queue and the code pointer from the family table. It uses the code pointer to fetch the cache-line containing this code pointer through the I-Cache. This cache-line also contains the register count word for the threads of the family. The register count word is the first word before the location pointed to by the code pointer and contains the number of local, shared and global registers that should be used for each thread's virtual register context (see section ??). Figure ?? shows the contents of this word. If the fetch hits the I-Cache, the family ID continues to the second stage. Otherwise, a flag is set in the I-Cache's line (see section ??) which will cause the I-Cache to wake up this process and advance it to the next stage when the cache-line has been loaded.

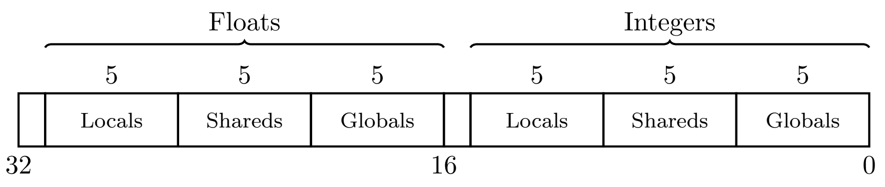

Figure 7.8: Contents of "register count" word on a RISC architecture with 5-bit register specifiers and 32-bit instruction words.

7.4.2 Family setup finalization

The next stage finalizes the family setup with the register information that was loaded in the previous stage. The shared register count is checked to determine if the family is independent (when all shared counts are 0). This information, combined with the number of cores that have been allocated to the family is used to calculate the thread distribution.

The number of threads in the family is determined from the start, limit and step fields as described earlier in this section. Based on the number of threads. The threads are evenly distributed across the allocated cores, where each core executes $\lceil \#threads/\#cores \rceil$. Should the number of threads not be an integer multiple of the number of cores, the last core executes fewer threads. Note that the number of allocated cores is always a power of two to simplify this division to a shift in hardware.

Threads of dependent families require their shareds communicated between threads. If these threads are distributed across multiple cores, the administration of keeping track of these dependencies is non-trivial. Considering, as well, that the global performance of dependent families is coupled with the size of the independent pre- of post-fix to the dependent part in the thread, the architecture does not distribute dependent families. Such families are executed on the first core, regardless of the number of cores that have been allocated. Should a dependent algorithm need to be distributed across multiple cores, a multi-family approach can be used where the top-level dependent family runs on the first core and creates a single dependent family on each of the allocated cores. The only non-local register communication is the initial and final shareds of the family.

Exclusive families, just like dependent families, execute all their threads on the first core as well.

With the number of threads and their distribution known, this stage advances the *start* field in the family entry to the correct value based on the number of threads per core and the core's offset to the first core in the family. The *limit* field is written with the number of threads that are left to be created on this core. Note that it thus takes on a different semantic meaning. The same field for both meanings is used to save space in the family table. This can be done because their uses are mutually exclusive.

7.4.3 Register allocation

The next step in the family create process is allocating the desired amount of registers from the register file. The context manager has an administration of available registers that is consulted to accomplish this task. Typically, the allocation granularity is not a single register, but a block of registers, at least being a single thread worth of registers. Having a larger allocation block size reduces the logic involved with allocating registers since there are fewer blocks to allocate, reduces fragmentation of the register file and makes it trivial to guarantee that a single context reservation is available as a block of consecutive registers.

The family's block size, as set by the program, is used to calculate the number of registers to allocate: `#globals + #shareds + block_size · (#locals + #shareds)`. This amount is rounded up to a multiple of the allocation block size and a contiguous range of registers of this size is allocated from the available registers, if possible. This is done for both the integer and floating-point registers.

If the context manager indicates that such there are not enough free registers, the block size is reduced by one, and the process is repeated. Since family allocation reserves one register context for the family, and given that `#globals + 2 · #shareds + #locals` should not be greater than a single context, this process is guaranteed to at least succeed if the block size goes down to one.

7.4.4 Create distribution

With the distribution determined and all resources allocated on this core, the next step forwards the create via the link network to the next core, if any. The message indicates the family entry index of the family on the next core (read from the family's *link* field). Once received, the network controller on the next core queues the create. When it gets handled, its registers are allocated, the family pushed onto the thread allocation queue (see next section) and another create message is sent to the next core over the link network.

7.4.5 Thread allocation queue

At this point, the family is ready to start creating threads. The family index is pushed on the core's *thread allocation queue*, a queue of family indices which indicate families that have been created and are ready to start executing threads. Section **??** discusses this mechanism.

7.4.6 Creation notification

To ensure that register writes from the parent and the first thread in the family do not occur when no registers have been allocated, creates are not notified until after the register have been allocated and the create has been sent to the other cores. This way, when the parent receives the create notification (i.e., the register specified at the create instruction is written), subsequent global writes sent to the first core in the family will succeed the create notification on the link network, thus ensuring that registers will always be allocated once the global write arrives.

This stage is the final stage in the family creation process.

7.5 Thread allocation

Once a family index has been pushed onto the thread allocation queue, it means its family entry is entirely setup and enough registers have been allocated to accomodate the block size. A hardware process is responsible for allocating threads to this family.

After a family has been created on a core, its index is pushed onto the *thread allocation queue*. This is a queue of families that need to have threads created for them and have fewer threads than the block size. The thread allocation hardware process continuously creates threads for the family at the head of this queue until it has reached the block size, or until the family has created all its threads.

Once a thread has terminated and its dependencies (see section **??**) have been resolved, the thread is pushed onto the cleanup queue. Since thread cleanup results in the reuse of the thread entry, thread cleanup is essentially thread allocation. Therefore, thread cleanup is performed by the same process as thread allocation. If the core's cleanup queue is not empty, the thread allocation process pops a thread off this queue. If the thread's family does not have its thread allocated yet, the thread entry is used to allocate the next thread in the family. Otherwise, the thread entry is put on the core's empty thread list (i.e., the entry is freed).

Only if the cleanup queue is empty, does the thread allocation process look at the thread allocation queue.

After allocating (or reusing) a thread entry, it must be initialized for initial execution of the thread. This involves copying various fields from the family entry to the thread entry and updating various fields in the family table.

Note that the thread's local register offset field only has to be initialized if the thread entry is freshly allocated. Otherwise, the thread can reuse the local from the entry's previous thread.

After the thread entry is initialized, the current index of the thread (as taken from the family's *start* field), is written into the first local register of the thread, if the thread has one or more local registers.

Finally, the thread is ready to be executed and its index is put on a thread ready list for the thread scheduler (see section **??**).

Chapter 8

Registers

Each thread has access to a section of the register file, which is visible to the thread through the register specifiers in its instructions. This view of the register file is called the thread's *logical* context, which is unrelated to the physical layout of the registers in the register file. The logical register context of a thread is separated into (up to four) different sections, each with different properties:

- Globals. The contents of these registers are provided by the parent thread of the family that the current thread belongs to. They contain read-only information which is independent from the index of the thread in the family and does not get modified during execution of the family. For instance, these registers can contain pointers to data structures, loop-invariants such a complex expressions, and so on.
- Dependents. These registers map to the "shared" registers of the previous thread in the family's index sequence. See the next item for more details.
- Shareds. These registers are available for reading by the next thread in the family's index sequence via its "dependent" registers. Combined, these two registers classes create a private, one-way communication channel between two adjacent threads in a family. Since such a channel exists between every adjacent pair of threads in the family, this creates a chain of possible efficient communication through the entire family without having to resort to memory for communicating small data between these threads.
- Locals. These registers are available for private use to the current thread and that thread alone. No other thread is able to read or write these registers at any time.

8.1 Virtual layout

The compiler determines the size of these four regions (with the shareds and dependents equally sized) based on the thread code, with the constraint that all these sections must fit in the original ISA's register context[1]. As a result, the compiler simply uses these section by using the correct register specifier from the base ISA. Figure **??** shows this mapping. Note that any or all of these sections can be of zero size.

8.2 Physical layout

The virtual registers as described in the previous section are mapped onto the core's physical register file. This register file contains the contexts of many threads. During the family creation process on a core, a section of contiguous registers are allocated to accomodate N threads (see section **??** for details and how N is chosen). This single block of registers is divided among the threads of that family on that core. However, this division is non-trivial. Trivially, by their nature, the global registers for every thread are mapped onto the same physical registers. Mapping the local registers is similary trivial, where a simple offset can be used to obtain a thread's private space in the block of allocated registers.

