Random Test Generators for Microprocessor Design Validation

Joel Storm
Staff Engineer, Hardware
Technology, Validation, & Test
Sun Microsystems Inc.

12 May 2006

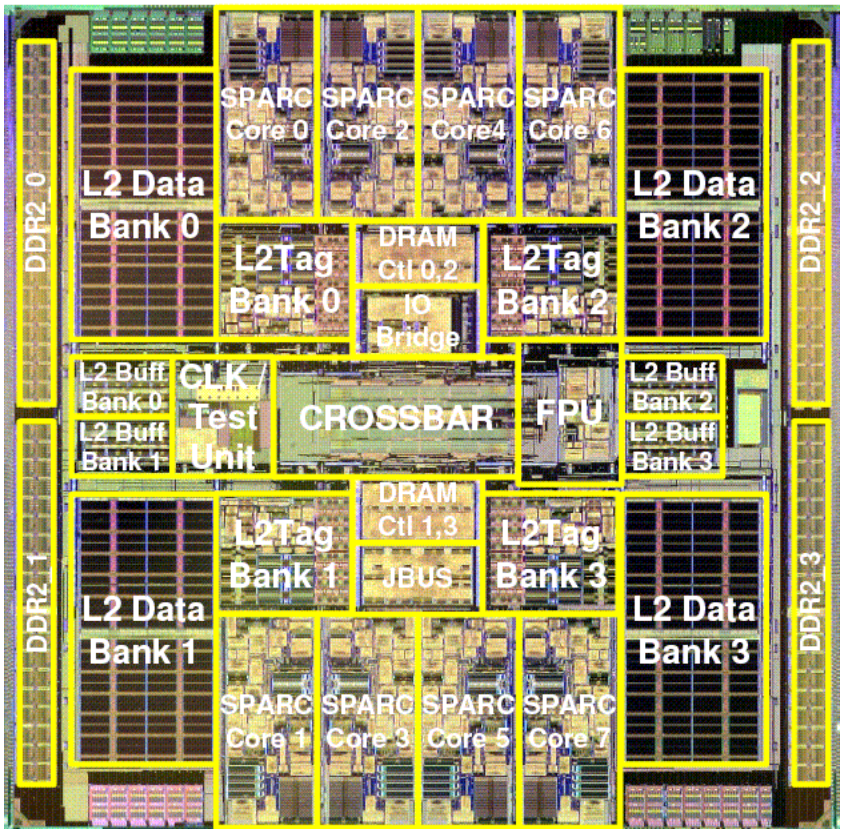
8th EMICRO Porto Alegre, RS, Brazil

http://www.inf.ufrgs.br/emicro

Agenda

- Design Verification Process Overview
  - Functional Verification Only
    - No performance
    - No timing
    - No electrical, circuit, power, etc.
- Random Instruction Generator Design
  - Full Featured Generator
    - Useful in simulation
    - Bootable on hardware
- OpenSPARC http://opensparc.sunsource.net/

UltraSPARC-T1: Some Design Choices



- Simpler core architecture to maximize cores on die
- Caches, dram channels shared across cores give better area utilization
- Shared L2 decreases cost of coherence misses by an order of magnitude
- On die memory controllers reduce miss latency
- Crossbar good for b/w, latency, functional verification
- 378mm2 die in 90nm dissipating ~70W
- http://opensparc.net

Importance of Design Verification

- Cost of manufacturing prototype silicon increasing
- Chip manufacturing turn around time increasing
- Cost of design faults in hardware increasing
  - Money
  - Electronics now used in “life critical” applications
- Time is money
  - A great product 2 years late is not so great
  - “Hacking” is too slow

PART 1: Design Verification Process

- Design for Verification
- Pre silicon Verification
- Post silicon Verification
- Review results to improve things for next time

Design for Verification

- Architecture Features
  - > Instruction monitors
  - > Address monitors
  - > Data saved on exceptions
- Microarchitecture Visibility
  - > Special access to microarchitecture for software
  - > External visibility: Scan
  - > Error injection for RAS testing
- Repeatability / Determinism
  - > Ability to see problem again (and again, and ....)

Bad Design for Verification

- Architecture Features
  - > Write Only Registers
  - > Undefined Fields or Actions
- Repeatability / Determinism
  - > No way to sync clock domains
  - > Random state after reset
- Typical reasons for poor Design for Verification
  - > Mistaken belief decision makes verification easier
  - > Concerns about impact to size or performance

Pre Silicon Verification

- Environments
  - RTL simulation
    - Stand Alone Test environments (SATs)
    - Fullchip
  - RTL simulation methods
    - Software
    - Hardware accelerated
  - Architectural simulation

Pre Silicon Verification 2

- Tools
  - > Formal verification
  - > Architectural directed tests
  - > Microarchitectural directed tests
  - > Pseudo Random tests
- Common problems
  - > No tests for “hard” cases
  - > Too many tests

Post Silicon Verification

- “Bootable” or “Bare Metal” or “Native Mode” tests
  - > From PROM
  - > Like Operating System
- Operating Systems based tests
  - > Runs like normal application program
  - > May use special debug/validation system calls

Design Verification Philosophy

- Test to the specification!!!
- Test all reasons for exceptions
- Do not limit testing to the “real” cases
  - > No one really knows all “real” cases
  - > New “real” cases will appear in the future
  - > Users will hit “no one would ever do that” cases by accident
- Do give priority to “real” case
  - > OK, honestly we do have a pretty good idea what users do

Design Verification Philosophy 2

- Know critical path in schedule (important!)
- Understand "chicken & egg" problem
  - What comes first? The test or the debug platform?
  - Best to work this out with software model
  - Bad to work it out with RTL

