QSigma, Inc.

a research company

Reconfigurable Compile-Time Superscalar Computer Architecture

Fundamental solutions to fundamental computing problems

www.qsigmainc.com

Earle Jennings, CTO

QSigma, Inc.

www.qsigmainc.com

earle.jennings@qsigmainc.com

Introduction to SMP Cores

The Simultaneous Multi-Processor (SMP) is a software-defined entity embodying a simultaneous process in hardware. Each SMP core includes multiple simultaneous processes.

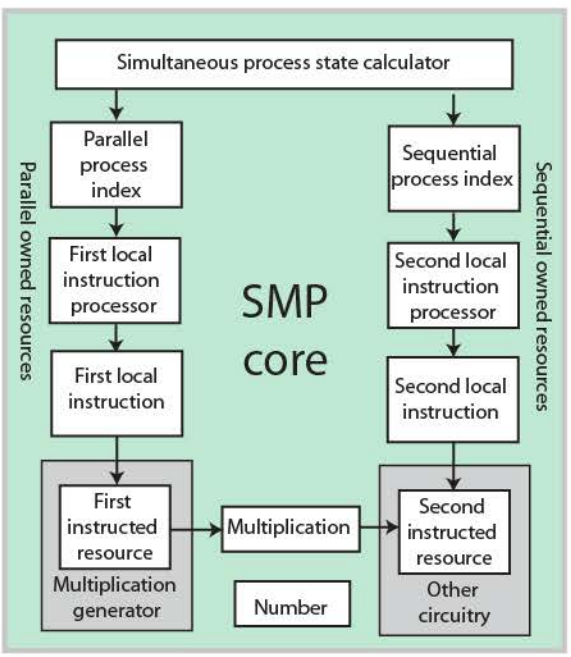
The simultaneous processor *owns* a component of a process state calculator, which generates a process state.

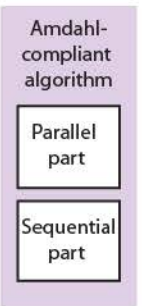
The simultaneous processor *owns* instructed resources.
Ownership may vary for different tasks, but is fixed in one task.

Each instructed resource includes a local instruction processor, which responds to the process state.

The local instruction processor generates a local instruction to instruct its data processing resource, such as a data memory port or an adder.

Owning a resource means only one process can stimulate its instruction processing, so there can be no resource collisions.

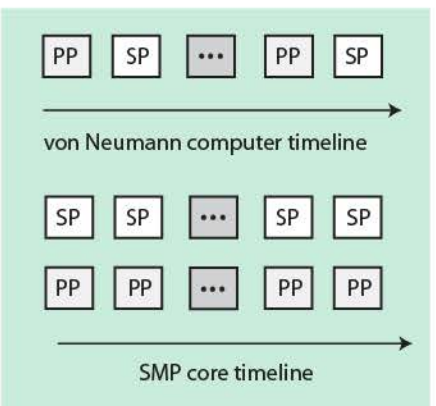




An Amdahl-compliant program has a sequential part (SP) and a parallel part (PP).

The Simultaneous Multi-Processor (SMP) core's simultaneous process state calculator issues two process states on each clock cycle, simultaneously executing the SP and the PP of the Amdahl-compliant program.

This contrasts with a scalar microprocessor (von Neuman computer), which executes, at most, one of the processes at a time.

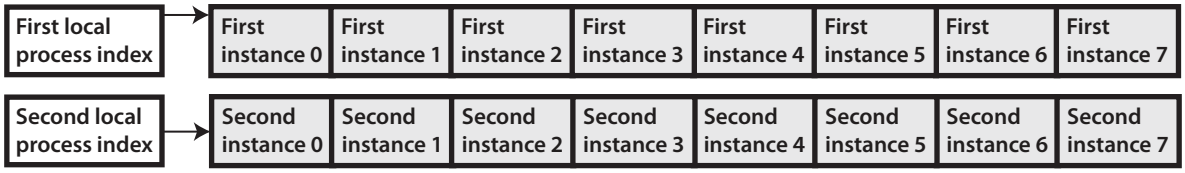


Superscalar microprocessors also support simultaneous performance of SP and PP processes, but only with a very large increase in circuitry and energy consumption. ***SMP cores substantially reduce energy consumption and silicon size.***

SMP Cores Enable Virtual VLIW Space

SMP cores make VLIW practical.

For example, assume the sequential processes and the parallel processes of an Amdahl-compliant program have eight, separately accessible, local, process-owned instructions.



A VLIW instruction memory supporting these same eight, independent operations requires a much larger VLIW memory of 64 instructions.

First instance 0	Second instance 0
First instance 0	Second instance 1
First instance 0	Second instance 2
First instance 0	Second instance 3
First instance 0	Second instance 4
First instance 0	Second instance 5
First instance 0	Second instance 6
First instance 0	Second instance 7

First instance 2	Second instance 0
First instance 2	Second instance 1
First instance 2	Second instance 2
First instance 2	Second instance 3
First instance 2	Second instance 4
First instance 2	Second instance 5
First instance 2	Second instance 6
First instance 2	Second instance 7

First instance 4	Second instance 0
First instance 4	Second instance 1
First instance 4	Second instance 2
First instance 4	Second instance 3
First instance 4	Second instance 4
First instance 4	Second instance 5
First instance 4	Second instance 6
First instance 4	Second instance 7

First instance 6	Second instance 0
First instance 6	Second instance 1
First instance 6	Second instance 2
First instance 6	Second instance 3
First instance 6	Second instance 4
First instance 6	Second instance 5
First instance 6	Second instance 6
First instance 6	Second instance 7

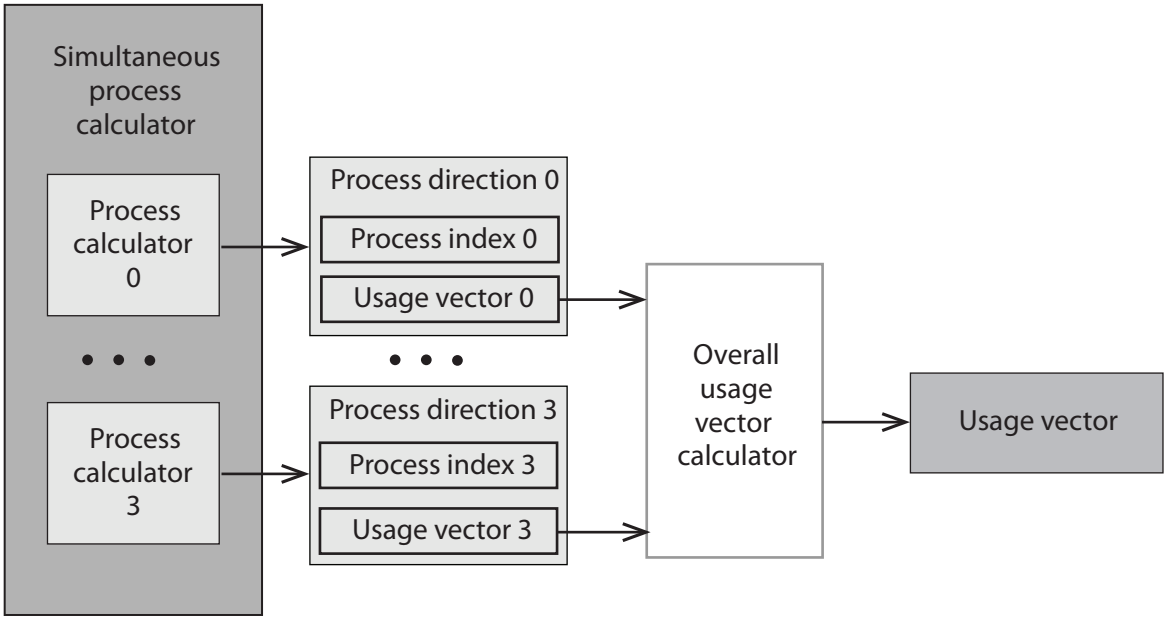
First instance 1	Second instance 0
First instance 1	Second instance 1
First instance 1	Second instance 2
First instance 1	Second instance 3
First instance 1	Second instance 4
First instance 1	Second instance 5
First instance 1	Second instance 6
First instance 1	Second instance 7

First instance 3	Second instance 0
First instance 3	Second instance 1
First instance 3	Second instance 2
First instance 3	Second instance 3
First instance 3	Second instance 4
First instance 3	Second instance 5
First instance 3	Second instance 6
First instance 3	Second instance 7

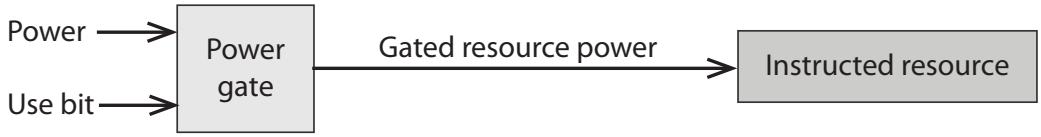
First instance 5	Second instance 0
First instance 5	Second instance 1
First instance 5	Second instance 2
First instance 5	Second instance 3
First instance 5	Second instance 4
First instance 5	Second instance 5
First instance 5	Second instance 6
First instance 5	Second instance 7

First instance 7	Second instance 0
First instance 7	Second instance 1
First instance 7	Second instance 2
First instance 7	Second instance 3
First instance 7	Second instance 4
First instance 7	Second instance 5
First instance 7	Second instance 6
First instance 7	Second instance 7

Power Management and Monitoring in SMP Cores



The process state calculator generates a usage vector for each process. This usage vector indicates which instructed resources are owned, and used, by a process on each execution wave front.



