Today's agenda

- **Act I:** STMs — Why should we care?

  The theoretical foundations

- **Act II:** STMs — How to build one?

  The gory details

*"…we need to explore new techniques like **transactional memory** that will allow us to get the full benefit of all those transistors and map that into higher and higher performance."*

Bill Gates, Businessman

Designing Transactional Memory Systems

Part I: Introduction

Pascal Felber
University of Neuchatel
Pascal.Felber@unine.ch

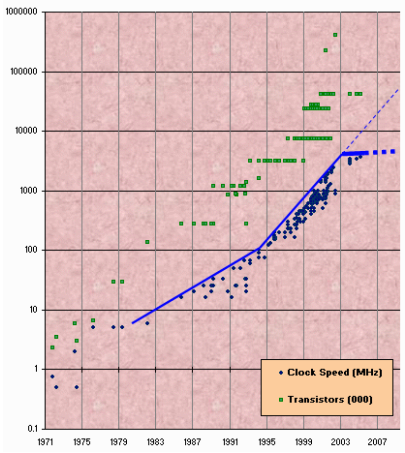
Institut d'informatique
Université de Neuchâtel

Based on joint work with Christof Fetzer & Torvald Riegel
with slides borrowed from several other people

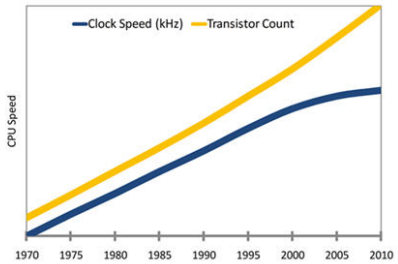
Agenda

- **Part I:** Introduction
  - Why do we need STMs?
  - What do STMs provide?
  - A brief history of STMs
- **Part II:** Obstruction-free STMs
- **Part III:** Lock-based STMs

Moore's Law and CPU speed



- Transistor count still rising according to Moore's Law
- Clock speed flattening



Multicores will be everywhere

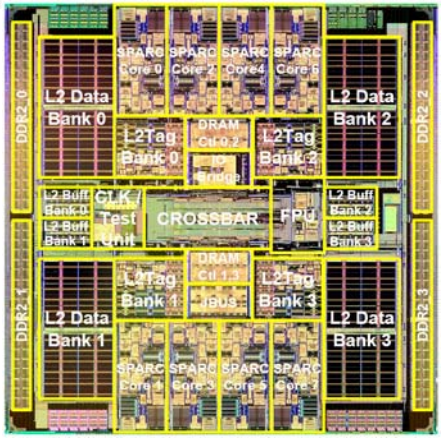
- Multicores are the answer to keeping up with increasing CPU performance despite:
  - The **memory wall** (gap between CPU and memory speeds)
  - The **ILP wall** (not enough instruction-level parallelism to keep the CPU busy)
  - The **power wall** (higher clock speeds require more power and create thermal problems)
- Consequence:
  - Single-thread performance doesn't improve…

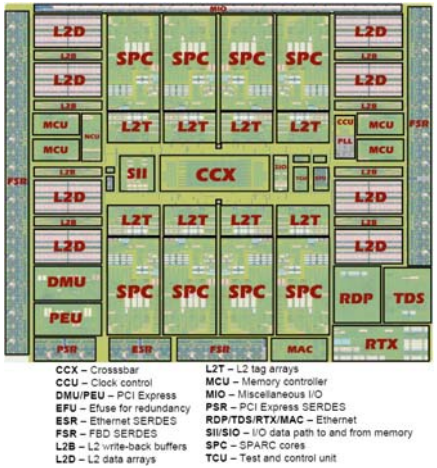
  … but we can put more cores on a chip

Multicores are everywhere

- **Dual-core** commonplace in laptops
- **Quad-core** in desktops
- **Dual quad-core** in servers
- All major chip manufacturers produce multicore CPUs
  - **SUN Niagara** (8 cores, 32 concurrent threads)
  - **Intel Xeon** (4 cores)
  - **AMD Opteron** (4 cores)
  - …

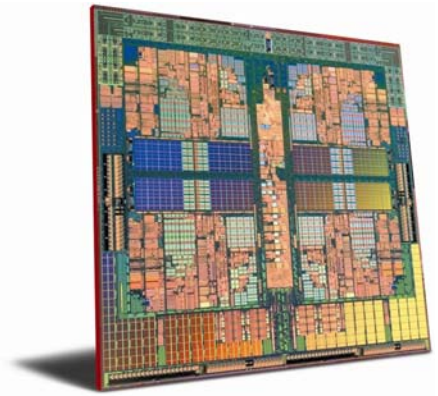
SUN's Niagara CPU2 (8 cores)