Priorities -> This is What's Important

- 1: Test Coverage
- 2: Usability
- 3: Efficiency (time/cycles to get to specific case)
- 4: Serious problems in 2 & 3 can affect #1
  - > Note that a common mistake is to use too many resources on 2 & 3.
    - >An intuitive interface and cycle efficient code that can test 80% of the design is not as useful as a crude interface and slow code that can test 95%

PART 2: Random Instruction Generator Design

- Full featured, do everything CPU test generator
  - Works in all simulation environments
  - Boots on hardware like an Operating System
  - Automatic stress test generation
  - User selectable features
  - Pseudo Random
    - Some built in intelligence on randomness
    - Completely random is not very usefull
  - Too big for 1 Engineer

Random Generator Usage

- Simulation environments
  - > Generate tests on good system
  - > Run only individual tests on target
- Hardware & hardware like environments
  - > Load (boot) test generator on target system
  - > Generate and run tests on target system
    - >Usually “infinite” loop
      - Generate test
      - Run test
      - Repeat

Architecture of Random Generator

- Three main sections
  - > Infrastructure
  - > Test Generator
  - > Run time environment

Architecture of Random Generator 2

- Infrastructure
  - User interface
    - Command reader
    - Output displays
  - Boot code
  - Debug aids
    - Event history tables
    - Instruction breakpoints

Architecture of Random Generator 3

- Test Generator
  - Memory Allocation
    - Data areas
    - Instruction areas
    - Address translations (Virtual to Physical)
  - Feature Selection
    - Architectural / Microarchitectural features
    - Instruction mix

Architecture of Random Generator 4

- Run time environment
  - Test start up code (context switch)
    - Stand alone tests
    - Switch from generator control code to test
  - Exception handlers (interrupts, traps)
    - Handle & recover from exception
    - Verify correct behavior
  - Test end code (context switch)
    - Switch back to generator control code
    - Final state checks

Important Considerations

- Not a “normal” application!
- Must be debugable on broken hardware
  - Limit outside dependencies (none is best)
    - Outside code may not work
    - Link to outside code may not work
  - Software deterministic
  - “Efficient” programing strategies not always good
    - Keep run time environment in 100% assembly
      - Debuggers will be stepping through this

Important Considerations 2

- Leverage other code carefully!
  - > Don't try to convert another program into a test
  - > Grabbing small functions if fine (stack, parse, print)
  - > Check if the code you want to use requires more code, that uses a library, that includes....
- If in doubt, write it yourself
  - > Time to convert code for verification use often longer than time to write new usage specific code.

Programming Techniques

- Data structure are better than logic!
  > Tables of data can greatly reduce the need for long switch/case logic statements
- Pointers are better than logic
  > Pass pointers to data structures
  > Use function pointers
- Just say “NO” to recursive algorithms
  > Remember: need to debug on broken hardware

Instruction Generator Goals

- Build all combinations of instructions
  - > Instruction X before and after instruction Y
  - > No limits on what can precede/follow instruction
- Conditional branches to/around any instruction
- Microarchitecture testing
  - > Fill instruction cache (subroutines a good way)
  - > Branches over “interesting” boundaries
    - > Cache lines
    - > Pages

Instruction Data Table

- All instruction data in one place
  - > Opcode (bit pattern that is unique to instruction)
  - > Mnemonic
  - > Number of operands
  - > Operand size
  - > Pointers to build/simulation/disassembly functions
  - > Flags for valid exceptions
  - > Instruction family flags (Branch, Floating-Point)

Instruction Data Table Example Entry

0x81a00d20 (opcode)
“fsmuld” (mnemonic)
SOURCE1_SINGLE (flag, mask, or constant for 1'st operand)
SOURCE2_SINGLE (flag, mask, or constant for 2'nd operand)
DESTINATION_DOUBLE (same info for destination reg)
FP_DISABLED | XYZ (flags for valid exceptions on this inst)
build_FP_s_to_d (function pointer to build code for this type)
sim_FP_s_to_d (function pointer to simulation code [if any])
disassemble_FP_s_to_d (fcn pointer to disassembly code)

Instruction Data Table 2

- Organize instruction table as a tree
  - > Use opcode to determine layout
  - > Leaf nodes are variable length arrays of instruction structures
  - > Easy to traverse
    - > With tree traversal algorithm
    - > Using opcode

Random Instruction Generation

- Once during program initialization (or when table is modified):
  - Traverse whole instruction tree and build sub tables
    - Array of pointers to all Floating-Point instructions
    - All branch instructions
    - All integer multiply instructions
    - etc.
- Used by code to stress specific features

Random Instruction Generation 2

- Once for each test built:
  - > Combine user specified adjustments with automatic instruction tuning
  - > Fill instruction mix array with pointers to instruction sub tables
    - >100 element array allows 1% adjustments
    - >1000 elements for 0.1% adjustments (duh)

Main Instruction Generation Loop

- Get a random number
- Use it to index into the instruction mix array
- Use another random number and pointer from instruction mix array to index into a sub table
- Follow pointer in sub table to instruction data structure
- Call instruction build function pointed to by function pointer in instruction data structure

Instruction Build Function

- One for each type of instruction (not each inst.)
  - > Example: 2 integer register sources and 1 integer register destination
  - > Floating-Point register load instructions
- Is passed a pointer to the instruction data entry
- Simple logic uses masks and flags from data entry to build a complete instruction
- Calls common functions for access to resources

Resource Allocation Functions

- One for each type of major resource
  - Integer registers
  - Floating-Point registers
  - Memory addresses
- Can track usage and force data dependencies
- Can force no dependencies
- Can reserve resources for special uses
  - Loop counters

“Split Up” Instruction Sequence

- Loop (backward branch sequence)
  - Initialize loop counter
  - Decrement/Increment counter
    - Set condition code
  - Conditional branch back to point after loop counter initialization
- Forward branch
  - Build branch skeleton (not finished instruction)
  - Fill in offset for target address