However, if the family's threads contain shareds and dependents, the dependents of thread $i+1$ have to be mapped to the shareds of thread i, with exceptions for the first and last thread in the family. The dependents of the first thread in a family have no shareds to be mapped onto, so they have their own section of registers. The parent thread can then fill these registers with a special register-move instruction; this mechanism is used to initialize the shareds-dependents chain through the family. Similarly, the shareds of last thread in a family

[1]For instance, on a traditional RISC architecture with 5-bit register specifiers with a single read-as-zero register, the sum of the sizes of these sections cannot exceed 31.

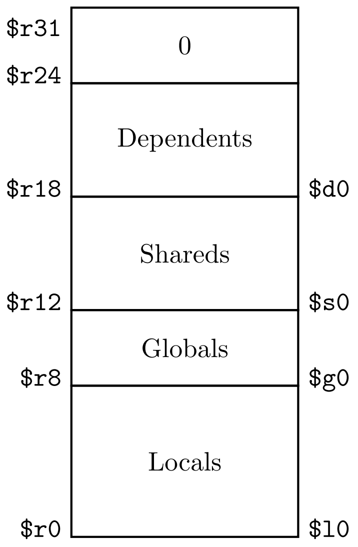

Figure 8.1: Virtual layout of a thread's integer registers with 8 locals, 4 globals and 6 shareds and dependents. On the left are the base ISA's register specifier. On the right are their alternative specifiers which indicate the sections. Note that only 24 of the 32 registers are used; `$r24`–`$r31` are read-as-zero. In this example the base ISA is from the DEC Alpha.

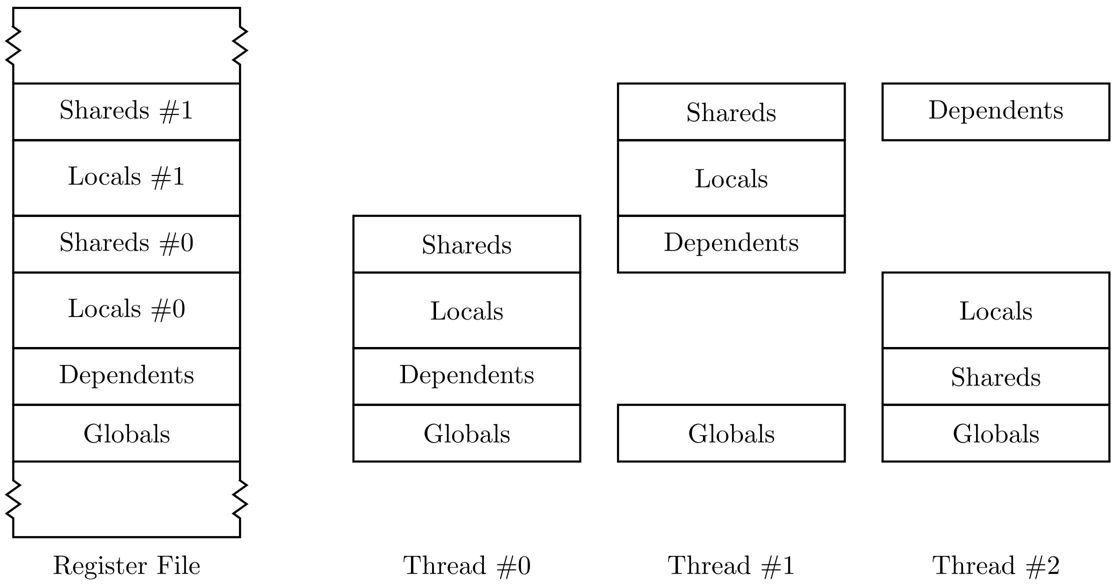

Figure 8.2: Phsyical and virtual layout of the integer registers of a family's threads. On the left is the physical register file with allocated registers. On the right are the first three threads in the family and their virtual register sections.

have no dependents mapped onto them. Instead, the parent thread uses a register-move instruction to copy their contents back to its own context, thus retrieving the "output" value of the shareds-dependents chain.

Since threads in a family are allocated in-order, the shareds-dependents chain can be constructed by mainting a "last allocated" pointer in the family table that points to the shareds of the last allocated thread of that family. Coupled with this, every thread has a pointer in the thread table that points to its dependents, which the pipeline can use to decode the virtual register specifier into a physical register specifier. In this case, thread creation involves copying the "last allocated" pointer from the family table to the newly created thread's dependent pointer and setting the "last allocated" pointer to this thread's shareds.

Figure **??** shows an example of this mapping of the virtual contexts of threads in a family onto the physical register file.

Note that the shareds of threads 2 overlap with the dependents of thread 0. One might assume that if thread 2 writes to its shareds, it will overwrite the dependents of thread 0. However, because there's only enough space in the allocated registers for two threads, at most two threads will exist at any time. As a result, if thread 2 exists, thread 0 must have terminated, freeing its locals and dependents for reuse.

8.2.1 Family Creation

When a family is created (see section **??**), the program can optionally specify a *block size*. This specifies a distribution size for a group family across multiple cores *and* an upper limit on the number of threads allocated at any time to this family, per core. The latter meaning of this value is of relevance to the allocation of registers. During family creation, the core attempts to allocate this many thread's worth of registers. If this cannot succeed, fewer and fewer threads worth of register will try to be allocated until a block of registers can be allocated which can accomodate N registers. This value is known as the *physical* block size, whereas the originally specified value is the *virtual* block size. The physical block size now determines how many threads can, at most, be allocated on this core.

After the core has finished allocating resources, including its register block, for the family, the family is moved to the *thread allocation list*, where another process will start allocating threads from it.

8.2.2 Thread Creation

A process on each core is responsible for taking allocated families and allocating threads to that family until the number of threads allocated to that family equals the family's physical block size (see section **??**). However, to avoid complicated allocation logic, when a thread can be cleaned up and its family still has threads in its index sequence that have not yet been creeated, the thread's entry is reused for the next thread in the family. This operation is performed by the same process. Thus, this process has two inputs: the family at the head of the thread allocation list, and the thread cleanup queue. The thread creation process is responsible for setting up the pointers in the thread table to the register file where the thread's locals, dependents and shareds are located (the address to the globals is thread independent and are stored in the family table). But, this operation dependents on whether the new thread uses a new entry, or reuses an old one.

When a new thread is created at an unused thread table entry, its register addresses must be initialized. The family maintains a counter, `thread_count`, which indicates how many physical threads have been allocated to this family. This counter starts at zero, is incremented with every newly allocated thread and will never be more than the number of allocated register contexts. This counter can be used to find the next context in the allocated registers which is still untouched. This context's offset is determined by skipping the base globals and dependents and `thread_count` worth of contexts (locals and shareds), which have already been allocated to existing threads. The new thread's shareds lie in this block as well, above the locals.

When a thread entry is reused for a new thread, the new thread can reuse the old thread's locals for its own. The only offsets that need to change are for the thread's dependents and shareds. The old thread's dependents are no longer in use and can be reused for the new thread's shareds, a simple copy from one field in the thread table to the other. The new thread's dependents are the last allocated thread's shareds.

Note in both cases, the new thread's dependents are the last allocated thread's shareds, which involves a simple copy from one field in the family table to the thread table.

Chapter 9

Thread Scheduler

The thread scheduler is the component in the core that handles thread wakeup requests and coordinates with the instruction cache to construct a list of threads that the pipeline can use whenever it performs a thread switch. This list of *active* threads is guaranteed to have the cache line with their next instruction available in the instruction cache, thus avoiding stalling the pipeline when it switches to them.

9.1 Overview

Figure **??** shows a conceptual layout of the thread scheduler. The thread scheduler's only process, P_a, reads a thread from the *ready list* (RL). This list hold threads that should be run by the pipeline, but are not guaranteed to have their cache-line present in the instruction cache. The scheduler then issues a fetch to the instruction cache based on the thread's program counter. If this fetch hits the cache, the scheduler appends the thread onto the *active list* immediately. Otherwise, when the fetch does not hit a cache entry with the data, the cache will append the thread to the cache entry's *waiting list* and allocate an entry and dispatch a request for the cache-line, if necessary. When the data returns from memory, the instruction cache will append the returned entry's list of waiting threads onto the active list.

When the pipeline performs a thread switch, it removes the first thread from the active list and fetches its information from the thread and family table, reads its instruction data from the instruction cache and commences execution of the thread. After a thread switch, the last instruction of the thread continues through the pipeline after which it will be added onto the ready list if the pipeline determined it can be executed again (e.g. on a branch).

9.2 Linked Lists

In two situations, it's highly desirable to move multiple threads from one list to another in a single cycle. These situations are as follows:

- When a global register (which is visible to multiple threads in a family) has not yet been written, all threads in the family on that core can suspend on the register. This necessitates that the architecture supports a (potentionally large) list of threads waiting on a register. When the register is written, all of these threads must be moved to the ready list in order to finally enter the pipeline and resume execution.

