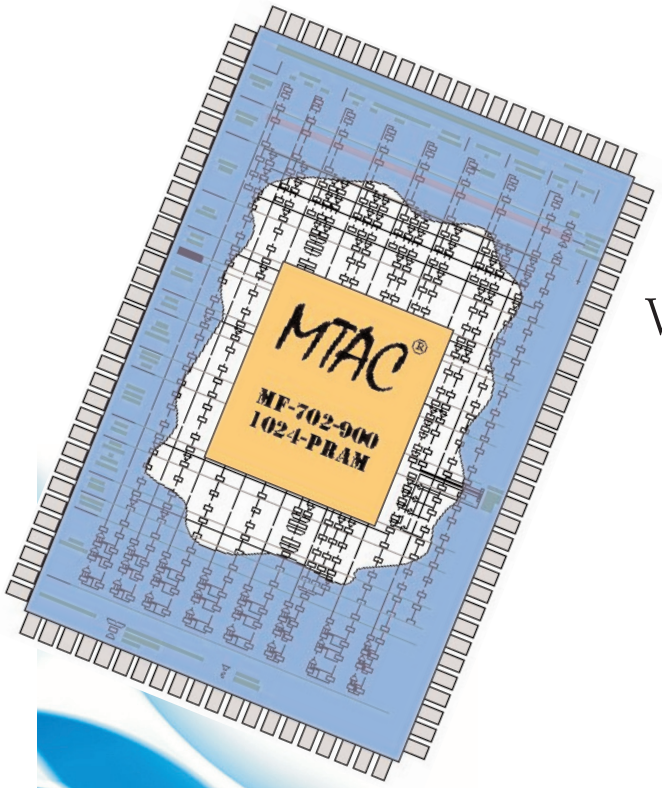
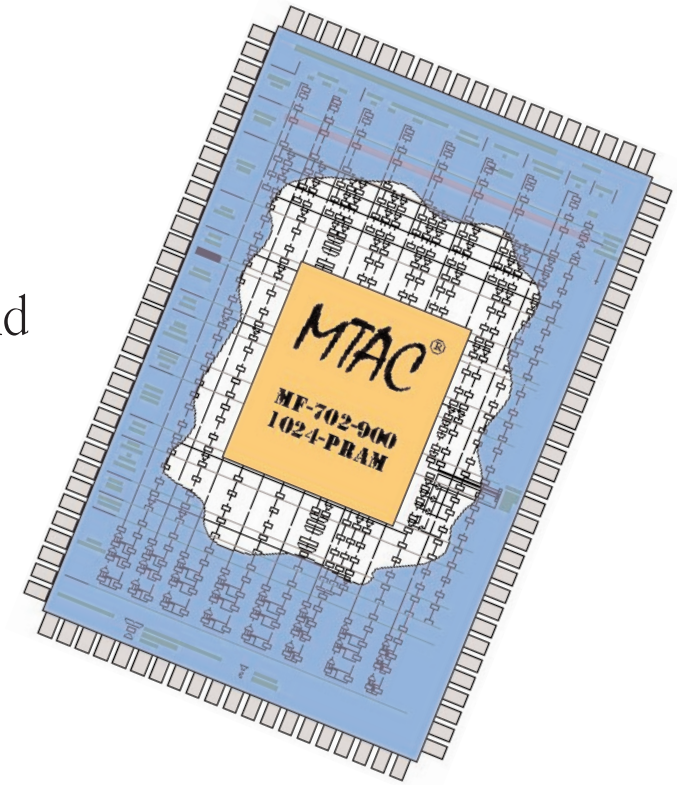
VTT

Implementing PRAM on a Chip



Martti Forsell
Platform Architectures Team
VTT—Technical Research Center of Finland
Oulu, Finland
Martti.Forsell@VTT.Fi

Advanced Parallel Programming:
Models, Languages, Algorithms
March 05, 2007
Linköping, Sweden



Contents

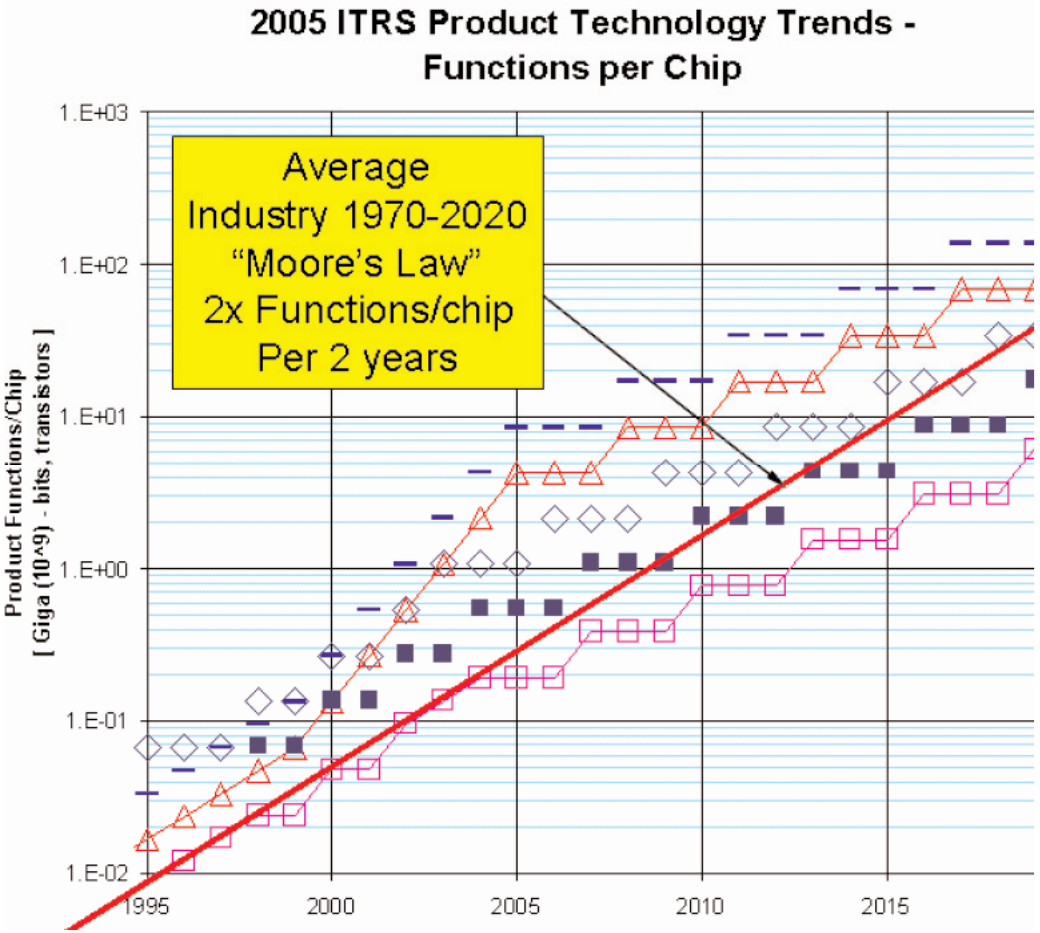
Part 1: Benefits of single chip designs

Vast potential intercommunication bandwidth

- Multiple layers of metal (up to 11 now, in 2020 14)
- Low power consumption (up to 1000 GOPS/W achiveable at 45 nm technology in 2009, more later)

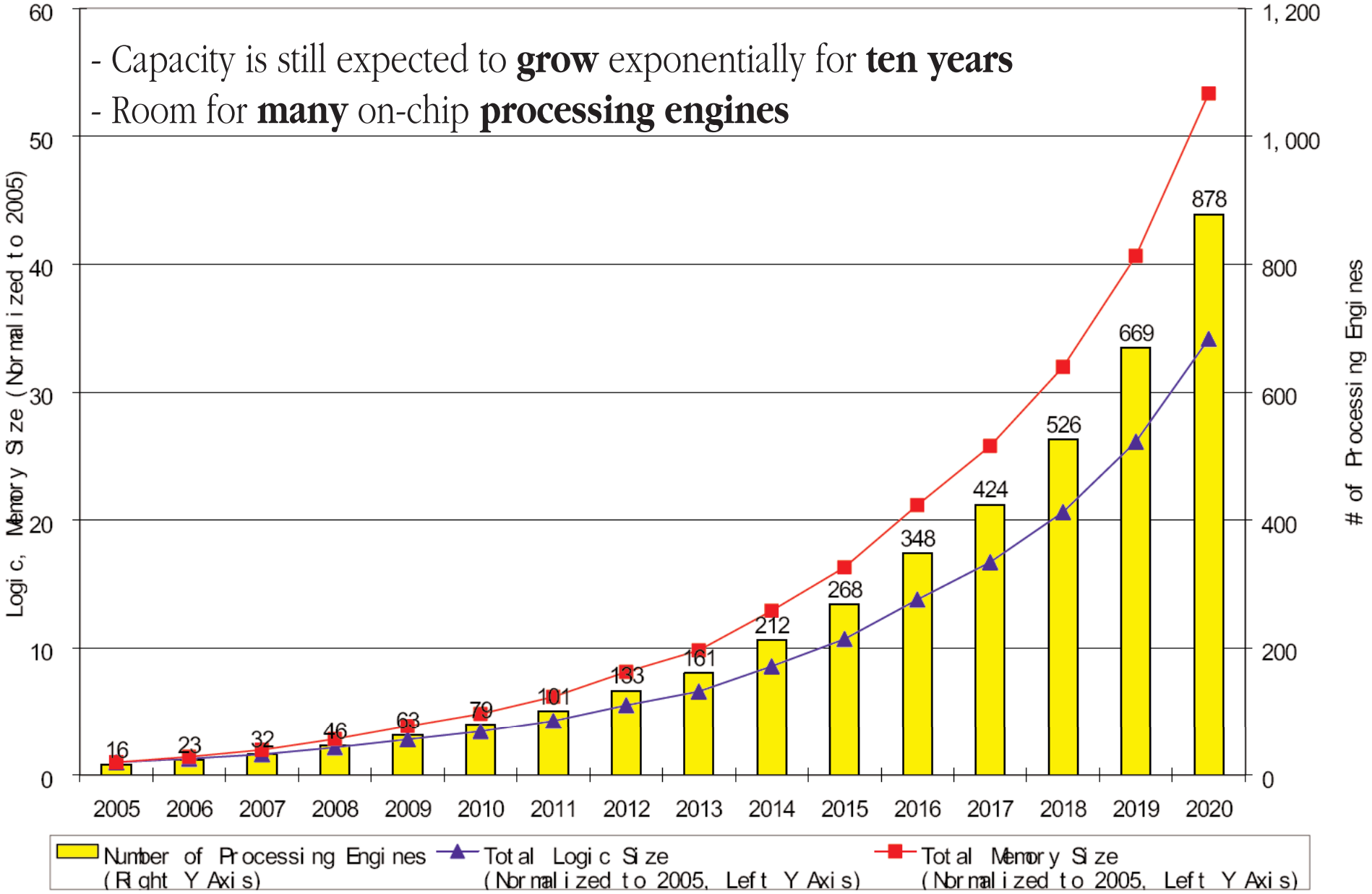
Large amount of logic/storage can be integrated to a single chip

- Extremely compact footprint



SOC-PE DESIGN COMPLEXITY TRENDS

- Capacity is still expected to **grow** exponentially for **ten years**
- Room for **many** on-chip **processing engines**



Challenges of (future) single chip designs

- **Power density** problems (have almost killed the clock rate growth)
- **Crosstalk** between parallel wires (makes interconnect design more complex, slows down communication)
- Increasing **static power** consumption is becoming a problem (see Figure)
- High number of **fast off-chip** connections increasingly difficult to implement (makes design of multichip systems more difficult)