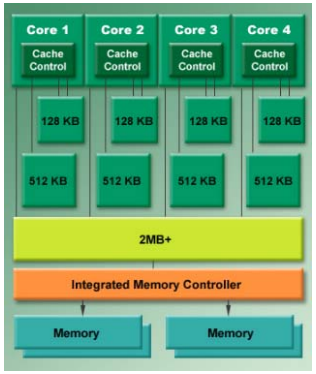




CCX – Crossbar
CCU – Clock control
DMU/PEU – PCI Express
EFU – Efuse for redundancy
ESR – Ethernet SERDES
FSR – FBD SERDES
L2B – L2 write-back buffers
L2D – L2 data arrays
L2T – L2 tag arrays
MCU – Memory controller
MIO – Miscellaneous I/O
PSR – PCI Express SERDES
RDP/TDS/RTX/MAC – Ethernet
SII/SIO – I/O data path to and from memory
SPC – SPARC cores
TCU – Test and control unit

AMD Opteron (4 cores)





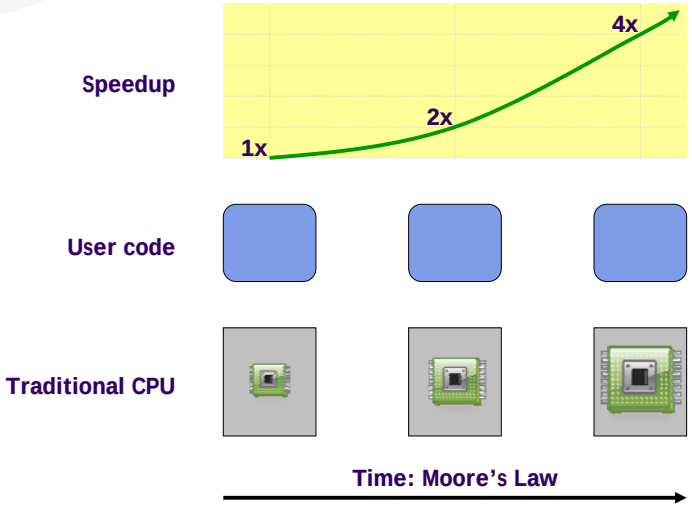
The "free ride" is over

- Cannot rely on CPUs getting faster in every generation
- Utilizing more than one CPU core requires thread-level parallelism (TLP)
- One of the biggest future software challenges: **exploiting concurrency**
  - Every programmer will have to deal with it
  - Affects HW/SW system architecture, programming languages, algorithms, …
  - Concurrent programming is hard to get right