- When a set of threads on the ready list want to move to the active list, but their cache-line is either not present, or does not yet contain data. In this case, the thread scheduler should not stall; it should be able to continue servicing other threads, and move them to the active queue so the pipeline can execute them while the first thread's cache line is being loaded. Therefore, every cache line can have one or more threads waiting on its load completion. When the cache-line is finally loaded, all these threads should be moved to the active list.

Since FIFO buffers cannot support moving arbitrary number of items from one to another, these two crucial use cases are supported by managing threads in these situations on a linked list. Every thread contains a `next` field in its entry in the thread table and every non-full register and non-full instruction cache-line has a head and tail pointer into the list of threads that are suspended on it. Such a linked list of threads can then be appended in its entirety to any other linked list in a single cycle. This requires that the ready list and active list are linked lists as well.

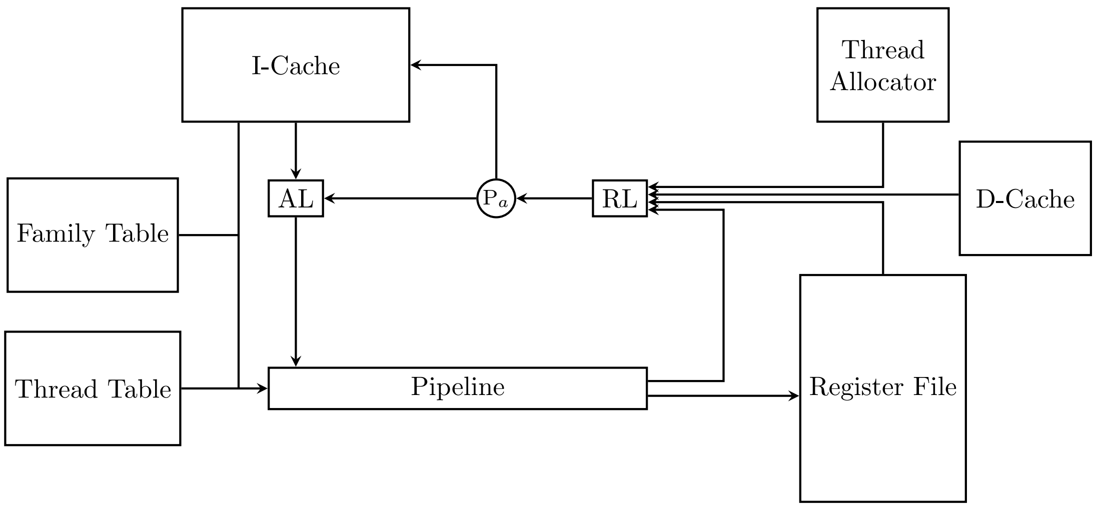

Figure 9.1: Conceptual overview of the thread scheduler. Shown is the ready list (RL), the active list (AL), the activation process (P_a) and the various components that supply thread information and schedule events.

9.2.1 Fairness

In the simplest linked list implementation, appended items are put on the front of the list. For the active and ready list this creates a problem in terms of fairness. A few threads could continously exist at the front of the lists, starving the threads behind it. As such, for these lists, threads should be appended to the *back* of the list, creating a FIFO list. This requires both a head and tail pointer for the list.

While the same could be said of the register waiting lists and the instruction cache waiting list, fairness there is actually not as important since there is no risk of starvation. As a consequence, these lists can be optimized to be FILO linked lists, reducing the number of write ports (see next sections).

9.3 Ready List

The ready list is the first list that threads enter when they are rescheduled (i.e., when they need to be run by the pipeline). The following situations can (re)schedule a thread onto a ready list:

- Thread creation. After a thread is created, it can immediately commence execution and is appended to the ready list.
- Write barrier completion. When a thread issues a write barrier and it still has pending (unacknowledged) writes, it sets a flag in its thread table entry indicating that it has suspended on the acknowledgement of its outstanding writes. When the data cache receives the last acknowledgement, and this flag is set, it will append the thread to the ready list.
- Register write. When a thread reads a register that is not full, it will append its ID onto the register's waiting list of threads, thus 'suspending' on it. When a write to this register fills it, this list of suspended threads is appended to the ready list.
- Thread reschedule. When the pipeline switches threads but the thread can continue with the next instruction (e.g., on a branch or crossing a cache-line boundary), the thread is appended to the ready list.

These four events could all occur at the same cycle, and since they would all access a shared state, arbitration on the ready list is required. However, this can be costly in terms of hardware or performance. A stall of either of these processes could have significant effects. Therefore, it might be worth giving some or all of these inputs their own list. Figure **??** shows two of these possible implementations of the ready list. When more than one ready list exists, the scheduler's activiation process chooses which list to activate the next thread from. For this, it uses round-robin priorization in order to avoid starving either of the lists. The tradeoff in choosing which implementation to use lies in the cost of additional hardware versus the benefit of avoiding stalls due to arbitration.

Note that only the register file can reschedule more than one thread in a single cycle. Thus, whichever ready list the register file is connected to, should be a linked list. The others can be linked lists (which would increase

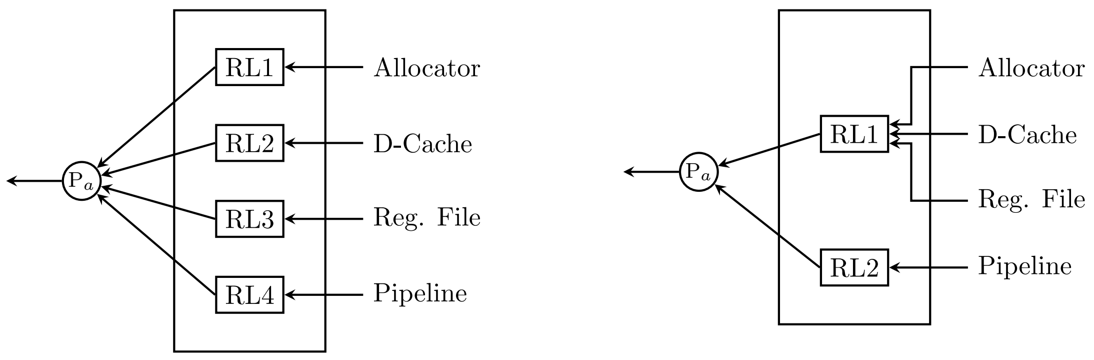

Figure 9.2: Some alternative implementations for the ready list. Left: every input has its own list. Right: only the pipeline has its own list.

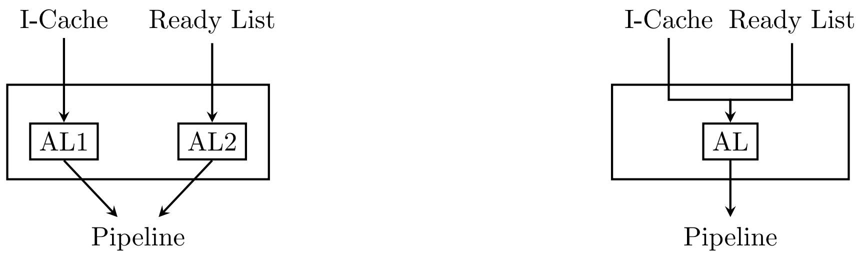

Figure 9.3: The two alternative implementations for the active list. Left: both inputs have their own list. Right: both inputs share a list.

the number of ports on the thread table's `next` field), or a simple FIFO buffer which is bounded by the number of entries in the thread table.

9.4 Active List

The active list is the second list that threads enter when they are rescheduled. After a thread has been chosen from the ready list, it will be put on the active list if (or when) the instruction cache contains the thread's cache line. Thus, there are two events that can put threads on the active list:

- Reschedule with I-Cache hit. When a thread is chosen for activation from the ready list and the I-Cache lookup results in a hit, it can immediately be put on the active list.
- Cache-line load completion. If the previous case results in a miss (i.e. the line and/or data is not present), the thread suspends on the cache-line. When the cache-line returns, all threads suspended on that cache-line are put on the active list.

Similarly to the ready list, these events could all occur at the same cycle, and need either arbitration or an individual list. Figure **??** shows the two possible implementations of the active list. Since only the instruction cache file can activate more than one thread in a single cycle, its list should be a linked list. For the other list, it can either reuse the same `next` field in the thread table at the cost of additional ports on this field, or use a dedicated FIFO buffer.