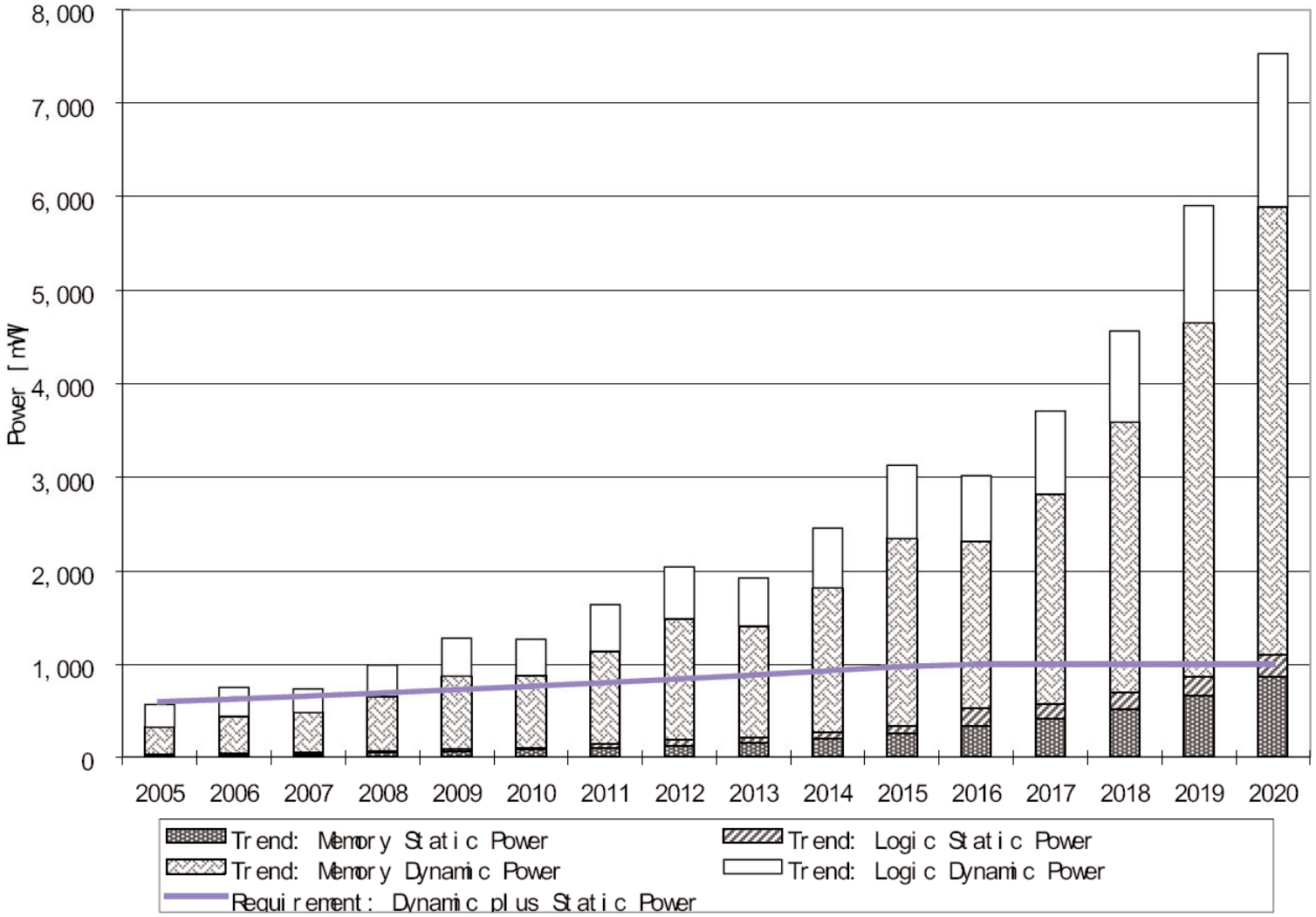
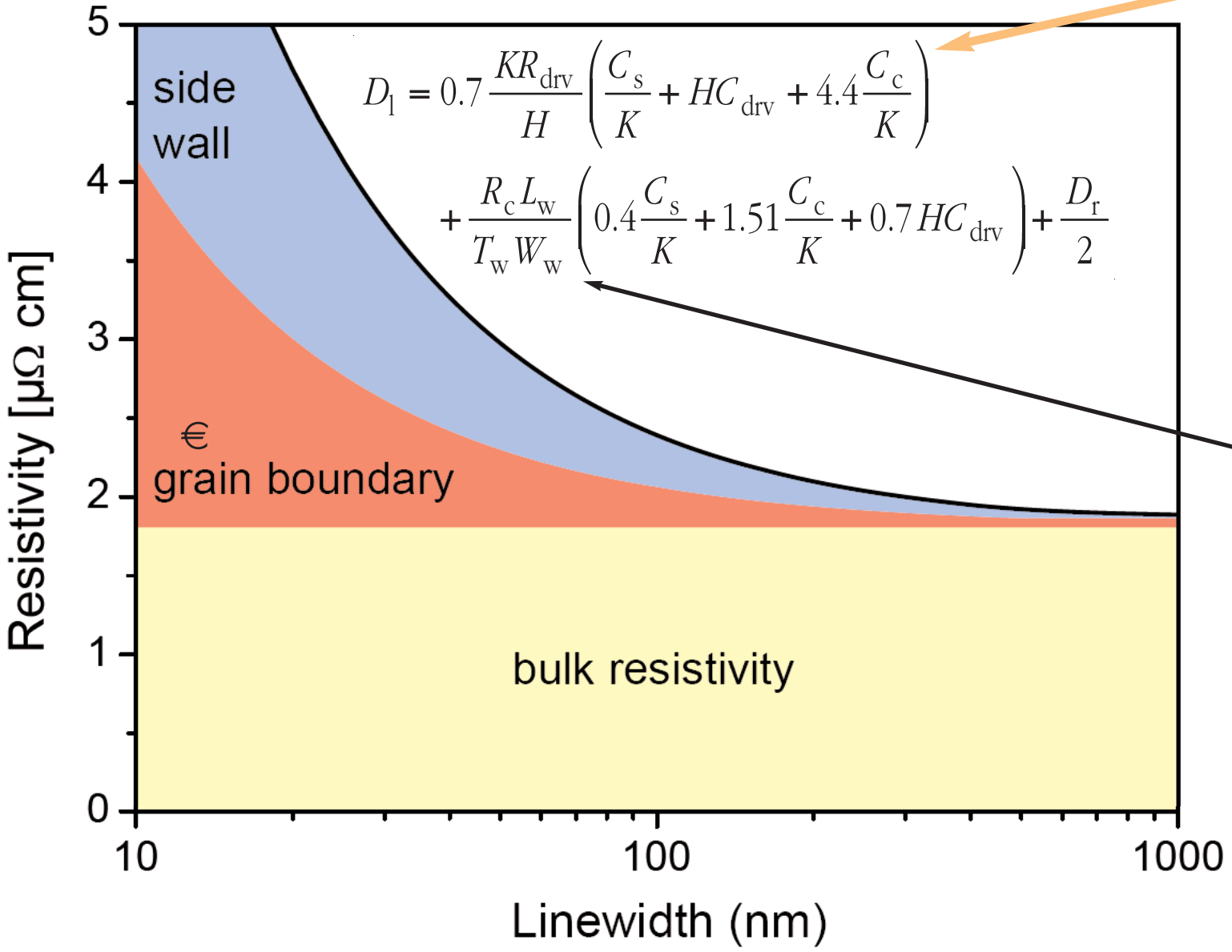


Figure 14 SOC-PE Power Consumption Trends

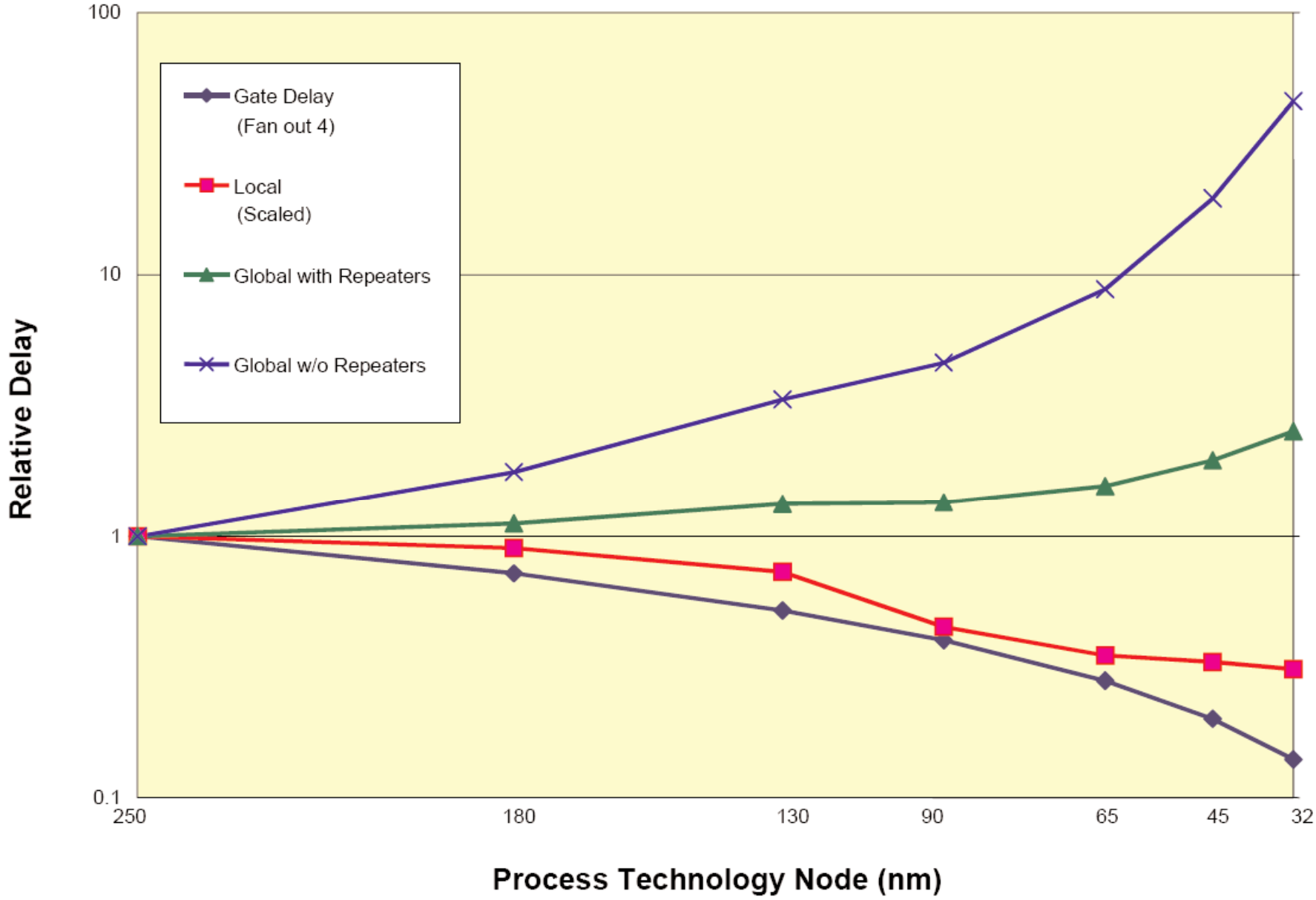
Cu resistivity



Raising conductor resistance slows down signal propagation on parallel interconnects

Resistance =
Resistivity * Length
/ Cross section area

Signal propagation slows down as feature size shrinks



Hierarchical Scaling of interconnect widths (ASIC)

Repeaters and **upscaling** the interconnect **cross-section** as the length increases helps a bit.

The rest must be compensated with **pipelining**.

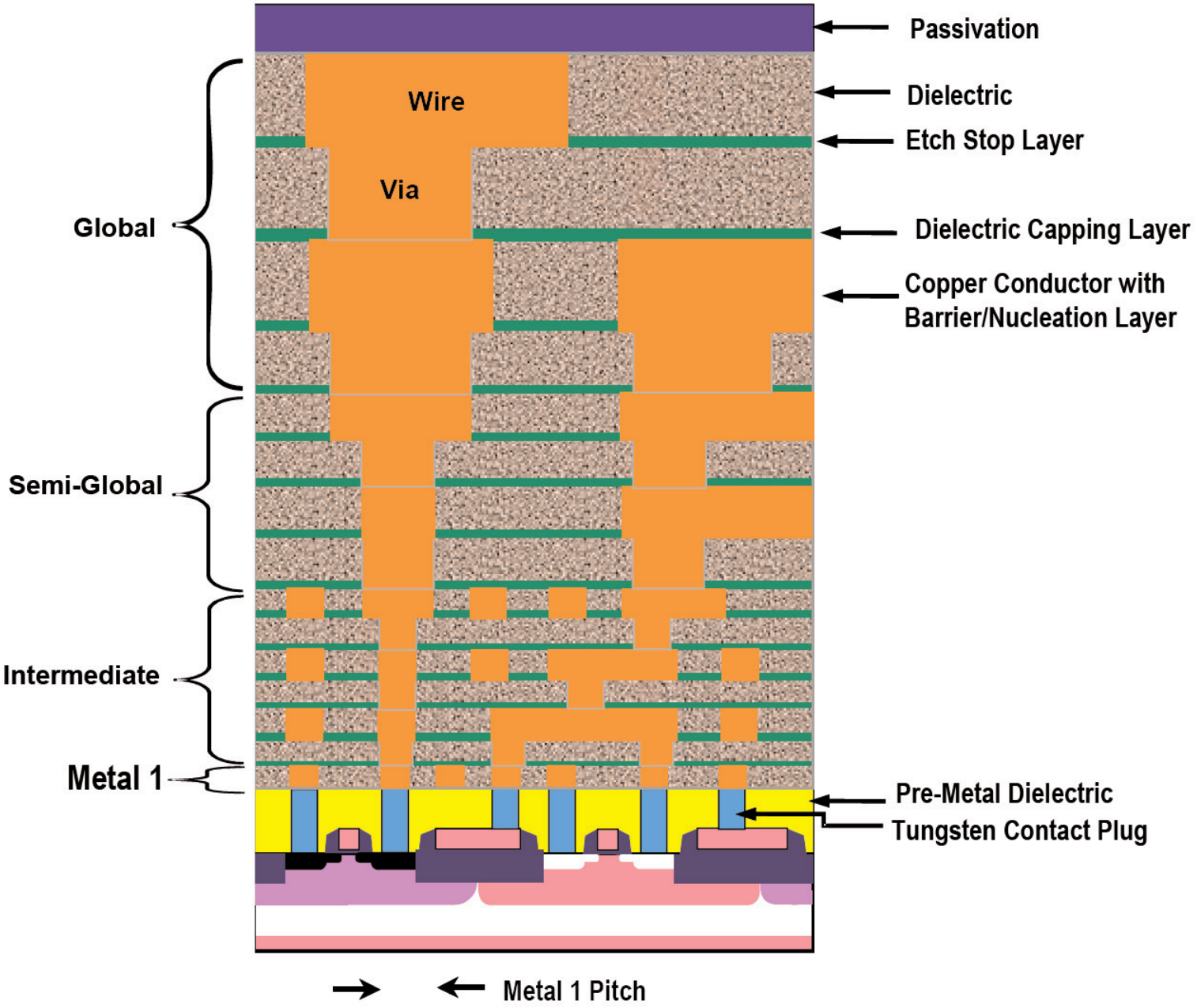
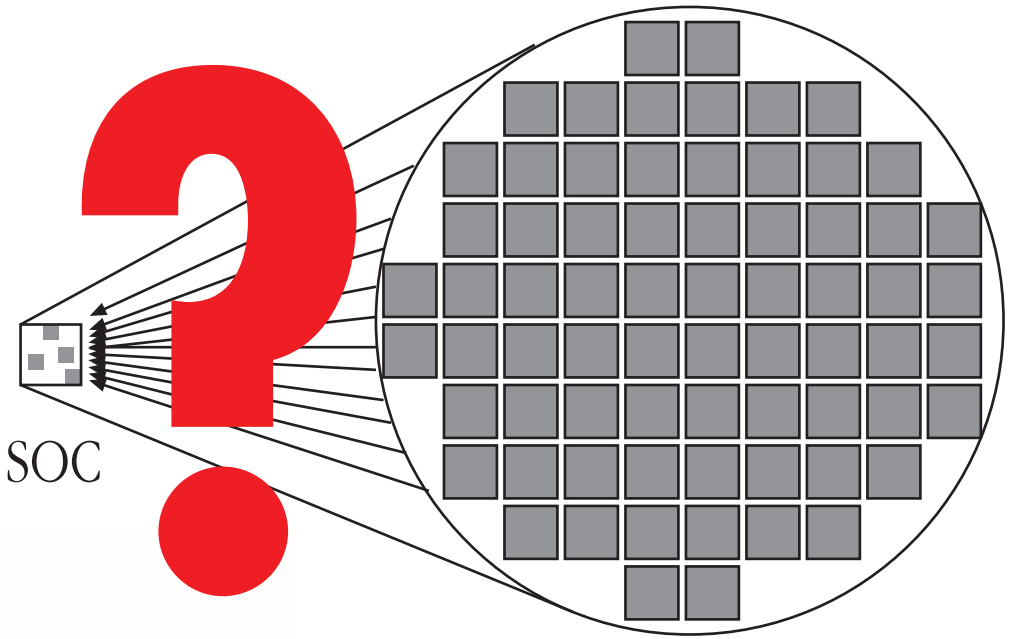


Figure 71 Cross-section of Hierarchical Scaling—ASIC Device

SOC memory bottleneck



Memories for a "balanced" system

Amdahl's rule of thumb:

"A balanced system has 1 MB memory and 1 Mbps I/O capacity per 1 MIPS"

According to this general-purpose SOCs are not realistic within 10 years at least without wafer-scale integration!

Do you believe in Amdahl?

- it is clear that this does not hold for special purpose processing, e.g. DSP
- what about general purpose case?