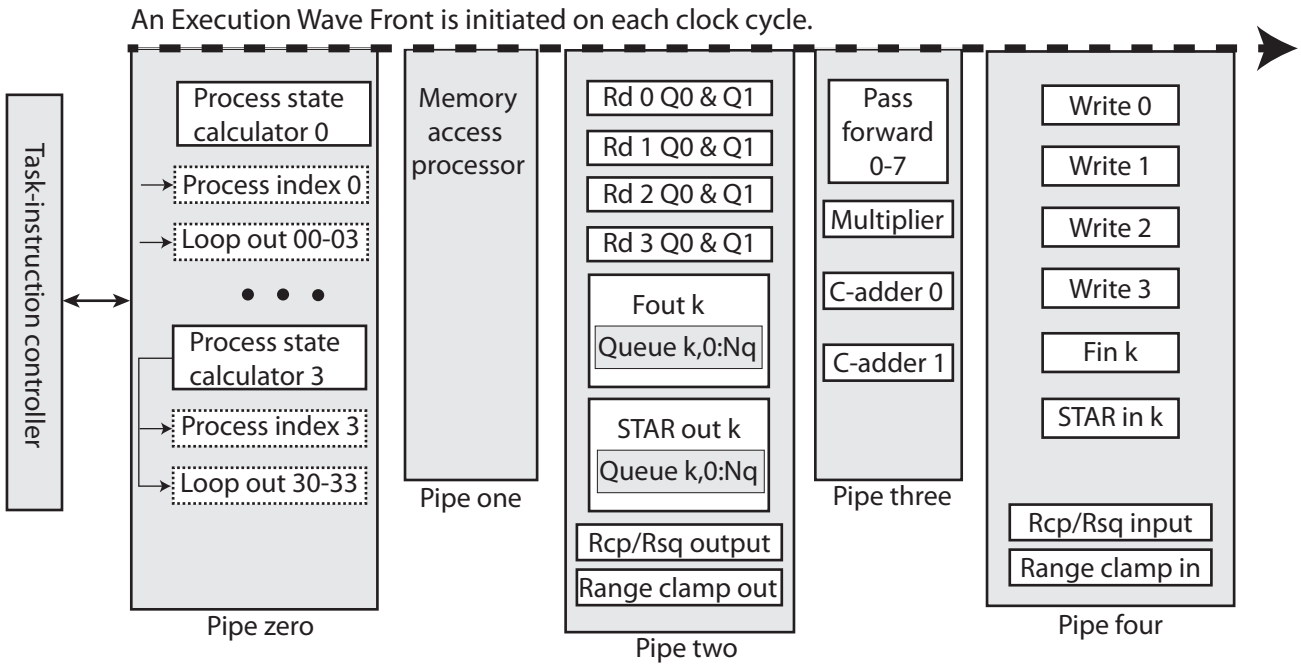
A gated resource power is generated by the power gate in response to the Use bit of the usage vector for an instructed resource.

The instructed resource uses the gated resource power as the execution wave front traverses the instructed resource.

Some implementations may gate the clock to effect control of the power.

Transforming a Microprocessor's Superscalar Interpreter into a Compile-time Utility

A Simultaneous Multi-Processor (SMP) core



Application compatibility

Application compatibility is achieved by insuring the SMP core modules are *semantically compatible* with an existing superscalar microprocessor.
Each core module includes floating point and integer typed SMP cores.

Semantic compatiblity

Semantic compatibility is achieved by using two targets:

1) a superscalar microprocessor
2) a core, or module of cores

Three components of the technical collateral of the superscalar microprocessor are used to create the semantically compatible core module of typed SMP cores:

1) A behavioral model of its superscalar interpreter.
2) A Register Transfer Layer (RTL) model of its data processing resources.
3) The verification and test sets used to confirm the microprocessor for production.

Source code written in the microprocessor assembly language generates executible programs for the two targets.
If every assembly program, when compiled and loaded into these two targets, generates the same output stream, in response to the same input stream, then the two targets are semantically compatible.

A semantically compatible SMP core can replace a superscalar microprocessor. SMP cores use substantially less power and take up a fraction of the silicon.

Programmable Execution Modules (PEMs)

Programmable Execution Modules (PEM)s contain multiple instances of the SMP core modules.

A core module contains a floating point core and an integer core, known as typed cores.
Typed cores can share instructed resources only among their simultaneous processes.

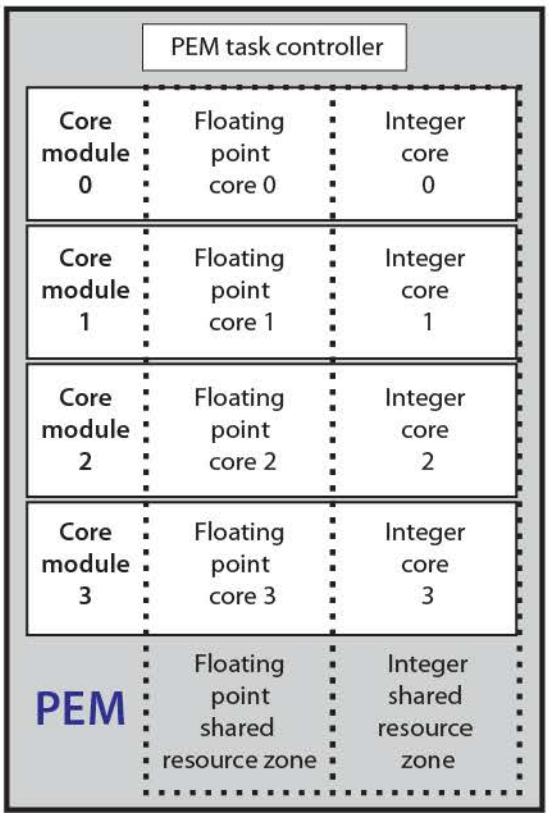
> An integer simultaneous process can own instructed resources in any integer core,
> but none in the floating point cores of the PEM.
> Conversely, a floating point simultaneous process can own instructed resources
> in any floating point core, but none in the integer cores.

These simultaneous processes respond only to data availability within their merged core,
their loop states, and their previous process state.

Data availability directs the operation of these processes, insuring correctness of results in response to *when the data becomes available in the PEM.*

Every PEM can implement up to 32 separate simultaneous processes,
under ***complete programmer compile-control.***

One PEM has, at least, the computing power of a quad-core micro-processor.