To avoid that subsequent activations from the ready list evict a cache-line that is required by a thread in the active list (which would stall the pipeline when it switches to that thread), cache-lines needed by threads in the active list and pipeline are locked and cannot be considered for eviction. This is explained in detail in section **??**.

9.5 Ports

The potentially many different linked lists of threads could create a strain on the `next` field in the thread table. To analyze the number of ports required on this field, we must first define the possible operations that add or remove threads from these lists:

1. Thread creation. An empty thread is allocated, initialized and put onto the ready list.
2. Register read. A thread must suspend on an empty register and is added to the register's *waiting* list.
3. Register write. The linked list of threads suspended on the register is appended onto the ready list.
4. Thread activation. A thread is popped from a ready list, hits the I-Cache and is appended onto the active list.
5. I-Cache miss. A thread is popped from a ready list, misses the I-Cache and is appended onto the line's *waiting* list.
6. I-Cache line completion. The linked list of threads suspended on the cache-line is appended onto the active list.
7. Thread switch. The pipeline pops the first thread off the active list for execution.
8. Thread reschedule. The pipeline pushes a thread onto the ready list for rescheduling.
9. Write barrier completion. The D-Cache pushes a thread that was waiting for write completion onto the ready list.

Note that all appends to a single list are arbitrated due to the access to the shared tail pointer of the list. Thus, one write port is required per concurrently-written list. Depending on the implementation of the ready and active lists as described in the previous sections, this could result in 4–8 write ports: 1–2 for the active list, 1–4 for the ready list, one for appending a thread to a register's waiting list and one for appending a thread to a cache-line's waiting list.

Similarly, only one process is responsible for removing items from a list, requiring only one read port per concurrently-read list. This results in 2 read ports. This number is independent from the implementation of the lists, since the alternative lists are never read at the same time and can thus share a read port on the `next` field. Likewise, the two waiting lists do not need a read port on the `next` field because they never have a single entry removed, but are appended in their entirety to another list, which does not require accessing their `next` fields. Thus, one read port is required for removing a single thread from the active list(s) and one for the removing a single thread from the ready list(s).

However, the queuing of threads assumes that the list of threads is properly terminated (i.e., the last thread's `next` field is "invalid"). If this cannot be guaranteed in any case, an additional write port is required per list to set this field. But, if we append threads to the front of those lists where there are no fairness constraints (see section **??**), these lists are guaranteed to be a properly terminated list. Applying this to all relevant lists, operations 1, 2, 3, 5, 6 and 9 in the list above do not require an explicit termination of the list. The scenarios 4 and 8 do, resulting in two additional write ports, bringing the total up to 6–10 write ports and 2 read ports.

9.5.1 Splitting

To reduce the number of ports, this single table with the `next` field can be split into different tables, each with fewer ports. This process of splitting the table is subject to the constraint that multi-thread appends can only occur between lists which use the same table for their `next` field. Since there are only two situations where a multi-thread append operation occurs (cache line completion for the active list and register write for the ready list), there can be 3 different tables for the `next` field:

- one for the active list and the cache lines' waiting lists, with 3–4 write ports and 1 read port.
- one for RL2 and the registers' waiting lists, with 2 write ports and 1 read port.
- one for RL1, with 2 write ports and 1 read port.

Although there is one read port more (because RL1 and RL2 are now split, they cannot share a read port), this division might prove better in terms of access times, area or power usage, depending on the production process.

Chapter 10

Instruction Cache

10.1 Overview

The instruction cache is an N-way set associative lock-up free [?] cache which uses the thread table entries as the miss information/status holding registers (MSHR). This allows efficient management of concurrent threads that require an instruction fetch without blocking the pipeline or cache.

Figure ?? shows a conceptual layout of the instruction cache. It receives instruction fetch operations from the core's thread scheduler. It can either return the data right away, which causes the thread to be scheduled onto the active queue right away, or buffer a request to the external memory system. This requests in this buffer are put on the memory bus by a hardware process. Replies and messages from other caches (currently, only writes are snooped) are received and handled. Read responses write the data into the cache immediately. The responses (minus the data) are buffered and are handled by a hardware process which wakes up the line's suspended threads.

10.2 Cache-line contents

Table ?? lists the fields in each entry in the instruction-cache. It is essentially a traditional cache-line extended with extra fields. The *tag* field is used as usual, to find the wanted cache-line in a set. The *state* field can be one of the following values:

`Empty` This state indicates that the entry is not being used.

`Loading` This state indicates that a read request has been sent to the memory and has no data.

`Invalid` This state indicates that the entry has threads that are suspended on the line, but it has been invalidated by the memory hierarchy. It cannot be cleared until the pending threads have run. To avoid a situation where the line is forever kept present due to continous instruction fetches, such fetches must stall when hitting an `Invalid` line.

`Full` This state indicates that the entry contains all data for a cache-line. Instruction fetches to this line do not need to suspend threads.

Figure ?? illustrates the states and their changes for a cache-line entry.

Name	Purpose	Bits
state	State of the line	2
tag	The address tag of the line stored in this entry	...
data	The data of the line	$8 \cdot$ Cache-line size
access	Last access time of this line	...
waiting	Head and tail pointer of the list of suspended threads	$2 \cdot \lceil log_2(\#\text{Threads}) \rceil$
creation	Is the core's create process suspended on this line?	1
references	Number of threads that need to use this cache-line.	$\lceil log_2(\#\text{Threads}+1) \rceil$

Table 10.1: Contents of an instruction-cache line

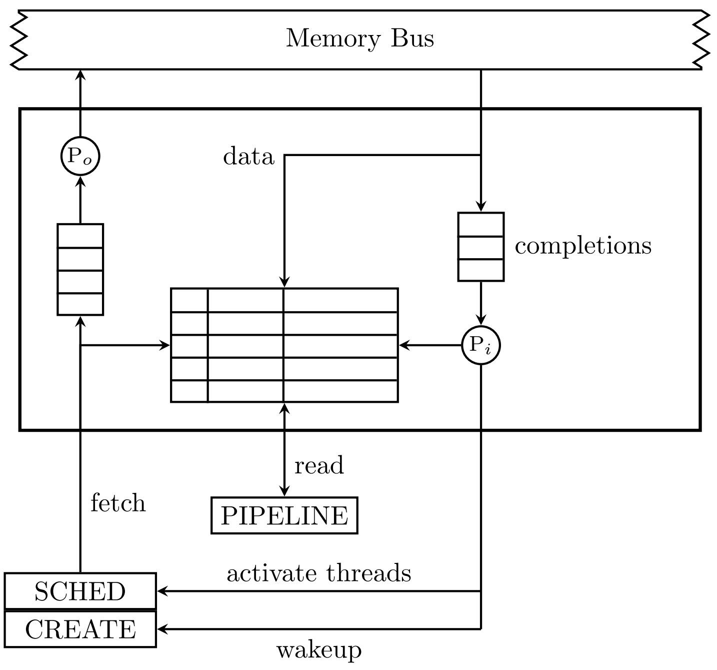

Figure 10.1: Overview of the instruction cache. Shown are the cache entries, buffers to and from memory, the hardware processes that handle the buffer items (P_i and P_o) and the connections to the core's thread scheduler (SCHED), pipeline and family creation process (CRE).

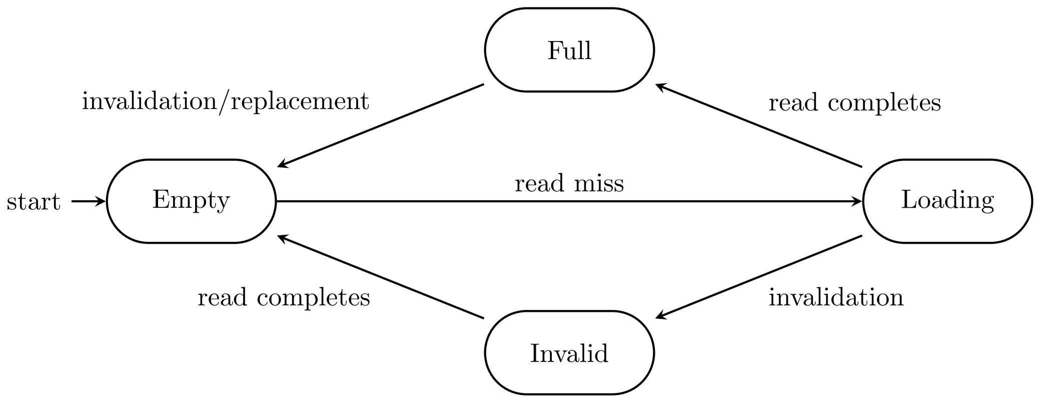

Figure 10.2: State transitions for an instruction-cache entry

10.2.1 Reference Counter

The instruction cache is used to activate threads by checking if they have their cache-line available. However, once a thread has passed the check and its cache-line is in the cache, the cache-line should not be evicted until the thread has been executed by the pipeline, and no longer needs the cache-line. To prevent this eviction, each cache entry has a reference counter that indicates how many threads still need that cache-line to be present in the cache. Lines can only be evicted if this counter is zero. The counter is incremented during thread activation when the cache is checked and decremented when the pipeline switches to another thread (the pipeline's current thread is used to index the cache).