SW solutions:	Memory usage minimization, in-place algorithms
HW solutions:	Virtual memory, multichip modules/wafer-scale integration

Part 2: Implementing PRAM on a Chip

For PRAM realization we need

- An ability to **hide the latency** of the intercommunication network/memory system (via multithreading and pipelined memory system; caches as they are defined now can not be used for latency hiding)
- **Sufficient bandwidth** for solving an arbitrary routing problem with a high probability
- A method to **avoid hot spots** in communication with a high probability
- Technique to **handle concurrent references** to a single location
- Support for **multioperations**
- **Efficient synchronization** mechanisms

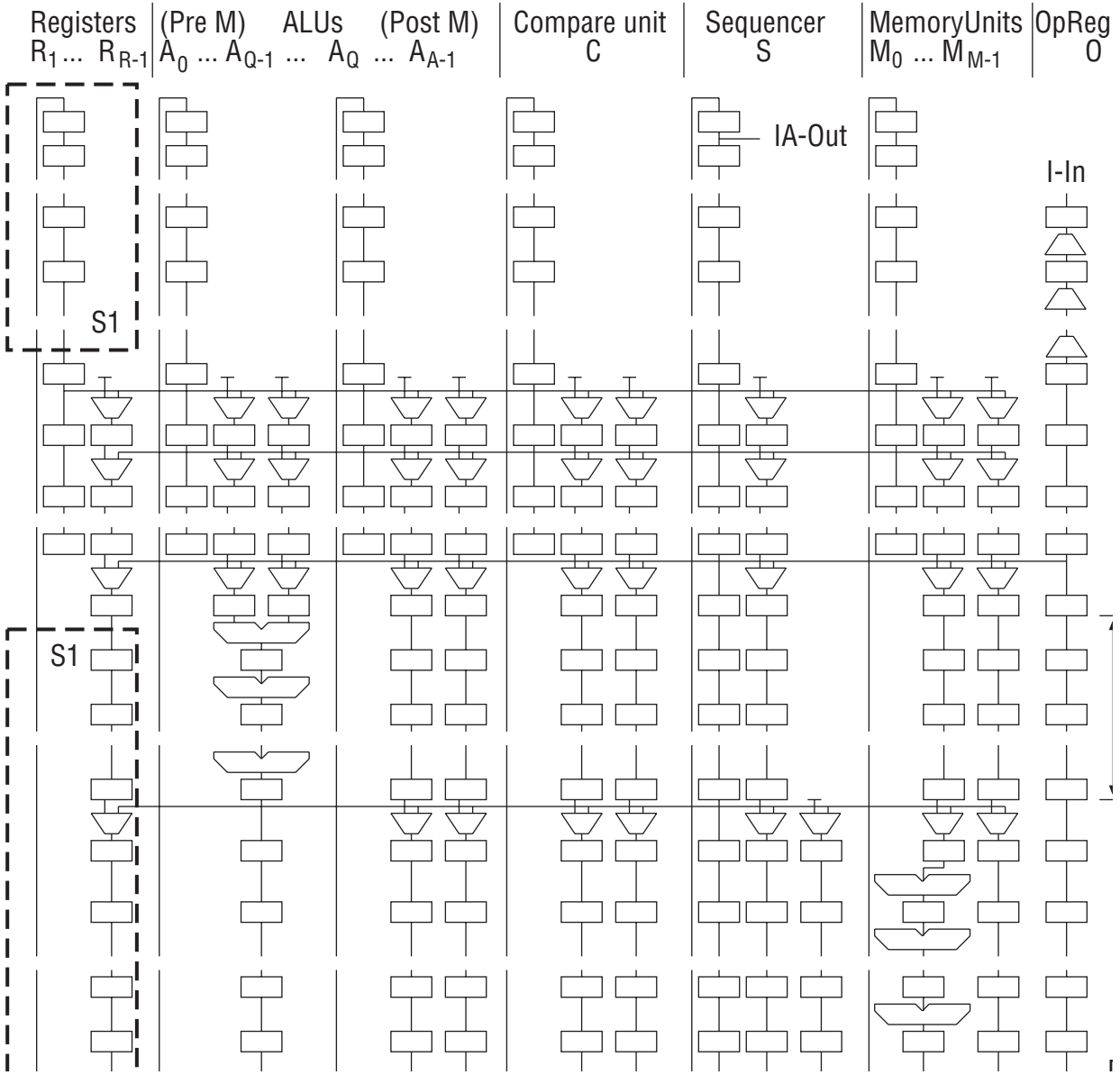
Single chip designs provide good possibilities for PRAM implementation!
In this part we will outline a single chip **EREW PRAM** implementation.

Processor—multithreading, chaining & inter-thread pipelining

For PRAM realization processors needs to be **multithreaded**. Previous presentation suggest also that **interthread superpipelining** and **chaining** should be used.

Such a processor is composed of

- *A* ALUs,
- *M* memory units (MU),
- a hash address calculation unit,
- a compare unit,
- a sequencer,and
- a distributed register file of *R* registers

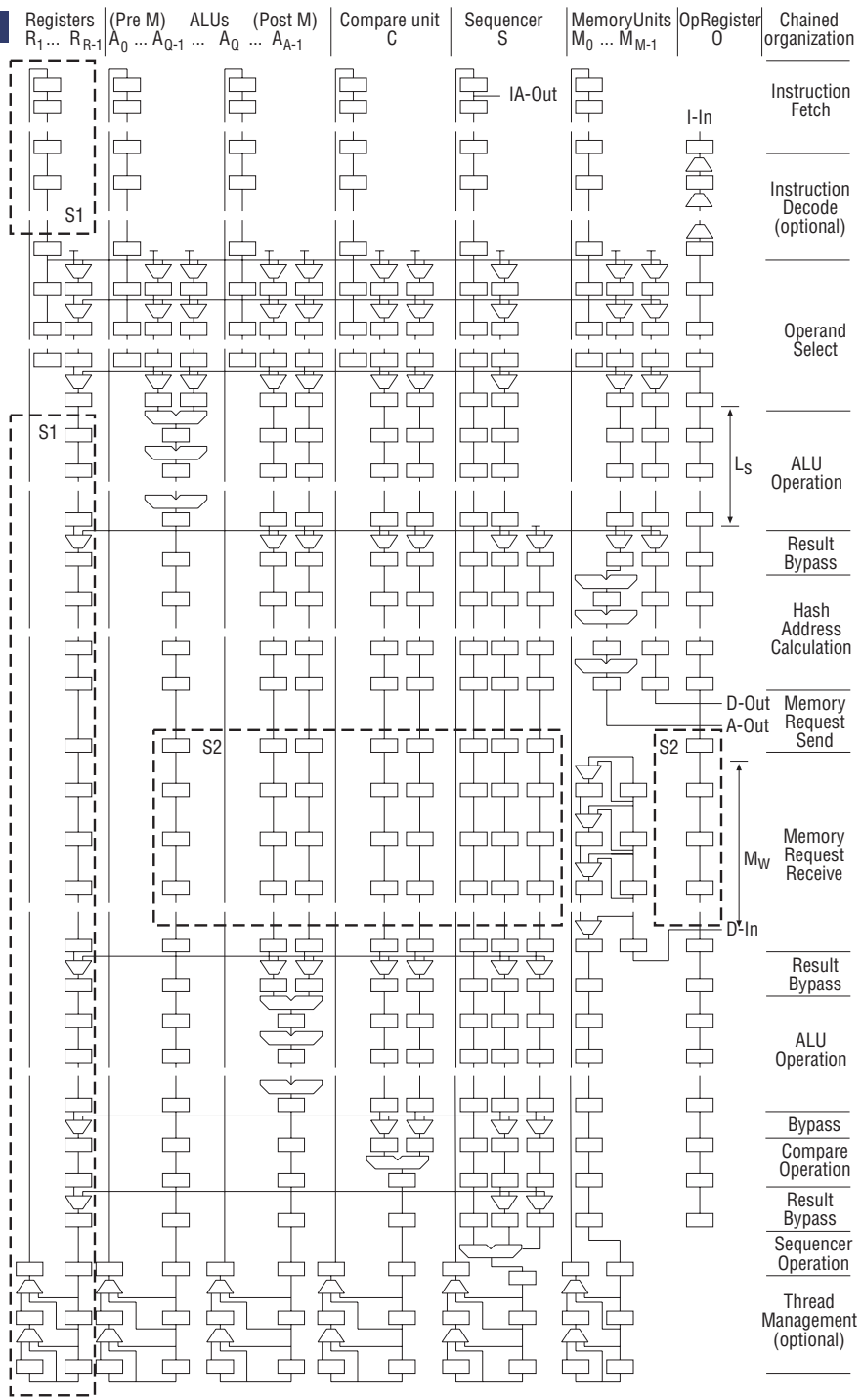


Features & raw performance

- **zero overhead** multithreading
- superpipelining for **clock cycle minimization**
- **hazard-free** pipelining with multiple functional units
- support for totally **cacheless memory** system
- Chaining of functional units to allow execution of **truly sequential** code in one step

According to our tests a such a processor (MTAC) with 4 FUs runs a suite of simple integer benchmarks 2.7 times faster on average than a basic five-stage pipelined RISC processor (DLX) with 4 FUs.

	DLX	MTAC	MTAC	MTAC
FUs	4	6	10	18
MUs	1	1	2	4
Performance	1	4.1	8.1	16.6

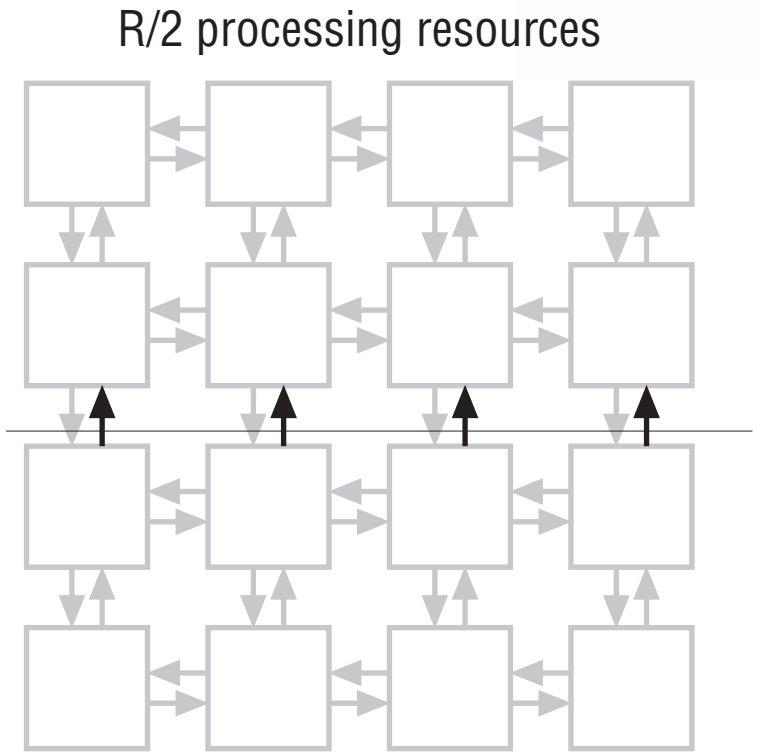


Network—sufficient bandwidth, deadlock/hot spot avoidance, physical feasibility, synchronicity

For PRAM realization intercommunication network needs to have **sufficient bandwidth**, **deadlock free** routing, mechanism for **hot spot avoidance**, and retaining **sychronicity** at step and thread levels. Finally, it needs to be **physically feasible**.

Although **logarithmic diameter** network topologies seem attractive, they are **not** physically **scalable** since the lower bound for a diameter for fixed length link topology is *square root P* rather than *log P*.

This seems to suggest that a **2D mesh** topology would be suitable, but unfortunately the 2D mesh **does not have** enough **bandwidth** for solving a random routing problem.



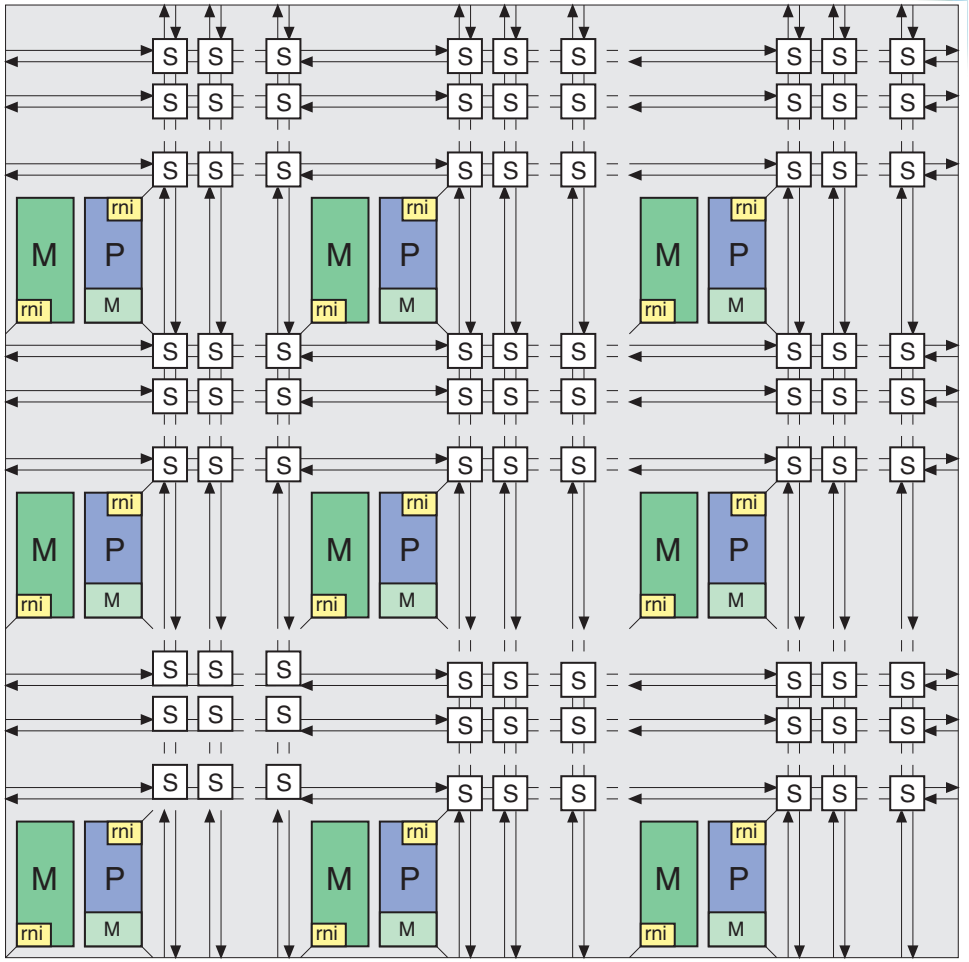
Typical networks, e.g. 2D meshes, do **not** provide **enough bandwidth** for random access patterns.

Solution: Acyclic sparse mesh, bandwidth scaling, hashing, synchronization wave

A scalable communication solution for PRAM realization is e.g. double acyclic sparse mesh network featuring

- *P* processors, *S* switches
- Constant degree switches
- Fixed length intercommunication wiring
- Chip-wide synchronization wave scheme
- Linear bandwidth scaling mechanism

Hashing does not have effect on network design since it can be computed at processors, networks just routes references.