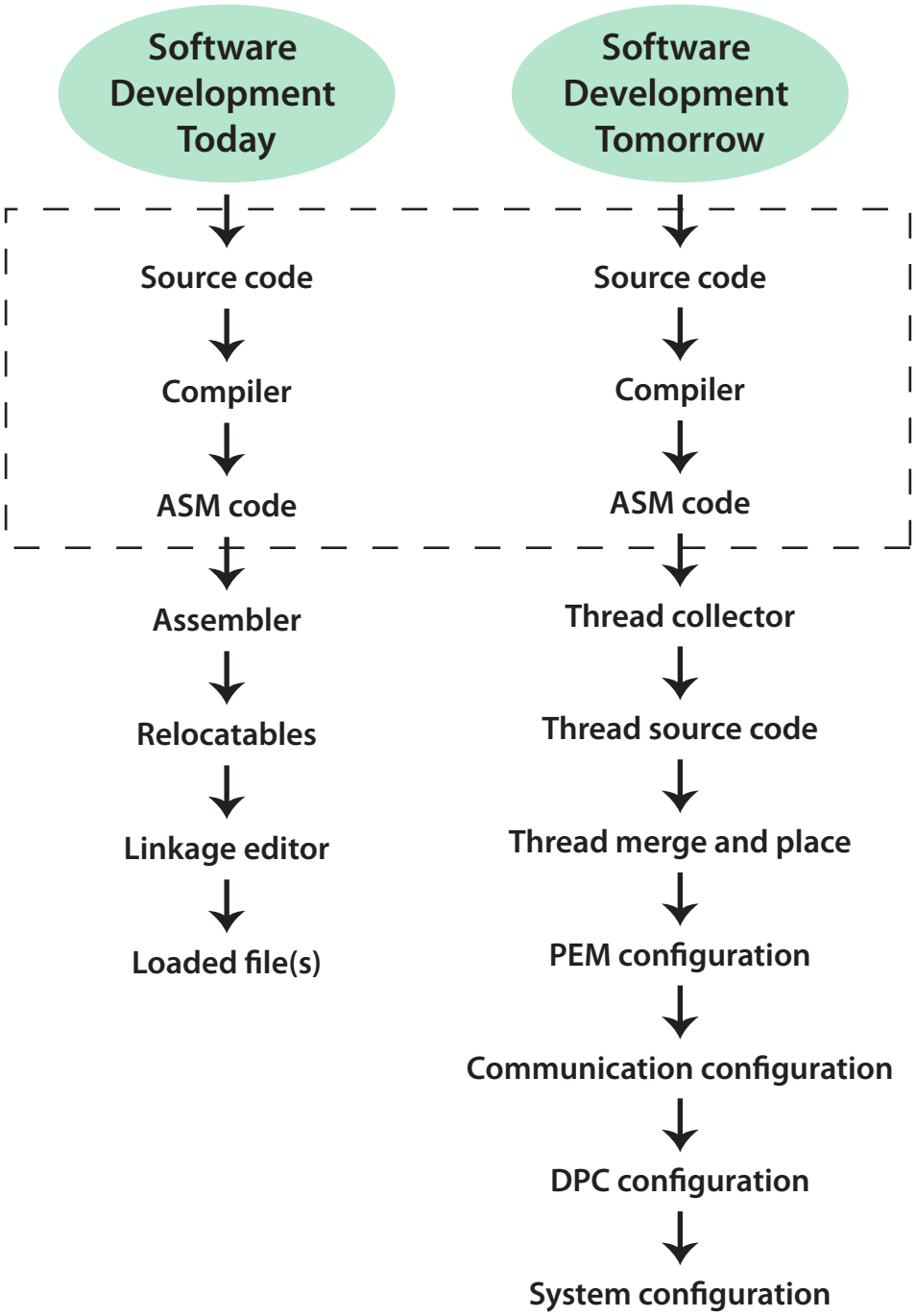


This PEM includes four instances of the core module, with the corresponding cores merged.

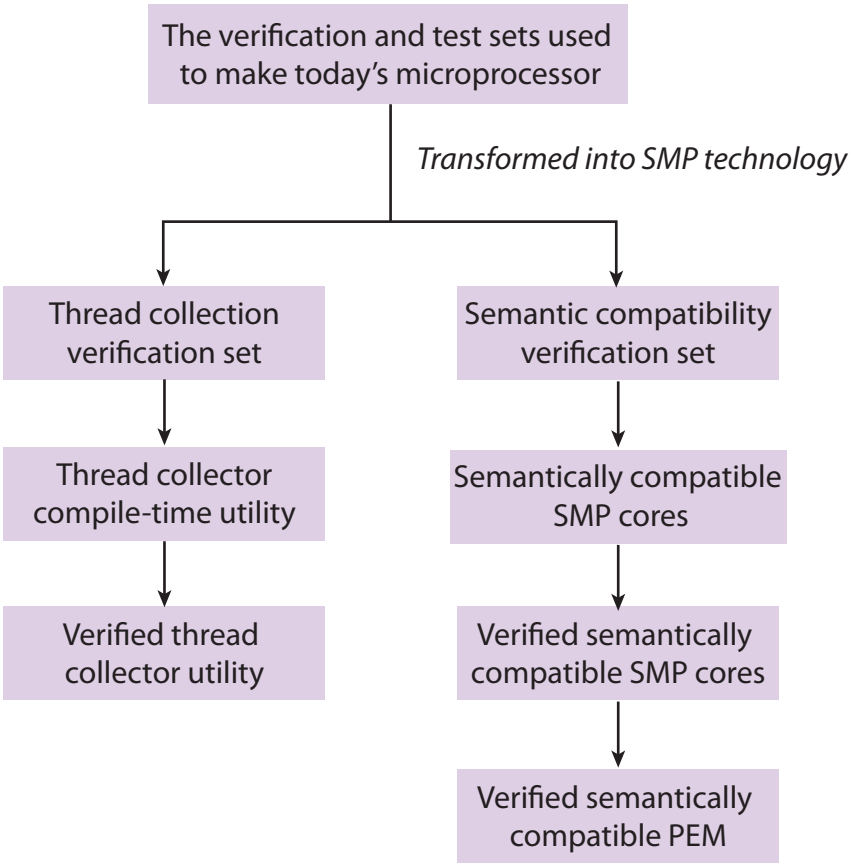
Assuming 256 process states per task, a PEM supports a VLIW instruction space of 256 ^ 32 of VLIW instructions per task.

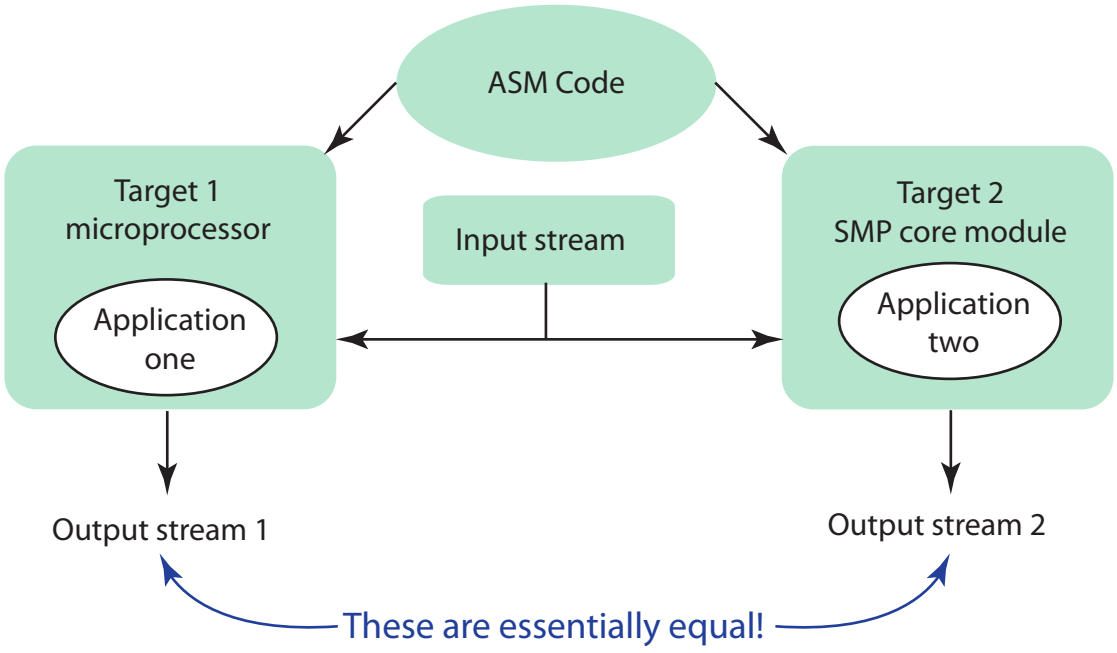
This VLIW instruction space removes the need for instruction caching

Software Application Development Process Today and Tomorrow



Creating Semantically Compatible Core Modules





SMP Thread Collection Utility

Today, a thread of execution is the smallest sequence of program instructions that can be managed independently by a scheduler in an operating system. In this architecture, threads are the operations of at least one typed core in a PEM. The application program is transformed into threads at compile-time, by a thread-collecting utility, based upon a superscalar interpreter.

In the QSigma architecture, threads are operations of at least one typed core.

An application program is ***transformed at compile-time into the computer and its operating environment.***

Instructed Resource	Dot Product Accumulate Process	First Filter Accumulate Process	Second Filter Accumulate Process	Calculate Maximum Process
In queues				Max-in queue
Feedback queues	Product fdbk 1, Dot accum 1 to n			Max-fdbk 2 to max-n
Memory read queue		Tap Read, Fir in	2nd-coef read, 2nd-pass data	
Multiplier				
C-adder 0	Yes	Yes	Yes	Yes
C-adder 1				
Memory write ports		1st write accumulate	2nd write accumulate	
Feedback in	Dot accumulate Feedback in	1st accumulate	2nd accumulate	C Max Feedback in
Output portal	Yes	Yes	Yes	Yes

Thread Collector Output Process Ownership

Transforming the superscalar interpreter into a compile-time utility ***removes the need for instruction caching.***

Algorithm State Machine (not your Dad's FPGA!)

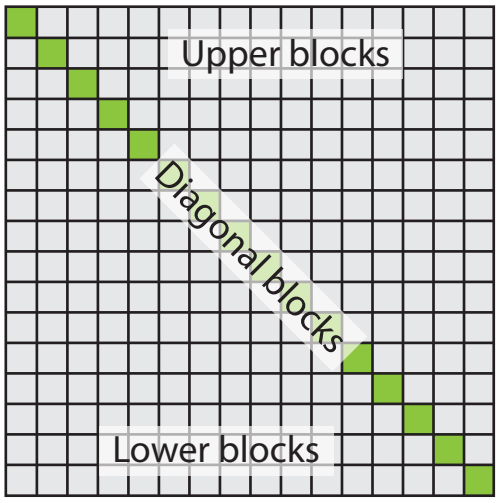
Dot Product Accumulation Process	1st Filter Accumulation Process	2nd Filter Accumulation Process	Max in & Accumulation Process	Merged Accumulation Process
DPaccum6	FIRaccum2	FFTaccum3	Max7	DPaccum6
DPaccum5	FIRaccum1	FFTaccum2	Max6	FIRaccum2
DPaccum4	FIRaccum0	FFTaccum1	Max5	FFTaccum3
DPaccum3		FFTaccum0	Max4	Max7
DPaccum2			Max3	DPaccum5
DPaccum1			Max2	FIRaccum1
DPaccum0			Max1	FFTaccum2
			Max0	Max6
				DPaccum4

Merging processes in terms of their respective process states creates a merged process of the merged thread. The highest priority process state is the top process state, and has a minimal probability. Merging a corresponding process means listing the highest priority state of each process in the merged process. The process merging continues in this way.