This could lead to the situation that all entries in the cache are locked. In such an event, no cache line can be evicted and whatever requests caused the eviction would have to stall or retry at a later time.

10.2.2 Creation flag

When a family is created, the create process on the core (see section **??**) requires the cache-line containing the entry point of the thread in order to allocate registers for the family. Since this cache-line also contains instructions, this cache-line is fetched through the instruction cache. The `creation` flag in each cache entry indicates that the core's create process is waiting for that line to be loaded. When this finally happens, the cache will send a signal to the core's create process, along with the cache-line data, indicating that the process can continue.

Note that as an alternative implementation, since the create process can only suspend on one cache-line at a time, this per-entry flag could be replaced by a single register containing the index of cache-line that the create process is waiting on.

10.3 Bus messages

This section describes the messages that can occur on the memory bus and how the instruction cache reacts to them.

10.3.1 Write

When the data cache snoops a write on the bus made by another cache it writes the data into its own copy of the line (and marks it as valid), if it has it. No care has to be taken in order to guarantee proper ordering of the write with respect to locally issued reads and writes because the microgrid's memory consistency model allows a non-deterministic result between these requests.

10.3.2 Read Response

If the higher memory returns a cache-line for a read, all caches on the bus use this response to store the data in their copy of the cache-line, if they have it. The entire cache-line is updated. Note that writes snooped while the line was loading will get overwritten, which is a valid outcome according to the microgrid's memory consistency model.

10.3.3 Invalidate

Depending on the higher memory's implementation, it can be required that all copies of the cache-line in the L1 caches should be invalidated. When this message is observed on the bus, and if it has the relevant line, the instruction cache will immediately clear the line if its state is Full. If the line's state is Loading, it will be set to Invalid, which will be set to Empty when the instruction-cache has woken up all threads suspended on that line.

10.4 Processes

This section describes the hardware processes in the instruction cache and what their responsibilities are.

10.4.1 Outgoing requests

All outgoing reads (i.e., those that miss the cache) are buffered to cope with stalls in the higher memory and contention on the bus. A hardware process is responsible for putting the requests from this buffer onto the memory bus.

10.4.2 Completed reads

Read responses observed on the bus are merged in their cache entry immediately (they could be buffered first, but since there is data attached, significantly more space would be required). The index of this entry is then pushed on the “returned reads” queue. A hardware process handles these lines one by one. It takes the entry from the head of the queue, wakes up suspended threads from this entry’s *waiting* list, optionally wakes up the creation process, and pops the entry’s index from the queue.

Chapter 11

Data Cache

11.1 Overview

The data cache is an N-way set associative lock-up free [?] cache which uses the destination register as the miss information/status holding registers (MSHR) instead of defining a separate and dedicated register file. This allows an outstanding memory read to every register without blocking the pipeline or cache.

Figure **??** shows a conceptual layout of the data cache. It receives read and write operations from the memory stage of the pipeline. It can either return the data right away, or buffer a request to the external memory system. This requests in this buffer are put on the memory bus by a hardware process. Replies and messages from other caches (currently, only writes are snooped) are received and handled. Read responses write the data into the cache immediately. The responses (minus the data) are buffered into separate queues for reads and writes. The read-response queue is handled by a hardware process which completes any pending writes of the returned cache-lines. The write-acknowledgement queue is handled by another hardware process which sends the acknowledgement on to the relevant parts of the core.

11.2 Cache-line contents

Table **??** lists the fields in each entry in the data-cache. It is essentially a traditional cache-line extended with extra fields. The *tag* field is used as usual, to find the wanted cache-line in a set. The *state* field can be one of the following values:

`Empty` This state indicates that the entry is not being used.

`Loading` This state indicates that a read request has been sent to the memory. Note that the entry may or may not have data in it. Writes from the pipeline and writes snooped from the bus may result in data being written to the cache-line. The bytes that have valid data is indicated by the entry's *valid* field and can be used to satisfy subsequent memory loads requests before the cache-line has been fetched from memory.

`Invalid` This state indicates that the entry has registers that still need to be processed, but it has been invalidated by the memory hierarchy. It cannot be cleared until the pending read requests have been processed. Although the line can still be used to service new local read requests, it should not be used to service local writes. If a write to the D-Cache hits an `Invalid` (or `Loading`) line, the write must stall, for the following reasons:

Name	Purpose	Bits
state	State of the line	2
tag	The address tag of the line stored in this entry	...
data	The data of the line	$8 \cdot$ Cache-line size
valid	Byte-granular bitmask of valid data	Cache-line size / 8
access	Last access time of this line	...
waiting	The first register in the list of pending registers	$\lceil \log_2(\#\text{Registers}) \rceil$
processing	Processing waiting queue?	1

Table 11.1: Contents of a Data-cache line

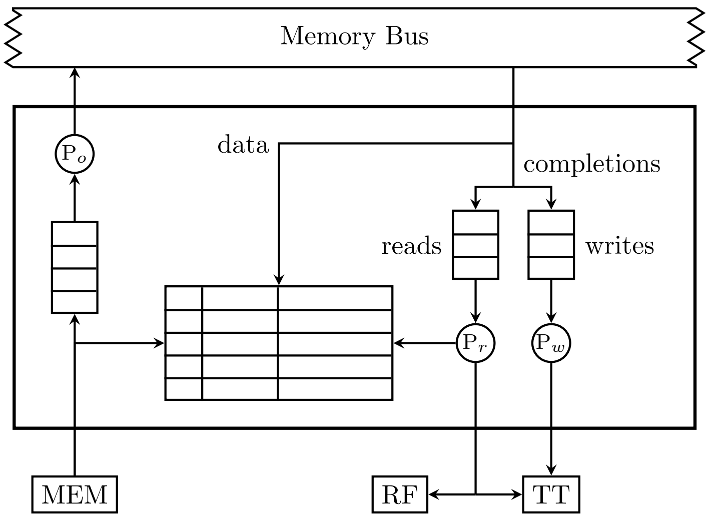

Figure 11.1: Overview of the data cache. Shown are the cache entries, buffers to and from memory, the hardware processes that handle the buffer items (P_r, P_w and P_o) and the connections to the Memory stage in the pipeline (MEM), Register File (RF) and Thread Table (TT).

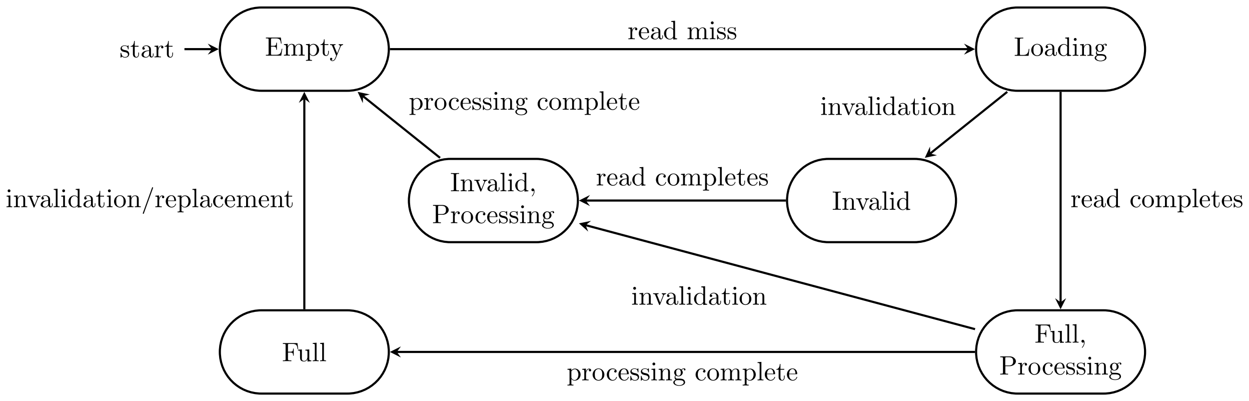

Figure 11.2: State transitions for a data cache entry

- A write into a `Loading` (or `Invalid`) entry might violate the sequential semantics of a single thread because pending reads might get the later write's data (WAR hazard). For instance, a read to location A misses the cache, allocates an entry and sets the state to Loading. A subsequent write (from the same thread) to A hits the cache and writes its data into the entry. The read completes and processes the pending reads. The register now gets the data from *after* the write, instead of before, as the thread would expect.