“Split Up” Instruction Sequence 2

- Other sequences
  - > Load subroutine address
  - > Call subroutine
- Two ways to implement
  - > Recursive calls from build function to main loop
  - > Scoreboard

Result Testing

- Simulation environments usually include built in checking
  > Results on RTL checked against software model
- Three major types of random generator built in testing
  > Sanity checks of exceptions
  > Two pass comparison (Very Powerful)
  > Instruction simulation
- End of test sanity checks

Sanity Checks of Exceptions

- Divide by zero trap
  - > Divide instruction?
  - > Operand was zero?
- TLB miss trap
  - > Load or Store?
  - > Miss was expected/allowed
- Illegal instruction trap
  - > Is it illegal
  - > Did the generator put it in this test (Important!)

Two Pass Comparison First Pass

- Run test in single step mode
  - > Use hardware feature or software traps
    - > Software trap method is self modifying code that walks trap instruction through test code
- Save state data at periodic checkpoints
  - > Registers
  - > Exceptions taken

Two Pass Comparison Second Pass

- Reset all data
- Run test normally (no single step)
- Check state data at periodic checkpoints
  - Registers
  - Exceptions taken
    - Can be part of checkpoint
    - Can be part of exception handling

Instruction Simulation

- Include simulation functions in random generator
  - > Connections to separate simulation engines don't work (trust me on this)
- Leverage pass one single step functions
  - > Easy to implement
  - > Allows partial simulation (don't need to simulate simple instructions)
  - > Use function pointer from instruction data table
    - > Easy implementation: look up instruction & call function from function pointer

End of Test Sanity Checks

- If CPU1 sent CPU7 12 messages, were they all received?
- Did all CPUs complete the test?
- Were expected exceptions actually seen?

Test Tuning

- Instruction mix
- Data stress
  - Boundary conditions
  - IEEE Floating-Point fun
- Microarchitecture stress
  - Instruction cache
  - Data cache
  - Store buffers
  - Data dependencies

Test Tuning 2

- Controls
  - > On
  - > Off
  - > Random
- Range variables
  - > Percent forward branches or loops
  - > Percent register or immediate operands

Usability Extras

- Common sense displays
  - Registers & addresses in Hexadecimal
  - Instruction dump disassembler
    - Use function pointer in instruction data table
    - Add helpful comments
- Access to microarchitecture: TLBs, Control registers
- Exception history logs
- Error logs

PART 3: OpenSPARC

- Freely available version of Sun's UltraSPARC T1 microprocessor
- Architecture documentation
- RTL
- Software models
- Verification tests
- http://OpenSPARC.net

OpenSPARC Introduction

All Following pages taken from David Yen's "Opening Doors to the Multicore Era" presentation for the Multicore Expo in March 2006

The Big Bang *Has* Happened — Four Converging Trends

Network Computing Is Thread Rich

Web services, Java™ applications, database transactions, ERP . . .

Moore's Law

A fraction of the die can already build a good processor core; how am I going to use a billion transistors?

Worsening Memory Latency

It's approaching 1000s of CPU cycles! Friend or foe?