After collecting the threads, the intermediate program representation is analyzed. This analyses determines whether sub-programs should be merged into the invoking thread, or separately instantiated.
For example, an 8 by 8 matrix inverse is small and performs quickly enough to optimally merge its thread into the invoking thread. However, a 32 by 32 matrix requires, roughly, 32K clock cycles. Placing this thread in a separate core may be more efficient.

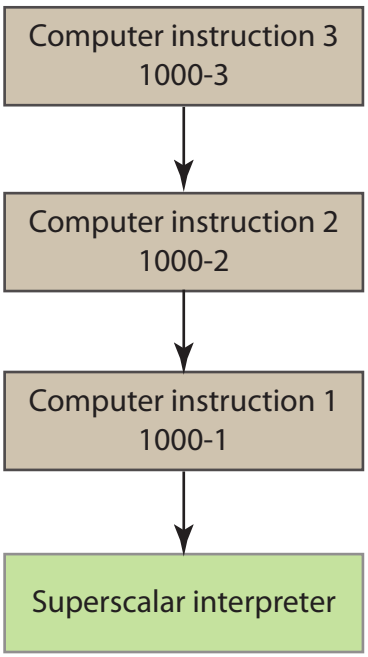
Compile-time merging and placement of threads maximizes system throughput with *no runtime overhead.*

Program (s)
+
SMP Computer
=
Algorithm State Machine Network

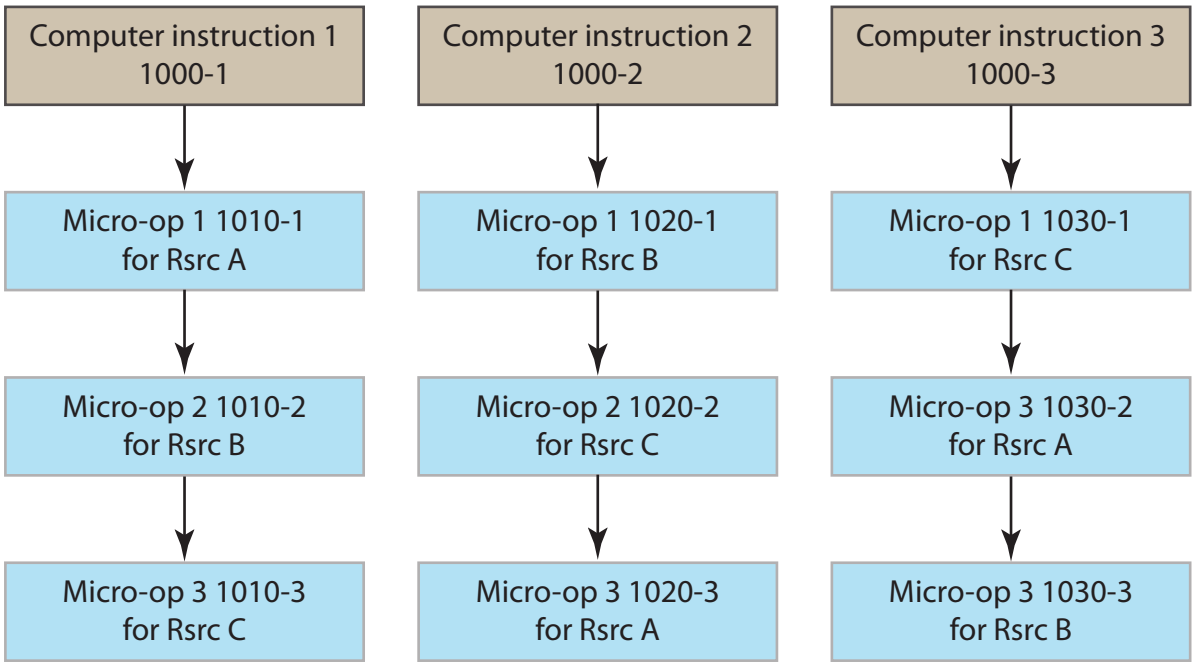


The result of merging and placement of threads implementing block LU decomposition (Linpack)

Threads
An Example



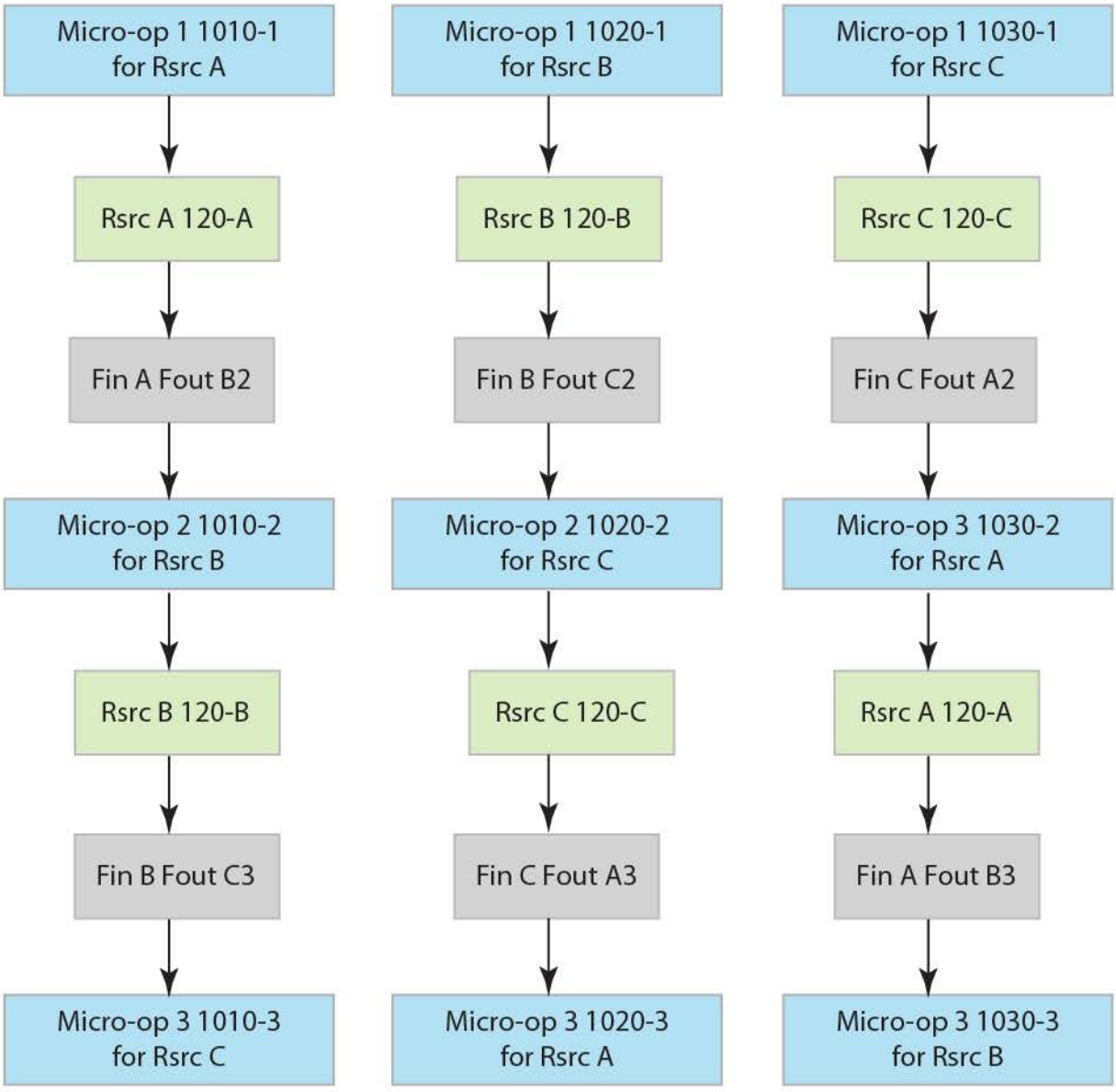
Three computer instructions about to enter a microprocessor superscalar interpreter



Assume that each micro-operation depends on finishing the result of the previous micro-operation of the instruction before it can run.

Thread Collector Utility Output
An Example

In this example of the thread collector utility outout, only Micro-op 1010-1, 1020-1 and 1030-1 need to be initiated to insure the operations of this thread are performed.



Collapse count 1010	Condition vector			
	Cond 0 1110-0	Cond 1 1110-1	•••	Cond K 1110-K

The thread condition vector is used to initiate the processes of the next thread to run and respond to test conditions.

When the current thread has completed its initiation, it releases the processes it owns, to operate based upon feedback and inputs.

Each element of the condition vector includes a 2 bit condition code:
'00' == 0 , '01==> 0, '10'== <0, and '11 means unavailable.