- The write cannot simply write-through while ignoring the entry because subsequent reads should get the new data and not the old, so the entry must be updated.

- The entry cannot be invalidated and a new line allocated, because read responses from the memory bus hit cache entries on address, and then there would be multiple valid lines of the same address. Read responses cannot identify cache lines by index, since this would put an unnecessary and undesirable strain on the design the higher memory architecture, where it would have to track cache-line indices of reads, which are most likely different for each cache on the memory bus.

`Full` This state indicates that the entry contains all data for a cache-line and no read requests are pending. The data in the entry can be used for reads and writes.

Figure **??** illustrates the states and their changes for a cache-line entry.

Name	Purpose	Bits
fid	Index of family whose thread issued the load	$\lceil \log_2(\#\text{Families}) \rceil$
ofs	The byte-offset in the cache-line to read from	$\lceil \log_2(\text{Cache-line size}) \rceil$
size	The number of consecutive bytes to read	$\lceil \log_2(\text{Register size}) \rceil$
sext	Sign-extend the result?	1
next	The next register in the linked list	$\lceil \log_2(\#\text{Registers}) \rceil$

Table 11.2: Contents of the MSHR part of registers

11.3 Registers

When a memory load from the pipeline results in a miss, the cache-line stores the destination register in the cache line's *waiting* field and returns the field's previous value to the pipeline. The pipeline will mark the destination register as *Pending* and fill out the MSHR part of the register. These fields, as listed in table **??** store all the information about the memory load which will be used by the data cache after the line has been fetched from memory. The cache line's previous value of the *waiting* field is stored in the register's *next* field. This mechanism effectively creates a linked list of pending registers, where each memory load that misses the cache attaches its destination register to the front of this list.

11.4 Timing issues

There is a possibility that the cache line is fetched from memory before the memory load instruction that caused the miss has propagated through the writeback stage. When the data cache tries to read the *next* field from the register identifies by the line's *waiting* field, it will contain invalid data. To counter this, the data-cache, when reading this register, must stall if the register is not in the *Pending* state, or without correct information in the MSHR fields (e.g. the *size* field is 0).

11.5 Bus messages

This section describes the messages that can occur on the memory bus and how the data cache reacts to them.

11.5.1 Write

When the data cache snoops a write on the bus made by another cache it writes the data into its own copy of the line (and marks it as valid), if it has it. This will ensure cache consistency between the caches sharing a single bus. It does not matter if the snooped write overwrite a local write, even when the entry is in the Loading state. The microthreading's memory consistency model states that independent writes made by different thread (and thus, since threads do not migrate, different cores) have non-deterministic consistency.

11.5.2 Read Response

If the higher memory returns a cache-line for a read, all caches on the bus use this response to store the data in their copy of the cache-line, if they have it. Only the invalid parts of the entry are updated with the data returned from memory. This guarantees that writes made after the read will not get overwritten by the read response. Note that because every cache uses the response, a cache may see a read response on the bus for a cache-line that is in the Full state. In such a case, the response is ignored.

11.5.3 Invalidate

Depending on the higher memory's implementation, it can be required that all copies of the cache-line in the L1 caches should be invalidated. When this message is observed on the bus, and if it has the relevant line, the data cache will immediately clear the line if its state is Full. If the line's state is Loading, it will be set to Invalid, which will be set to Empty when the data-cache has finished processing all pending requests from that line.

11.6 Processes

This section describes the hardware processes in the data cache and what their responsibilities are.

11.6.1 Outgoing requests

All outgoing reads (i.e., those that miss the cache) and writes (i.e., all writes, because the data cache is write-through) are buffered to cope with stalls in the higher memory and contention on the bus. A hardware process is responsible for putting the requests from this buffer onto the memory bus.

11.6.2 Completed writes

Write completion notifications observed on the bus are buffered in the "write completion buffer" so that the memory bus is not unnecessarily occupied when the notification cannot be handled immediately by the core. A hardware process removes these notifications from the queue and passes them on to the core, which involves decrementing a counter in the thread table and acting when it's reached zero. This can be:

- waking up the thread if its thread table entry indicates it is waiting on a memory write barrier.
- cleaning up the thread if all other dependencies have already been satisfied.

11.6.3 Completed reads

Read responses observed on the bus are merged in their cache entry immediately (they could be buffered first, but since there is data attached, significantly more space would be required). The index of this entry is then pushed on the "returned reads" queue. A hardware process handles these lines one by one. It takes the entry from the head of the queue and resolves pending reads from this entry's *waiting* list of registers. When all registers have been handled and the list is empty, the entry's index is popped from the queue.

Chapter 12

Memory

The execution model (see section **??**) provides a single, flat memory space to programs with weak consistency. This memory model is currently implemented in the microgrid via a memory interface on each core to a chip-wide memory network that provides access to the single, flat address space. The current memory implementations provide global consistency (i.e., every write is guaranteed to be visible to every core at some point in the future). This model has been chosen because the current weak memory consistency model in SVP is more costly to implement in a chip-wide memory network.

12.1 Core Interface

Every core on a microgrid has a single interface to memory. The core's instruction and data caches are connected, via a bus, possibly with caches from other cores, to the memory system. The following messages are possible on the bus:

12.1.1 Read Request

When a cache needs to read data from memory, it sends out a read request to the memory system. This request is for an entire cache-line. The request simply contains the line-aligned address of the desired cache-line. Eventually, the memory system will respond with a read response, as described below. This message is usually not relevant to the other caches on the bus.

12.1.2 Read Response

If memory returns a cache-line for a read request, all caches on the bus use this response to store the data in their copy of the cache-line, if they have an entry allocated to that line. Only the invalid parts of the entry are updated with the data returned from memory. This guarantees that writes made after the read request will not get overwritten by the read response. Note that because every cache uses the response, a cache may see a read response on the bus for a cache-line that it already has the data for. In such a case, the response is ignored. Therefore, there is no strict coupling between requests and responses. If N caches on a single bus issue a request, anywhere from 1 to N responses may be returned, in any order.

12.1.3 Write Request

When a core issues a memory write, this write is written through or around the data cache to the memory bus. The request contains the byte-aligned address of the data to write, the data to write, and a tag consisting of the thread table index of the thread that made the write. Every write request is matched with a write acknowledgement (see below) that is returned by the memory system when a write has been made consistent with the rest of the system. This acknowledgement contains the same tag that was present in the write request. This mechanism allows the core to keep track of pending writes for every thread, and thus family. This feature is required because the programming model's consistency model dictates several things:

- a family cannot be considered 'complete' until all its thread's memory writes have been made consistent.
- a family create is an implicit memory write barrier, i.e., all preceding writes of the thread must be consistent in memory.
- a thread may want to issue a memory write barrier before writing a pointer to a shared to guarantee that writes to the pointed-to memory are consistent before the next thread in the family receives the pointer.

When the data cache snoops a write request on the bus made by another cache it writes the data into its own copy of the line (and marks it as valid), if it has it. This will ensure cache consistency between the caches sharing a single bus. It does not matter if the snooped write overwrite a local write, even when the entry is still being loaded. SVP's memory consistency model states that independent writes made by different thread (and thus, since threads do not migrate, different cores) have non-deterministic consistency.

12.1.4 Write Acknowledgement

Write acknowledgements are sent by the memory system in response to write requests at the time when the write request is guaranteed to be consistent for all cores in the microgrid within the rules of SVP. Write acknowledgements are targeted to the core that the corresponding write request originated from and tagged with the same tag from the write request. Upon receiving a write acknowledgement, the core can update its internal administration for the thread and family specified in the tag.

Note that, since write acknowledgements are meant for one core only, the other caches on the bus must ignore acknowledgements not meant for them.

12.1.5 Invalidate

Depending on the memory system's implementation, it can be required that all copies of the cache-line in the caches should be invalidated. If a cache observes this message on the bus, and if it has the relevant line, the cache will immediately clear the line. If, however, the line is still being loaded, the cache must mark the line as invalid, which will cause the line to be cleared when the data has returned and the cache has finished processing all pending requests for that line.

12.2 COMA

The currently implemented memory system for the Microgrid is a Cache-Only Memory Architecture, or COMA, memory system. It consists of caches (henceforth called L2 caches, considering the caches in the core are L1 caches) which are connected in a hierarchy of unidirectional rings to one or more external memory chips. Figure **??** shows the conceptual layout of the COMA memory system. The design of the memory system is essentially one big ring of all caches. However, since data access tends to be localized in the microgrid, it's quite feasible that large sections of caches will never have certain cache-lines, the ring is shortcut across several points (creating the hierarchy of rings) with directories to allow messages bypass entire groups of caches that do not have the cache-line that the message refers to.