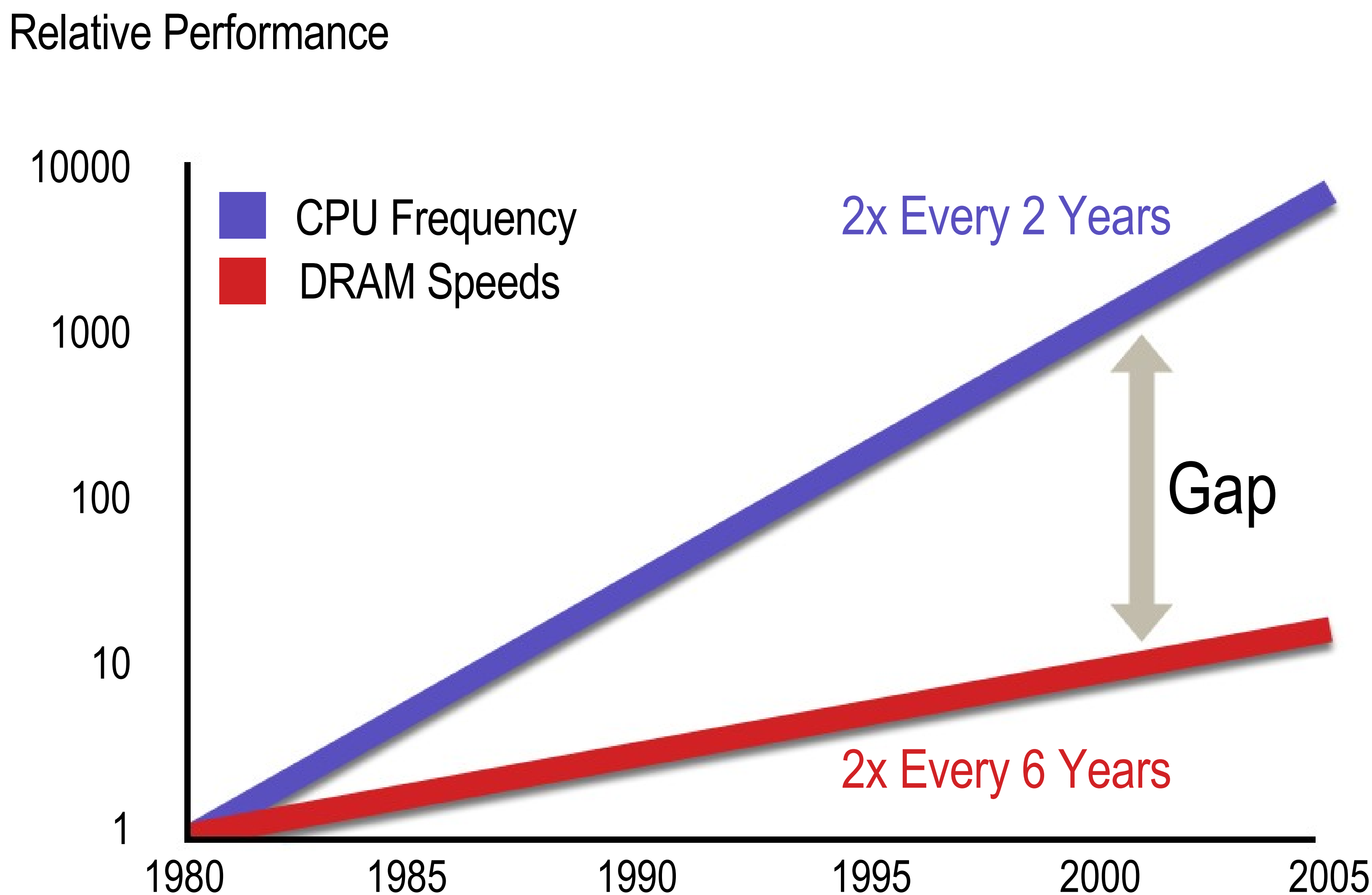
Growing Complexity of Processor Design

Forcing a rethinking of processor architecture – modularity, less is more, time-to-market

Attributes of Commercial Workloads

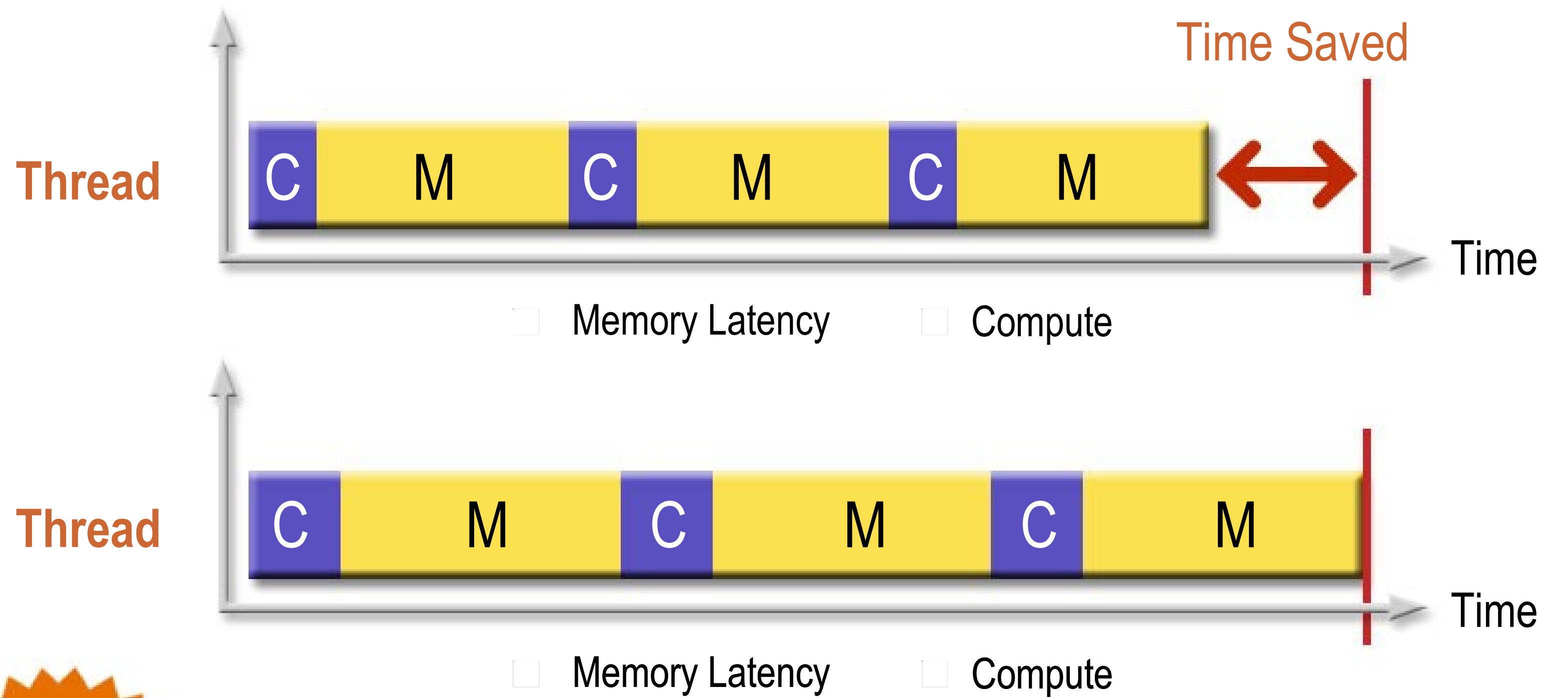
	Web Services			Client Server		Data Warehouse
Attribute	TIER1 Web (Web99)	TIER2 App Serv (IBB)	TIER3 Data (TPC-C)	SAP 2T	SAP 3T (DB)	DSS (TPC-H)
Application Category	Web Server	Server Java	OLTP	ERP	ERP	DSS
Instruction-level Parallelism	Low	Low	Low	Medium	Low	High
Thread-level Parallelism	High	High	High	High	High	High
Instruction/Data Working Set	Large	Large	Large	Medium	Large	Large
Data Sharing	Low	Medium	High	Medium	High	Medium