(Sub)program for Merged Threads
An Example

Example (Sub)program with Thread Condition Register States for Merged Threads

```
subprogram X1(parm_list1){
  code 1              // Cond State Length = 0 Cond collapse = 0
  if test 1           //                   = 0       0
  { code 2 }          //                   = 1       0
  else {
    code 3            //                   = 1       0
    if test 2         //                   = 1       0
    { code 4          //                   = 2       0
    } else {
      if test 3       //                   = 2       0
      { code 5        //                   = 3       0
      } else {
        code 6        //                   = 3       0
      }
      code 7          //                   = 2       1
    }
    code 8            //                   = 1       2
  }
  code 9              //                   = 0       3
}
```

Summary of Basic Branch Related Constructs and the Thread Condition Register:

Arithmetic IF from Fortran,
switch conditionals of C, and
range limiting by the clamp circuits, for transcendental functions,
all use the trinary coding scheme for the conditionals.

All looping constructs are converted into two basic C templates:

while some_test1
do loop-body1;

do loop_body2
while some_test2;

The Non-Linear Function Accelerator (NLA)

The QSigma NLA features a log2 calculator which performs
an exact factorization of a double precision mantissa with 2 guard bits.

> The logarithms of the factors are generated by table lookup with 6+ bits of additional fractional part, so that calculations such as X^64 to X^-64 can be carried out accurately to within 1/2 the least significant guard bit.
>
> The log2 calculator, and subsequent NLA components, enable an exact representation of log2(0) as negative infinity, and insure that within the NLA, negative infinity added to any other log domain number yields negative infinity.

2^(negative infinity) is exactly 0.0 in the floating point core, removing one of the persistent problems with log domain calculators.

The NLA incorporates the correct circuitry to calculate the exponential of a complex number accurate to within 1/2 the LSB of the mantissa.

The NLA improves performance for :

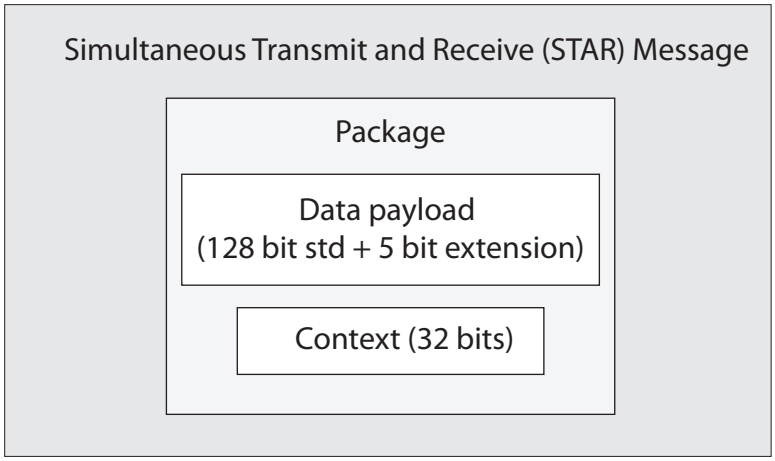
> Log and exponential functions by 10X.
> General polynomial and rational function evaluation by 2X-3X.
> Exponentials of complex numbers by 3X-4X.

The NLA augmentation doubles performance for matrix multiplied by matrix operations.

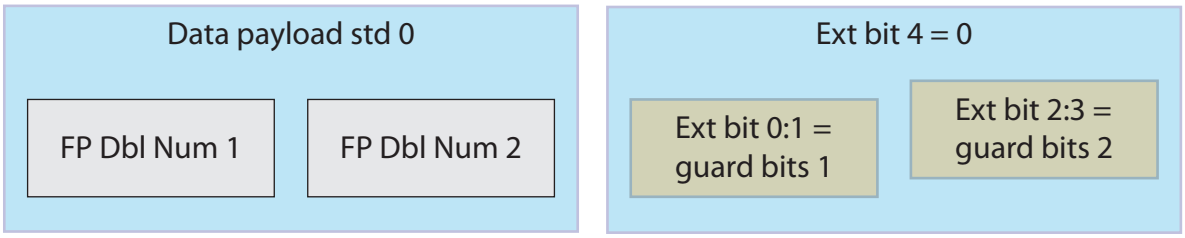
> For example, performance of Block LU Decomposition in Linpack is doubled with the addition of the NLA.

A one exaflop computer becomes a two exaflop computer when the NLA is used in each core module.

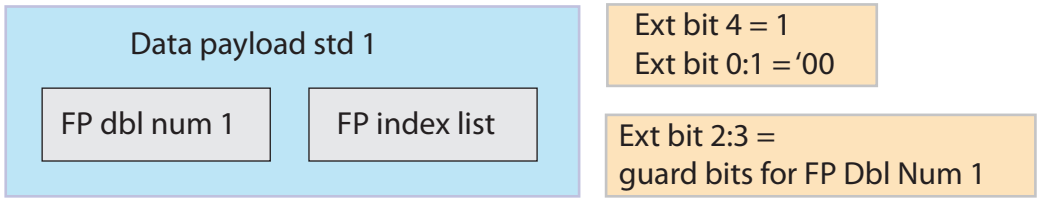
The STAR Message Protocol
An Introduction



The STAR message protocol transmits and receives a STAR message on every local clock cycle, except when responding to an uncorrectable error on reception.
The response to such errors is automatic channel component replacement within, at most, a microsecond.
Assuming a 1 ns clock, 200 Gbits/second can be delivered, and sent, on each STAR channel.
The context of the message is interpreted at every STAR message core to determine its disposition and transfer.
The context, and its interpretation, is under complete control of the program.

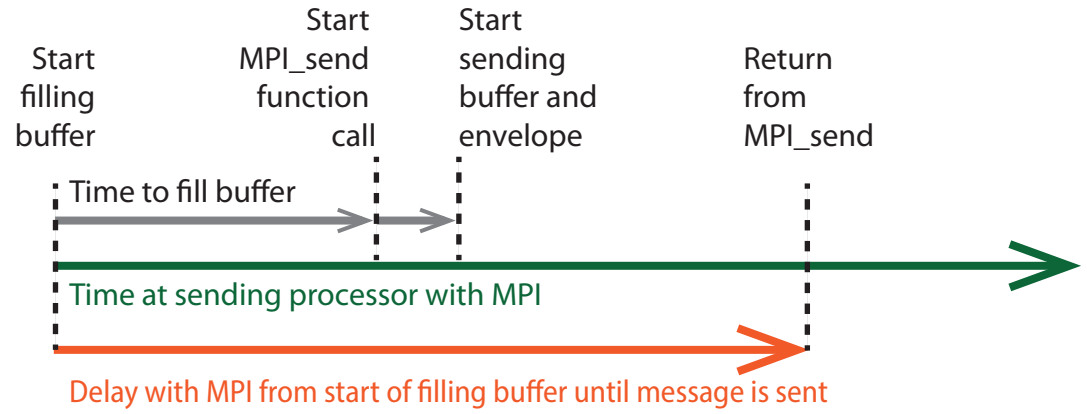


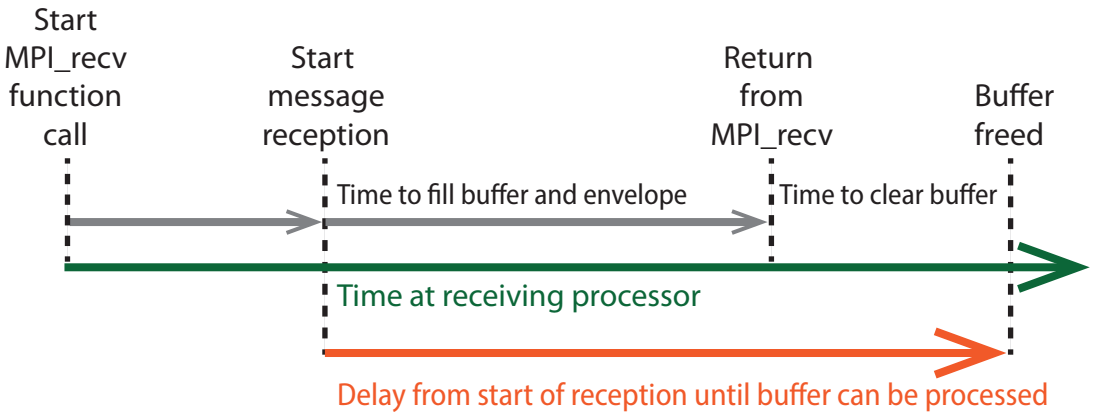
Example of data payload supporting 2 guard bits for each of two double precision numbers.