… better not hit the **productivity wall**

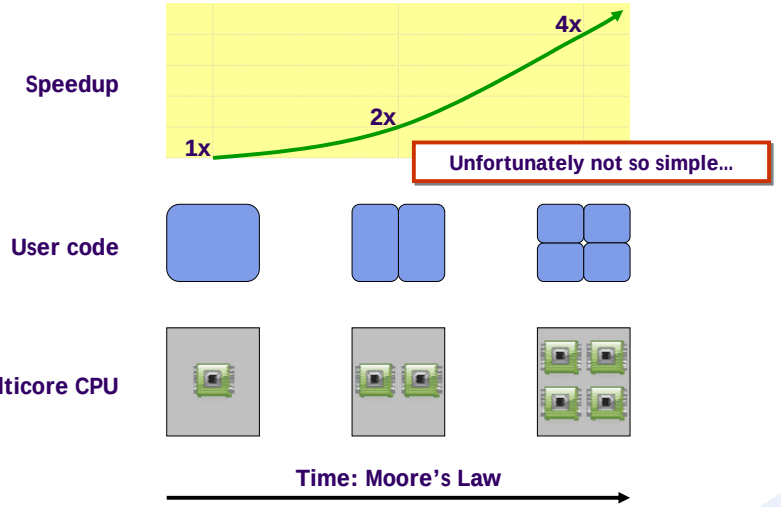
Based on slide by Herlihy & Shavit

Traditional scaling process



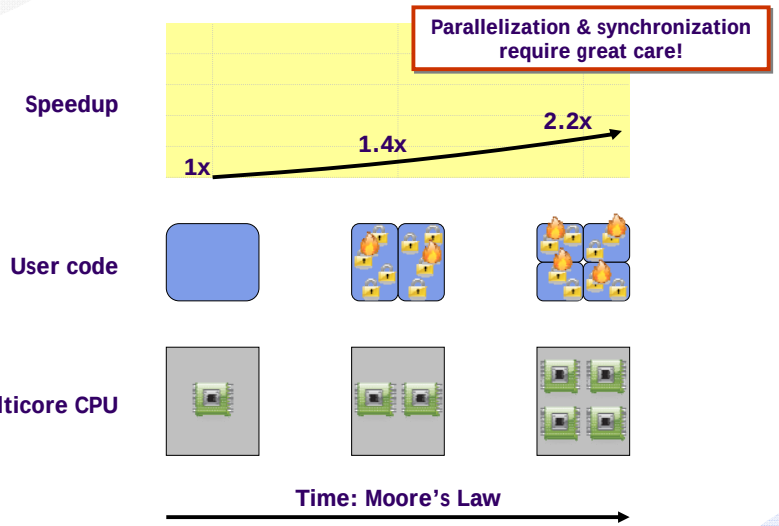
Based on slide by Herlihy & Shavit

Multicore scaling process



Based on slide by Herlihy & Shavit

Real-world scaling process



Concurrent programming is hard

- Hard to make **correct** and **efficient**
  - We need to exploit parallelism
- The human mind tends to be sequential
  - Concurrent specifications
  - Non-deterministic executions
- What about races? deadlocks? livelocks? starvation? fairness?
  - Need synchronization (correctness)…
  - … but not too much (performance)

Parallelization (1)

- **Data parallelism**
  - Split data into partitions
  - Let threads process partitions
  - Threads join after finishing their work
  - **Example:** image processing, sequencing, etc.
- Good data partitioning is difficult (correctness, load balancing, …)
- Synchronization required
  - Join
  - Load balancing during runtime

Parallelization (2)

- **Pipeline parallelism**
  - Split work into phases
  - Let each thread work on jobs in one of the phases
  - **Example:** event stream processing
    input → parse → process → format → output
- Not every program has such phases, some phases are longer than others
- Synchronization required
  - One phase's results become next phase's input
  - Complex access patterns

Parallelization (3)

- **Speculative/optimistic parallelism**
  - Just execute concurrent jobs
  - Assumption: most jobs do not conflict
  - Partitioning or phases not required but possible
  - **Example:** event handlers, application servers, graph algorithms, etc.
- There must be little contention on shared data structures
- Synchronization required
  - Every access can potentially interfere with accesses from another thread

Shared memory synchronization

- Cores share main memory
  - Some hardware instructions are (or can be made) atomic: **`inc`**, **`dec`**, **`cmpxchg`**, …
- Loads/store usually atomic, not necessarily ordered (no sequential consistency!)
  - Must use memory barriers to enforce order
- Current state of concurrent programming:
  - Use locks built from these instructions
  - Build concurrent (non-blocking) algorithms from these instructions

Based on slide by Grossman

Why transactions?

```
void deposit(…)  { synchronized(this) { … } }
void withdraw(…) { synchronized(this) { … } }
int  balance(…)  { synchronized(this) { … } }

void transfer(account from, int amount) {

   if (from.balance() >= amount) {
     from.withdraw(amount);
     this.deposit(amount);
   }


}
```

No concurrency control: race!

Based on slide by Grossman

Why transactions?

```
void deposit(…)  { synchronized(this) { … } }
void withdraw(…) { synchronized(this) { … } }
int  balance(…)  { synchronized(this) { … } }

void transfer(account from, int amount) {

  synchronized(this) {
   if (from.balance() >= amount) {
     from.withdraw(amount);
     this.deposit(amount);
   }
  }

}
```

Race!

Why transactions?

```
void deposit(…)  { synchronized(this) { … } }
void withdraw(…) { synchronized(this) { … } }
int  balance(…)  { synchronized(this) { … } }

void transfer(account from, int amount) {
 synchronized(this) {
  synchronized(from) {
   if (from.balance() >= amount) {
     from.withdraw(amount);
     this.deposit(amount);
   }
  }
 }
}
```

Deadlock!

Atomic blocks

```
void deposit(int x) {
 synchronized(this) {
  int tmp = balance;
  tmp += x;
  balance = tmp;
 }
}
```

Lock acquire/release

```
void deposit(int x) {
 atomic {
  int tmp = balance;
  tmp += x;
  balance = tmp;
 }
}
```

(As if) no interleaved computation

Easier-to-use primitive
(but harder to implement)

Atomic blocks

```
void deposit(…)  { atomic { … } }
void withdraw(…) { atomic { … } }
int  balance(…)  { atomic { … } }

void transfer(account from, int amount) {

   if (from.balance() >= amount) {
     from.withdraw(amount);
     this.deposit(amount);
   }


}
```

No concurrency control: race!

Atomic blocks

```
void deposit(…)  { atomic { … } }
void withdraw(…) { atomic { … } }
int  balance(…)  { atomic { … } }

void transfer(account from, int amount) {

  atomic {
   if (from.balance() >= amount) {
     from.withdraw(amount);
     this.deposit(amount);
   }
  }

}
```

Correct and enables parallelism!

Why STM?

- Transactions: a simple paradigm

A sequence of instructions, executed atomically

- Software transactions are good for:
  - Software engineering (simple programming, avoid races & deadlocks, composability)
  - Performance (when no conflict, high parallelism and no locking overhead)

A "universal" synchronization construct

Don't care how transactions are implemented!