12.2.1 Protocol

The COMA protocol is a token protocol. This means that every copy of the cache-line in the system, whether in transit, or in a cache, has a certain number of tokens associated with it. Based on this, there are several basic rules that govern the COMA protocol:

- When a cache-line is introduced into the COMA system (i.e., read from external system), it is given a number of tokens equal to the number of caches in the system.
- A cache-line must always have at least one token.
- In order to service a read from a core, a cache must have a copy of the cache-line with at least one token.
- In order to acknowledge a write from a core, a cache must have the only copy of the cache-line (with all the tokens).

These rules guarantee a correct consistency operation in the memory network. The protocol itself implements these rules.

Messages

The protocol is implemented via messages on the ring network. The contents of a message is shown in table **??** and the protocol consists of the following message types:

- **Read Request**. (`RR`) This message signals a request by a cache for a copy of the specified cache-line. The request has neither data nor tokens attached to it. The request will keep going around the ring until it has acquired both.

Name	Purpose	Bits
type	Type of this message	3
address	Byte-aligned address of the cache-line this message	N
sender	ID of the cache that sent this message	$\lceil \log_2(\#\text{Caches}) \rceil$
ignore	Should this message be ignored?	1
data	Data of the line	512
tokens	Number of tokens held by this line copy	$\lceil \log_2(\#\text{Caches}) \rceil$
dirty	Has this copy of the line been modified?	1
client	Client on the sender's bus that caused the message	N
tid	ID of the thread on the client that caused the message	$\lceil \log_2(\#\text{Threads}) \rceil$

Table 12.1: Contents of a message

Field	RR	RD	RDT	UP	EV
type	X	X	X	X	X
address	X	X	X	X	X
sender	X	X	X	X	X
ignore	X	X	X	X	X
data		X	X	X	X
tokens			X		X
dirty					X
client				X	
tid				X	

Table 12.2: Which message fields are used for which message types

- **Read Request, with Data**. (`RD`) When a read request passes by a cache or root directory and gets the data, its type is 'promoted' to a Read Request, with Data. It keeps going around the ring until it has acquired tokens as well.
- **Read Request, with Data and Tokens**. (`RDT`) When a read request with data passes by a cache or root directory and gets one or more tokens, its type is 'promoted' to a Read Request, with Data and Tokens. When this message arrives back at the sender cache, the read request is completed.
- **Update**. (`UP`) A write from a client below a cache will trigger an Update message to be sent around the ring once, updating every copy of the cache-line it comes across. When this message returns to its sender cache, it has updated every cache and the message is dropped.
- **Eviction**. (`EV`) When a read or write request from a client requires an entry in the cache, but none are available, an entry is chosen and evicted from the cache in an Eviction message. An eviction message will travel to the root directory, possibly injecting in other caches on the way if it can, to save the cost of writing the data back to memory or reading it from memory when required at a later time.

Table **??** shows which fields are used for which message types.

Deadlock avoidance

Special care is taken to guarantee that the COMA protocol is both deadlock-free and livelock-free. The first aspect of this is to guarantee that the network never stalls. If we look at a flat ring network with caches as nodes and memory clients attached to the caches, only memory clients can insert messages into the network. The network itself can only modify, forward, or remove messages. If we had a naive implementation where, at a node, either forwarding messages or inserting messages took priority, it would be possible for the ring network to stall because all buffers filled up and no message could be handled. There exists, in essence, a cyclic dependency that can cause this deadlock.

The solution to this, as proven by [**?**], is to ensure that every node has at least two buffer entries available and only accepts 'new' messages into the ring from the memory clients when the buffer has more than one entry free. This ensures that the network is never full and messages can always be forwarded.

However, a problem occurs due to the shortcuts provided by the directories. While, conceptually, the COMA network is one large ring, and the above buffering protocol can avoid deadlock on that single ring, a problem exists when considering these shortcuts. At a directory, a single output buffer can be filled by two input buffers. One from the 'upper' ring (the shortcut) and one from the 'lower' ring (the normal path). As a result, one buffer

Name	Purpose	Bits
state	Current state of the cache entry	2
tag	Tag part of the address of the line	N
data	Data of the line	512
access	(Psuedo-) last access time	N
tokens	Number of tokens held by this line copy	$\lceil \log_2(\#\text{Caches}) \rceil$
dirty	Has this copy of the line been modified?	1
updating	Number of unacknowledged update requests for this line	N
valid	Data validity bitmask	64

Table 12.3: Contents of a cache entry

has to have priority over the other when both want to output at the same time. If the lower ring's input buffer has priority, it's possible that deadlock occurs on the upper ring. Likewise, if the upper ring's input buffer has priority, it's possible that deadlock occurs on the lower ring.

To solve this issue, the COMA protocol provides deadlock-guaranteed fallback routing. If a message arrives at a directory from its upper ring, and the directory indicates that the message can continue on the upper ring (i.e., take the shortcut), and if the output buffer for that ring does not have at least 2 free entries available, the message is instead routed to the lower ring, with the `ignore` flag set. This flag will cause the message to be ignored by every cache in the ring until it reaches the directory again, where it is then forwarded onto the upper ring. In other words, if the shortcut cannot be taken, the message is sent the long-way round, which has guaranteed deadlock freedom as described above.

12.2.2 Cache

This section describes the caches in a COMA system and how the cache reacts on messages that it can receive. The cache is the leaf in COMA's hierarchy of rings. Only caches contain data (although data can also be in transit elsewhere in the memory network), and only caches interact with the memory clients, such as processing cores.

Structure

Table **??** lists the fields in a single cache entry. Note that, in order to accomodate all possible lines in the memory clients below the cache, the number of entries in the L2 caches must be the sum of the cache-lines in the clients below the cache. Also note that this is achieved by maintaining the same number of sets, and adding the associativities. For example, if an L2 cache is connected with four cores, each with two 1 kB 4-way set-associative L1 caches, then the L2 cache must be at least an 8 kB, 32-way set associative cache.

Bus Read

When a client on the bus below a cache issues a cache line read, a copy of the cache-line is returned immediately if the cache has such a copy with one or more tokens. If the cache has the line, but it is still being loaded, the request is dropped, as the load completion will send a read response onto the bus, also satisfying this read.

If the cache does not have the line, an entry is allocated and a Read Request send out on the ring network. If no free entries are available, a line is evicted (see section **??**) and freed. The request then stalls for one cycle, causing it to be handled succesfully next cycle. This is done. If no line can be evicted, the bus read request stalls.

Bus Write

When a client on the bus below a cache issues a memory write, an update message is sent out on the bus to update all other copies of the cache-line. This message is not sent if the cache has the only copy of the cache-line (i.e., with all tokens). In that case, the write is acknowledged immediately to the memory client on the bus that sent it. If the cache does not have a copy of the cache-line, it requests a copy of the cache-line by sending out a read request. It stores the written data in the cache-line, using the entry's dirty mask to mark which bytes have been written. When the read response arrives, the dirty mask is used to prevent those bytes from being overwritten by the returned data.

Read Request

If a cache receives this mesage, it will check if it has the line. If not, the message is forwarded as-is. Otherwise, if the cache has at least two tokens in its copy, it will copy the data from its entry into the message, give the message one of its tokens, upgrade the message to *Read Request, With Data and Tokens* and forward it so it can return to its sender. If the cache does not have enough tokens (i.e., just one token), it will only copy the data into the message and upgrade it to *Read Request, With Data*.

Read Request, with Data

If a cache receives this message, it will check if it has the line. If not, the message is forwarded as-is. Otherwise, if the cache has at least two tokens in its copy, it will copy the data from its entry into the message, give the message one of its tokens, upgrade the message to *Read Request, With Data and Tokens* and forward it so it can return to its sender. If the cache does not have enough tokens (i.e., just one token), the message if forwarded as-is.

Note that the data is copied along with the tokens, even though the message already has the data. This was done to simplify the implementation, such that this request type and a simple read request can share much of the same functionality.

Read Request, with Data and Tokens

If a cache receives this message, it will check the message's `sender` field to see if the message originated from (and thus, is meant for) itself. If not, the message is forwarded as-is. Otherwise, the cache will handle this message as a read response for an earlier read request. The data in the message is stored in the cache-line, but only for those bytes where the entry's `valid` mask is not set. This prevents the read response from overwriting writes made after the read request was sent. Then, pending bus writes of this cache (i.e., writes from memory clients that have not yet been processed by the cache) are merged with the data and this final cache-line is sent to the memory clients as a read response. The message itself is also destroyed.

