Multicores from the Compiler's Perspective
A Blessing or a Curse?

Saman Amarasinghe

Associate Professor, Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
Computer Science and Artificial Intelligence Laboratory

CTO, Determina Inc.
Multicores are coming!

- **AMD Opteron**
- **Intel Montecito**
  - 1.7 Billion transistors
  - Dual Core IA/64
- **Intel Tanglewood**
  - Dual Core IA/64
- **Intel Pentium D (Smithfield)**
- **Intel Pentium Extreme**
  - 3.2GHz Dual Core
- **Intel Dempsey**
  - Dual Core Xeon
- **Intel Yonah**
  - Dual Core Mobile
- **AMD Opteron**
  - Dual Core
- **IBM Power 4 and 5**
  - Dual Cores Since 2001
- **IBM Cell**
  - Scalable Multicore
- **IBM Power 6**
  - Dual Core
- **Sun Olympus and Niagara**
  - 8 Processor Cores
- **MIT Raw**
  - 16 Cores
  - Since 2002
- **Intel Tejas & Jayhawk Unicore (4GHz P4)**
- **Intel Tejas & Jayhawk**
  - 1.7 Billion transistors
  - Dual Core IA/64
- **Intel Tanglewood**
  - Dual Core IA/64
- **Intel Pentium D (Smithfield)**
- **Intel Pentium Extreme**
  - 3.2GHz Dual Core
- **Intel Dempsey**
  - Dual Core Xeon
- **Intel Yonah**
  - Dual Core Mobile
- **AMD Opteron**
  - Dual Core
- **IBM Power 4 and 5**
  - Dual Cores Since 2001
- **IBM Power 6**
  - Dual Core
- **Sun Olympus and Niagara**
  - 8 Processor Cores
- **MIT Raw**
  - 16 Cores
  - Since 2002
- **Intel Tejas & Jayhawk Unicore (4GHz P4)**
- **Intel Tejas & Jayhawk**
  - 1.7 Billion transistors
  - Dual Core IA/64
- **Intel Tanglewood**
  - Dual Core IA/64
- **Intel Pentium D (Smithfield)**
- **Intel Pentium Extreme**
  - 3.2GHz Dual Core
- **Intel Dempsey**
  - Dual Core Xeon
- **Intel Yonah**
  - Dual Core Mobile
- **AMD Opteron**
  - Dual Core
- **IBM Power 4 and 5**
  - Dual Cores Since 2001
- **IBM Power 6**
  - Dual Core
- **Sun Olympus and Niagara**
  - 8 Processor Cores
- **MIT Raw**
  - 16 Cores
  - Since 2002
- **Intel Tejas & Jayhawk Unicore (4GHz P4)**
- **Intel Tejas & Jayhawk**
  - 1.7 Billion transistors
  - Dual Core IA/64
- **Intel Tanglewood**
  - Dual Core IA/64
- **Intel Pentium D (Smithfield)**
- **Intel Pentium Extreme**
  - 3.2GHz Dual Core
- **Intel Dempsey**
  - Dual Core Xeon
- **Intel Yonah**
  - Dual Core Mobile
- **AMD Opteron**
  - Dual Core
- **IBM Power 4 and 5**
  - Dual Cores Since 2001
- **IBM Power 6**
  - Dual Core
- **Sun Olympus and Niagara**
  - 8 Processor Cores
- **MIT Raw**
  - 16 Cores
  - Since 2002
- **Intel Tejas & Jayhawk Unicore (4GHz P4)**

Timeline:
- **2H 2004**
- **1H 2005**
- **2H 2005**
- **1H 2006**
- **2H 2006**
What is Multicore?

- Multiple, externally visible processors on a single die where the processors have independent control-flow, separate internal state and no critical resource sharing.

- Multicores have many names…
  - Chip Multiprocessor (CMP)
  - Tiled Processor
  - ....
Why move to Multicores?

- Many issues with scaling a unicore
  - Power
  - Efficiency
  - Complexity
  - Wire Delay
  - Diminishing returns from optimizing a single instruction stream
Moore’s Law: Transistors Well Spent?