Memory Bottleneck



Source: Sun World Wide Analyst Conference Feb. 25, 2003

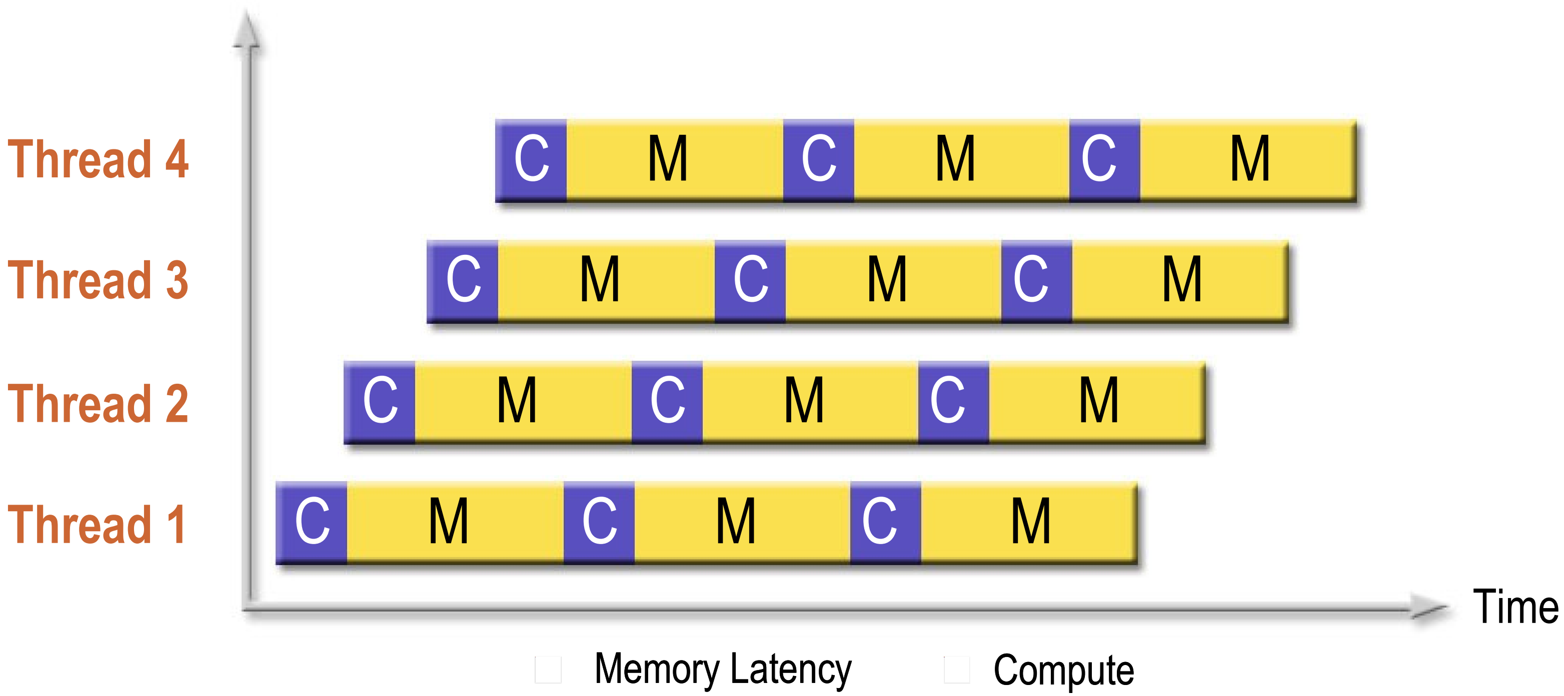
Typical Complex High Frequency Processor



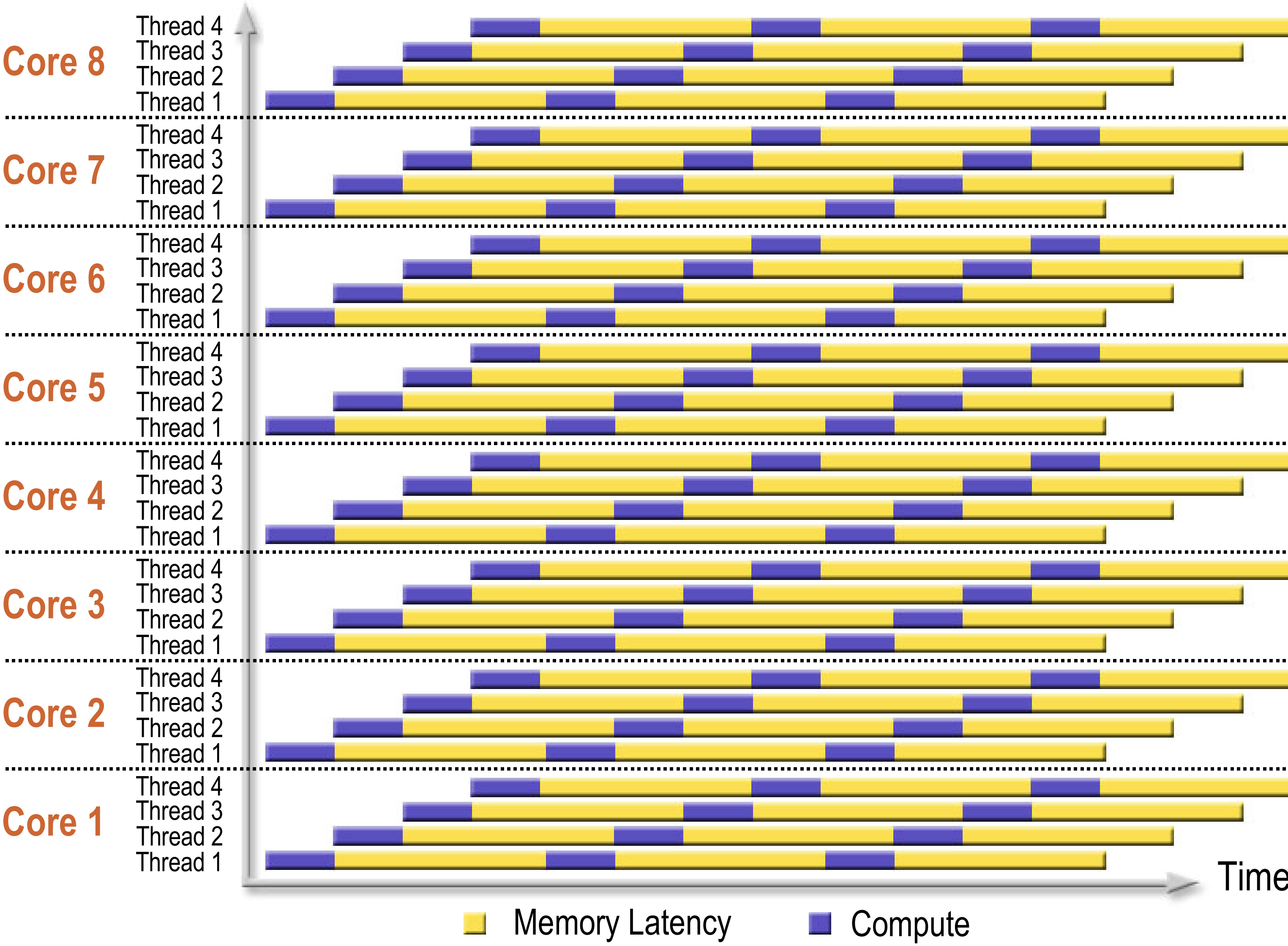
HURRY UP AND WAIT!