Update

If a cache receives this message, it will check the message's `sender` field to see if it originated from itself. If so, it will use the message's `client` and `tid` fields to send a *write acknowledgement* to the specified memory client. The message itself is destroyed. If the update did not originate from the cache, and if the cache has the line, the data from the message is written (in essence merging the write) to the cache-line and put on the memory client's bus so they can snoop the write and update their caches accordingly. In either case, whether the cache has the line or not, the message is forwarded as-is.

Eviction

If a cache receives this message, it will check if it has the line and/or if it has a free entry for this line. If the cache does not have the line, but has a free entry, the line's data, token count and dirty flag are copied from the message into the new entry and the message is destroyed. If the cache does not have the line, and no free entry, the message is forwarded as-is. If the cache has the line, and it is not being loaded, the cache merges the evicted line with the entry by adding the tokens in the message to the tokens in the line and combining the dirty flag in the message (via a logical OR) with the dirty flag in the entry.

Note that if the eviction message is merged with the exiting cache entry, the data from the message can be ignored because any copy of the cache-line will be consistent due to the nature of the write-update part of the COMA protocol. Any write that updated the evicted entry will have also updated the entry with which it is merged.

12.2.3 Directory

This section describes the directories in a COMA system and how the directory reacts on messages that it can receive. Directories act as short-cut points on the conceptually single COMA ring. By having such short-cuts, entire sections of the ring can be skipped if the directory indicates that the ring below it does not contain a copy of the cache-line that a message is interested in. Directories do not contain data themselves, instead they only keep a light administration of the presence of cache-lines in the ring below it. Note that it is possible to have a multiple-level hierarchy, with directories at every level.

Name	Purpose	Bits
valid	Whether this entry is in use	2
tag	Tag part of the address of the line	N
tokens	Number of tokens held by the caches in this ring for this line	$\lceil \log_2(\#\text{Caches}) \rceil$

Table 12.4: Contents of a directory entry

Structure

Table **??** lists the fields in a single directory entry. Note that, in order to accomodate all possible lines in the caches below the directry, the number of entries in the directories must be the sum of the lines in the caches below the directory. Also note that this is achieved by maintaining the same number of sets, and adding the associativities. For example, if an directory manages a ring with eight 8 kB 32-way set-associative L2 caches, then the directory must be at least an 64 kB, 128-way set associative directory.

Read Request

Read Request, with Data

Read Request, with Data and Tokens

Update

Eviction

12.2.4 Root Directory

This section describes the root directories in a COMA system and how the root directory reacts on messages that it can receive. The root directory is both a directory, as mentioned above, as well as a memory controller. It acts as a directory in that it keeps track of the presence of cache-lines in the ring below it. Given that the root directory is located at the top-level of the COMA hierarchy, it therefore indicates whether or not a cache-line is present in the whole network, or not. Next to this, a root directory also acts as a controller or interface with external memory, e.g., DDR modules. Read or writeback requests eventually come up to the root directory, where the root directory communicates with the memory to read or write the desired data.

Structure

Read Request

Read Request, with Data

Read Request, with Data and Tokens

Update

Eviction

12.2.5 Virtual Memory

A topic that has yet remained untouched is the issue of virtual memory. In a typical microgrid, the address space is 64-bits large. It is, however, unfeasible to assume that 16 exabytes of memory storage is available. As in traditional systems, address translation has to be applied at some point to mape the 64-bit virtual address space to a reduced physical address space, possibly backed with paging memory.

The current idea is to apply this translation layer at the root directories, where the off-chip communication occurs, as this seems a cheaper solution than doing address translation at the cache layer.

Chapter 13

Input / output

The microgrid as described so far is a standalone processor architecture and execution model with volatile storage. To enable non-volatile storage and the ability to affect the environment outside the system, an I/O mechanism is required. This is implemented in the form of one or more I/O busses which are connected to one or more dedicated I/O cores, such that every I/O core is connected to one I/O bus. These I/O cores, while fully microthreaded, are intended to be used for I/O communication only[1].

13.1 General mechanisms

In general, I/O is based on two general forms of communication: *transactions* and *broadcasts*.

Transactions happen when one device performs a *request* to a peer and expects a *response* back from the peer device. This happens for example when a host queries a keyboard controller for the last pressed key. So-called "master" devices can perform a request to a host system to provide asynchronously data to an autonomous process, for example using DMA to fill in a disk buffer.

Broadcasts happen when a device *notifies* one or more peers of some event without expecting a response back. For example an Ethernet interface notifies a host system of an incoming frame.

How these forms of communication are implemented in hardware further depends on the hardware substrate where I/O communication is to take place.

Traditionally, I/O was performed on a bus interface with address, data and interrupt lines. This type of medium encourages inefficient operating semantics based on polling and interrupts. For example with buses, a transaction entails a 5-phase operation for the client (set address, set data, select; wait; read) during which no other communication can occur. For a broadcast, the initiator raises an interrupt, and all "interested" peers then scan the bus looking for which device initiated the interrupt.

More recently, a number of I/O transport technologies based on *packed-based networks* have been designed, primarily to overcome physical limitations, and with the added benefit that they encourage operating semantics based on transactions and broadcasts instead of polling and interrupts. This is the case for example with PCI-e, HyperTransport and QuickPath Interconnect. With networks, a transaction on the network mirrors the logical transaction: the client can send its the request as a packet, and receive the response as another packet. A broadcast operation simply amounts to sending a packet to all peers.

13.2 Mapping I/O to dataflow constructs

From a dataflow perspective, the general two forms of communication entail distinct mechanisms.

For transactions, the initiator of a transaction can be seen as issuing a dataflow operation (possibly with arguments), and waiting for its completion (when the transaction ends). The assymetry thus created defines a requestor (client) and requestee (service). Normal operation entails that services can accept requests, and that they respond to clients within a bounded latency, otherwise the dataflow operation deadlocks.

For broadcasts, the initator fires the event and does not expect an acknowledgement. Moreover, broacasts may be initiated unexpectedly, that is without a previous transaction. From a dataflow perspective, there are two ways to deal with this situation: either introduce a *domain boundary* at which all broadcasts are translated to discrete dataflow operations to an arbitrary selected peer; or define a "wait for broadcast" operation which completes automatically as soon as a broadcast is received. We prefer the latter approach because it provides

[1] As an optimization, these cores can have reduced functionality due to their dedicated nature; for instance: no FPU or a smaller thread table.

the lowest latency between the availability of an event and its handling by a software thread (because the software thread already exists and waiting when the notification is delivered).

The actual implementation of the dataflow primitives in hardware depends on the substrate.

With a bus, the interface would be relatively straightforward although it entails polling the bus for a response while a dataflow request is pending.

With the Microgrid we assume instead that the I/O substrate is a packet switched network. We allows both processors (cores) on the grid and I/O devices to initiate transactions: transactions initiated by cores are translated to traditional read/write request/response cycles, whereas transactions iniated by devices are translated to direct memory access (DMA) via the cache network.

The hardware interfaces and components are described in note [mgsim14]. From a programmer's perspective, the I/O space is memory-mapped onto the address space of I/O cores.

Memory mapping means that the memory space (starting from the L1 cache) cannot be accessed for these addresses from the I/O core. From a system integration perspective, either the I/O address space is configured to be separate from the main memory space, or address translation is implemented, or the software running on the I/O cores can use another core as a "proxy" for main memory.

The I/O space is further partitioned in two areas: the transaction area and the notification area.

The transaction area is split into separate address ranges for each possible peer on the I/O network. When a memory load/store operation is performed on the transaction area, the address in the I/O core's address space is translated into a combination of network address and device address. The network address is used to direct the message to the correct peer. The device address is an internal address within the peer, for example a hardware register in a I/O device. Stores are translated to I/O writes and are not acknowledged. Loads are translated to I/O reads and complete when the response is returned from the peer.

The notification area is split into separate addresses for each possible notification channel on the I/O network, one memory word per channel. When a load is issued on such a word, the operation is suspended until a broadcast is notified on that channel, or completes immediately if a broadcast had been received while not waiting. If two interrupt broadcast are received without a listener active, they are merged into one pending notification, such that only one interrupt is notified when the program eventually loads from the channel word.

However next to interrupts we also assume that the I/O substrate supports general notification messages with a payload. This is used for example with SATA devices to indicate which transaction has completed. Instead of being merged, this type of messages are buffered at the I/O interface as discrete events that must be received in turn.