Pentium 4 (217mm² / .18µ)
- Integer Core
- FPU
- MMX/SSE
- L1 Data Cache
- L2 Data Cache
- Bus Logic
- L3 Tag
- L2 Tag
- Trace Cache
- Decode
- Fetch
- Bus Control
- Op Scheduling

4004 (12mm² / 8µ)
- Integer Core
- FPU
- MMX/SSE
- L1 Data Cache
- Cache

Itanium 2 (421mm² / .18µ)
- Floating Point Integer Core
- L1 Cache
- L2 Data Cache
- Bus Logic
- L3 Tag
- L2 Tag
- Trace Cache
- Decode Fetch
- Bus Control

Moore’s Law:
- 4004 (12mm² / 8µ)
- 4080
- 8008
- 4004
- 8080
- 8086
- 286
- 386
- 486
- Pentium
- P2
- P3
- P4
- Itanium
- Itanium 2

transistors

1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
1,000

Outline

- Introduction
- Overview of Multicores
- Success Criteria for a Compiler
- Data Level Parallelism
- Instruction Level Parallelism
- Language Exposed Parallelism
- Conclusion
Impact of Multicores

- How does going from Multiprocessors to Multicores impact programs?

- What changed?

- Where is the Impact?
  - Communication Bandwidth
  - Communication Latency
Communication Bandwidth

- How much data can be communicated between two cores?

- What changed?
  - Number of Wires
    - IO is the true bottleneck
    - On-chip wire density is very high
  - Clock rate
    - IO is slower than on-chip
  - Multiplexing
    - No sharing of pins

- Impact on programming model?
  - Massive data exchange is possible
  - Data movement is not the bottleneck
    → locality is not that important

32 Giga bits/sec ~300 Tera bits/sec

10,000X
Communication Latency

- How long does it take for a round trip communication?

- What changed?
  - Length of wire
    - Very short wires are faster
  - Pipeline stages
    - No multiplexing
    - On-chip is much closer

- Impact on programming model?
  - Ultra-fast synchronization
  - Can run real-time apps on multiple cores
Past, Present and the Future?

Traditional Multiprocessor

Basic Multicore
IBM Power5

Integrated Multicore
16 Tile MIT Raw

Memory

Memory

Memory

Memory
Outline

- Introduction
- Overview of Multicores
- Success Criteria for a Compiler
- Data Level Parallelism
- Instruction Level Parallelism
- Language Exposed Parallelism
- Conclusion
When is a compiler successful as a general purpose tool?

- **General Purpose**
  - Programs compiled with the compiler are in daily use by non-expert users
  - Used by many programmers
  - Used in open source and commercial settings

- **Research / niche**
  - You know the names of all the users
Success Criteria

1. Effective
2. Stable
3. General
4. Scalable
5. Simple
1: Effective

- Good performance improvements on most programs

- The speedup graph goes here!
2: Stable

- Simple change in the program should not drastically change the performance!
  - Otherwise need to understand the compiler inside-out
  - Programmers want to treat the compiler as a black box
3: General

- Support the diversity of programs
  - Support Real Languages: C, C++, (Java)
    - Handle rich control and data structures
    - Tolerate aliasing of pointers
  - Support Real Environments
    - Separate compilation
    - Statically and dynamically linked libraries
- Work beyond an ideal laboratory setting
4: Scalable

- Real applications are large!
  - Algorithm should scale
    - polynomial or exponential in the program size doesn’t work
- Real Programs are Dynamic
  - Dynamically loaded libraries
  - Dynamically generated code
- Whole program analysis tractable?
5: Simple

- Aggressive analysis and complex transformation lead to:
  - Buggy compilers!
    - Programmers want to trust their compiler!
    - How do you manage a software project when the compiler is broken?
  - Long time to develop

- Simple compiler $\Rightarrow$ fast compile-times

- Current compilers are too complex!