Composability

- Ability to build large, complex systems from small, simple pieces
  - Enables divide-and-conquer strategy
- Locks are **not** composable
  - Must be exposed and use "compatible" strategies
- Custom concurrent algorithms are typically **not** composable
- Composability challenges
  - Must preserve correctness
  - Should not hamper performance
- STM **is** composable

Can we make STM fast?

- Typically, problem-specific handcrafted algorithms are **more efficient**…
  - Whether lock-free or using fined-grained locking
  - Programmers can use knowledge of data flow relationships to control contention

… but STM is **simpler and safer** to use…

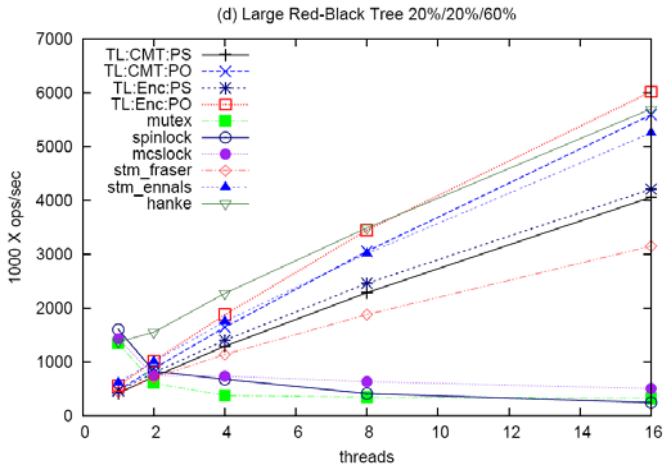
  - Programmer does not need to reason about concurrency

… and there is room for **optimism**

  - Performance can be as good or better for complex data structures

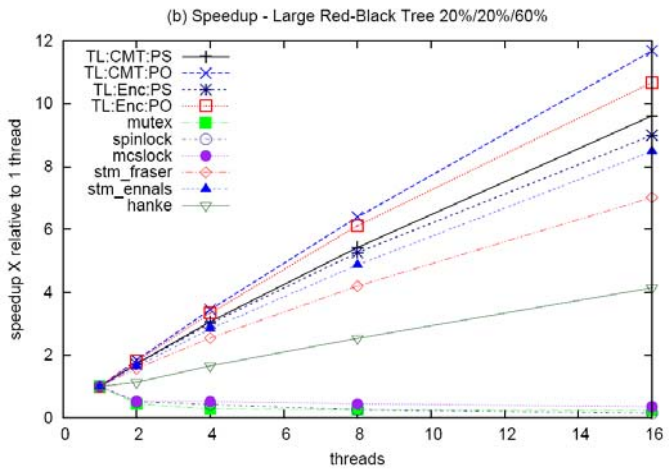
STM performance [Dice & Shavit]

STM can be as efficient as handcrafted lock-based implementation!



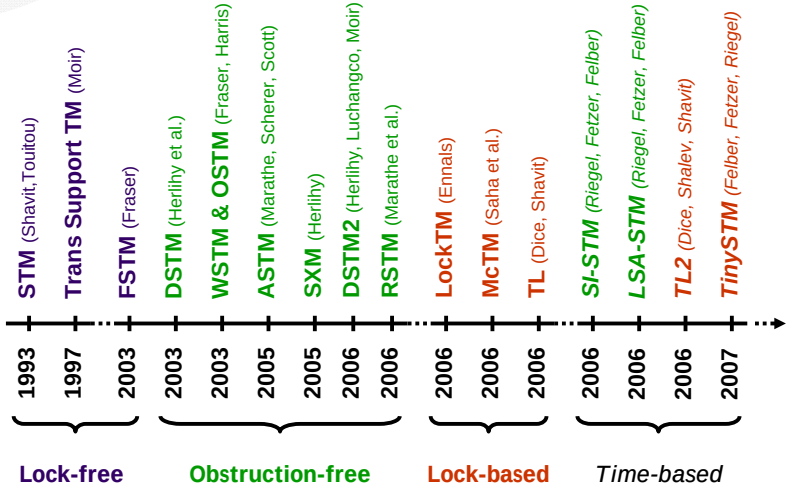
STM scaling [Dice & Shavit]

STM scales better than handcrafted lock-based implementation!



Based on slide by Shalev & Shavit

A brief (partial) history of STM



Conclusion (Part I)

- **Multicores are here!**
  - No more scalability for free
  - We need the right tools to exploit their power
- **Concurrent programming is hard**
  - Synchronization necessary for correctness, parallelism for efficiency
- STM exploits **disjoint access parallelism**
  - Execute sequences of operations atomically (don't care how this is done!)
  - Optimistic concurrency control
  - Transactions are composable