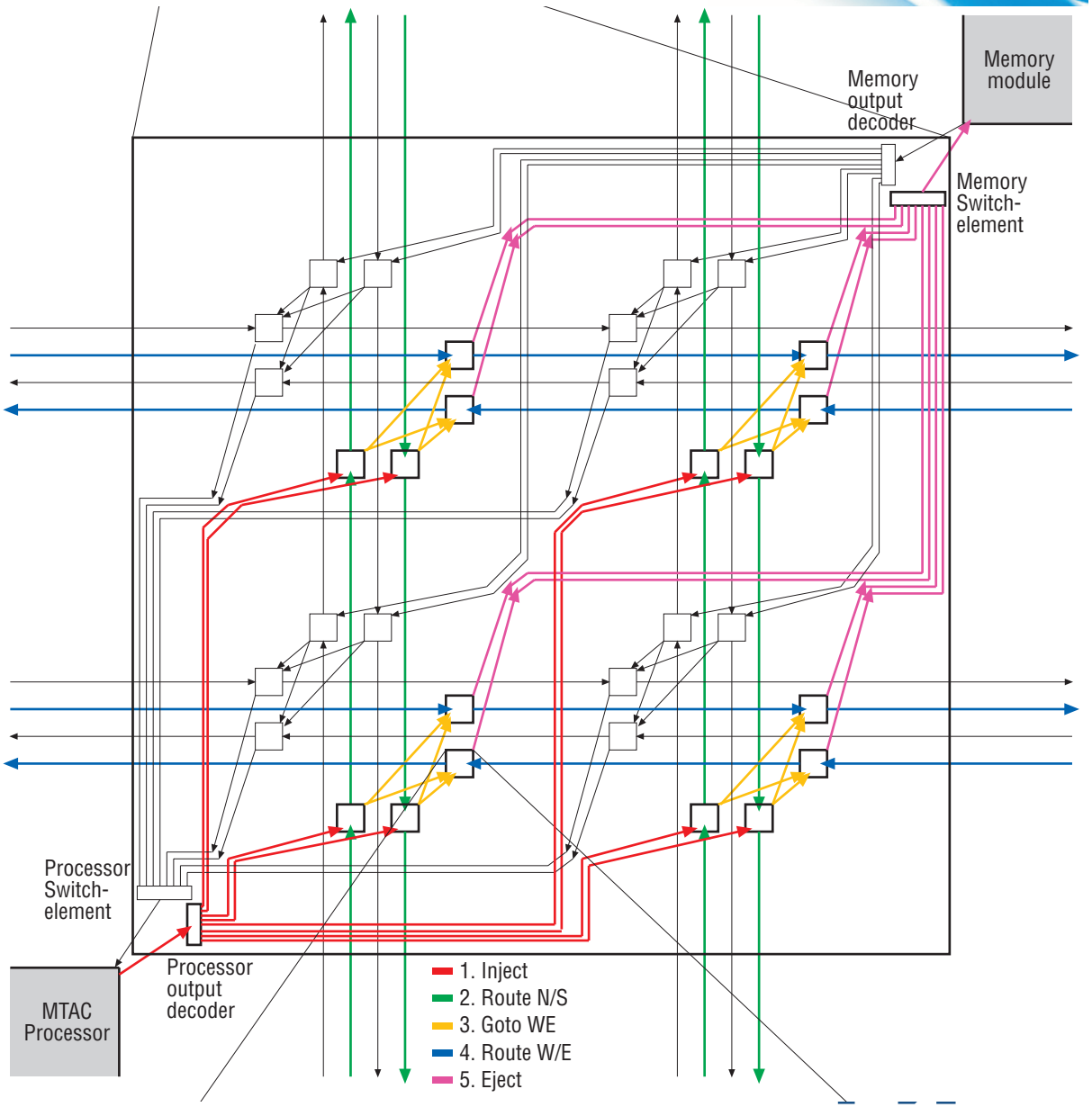
Superswitch

Switches related to each resource pair are grouped as a single **superswitch** to keep switches constant degree.

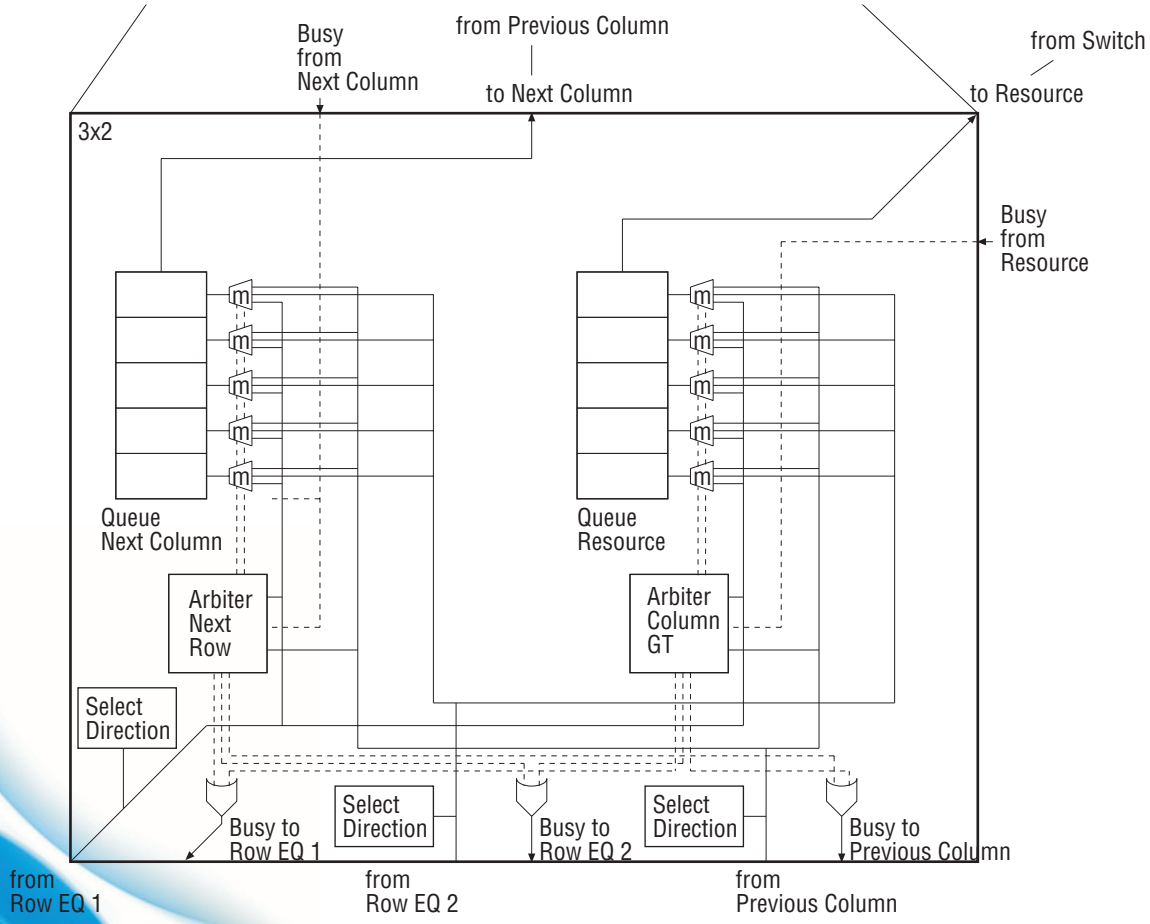
A superswitch is composed of *P*/16 switches, two switch elements and decoders.

A switch is composed of four 2x3 and four 3x2 **switch elements**

Processor and memory can send replies simultaneously
=> Separate lines (networks) for messages going from processors to memories and from memories to processors



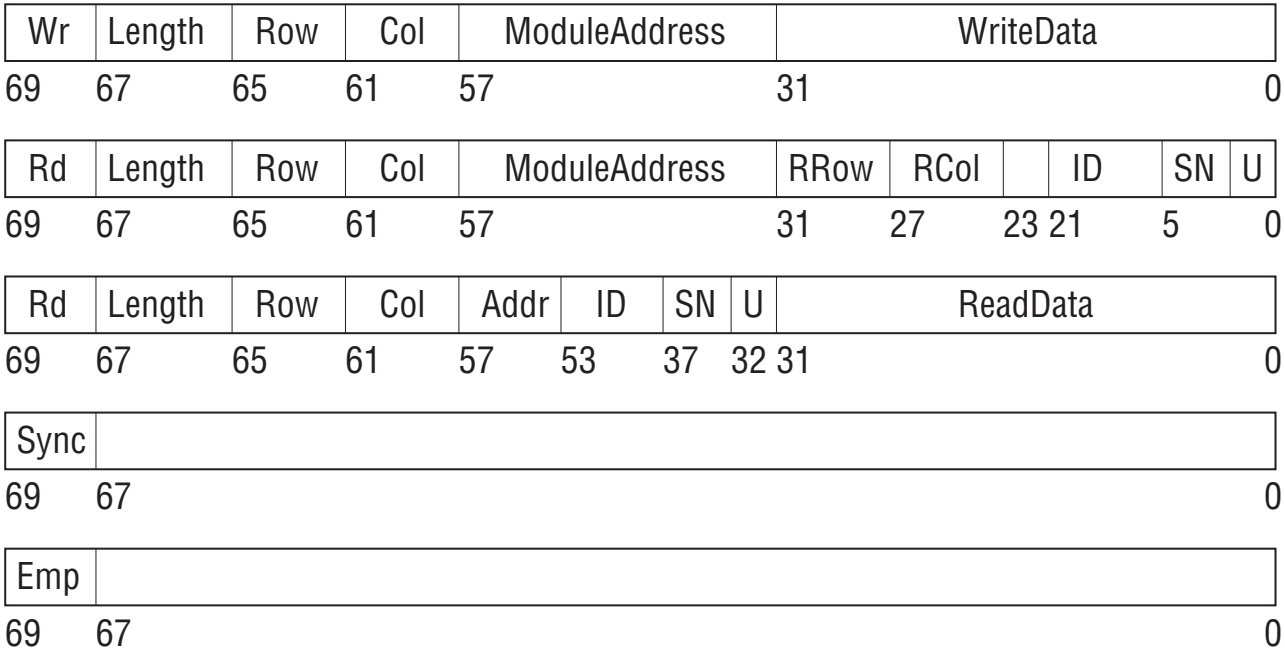
3x2 switch element



A 3x2 switch element is composed of two *Q*-slot **FIFO-queue** elements which are directed by **two arbiter** units

Each FIFO queue accepts up to the number of inputs messages (3 for 3x2 switch element) each clock cycle if there is enough room for them

Messages



Wr	Write message
Rd	Read or reply message
Emp	Empty message
Sync	Synchronization wave message
Length	Log2 (lenght of data reference in bytes)
Row	The row of the target resource
Col	The column of the target resource
Addr	Low 4 bits of the module address
ID	Thread identification number
RRow	The source row of the sender
RCol	The source column of the sender
SN	Switch number for returning messages
U	Unsigned read (0=Unsigned,1=Signed)

The processors can send memory requests (reads and writes) and synchronization messages to the memory modules and modules can send replies and synchronization messages back to processors.

There are four types of messages which are identified by the two-bit header field.

0 = Empty
1 = Read (or read reply)
2 = Write
3 = Synchronization

Deadlock free routing algorithm

Go via two intermediate targets (randomly chosen switches in the superswitches related to sender and target) at the rate of one hop per clock cycle if there is room in the queues:

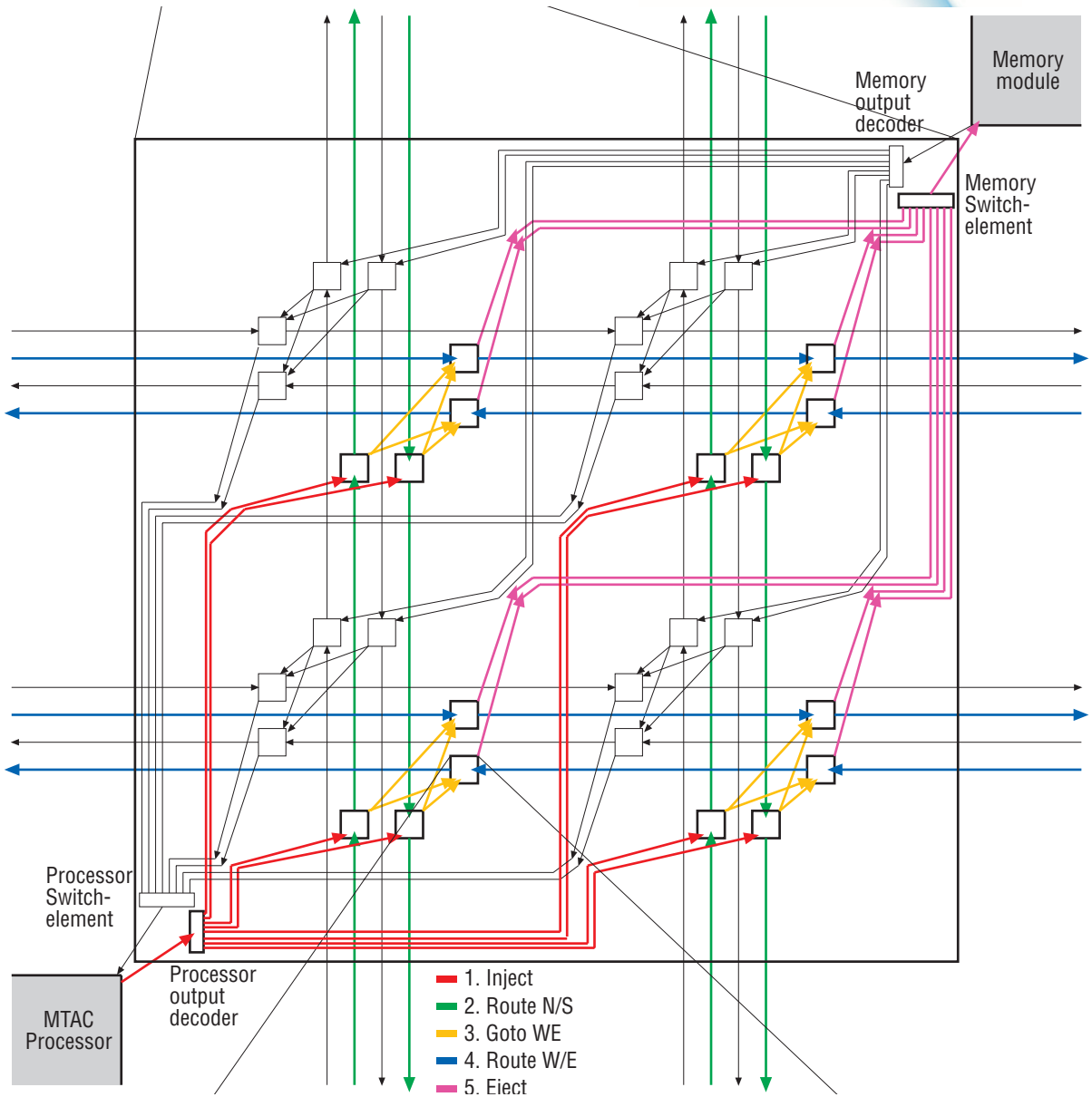
1. Go to the first intermediate target

Go to the second intermediate target greedily:

2. Go to the right row using N/E lines
3. Switch to the W/E network
4. Go go to the right column using W/E lines

5. Go from the second intermediate target to the target resource.

Deadlocks are not possible during communication because the network formed by possible routing paths is acyclic.