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNU GCC</td>
<td>~1.2 million</td>
</tr>
<tr>
<td>SUIF</td>
<td>~250,000</td>
</tr>
<tr>
<td>Open Research Compiler</td>
<td>~3.5 million</td>
</tr>
<tr>
<td>Trimaran</td>
<td>~800,000</td>
</tr>
<tr>
<td>StreamIt</td>
<td>~300,000</td>
</tr>
</tbody>
</table>
Data Level Parallelism

- Identify loops where each iteration can run in parallel
  - DOALL parallelism

- What affects performance?
  - Parallelism Coverage
  - Granularity of Parallelism
  - Data Locality

```
TDT = DT
MP1 = M+1
NP1 = N+1
EL = N*DX
PI = 4.D0*ATAN(1.D0)
TPI = PI+PI
D1 = TPI/M
DJ = TPI/N
PCF = PI*PI*A*A/(EL*EL)

DO 50 J=1,NP1
   DO 50 I=1,MP1
       PSI(I,J) = A*SIN((I-.5D0)*DI)*SIN((J-.5D0)*DJ)
P(I,J) = PCF*(COS(2.D0)
CONTINUE

DO 60 J=1,N
   DO 60 I=1,M
       U(I+1,J) = -(PSI(I+1,J+1)
       -PSI(I+1,J))/DY
       V(I,J+1) = (PSI(I+1,J+1)
       -PSI(I,J+1))/DX
CONTINUE
```
Parallelism Coverage

- **Amdahl’s Law**
  
  *Performance improvement to be gained from faster mode of execution is limited by the fraction of the time the faster mode can be used*

- **Find more parallelism**
  - Interprocedural analysis
  - Alias analysis
  - Data-flow analysis
  - ……
SUIF Parallelizer Results

SPEC95fp, SPEC92fp, Nas, Perfect Benchmark Suites
On a 8 processor Silicon Graphics Challenge (200MHz MIPS R4000)
Granularity of Parallelism

- Synchronization is expensive
- Need to find very large parallel regions → coarse-grain loop nests
- Heroic analysis required
Granularity of Parallelism

- Synchronization is expensive
- Need to find very large parallel regions → coarse-grain loop nests
- Heroic analysis required
- Single unanalyzable line →

turb3d in SPEC95fp
Granularity of Parallelism

- Synchronization is expensive
- Need to find very large parallel regions → coarse-grain loop nests
- Heroic analysis required
- Single unanalyzable line →
  - Small Reduction in Coverage
  - Drastic Reduction in Granularity

turb3d in SPEC95fp
SUIF Parallelizer Results

Parallelism Coverage vs. Granularity of Parallelism Graph

Speedup
Data Locality

- Non-local data →
  - Stalls due to latency
  - Serialize when lack of bandwidth

- Data Transformations
  - Global impact
  - Whole program analysis
DLP on Multiprocessors: Current State

- Huge body of work over the years.
  - Vectorization in the ’80s
  - High Performance Computing in ’90s

- Commercial DLP compilers exist
  - But…only a very small user community

- Can multicores make DLP mainstream?
Effectiveness

Main Issue
- Parallelism Coverage

Compiling to Multiprocessors
- Amdahl’s law
  - Many programs have no loop-level parallelism

Compiling to Multicores
- Nothing much has changed
Stability

- Main Issue
  - Granularity of Parallelism

- Compiling for Multiprocessors
  - Unpredictable, drastic granularity changes reduce the stability

- Compiling for Multicores
  - Low latency $\rightarrow$ granularity is less important
Generality

- **Main Issue**
  - Changes in general purpose programming styles over time impacts compilation

- **Compiling for Multiprocessors** (*In the good old days*)
  - Mainly FORTRAN
    - Loop nests and Arrays

- **Compiling for Multicores**
  - Modern languages/programs are hard to analyze
    - Aliasing (C, C++ and Java)
    - Complex structures (lists, sets, trees)
    - Complex control (concurrency, recursion)
    - Dynamic (DLLs, Dynamically generated code)
Scalability

- **Main Issue**
  - Whole program analysis and global transformations don’t scale

- **Compiling for Multiprocessors**
  - Interprocedural analysis needed to improve granularity
  - Most data transformations have global impact

- **Compiling for Multicores**
  - High bandwidth and low latency → no data transformations
  - Low latency → granularity improvements not important
Simplicity

- Main Issue
  - Parallelizing compilers are exceedingly complex

- Compiling for Multiprocessors
  - Heroic interprocedural analysis and global transformations are required because of high latency and low bandwidth

- Compiling for Multicores
  - Hardware is a lot more forgiving…
  - But…modern languages and programs make life difficult
Outline