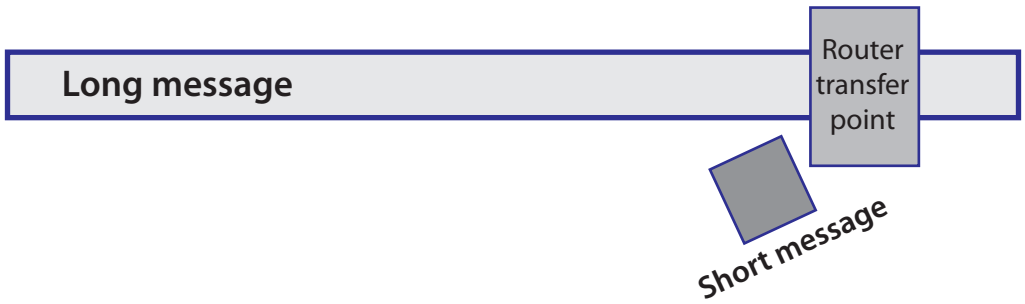
Double precision number and index list suitable for:
- HPL pivot entry calculations
- Multi-grid operations
- Other sparse matrix operations

Star Messaging and MPI





STAR messages take only one clock cycle to be sent and received



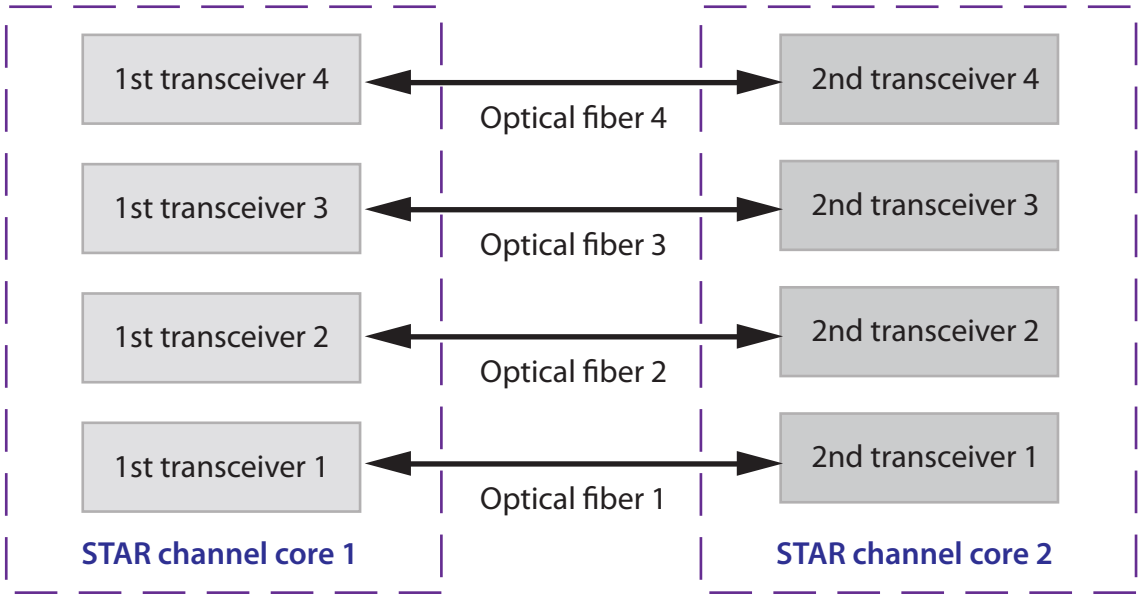
In MPI the long message can block the short message at a router transfer point

STAR messages are all the same length

STAR messages traverse each communication resource pipe in a single clock

Multi-fiber STAR Message Channel
An Example

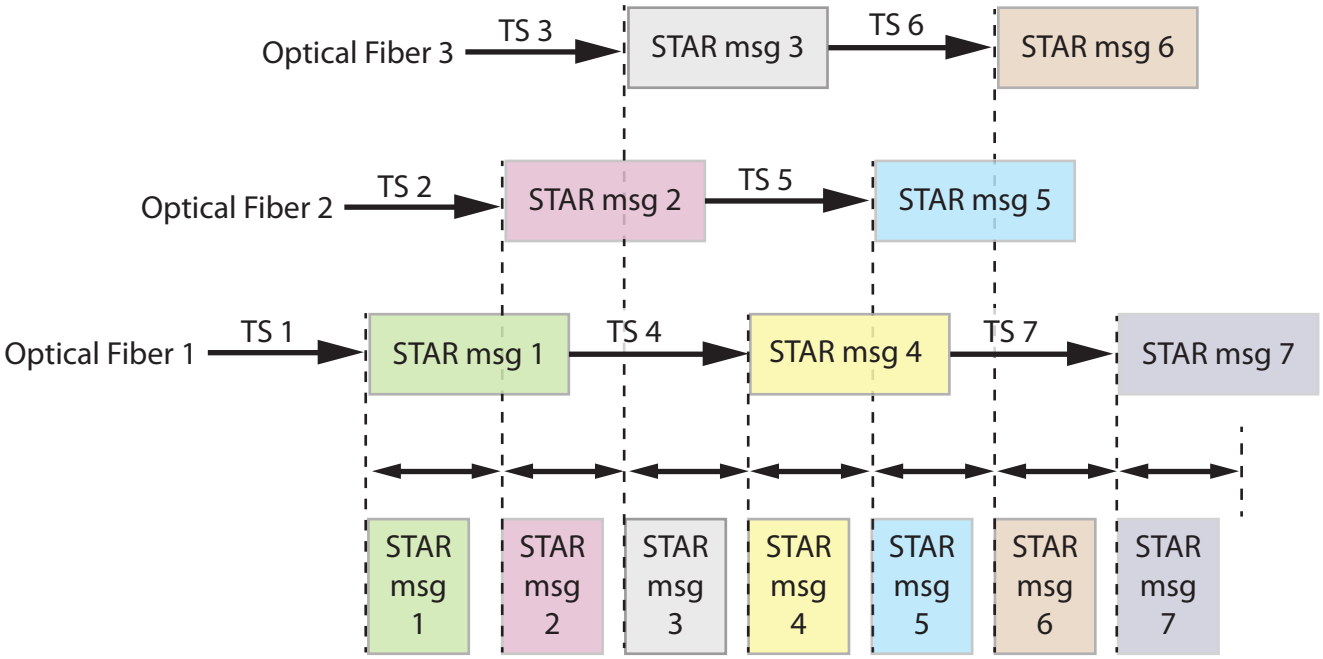
Example of a multi-fiber STAR message channel



Optical fiber 3 serves as a primary delivery vehicle for the STAR messaging when the timing at the transceivers is about 100 GHz.

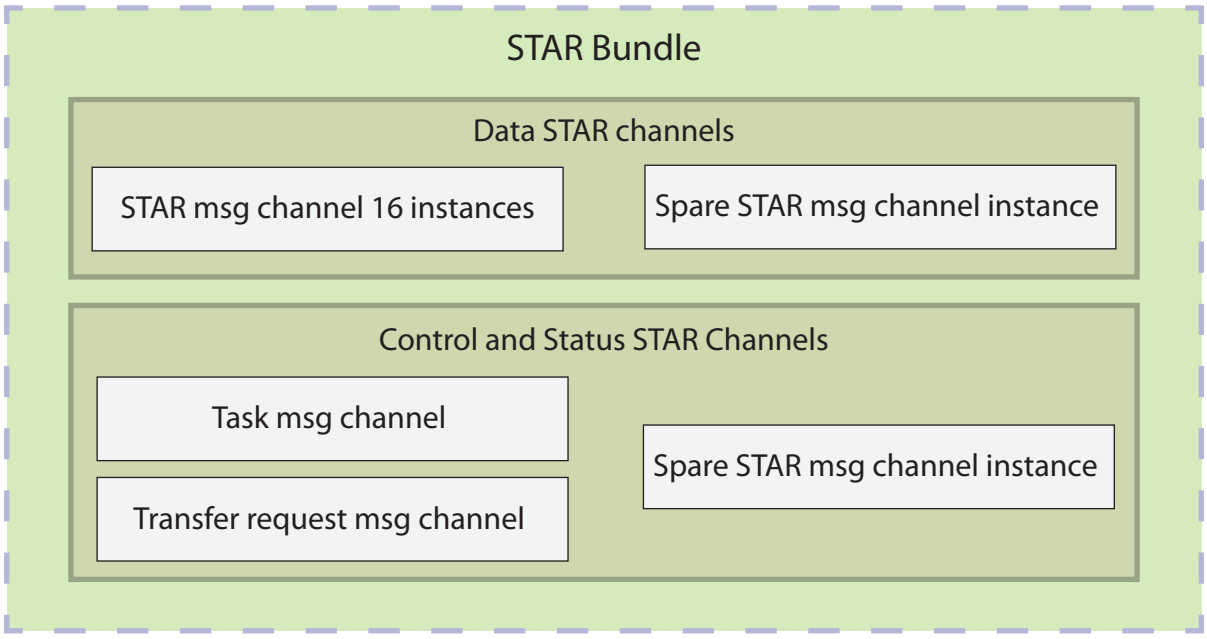
Optical Fiber 4 is a spare, possibly powered down until needed.

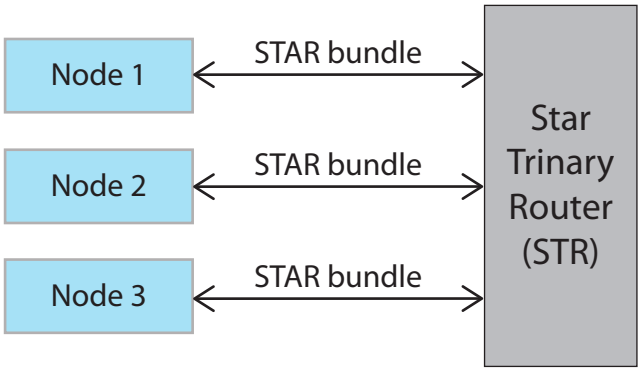
Transmitting STAR messages on this channel



TS k stands for the Training Sequence for the STAR Msg k, for k = 1 to 7.