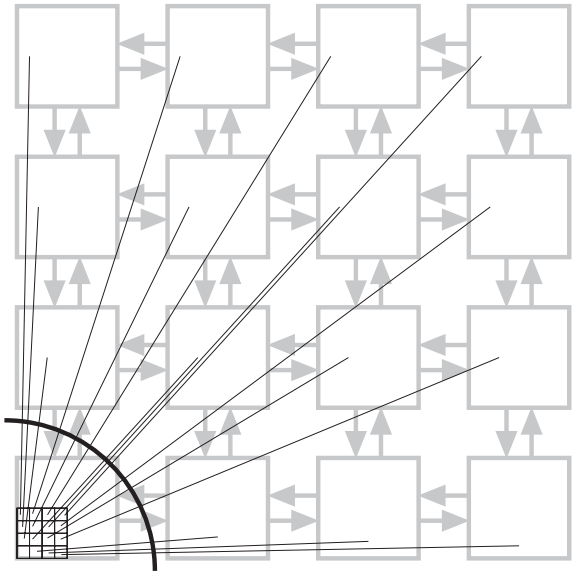


Bandwidth scaling & hot spot avoidance

The network should provide sufficient bandwidth for random routing problem

- A linear bandwidth scaling mechanism
  $S \geq P^2/16$=> Bisection bandwidth $\geq P/2$
- Randomized hashing of memory locations to avoid congestion/hot spots of references

Processors	16	64	256
Switches	16	256	4096

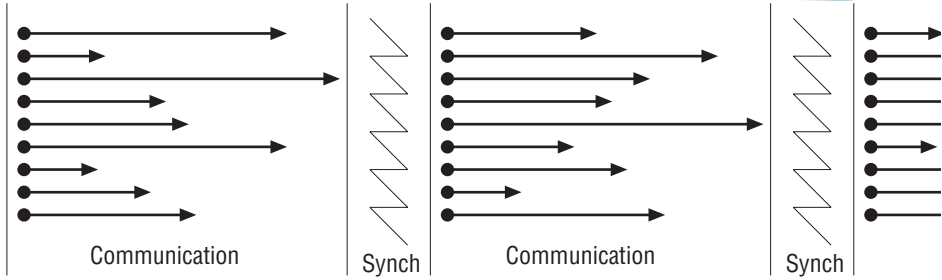


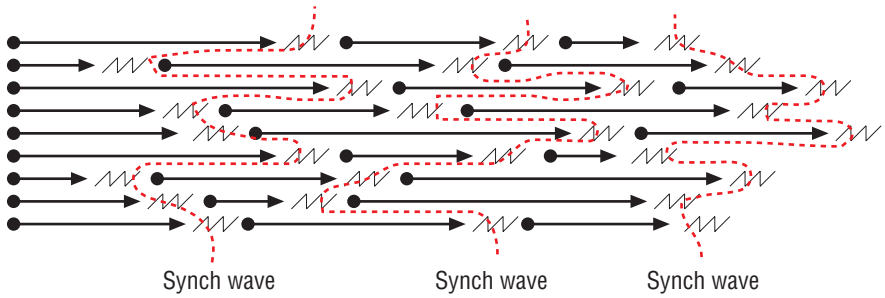
==> For large values of P a 3D sparse mesh is needed!

Synchronization mechanisms

Implicit — *Advanced synchronization wave*

- Used to implement synchronous execution of instructions
- Separates the references belonging to subsequent steps
- Sent by the processors to the memory modules and vice versa
- Principle: when a switch receives a synchronization message from one of its inputs, it waits, until it has received a synchronization message from all of its inputs, then it forwards the synchronization wave to all of its outputs





Explicit — *Arbitrary barrier synchronizations*

- Used to synchronize after thread-private control structures
- Executes in constant number of steps
- Realized via partial CRCW/active memory operations

Area reduced alternative

Eliminate all switches except those on the diagonal of superswitches

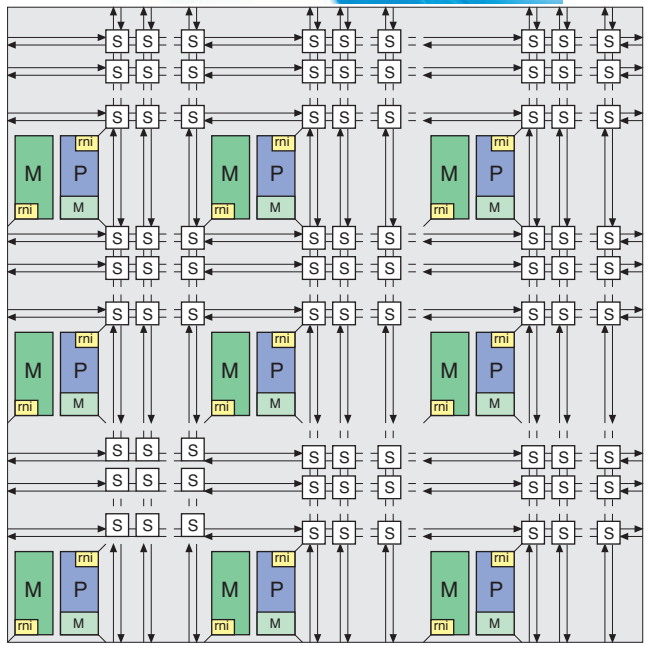
- Topology becomes $\sqrt{S/P}$ parallel acyclic two-dimensional meshes
- Area of superswitches reduces by the factor of $\sqrt{S/P}$.
- Can be seen as a 3D sparse mesh that is flatten onto 2D surface.
- The diameter of the network reduces to $O(\sqrt{P})$
- Consists of $O(P^{1.5})$ switches => can hide the latency of messages sent by $O(P)$ processors

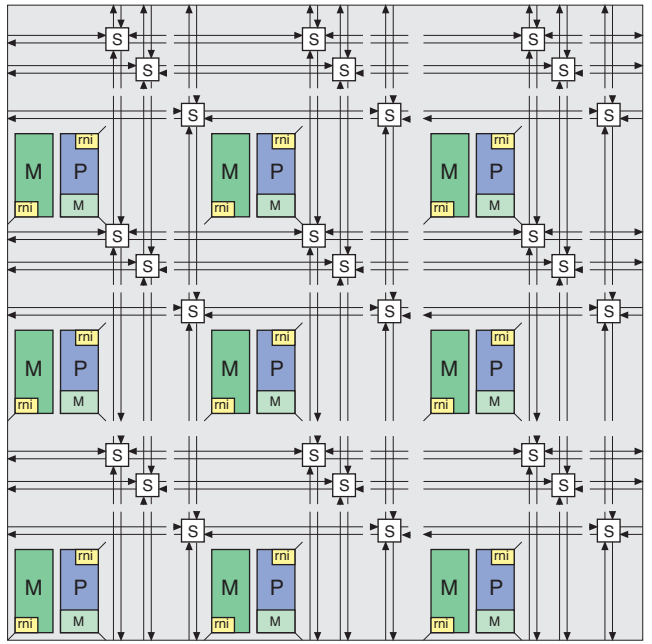
€

- Routing like in the baseline network, but both intermediate targets must belong to the same mesh

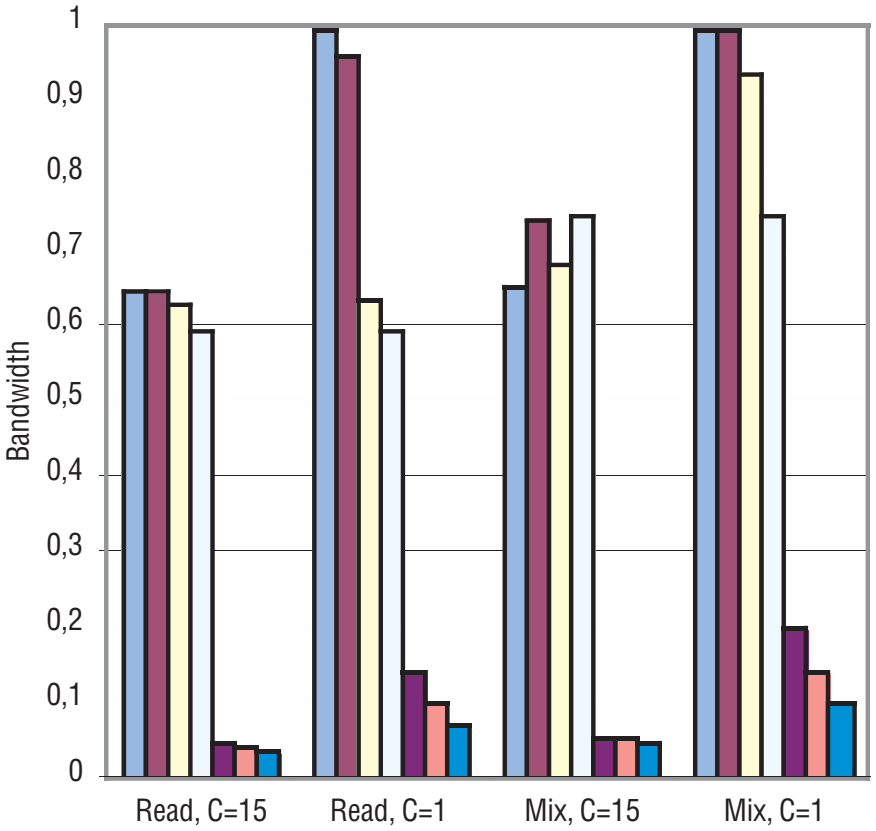
In practice the complexity of superswitches becomes more acceptable

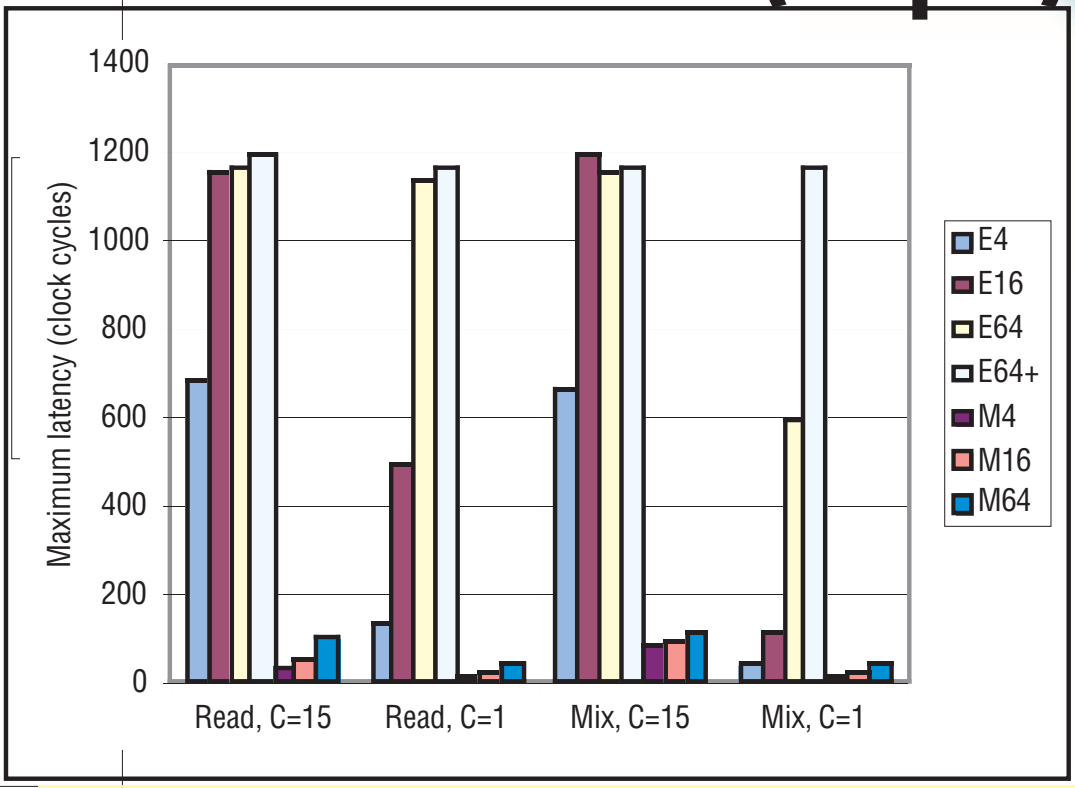
Processors	16	64	256
Switches	16	128	1024





Evaluation results (Eclipse)



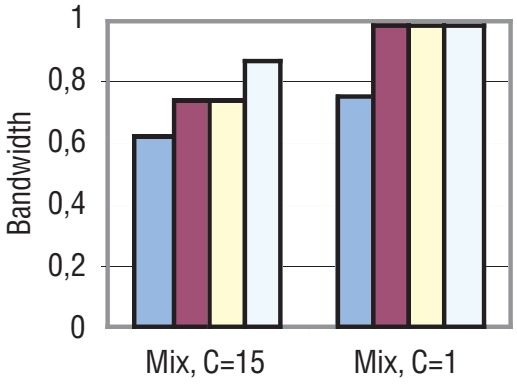


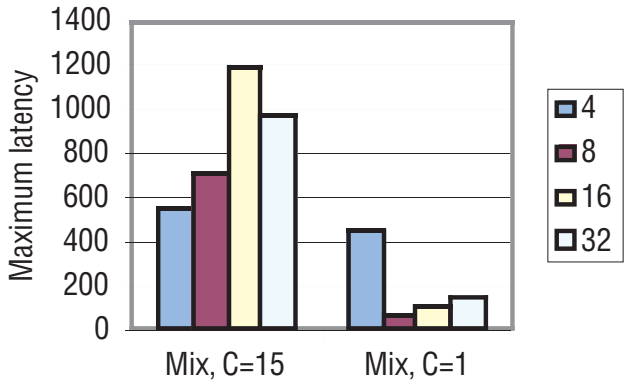
	E4	E16	E64	E64+	M4	M16	M64
Processors	4	16	64	64	4	16	64
Switches	4	16	256	128	4	16	64
Word length	32	32	32	32	32	32	32
FIFOs	16	16	16	16	16	16	16
Area reduced	no	no	no	yes	-	-	-

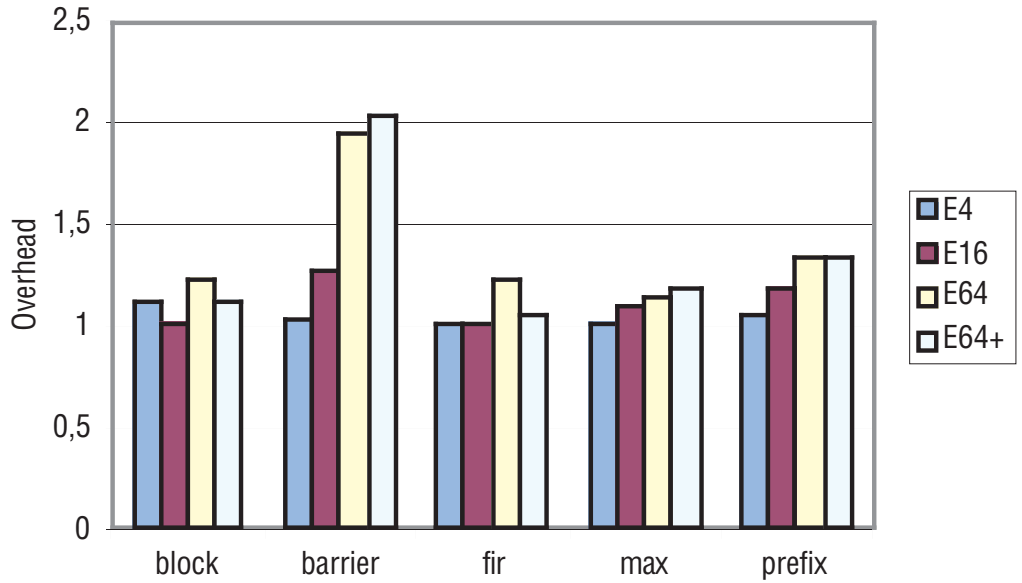
The **relative bandwidth** compared to the ideal case and **maximum latency** in random communication

- *C*=the processor/memory clock speed ratio
- *Read*=read test
- *Mix*=read and write mix in 2:1.