Note: Up to 75% Cycles Waiting for Memory

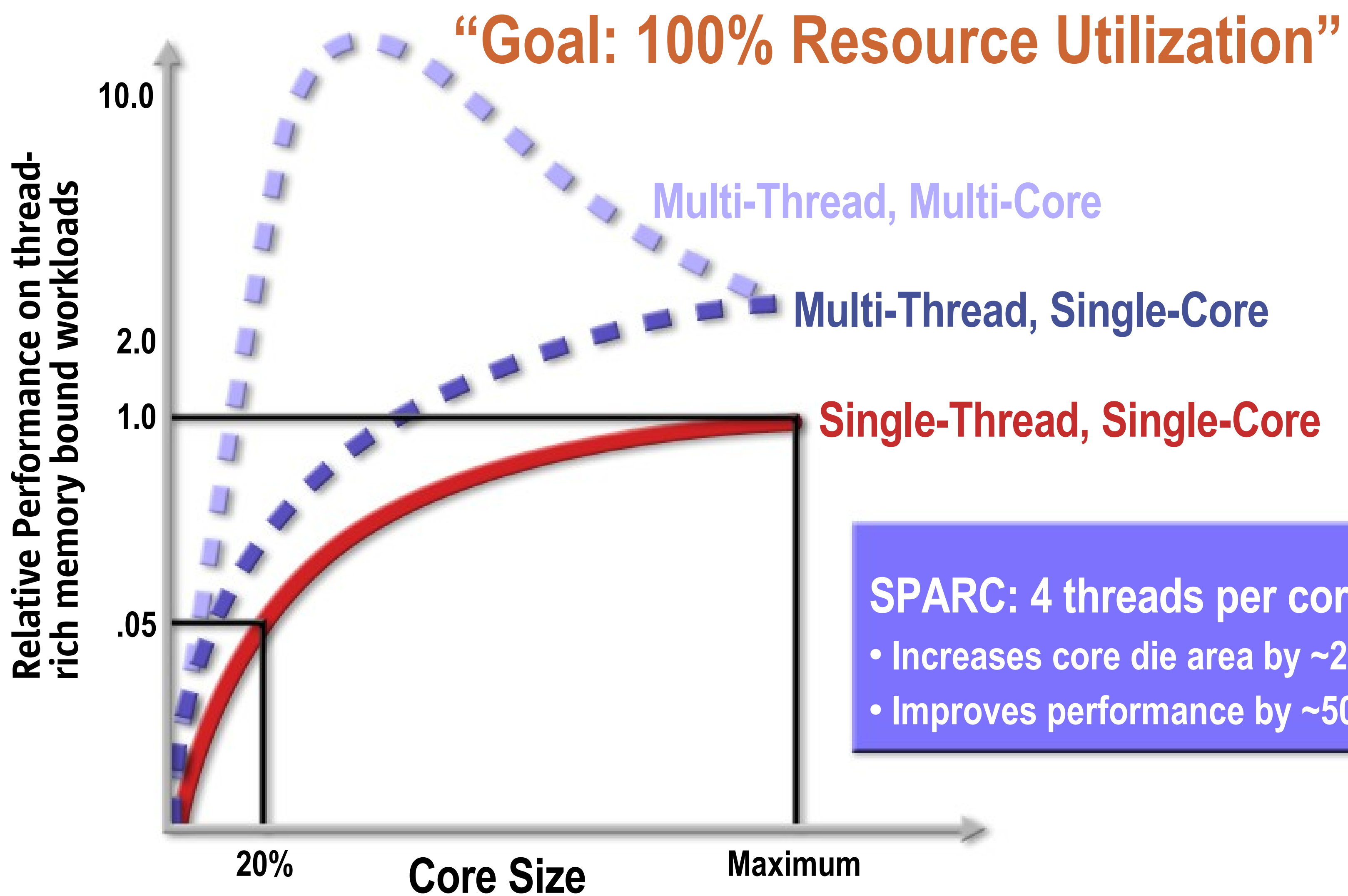
Chip Multithreading (CMT)