The STAR Bundle and STAR Trinary Router





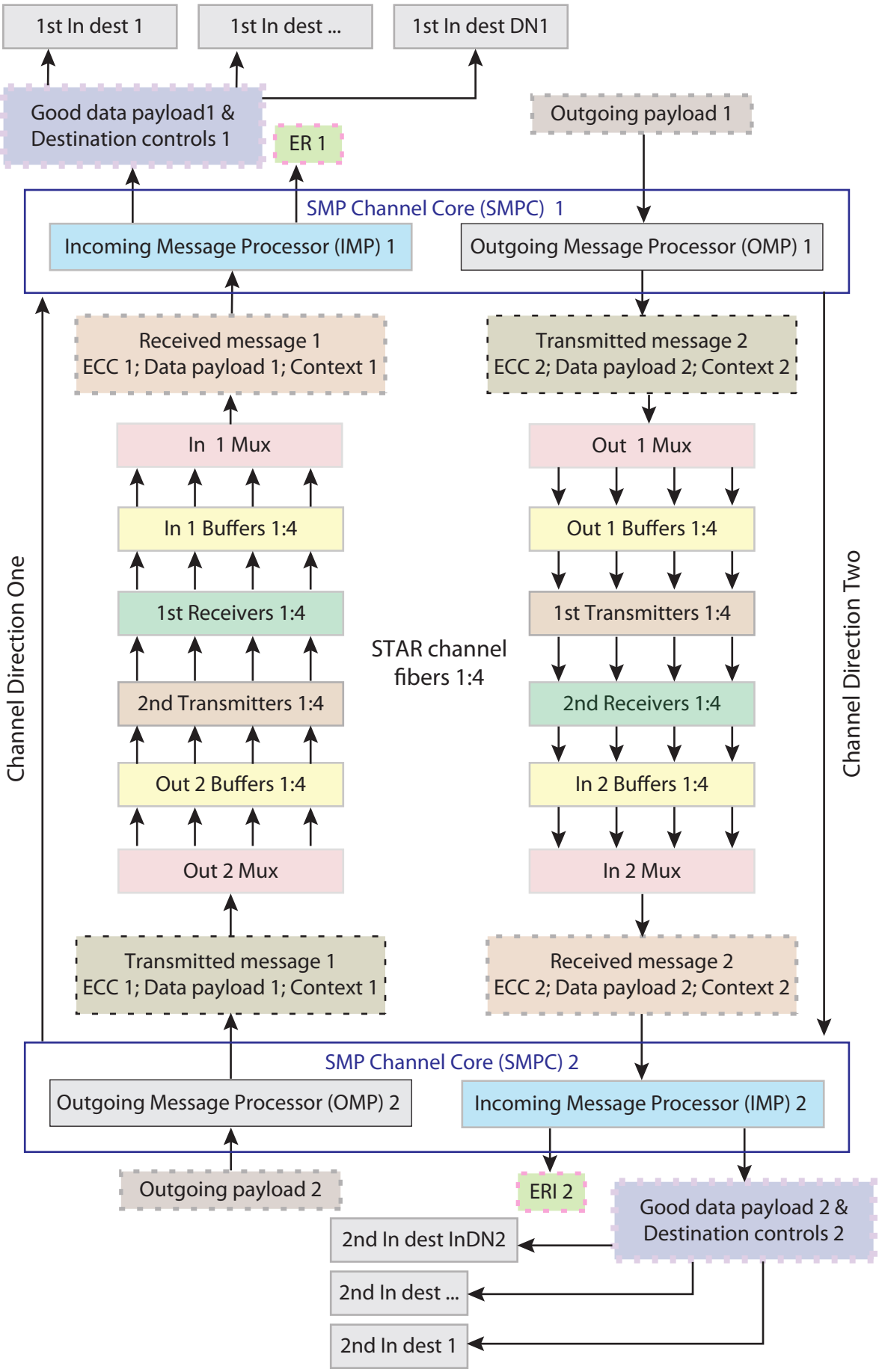
Each node can send and receive16 data payloads every clock (1 ns), with the STAR Trinary Router (STR).

Each node has a data bandwidth of 2 Terabits/sec in and out.

A node can be a PEM, or a chip.

STRs are implemented inside a chip as a communication module paired with a PEM.

Star Channel Core



The STAR Mostly Binary Network Interface for HPC

Binary trees leave HPC system components vulnerable to communications bottlenecks.

By extending the network to a Mostly Binary Tree, chipstacks, optical PCBs, etc, can have dual, or better, interfaces.

There are 30 to 33 Star bundles available at each cabinet to interface to the datacenter.

Each optical channel is physically compatible to ethernet, therefore the data STAR channels of each STAR bundle can interface to 32 to 48 Gbit ethernet networks.

Exascale can be achieved with 256 cabinets.
Each cabinet can simultaneously communicate with up to 1K ethernet networks of 100 Gbit/second bandwidth.

The Star Mostly Binary Interface canonically resolves bandwidth delivery to and from supercomputers for years to come.

Architectural Support for Fault Resilience

Four degrees of freedom in managing the STAR communication network support automatic local fault resilience:

1) The opto-transceiver clocks can be slowed down.

2) A faulty transmitter link to a receiver in one direction, can be replaced by the spare optical fiber in that direction.

3) A faulty STAR message channel instance in one, or both, directions can be replaced with the spare STAR message channel.

4) *Two* opto-fibers in a STAR message channel are operated, rather than three, lowering the bandwidth to two STAR messages every three clock cycles.

Multiple layers of arithmetic provide support for automatic local fault resilience:

Each C-adder incorporates two fully functional 2 operand adders, and can be configured to operate on just one adder, should the other be faulty.

Each Floating Point (FP) 2 operand adder has two identical stages, so if one of them is faulty, the other can do its job.

Each 2 operand multiplier has two identical stages, if one is faulty, the other can do its job.

The FP multiplier can either be augmented by the Non-Linear Accelerator (NLA), or be replaced should it be faulty.

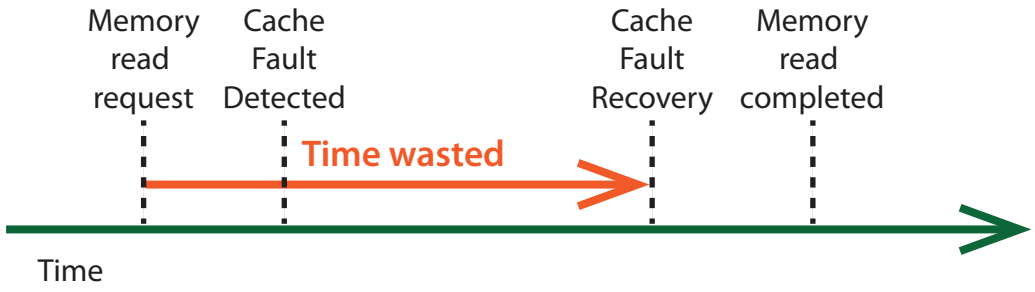
Additional fault resilience :

Each pair of PEMs have a spare core module.

Anticipation, not Hindsight

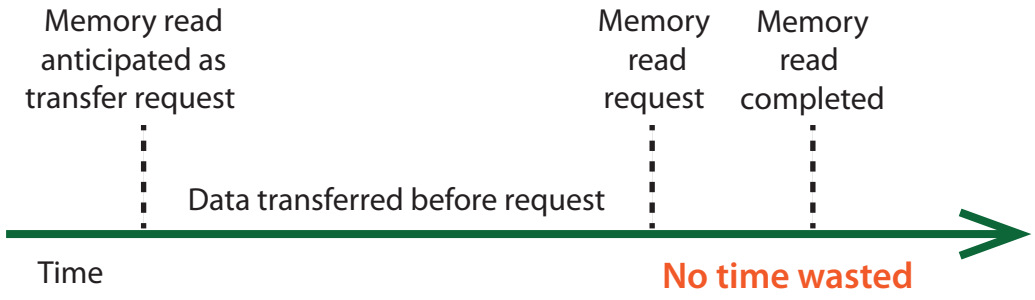
In the QSigma architecture, ***memory access is anticipated***, not initiated after it is requested, in hindsight, by a cache fault.