Evaluation results 2







Relative bandwidth compared to the ideal case and **latency** as the functions of the length of output queues.

Execution time overhead with real parallel programs in respect to ideal PRAM.

The **silicon area excl. wiring** of various switch logic blocks synthesized with a **0.18 um process** as mm^2

Switch element 3x2 (E4, E16, E64, E64+)	0.242
Switch (E4, E16, E64, E64+)	2.126
Processor output decoder 1x8 (E4, E16, E64, E64+)	0.104
Processor switch element 8x1 (E4, E16, E64, E64+)	0.465
Superswitch 1x1 (E4, E16)	2.376
Superswitch 2x2 (E64)	9.643
Superswitch 2x2 (E64+)	4.822
Switch (estimated for M4, M16. M64)	1.015

Preliminary area model for ECLIPSE:

- Communication network logic takes 0.6% of the overall logic area in a E16 with 1.5MB SRAM per processor.

Implementation estimations

Based on our **performance-area-power model** relying on our analytical modeling and ITRS 2005 we can estimate that:

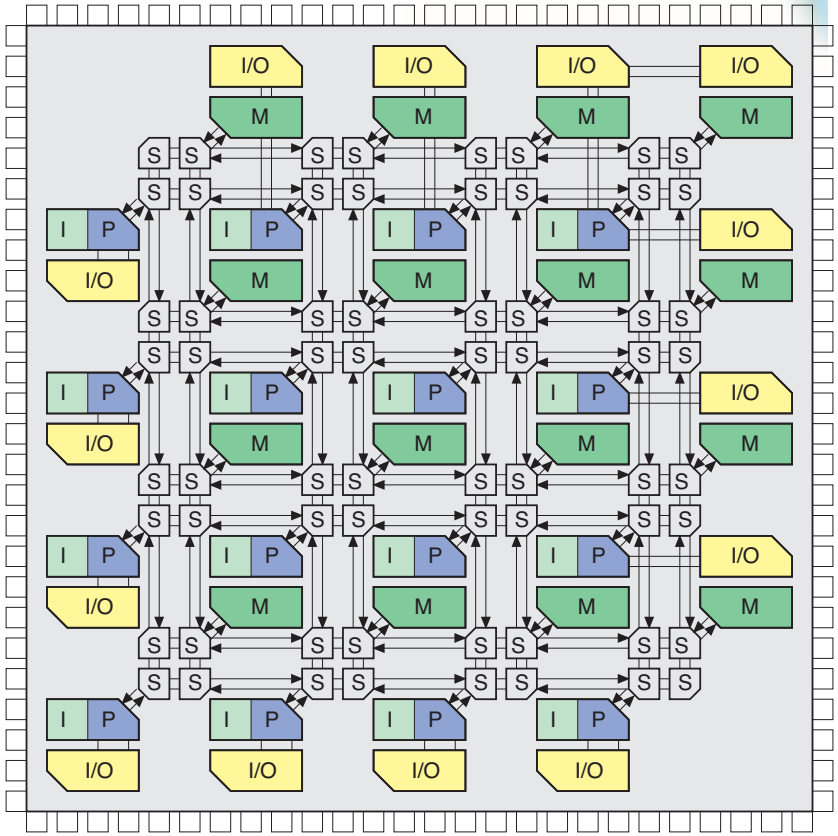
- A 16-processor Eclipse with 1 MB data memory per processor fits into 170 mm^2 with state of the art 65 nm technology
- An version without power-saving techniques would consume about 250 W power with the clock frequency of 2 GHz

Part 2: Extension to CRCW PRAM

Part 1 outlined a single chip EREW PRAM realization

Missing: Ability to provide CRCW memory access

CRCW provide logarithmically faster execution of certain algorithms.



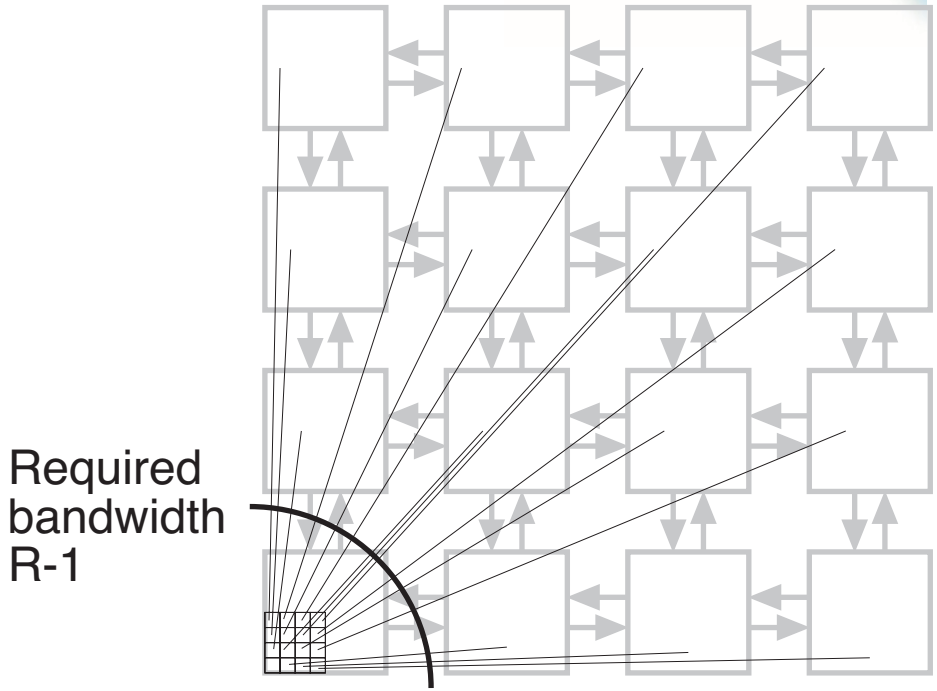
Existing solutions

Sequentilization of references on memory banks

- Causes severe congestion since an excessive amount of references gets routed to the same target location
- Requires fast SRAM memory
- Used e.g. in the baseline Eclipse architecture

Combining of messages during transport

- Combine messages that are targeted to the same location
- Requires a separate sorting phase prior to actual routing phase => decreased performance and increased complexity
- Used e.g. in Ranade's algorithm for shared memory emulation and in SB-PRAM



Approach

Can CRCW be implemented easily on a top of existing solutions?

- **Possible solutions:** 1. Combining
  2. Combining-serializing hybrid

We will describe the latter one since it seems currently more promising. It will be based on **step caches**.

Step caches are caches which **retain data** inserted to them **only until the end of on-going step** of execution.

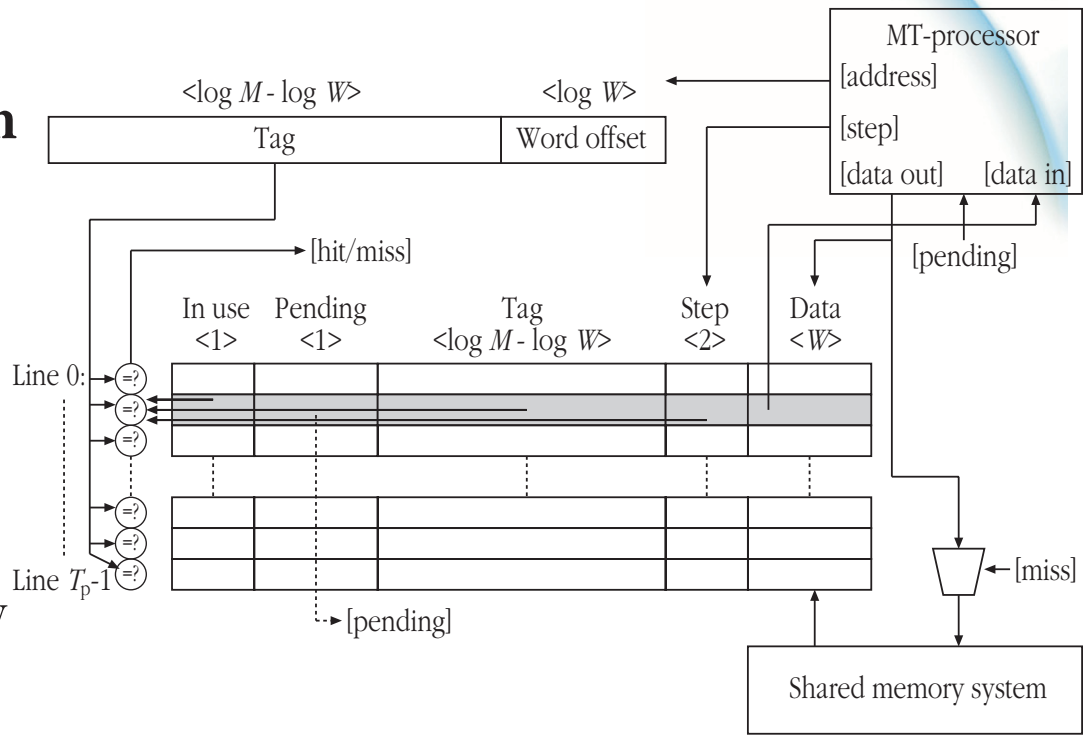
Step caches—A novel class of caches

A **C line, single W-bit word per line cache with**

- two special fields (pending, step)
- a slightly modified control logic
- step-aware replacement policy

Fields of a cache line:

In use	A bit indicating that the line is in use.
Pending	A bit indicating that the data is currently being retrieved from the memory
Tag	The address tag of the line
Step	Two least significant bits of the step of the data write
Data	Storage for a data word



A fully associative step cache consists of

- *C* lines
- *C* comparators
- *C* to 1 multiplexer
- simplified decay logic matched for a T_p-threaded processor attached to a *M*-word memory.

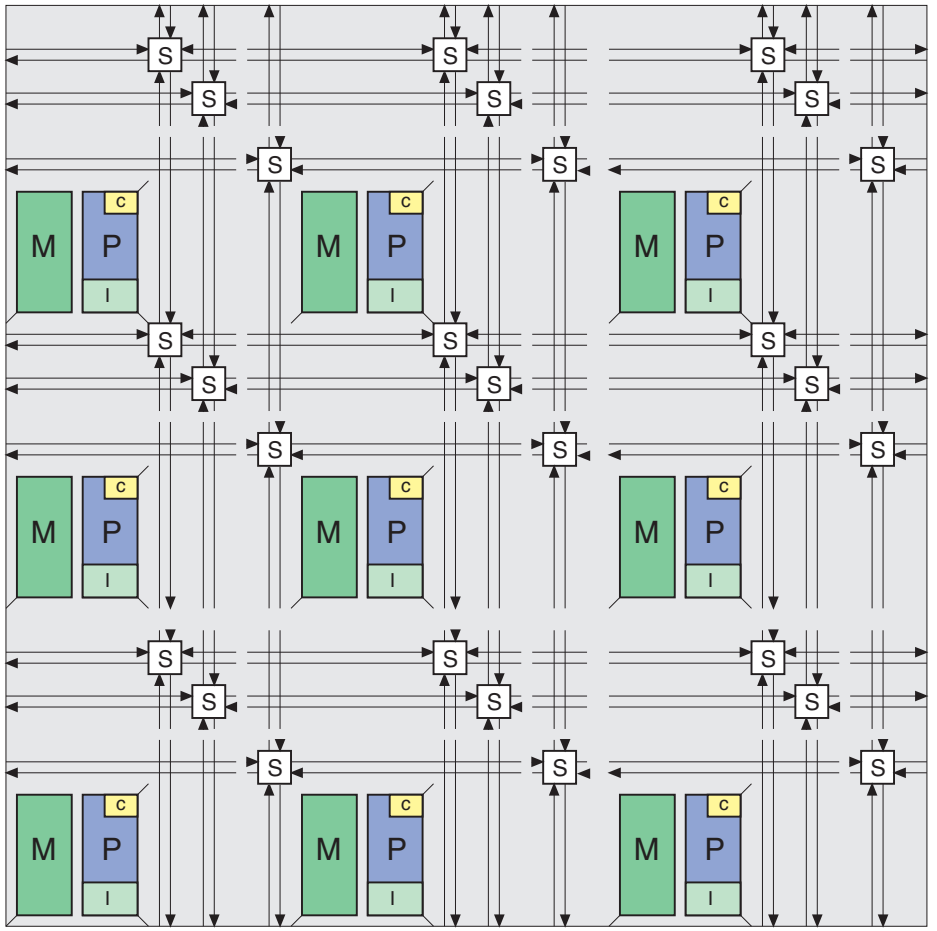
Implementing concurrent access

Consider a shared memory MP-SOC with P T_p-threaded processors attached to a shared memory:

Add fully associative T_p-line **step caches** between processors and the memory system.

- Step caches **filter out** step-wisely **all but** the **first** reference for each referred location and **avoid** conflict **misses** due to sufficient **capacity** and step-aware **replacement** policy

- **Traffic** generated by concurrent accesses **drops** radically: **At most** P **references** can occur per a memory location **per** single **step**.



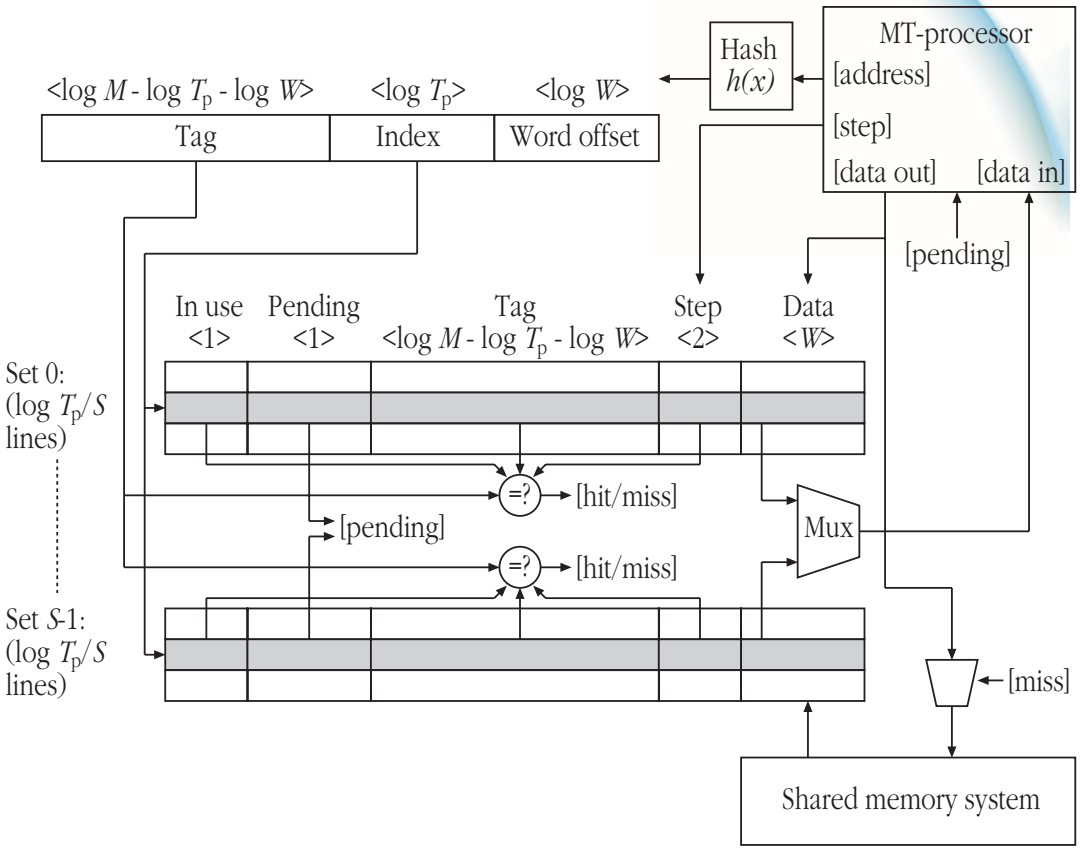
Reducing the associativity and size

Allow an initiated **sequence** of references to a memory location **to be interrupted** by another reference to a different location **if** the **capacity** of the referred set of cache lines is **exceeded**