CMT – Multiple Multithreaded Cores



Why CMT Works

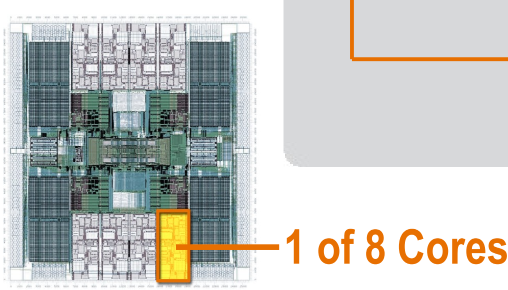


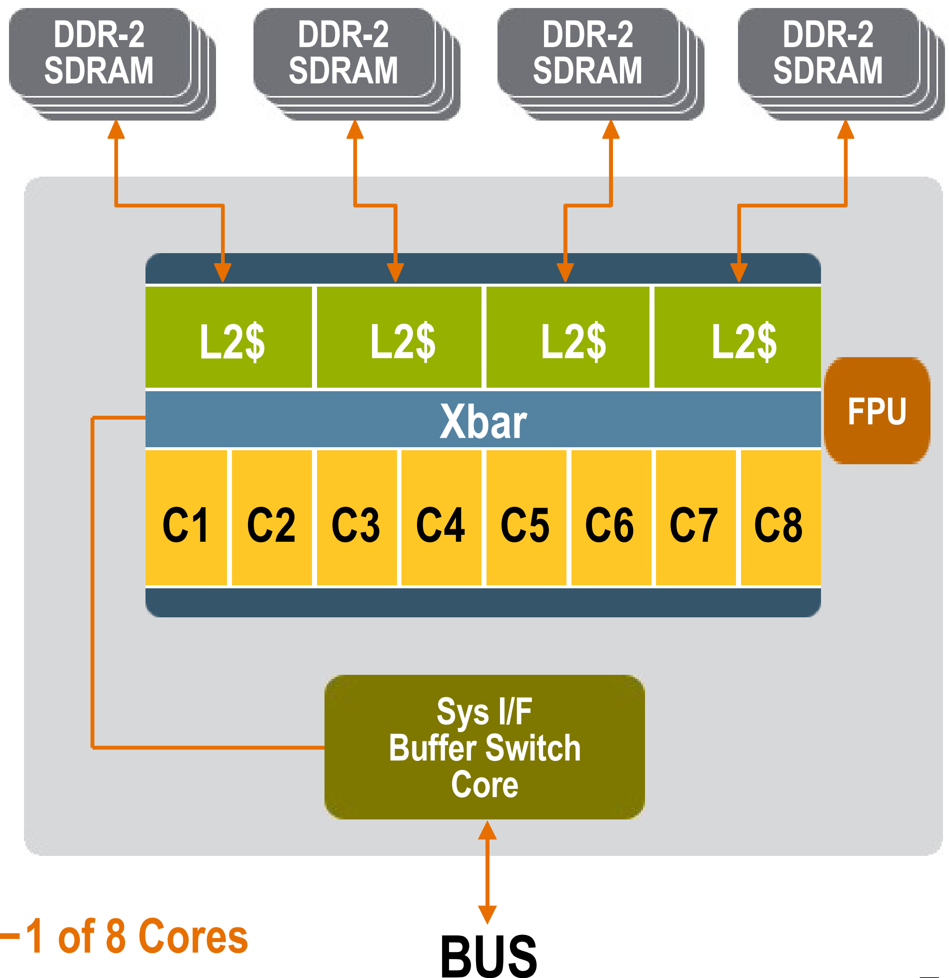
SPARC: 4 threads per core

- Increases core die area by ~20%
- Improves performance by ~50–100%

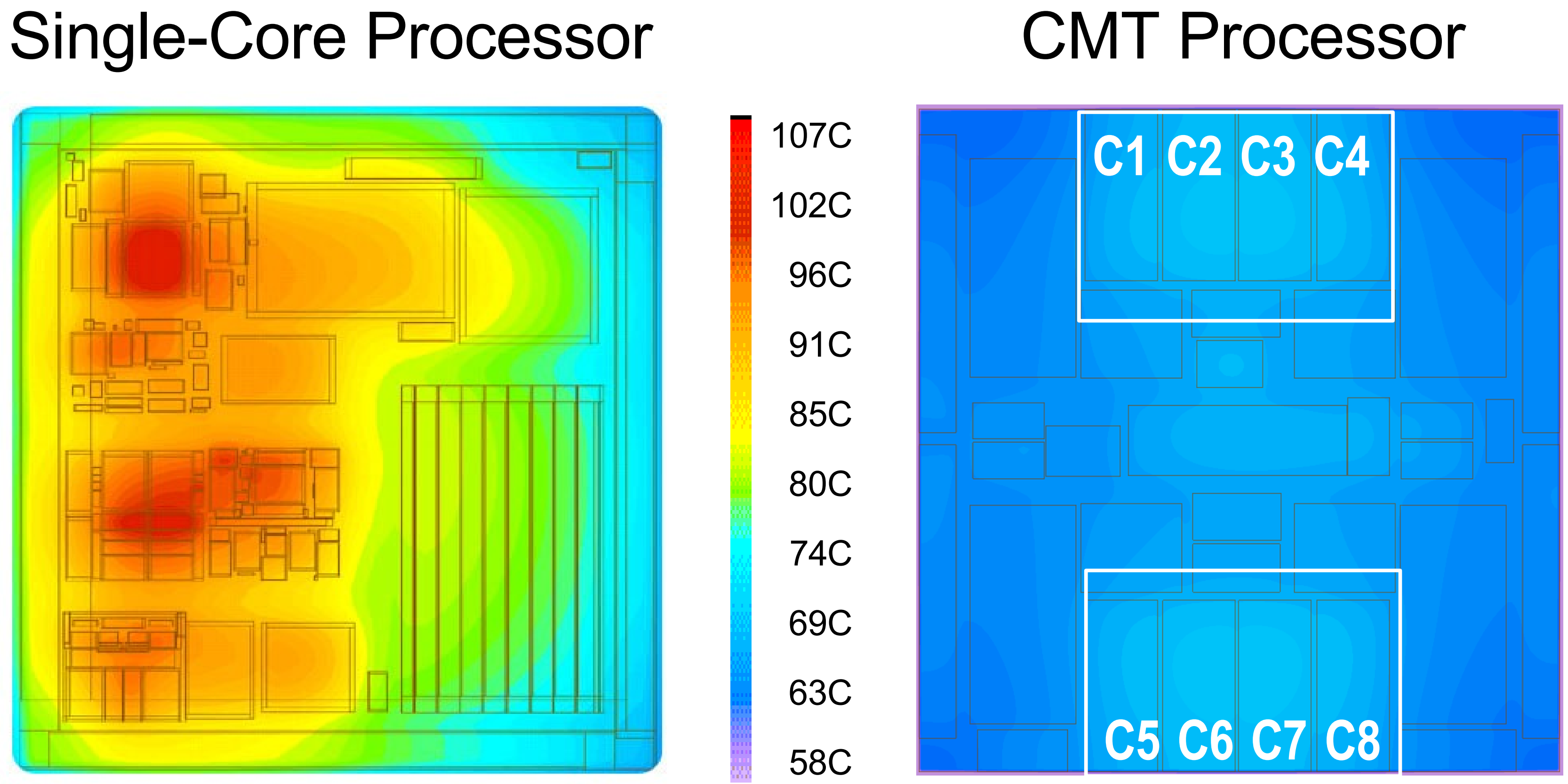
UltraSPARC T1 Processor

- SPARC V9 implementation
- Up to eight 4-way multi-threaded cores for up to 32 simultaneous threads
- All cores connected through a 134.4GB/s crossbar switch
- High-bandwidth 12-way associative 3MB Level-2 cache on chip
- 4 DDR2 channels (23GB/s)
- Power : < 80W
- ~300M transistors
- 378 sq. mm die





Faster Can Be Cooler

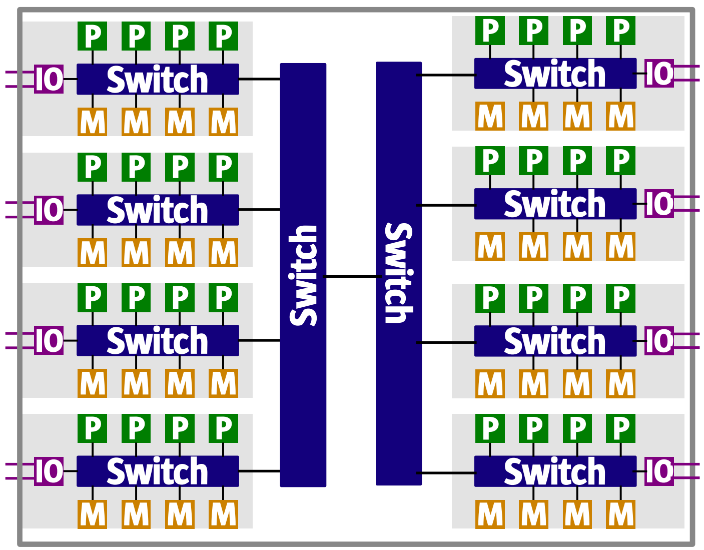


(Not to Scale)

CMT: On-chip = High Bandwidth

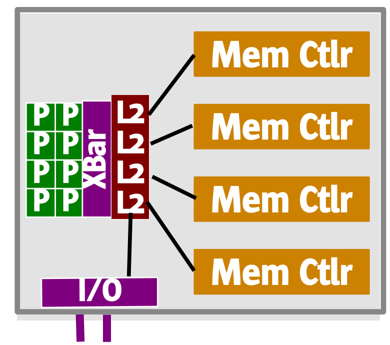
Traditional SMP: 32 Threads

Example: Typical SMP Machine Configuration



Niagara: 32 Threads

One motherboard, no switch ASICs



Direct crossbar interconnect

Lower cost, better RAS, lower BTUs, lower and uniform latency, greater and uniform bandwidth. . .

CMT Benefits

Performance

Cost

- Fewer servers
- Less floor space
- Reduced power consumption
- Less air conditioning
- Lower administration and maintenance

Reliability

OpenSPARC: New Frontier in Choice!

- Sun's OpenSPARC initiative intends to open source UltraSPARC T1 design point
  - > Announced Dec. 6, 2005
  - > RTL in Verilog released March 21, 2006
- Initial publications also include:
  - > A verification suite and simulation models
  - > ISA specification (UltraSPARC Architecture 2005)
  - > UltraSPARC T1-specific ISA supplement
  - > A Solaris port

About the Community: opensparc.net

Clustermaps for http://opensparc.net

Innovation Happens Everywhere

開放的
열린
مفتوح
libre
मुक्त
ಮುಕ್ತ
livre
libero
ముక్త
开放的
açık
open
nyílt
פתוח
オープン
livre
ανοικτό
offen
otevřený
öppen
открытый
வெளிப்படை

open

64 bit, 32 threads, free

www.opensparc.net

Get the code. Start innovating.

Multi-threaded algorithms and applications, Operating Systems, System Architecture, EDA Tools/Methodology, Circuit implementations, Compiler Tools, System Modeling, System on a Chip, Debug tools, Performance analysis and benchmarking

Random Test Generators for Microprocessor Design Validation

Joel Storm
Joel.Storm@sun.com