Hindsight:

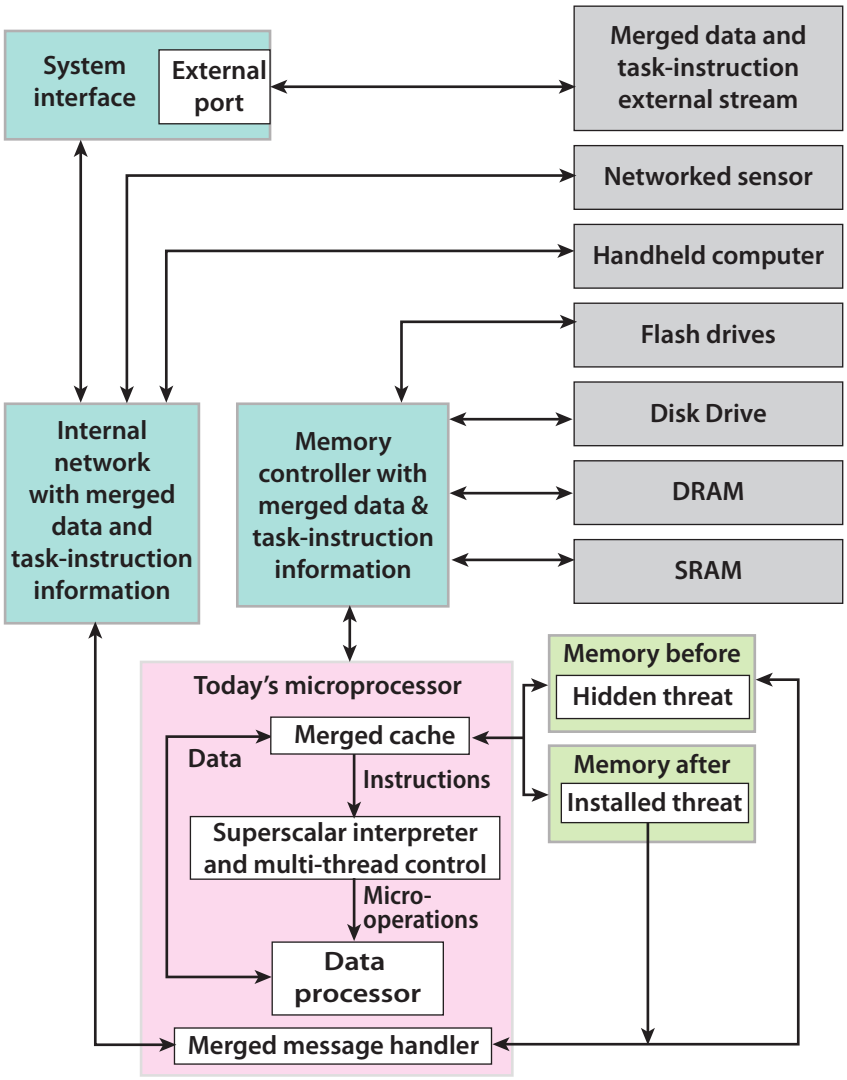


By anticipating the memory access, the system can fetch data before it is requested, preventing the data processing resources from stalling.

Anticipation:



Today's Vulnerable Data Center



Block diagram of a vulnerable data center.

The external portal does not usually separate data related information from task/instruction related information, so any mistake in its firewall can allow hidden threats into the system.

The internal network does not separate the data related information from task/instruction related information, so any hidden threat has an opportunity to further infiltrate the system.

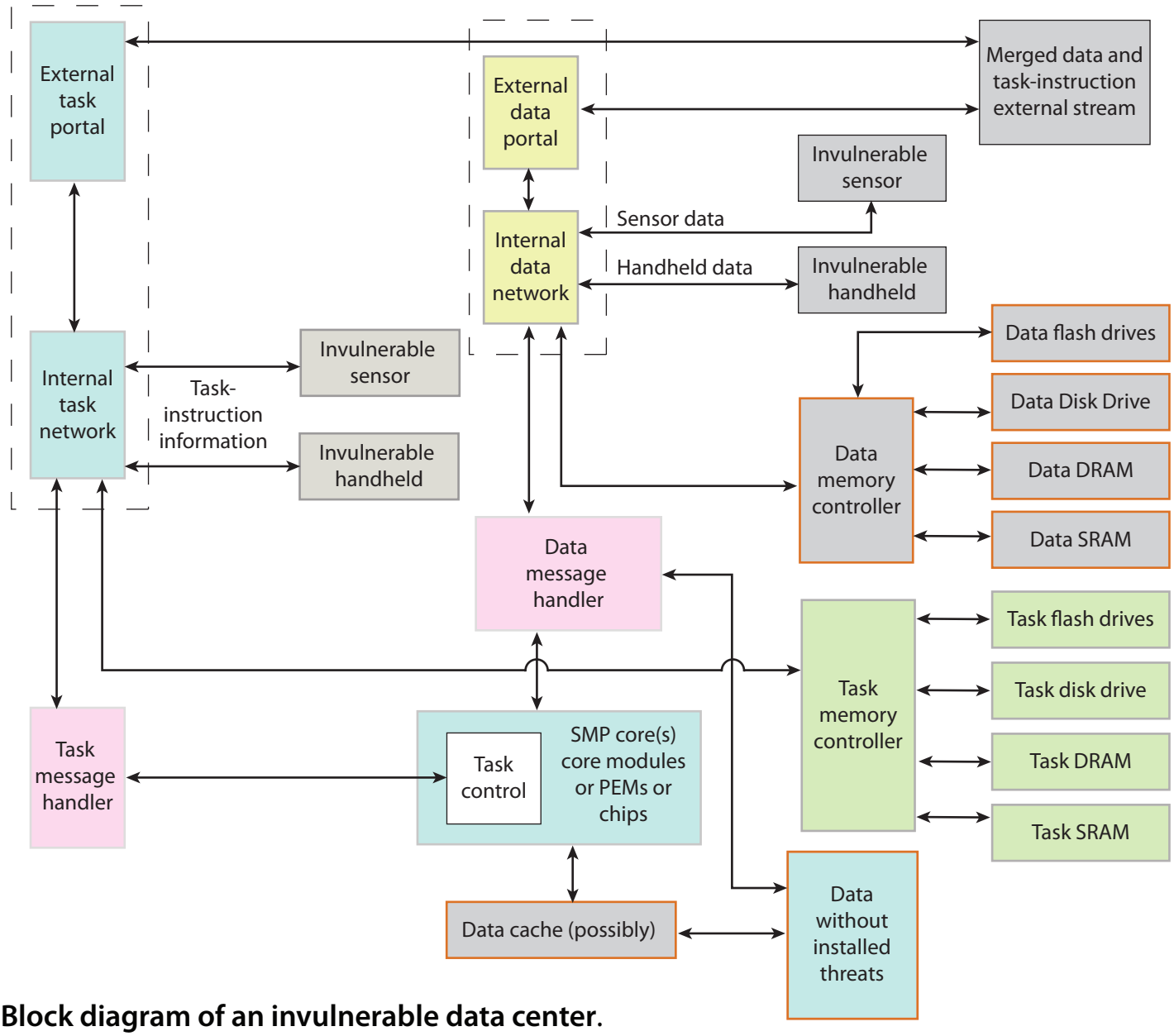
Memory controllers tend to access a single memory domain, which is supposed to have subdomains for programs and data. Very small mistakes in a program, or operating environment, can turn hidden threats into installed threats, where they can persist indefinitely.

Flash drives are another entry point for hidden threats. They also operate a single memory domain of files, which can also hide threats.

The internal network of these data centers interfaces with merged message handlers which deliver the hidden threat into memory, creating the installed threat. The message handler can also send messages containing the hidden threat elsewhere.

Today's microprocessors use a merged cache which can readily load a hidden threat into its memory. After the program error has occurred, the microprocessor then fetches and executes the installed threat.

Tomorrow's Invulnerable Data Center



Block diagram of an invulnerable data center.

The invulnerable data center interfaces to less secure, general purpose, networks through a new interface.

This new interface operates two primary portals to two separate internal networks.
One portal supports access to task management and program configuration.
A second portal supports data transfers.

The invulnerable data center physically separates data memory, task-instruction memory, and their memory controllers.

There are no transfer paths from one form of memory to the other.

No data-related operation can alter a task, or an instruction, residing in the task-instruction memory. This organization removes the opportunity for viruses and rootkits to infect SMP cores, handhelds and networked sensors.

An Exascale Reconfigurable Compile-time Superscalar Computer

QSigma's architecture can achieve exaflop performance while:

(1) Consuming minimum power,

(2) Maintaining maximum system reliability,

(3) Maintaining application compatibility,

(4) Supporting the maximum number of super computer programs,

(5) Providing maximum scalability, and

(6) Providing far greater security.

Systematically built from the fundamentals of hardware, software, systems, and algorithms interacting together, beyond co-design, the QSigma architecture is ***quad-designed***.

(1) Consumes minimal power because the SMP cores replace instruction caching and superscalar interpretation with a compile-time thread collection utility. This innovation, along with the STAR communication components, means the QSigma data processor chips require less than 10% of the circuitry needed by contemporary parallel processor chips to deliver comparable performance. Any circuit not in use, in any single clock cycle, is automatically powered off.

(2) Maintains maximum system reliability through automated fault resiliant operation of arithmetic components, and communication channels, with support for runtime system testing as an automatic part of the operating environment.

(3) Maintains application compatibility because the SMP cores are semantically compatible with an existing superscalar microprocessor. Assembly language programs can target the microprocessor and the SMP cores.

(4) **Supports the maximum number of super computer programs** because this reconfigurable architecture transforms programs into an Algorithm State Machine made of semantically compatible core resources.
Every program reconfigures the computer into it's own Algorithm State Machine.

(5) Provides maximum scalability through the use of consistent small component ensembles. These ensembles scale seamlessly from inside chips to a computer floor of cabinets.

(6) Provides far greater security because data access operations cannot alter instruction memories, removing the standard mechanism of attack by viruses and rootkits.

QSigma's reconfigurable compile-time superscalar computer can be built with today's technology, and will deliver essentially linear performance improvements for any number of cores, data processor chips, and/or cabinets, from a test chip, to a super computer, to an exascale computer, and beyond.

QSigma's reconfigurable compile-time superscalar computer can integrate seamlessly with exotic materials, such as memristers and molecular gates, with only minor changes to ported programs. Exascale can be achieved with 256 cabinets.