In order to **distribute** this kind of **conflicts** (almost) **evenly** over the cache lines, access **addresses are hashed** with a randomly selected hashing function

- Implements **concurrent** memory **access** with a **low overhead** with respect to the fully associative solution with a **high probability**

- **Cuts** the **size dependence** of step caches on the number of threads per processor, T_p, but with the cost of increased memory traffic



S-way set associative step cache

Evaluation on Eclipse MP-SOC frame-work

Configurations:

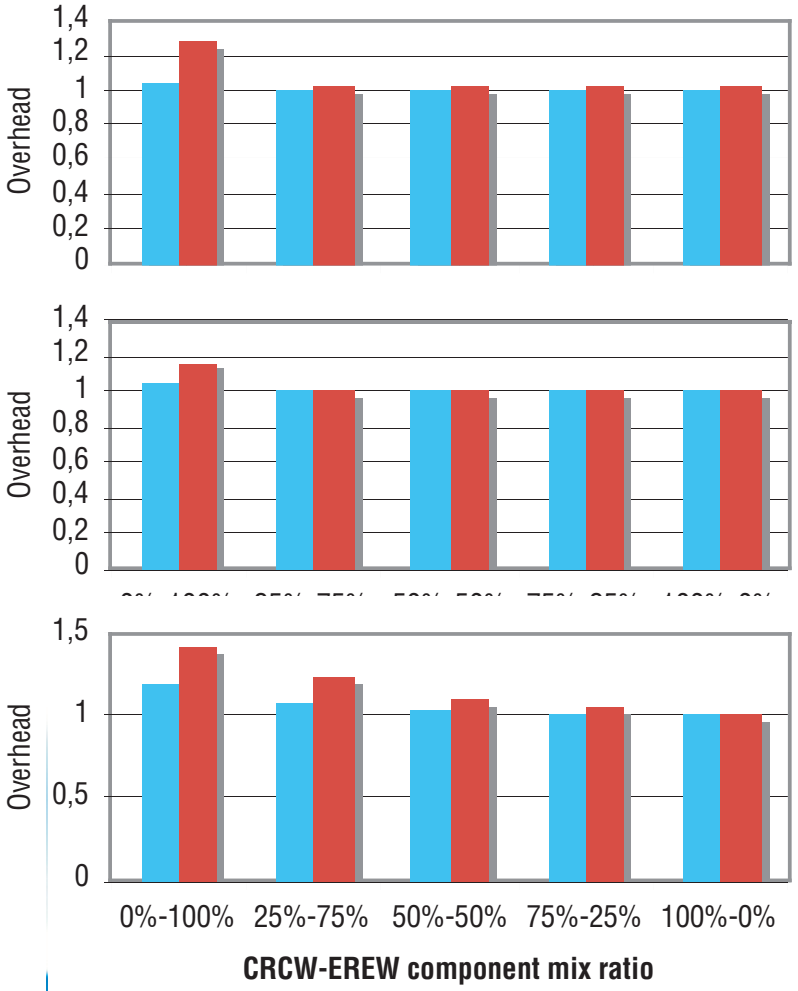
	E4	E16	E64
Processors	4	16	64
Threads per processor	512	512	512
Functional units	5	5	5
Bank access time	1 c	1 c	1 c
Bank cycle time	1 c	1 c	1 c
Length of FIFOs	16	16	16

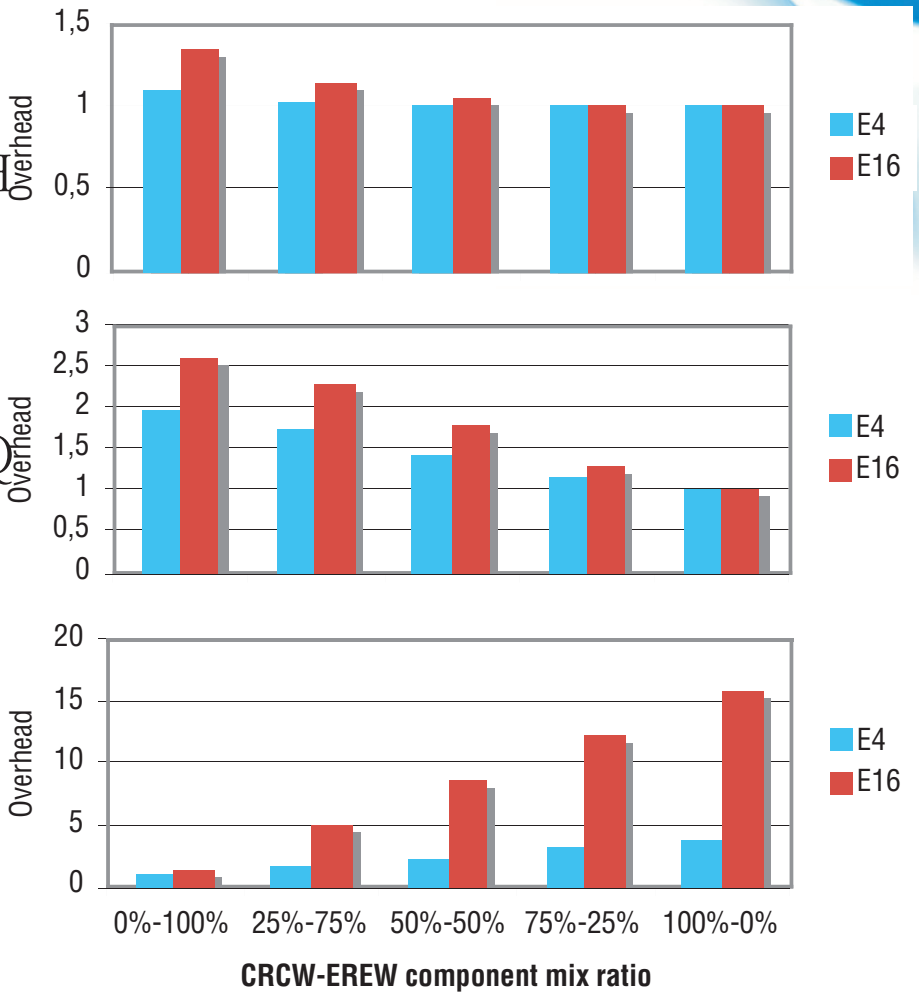
A two-component benchmark sheme:

CRCW	A parallel program component that **reads and writes concurrently** a given sequence of locations in the shared memory
EREW	A parallel program component issuing a memory pattern extract-ed from random SPEC CPU 2000 data reference trace on Alpha architecture for each thread.

Components are mixed in 0%-100%, 25%-75%, 50%-50%, 75%-25% and 100%-0% ratios to form benchmarks

Execution time overhead



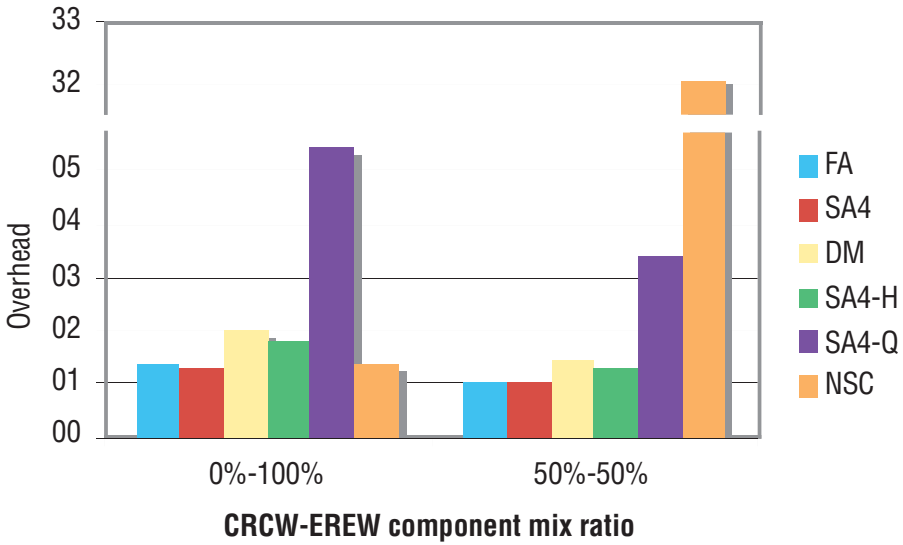


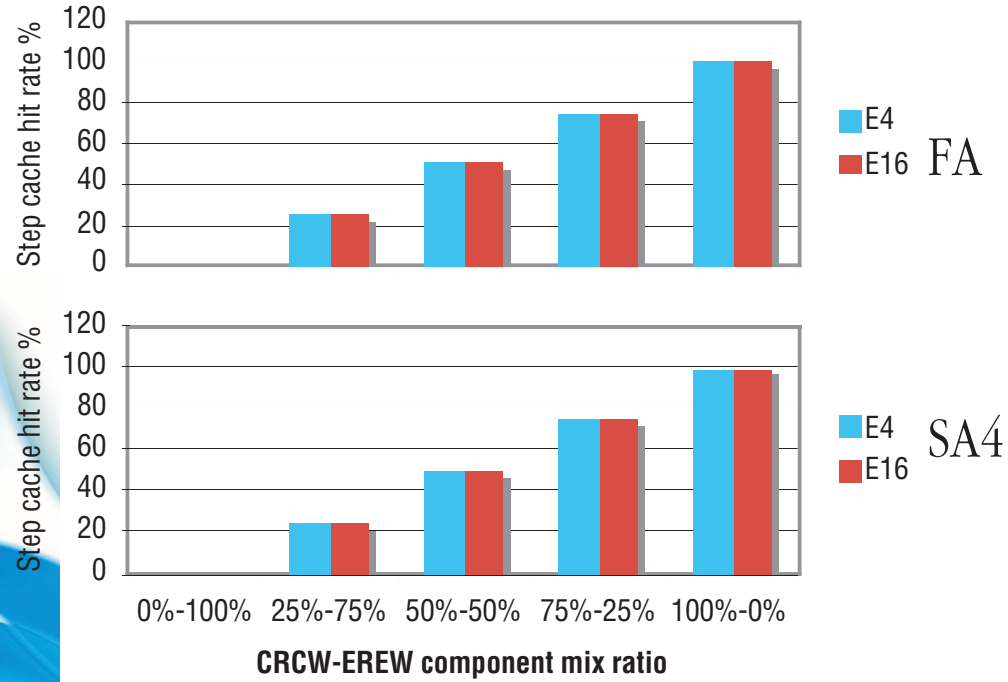
FA=fully associative, SA4=4-way set associative, DM=direct mapped, SA4-H=4-way set associative half size, SA4-Q=4-way set associative quarter size, NSC=non-step cached.

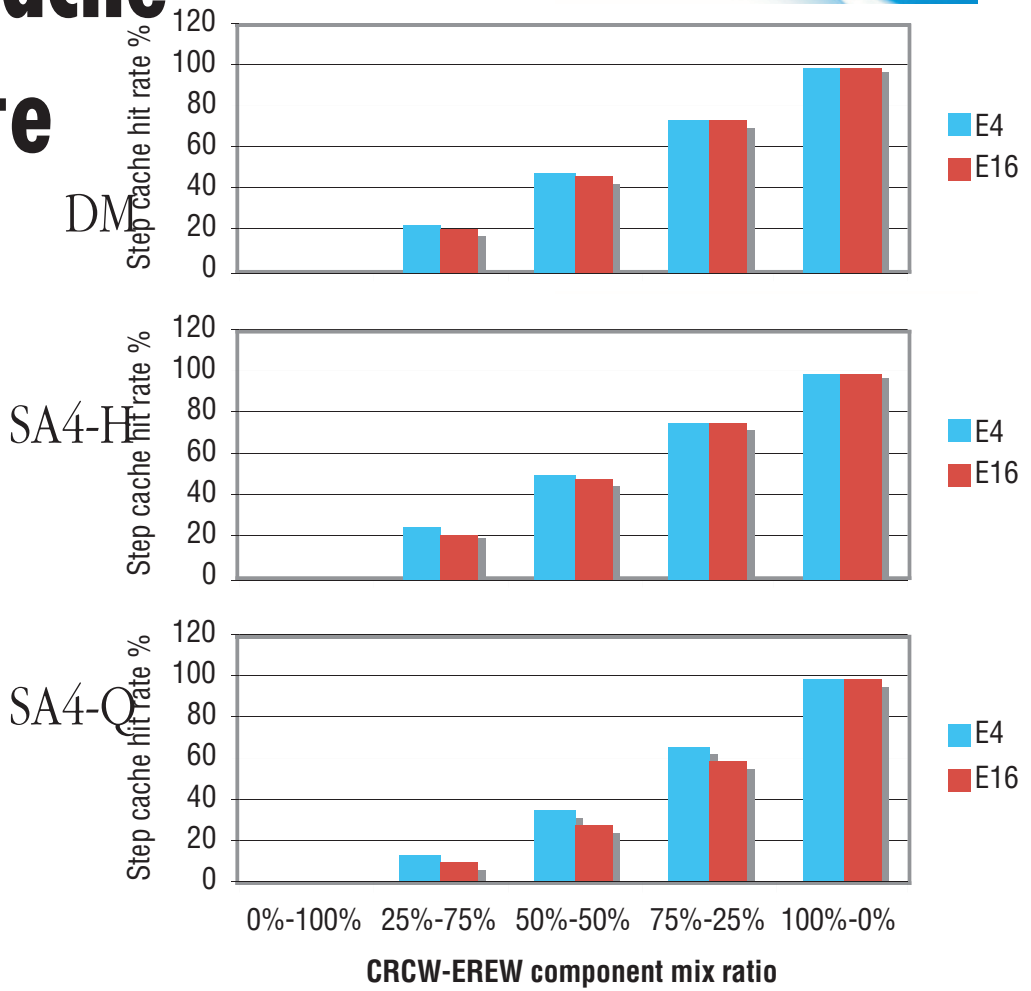
- Decreases as the degree of CRCW increases
- Remains low for FA and SA4
- Comes close to #processors for NSC
- Increases slowly as the #processors increases

VTT

Step cache hit rate







- Comes close to the mix ratio except in SA4-Q
- Decreases slowly as #processors increases

FA=fully associative, SA4=4-way set associative, DM=direct mapped, SA4-H=4-way set associative half size, SA4-Q=4-way set associative quarter size

Area overhead of CRCW

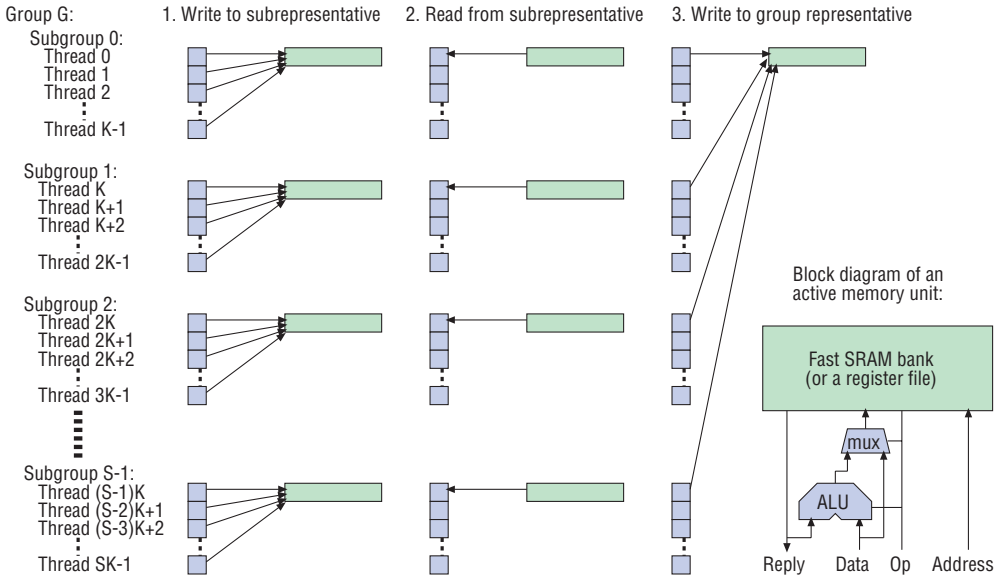
According to our performance-area-power model:

- The silicon area overhead of the CRCW implementation using step caches is typically **less than 0.5 %**.

Part 3: Extension to MCRCW

Can MCRCW be implemented on a top of the solution described in Part 2?

The answer is yes, although the implementation is not anymore trivial.

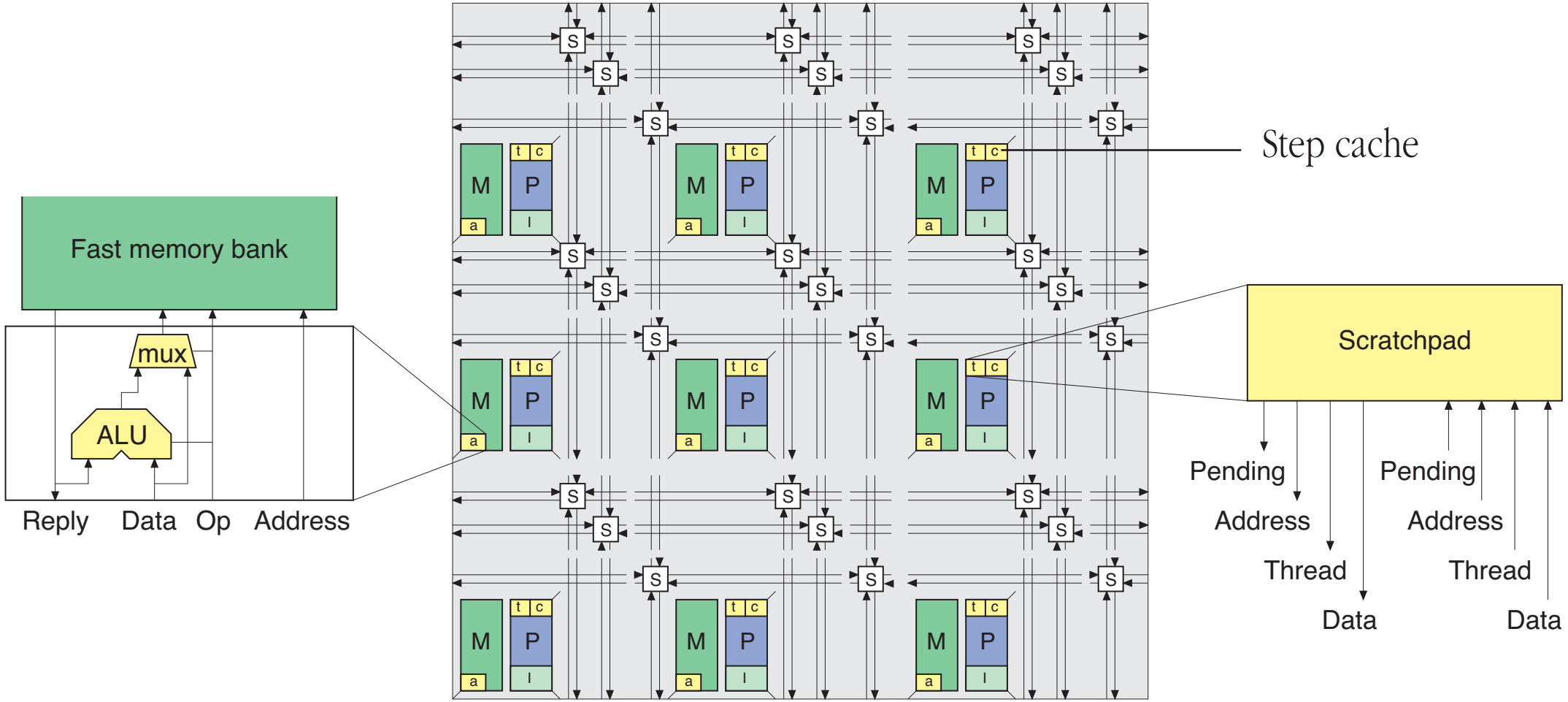


Multioperation solution of original Eclipse

- **Add** special **active memory units** consisting of a simple ALU and fetcher to memory modules, and switch memories to faster ones allowing a read and a write per one processor cycle
- **Implement multioperations** as two consecutive single step operations
- **Add** separate processor-level multioperation memories, called **scratchpads**
  (We need to store the id of the initiator thread of each multioperation sequence to the step cache and internal initiator thread id register as well as reference information to a storage that saves the information regardless of possible conflicts that may wipe away information on references from the step cache)

Implementing multioperations

—add active memory units, step caches and scratchpads



```
PROCEDURE Processor::Execute::BMPADD (Write_data,Write_Address)
Search Write_Address from the StepCache and put the
result in matching_index
IF not found THEN
    IF the target line pending THEN
        Mark memory system busy until the end of the current cycle
    ELSE
        Read_data := 0
        StepCache[matching_index].data := Write_data
        StepCache[matching_index].address := Write_Address
        StepCache[matching_index].initiator_thread := Thread_id
        ScratchPad[Thread_id].Data := Write_data
        ScratchPad[Thread_id].Address := Write_Address
ELSE
    Read_data := StepCache[matching_index].data
    StepCache[matching_index].data:= Write_data + Read_data
    ScratchPad[Initiator_thread].Data := Write_data + Read_data
    Initiator_thread := StepCache[matching_index].Initiator_thread
```

Algorithm 1. Implementation of a two step MPADD multioperation 1/3

```
PROCEDURE Processor::Execute::EMPADD (Write_data,Write_Address)
IF Thread_id<>Initiator_thread THEN
    IF ScratchPad[Initiator_thread].pending THEN
        Reply_pending := True
    ELSE
        Read_data := Write_data + ScratchPad[Initiator_thread].Data
ELSE
    IF Write_Address = ScratchPad[Initiator_thread].Address THEN
        Send a EMPADD reference to the memory system with
            - address = Write_Address
            - data = ScratchPad[Initiator_thread].Data
        ScratchPad[Thread_id].pending := True
    ELSE
        Commit a Multioperation address error exception
```

Algorithm 1. Implementation of a two step MPADD multioperation 2/3

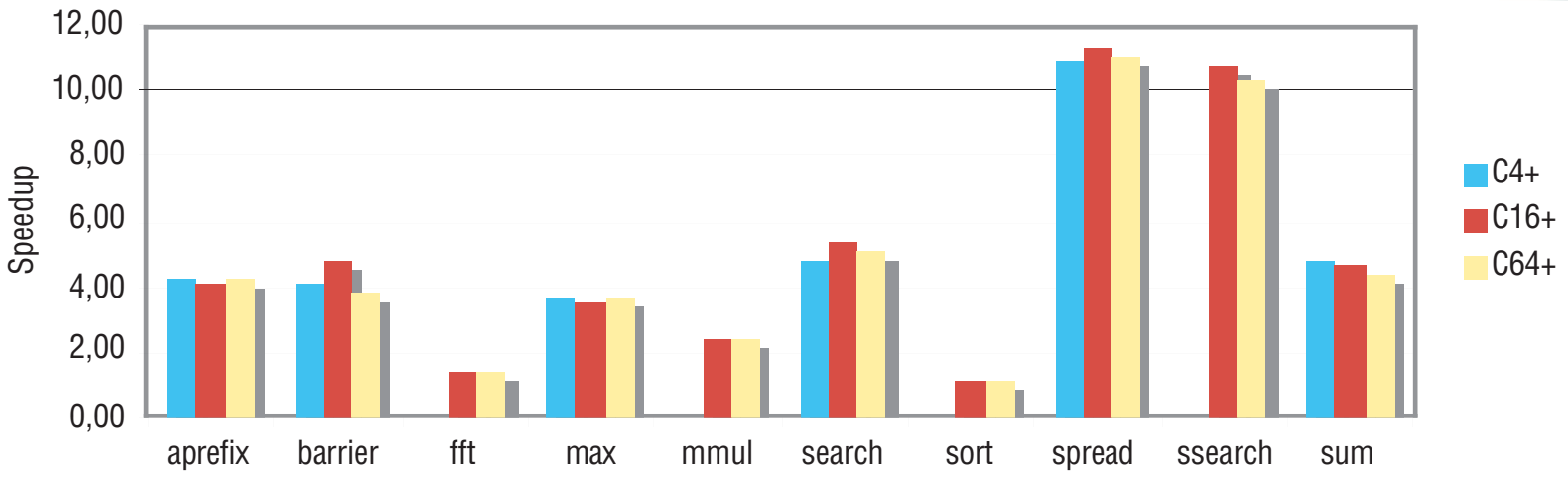
```
PROCEDURE Module::Commit_access::EMPADD ( Data , Address )
Temporary_data := Memory [Address]
Memory[Address] := Memory[Address] + Data
Reply_data := Temporary_data

PROCEDURE Processor::Receive_reply::EMPADD (Data,Address,Thread)
Read_Data[Thread] := Data
ScratchPad[Thread].Data := Data
ScratchPad[Thread].Pending := False
ReplyPending[Thread_id] := False
FOR each successor of Thread DO
    IF ReplyPending[successor] THEN
        Read_data := Data
        ReplyPending[successor] := False
```

Algorithm 1. Implementation of a two step MPADD multioperation 3/3

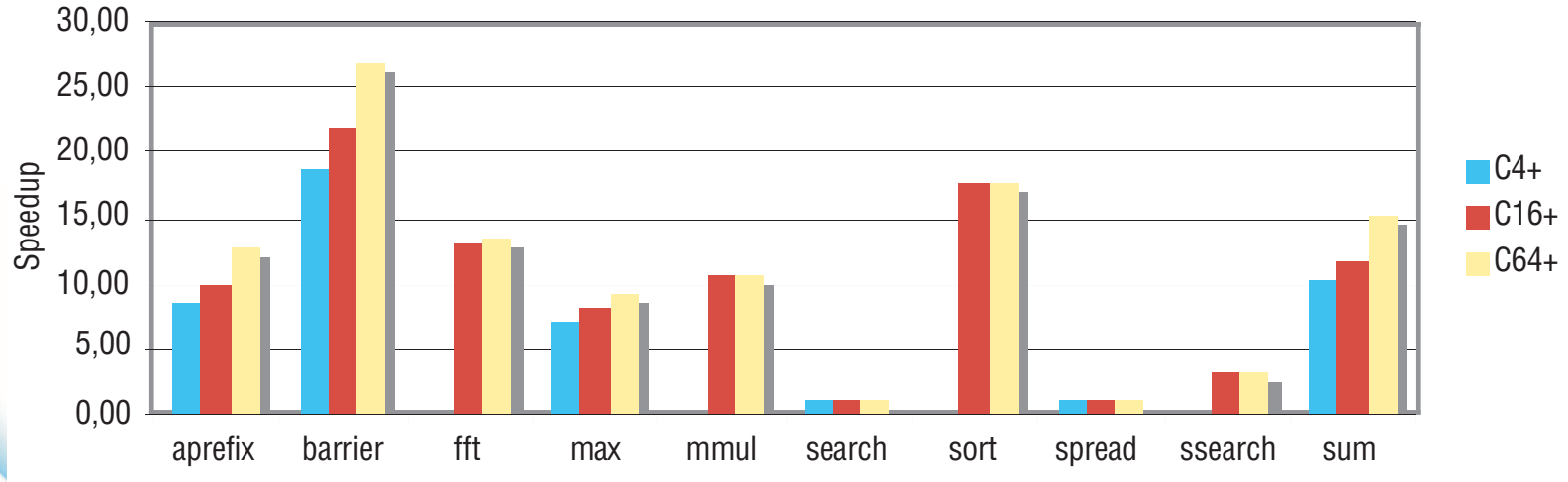
Evaluation

	EREW				MCRCW		
Name	**N**	**E**	**P**	**W**	**E**	**P=W**	**Explanation**
aprefix	T	log N	N	N log N	1	N	Determine an arbitrary ordered multiprefix of N integers
barrier	T	log N	N	N log N	1	N	Commit a barrier synchronization for a set of N threads
fft	64	log N	N	N log N	1	N^2	Perform a 64-point complex Fourier transform
max	T	log N	N	N log N	1	N	Find the maximum of a table of N words
mmul	16	N	N^2	N^3	1	N^3	Compute the product of two 16-element matrixes
search	T	log N	N	N log N	1	N	Find a key from an ordered array of N integers
sort	64	log N	N	N log N	1	N^2	Sort a table of 64 integers
spread	T	log N	N	N log N	1	N	Spread an integer to all N threads
ssearch	2048	M+log N	N	MN +log N	1	MN	Find occurrences of a key string from a target string
sum	T	log N	N	N log N	1	N	Compute the sum of an array of N integers

Evaluated computational problems and features of their EREW and MCRCW implementations (E=execution time, M=size of the key string, N=size of the problem, P=number of processors, T=number of threads, W=work).



The speedup of the proposed MCRCW solution versus the existing limited and partial LCRCW solution.



The speedup of the proposed MCRCW solution versus the baseline step cached CRCW solution.

Area overhead of MCRCW

According to our performance-area-power model:

- The silicon area overhead of the MCRCW implementation using step caches is typically **less than 1.0 %**.

Conclusions

We have described a number of single chip PRAM realizations

- EREW PRAM (multithreaded processors with chaining, 2D acyclic sparse mesh interconnect)
- Extension to CRCW using step caches
- Extension to MCRCW using stepc caches and scratchpads

Single chip silicon platform seems suitable for PRAM implementation, but there are still a number technical/commercial/educational issues open

- How to provide enough memory?
- How to survive if there is not enough parallelism available?
- The interconnection network appears to be quite complex, are there (simpler) alternatives?
- Who will try manufacturing PRAM on-chip?
- How to migrate to (explicit) parallel computing from existing code base?
- How will teach this kind of parallel computing to masses?