- Introduction
- Overview of Multicores
- Success Criteria for a Compiler
- Data Level Parallelism
- **Instruction Level Parallelism**
- Language Exposed Parallelism
- Conclusion
Instruction Level parallelism on a Unicore

\[
\begin{align*}
tmp0 &= (seed \times 3 + 2) / 2 \\
tmp1 &= seed \times v1 + 2 \\
tmp2 &= seed \times v2 + 2 \\
tmp3 &= (seed \times 6 + 2) / 3 \\
v2 &= (tmp1 - tmp3) \times 5 \\
v1 &= (tmp1 + tmp2) \times 3 \\
v0 &= tmp0 - v1 \\
v3 &= tmp3 - v2
\end{align*}
\]

- Programs have ILP
- Modern processors extract the ILP
  - Superscalars → Hardware
  - VLIW → Compiler
Scalar Operand Network (SON)

- Moves results of an operation to dependent instructions
- Superscalars → in Hardware
- What makes a good SON?
Scalar Operand Network (SON)

- Moves results of an operation to dependent instructions
- Superscalars → in Hardware
- What makes a good SON?
  - Low latency from producer to consumer

```
pval5 = seed.0 * 6.0
seed.0 = seed
```
Scalar Operand Network (SON)

- Moves results of an operation to dependent instructions
- Superscalars → in Hardware
- What makes a good SON?
  - Low latency from producer to consumer
  - Low occupancy at the producer and consumer
Scalar Operand Network (SON)

- Moves results of an operation to dependent instructions
- Superscalars → in Hardware
- What makes a good SON?
  - Low latency from producer to consumer
  - Low occupancy at the producer and consumer
  - High bandwidth for multiple operations
Is an Integrated Multicore Ready to be a Scalar Operand Network?

<table>
<thead>
<tr>
<th></th>
<th>Traditional Multiprocessor</th>
<th>Basic Multicore</th>
<th>Integrated Multicore</th>
<th>VLIW Unicore</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Latency</strong></td>
<td>60</td>
<td>4</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>(cycles)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Occupancy</strong></td>
<td>50</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(instructions)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Bandwidth</strong></td>
<td>1</td>
<td>2</td>
<td>16</td>
<td>6</td>
</tr>
<tr>
<td>(operands/cycle)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Scalable Scalar Operand Network?

- Unicores
  - N2 connectivity
  - Need to cluster → introduces latency

- Integrated Multicores
  - No bottlenecks in scaling
Compiler Support for Instruction Level Parallelism

- Accepted general purpose technique
  - Enhance the performance of superscalars
  - Essential for VLIW

- Instruction Scheduling
  - List scheduling or Software pipelining
ILP on Integrated Multicores: Space-Time Instruction Scheduling

- Partition, placement, route and schedule
- Similar to Clustered VLIW
Handling Control Flow

- Asynchronous global branching
  - Propagate the branch condition to all the tiles as part of the basic block schedule
  - When finished with the basic block execution asynchronously switch to another basic block schedule depending on the branch condition

\[
x = \text{cmp } a, b
\]

\[
\text{br } x
\]
Raw Performance

- Dense Matrix
- Multimedia
- Irregular

- Cholesky
- Mxm
- Tomcatv
- Vpenta
- Btrix
- Life
- Jacobi
- Adpcm-encode
- SHA
- MPEG-kernel
- Moldyn
- Unstruct

• 32 tile Raw
## Success Criteria

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Effective</td>
</tr>
<tr>
<td></td>
<td>- If ILP exists → same</td>
</tr>
<tr>
<td>2.</td>
<td>Stable</td>
</tr>
<tr>
<td></td>
<td>-Localized optimization → similar</td>
</tr>
<tr>
<td>3.</td>
<td>General</td>
</tr>
<tr>
<td></td>
<td>- Applies to same type of applications</td>
</tr>
<tr>
<td>4.</td>
<td>Scalable</td>
</tr>
<tr>
<td></td>
<td>- Local analysis → similar</td>
</tr>
<tr>
<td>5.</td>
<td>Simple</td>
</tr>
<tr>
<td></td>
<td>- Deeper analysis and more transformations</td>
</tr>
</tbody>
</table>
Outline

- Introduction
- Overview of Multicores
- Success Criteria for a Compiler
- Data Level Parallelism
- Instruction Level Parallelism
- Language Exposed Parallelism
- Conclusion
Languages are out-of-touch with Architecture

- Two choices:
  - Develop cool architecture with complicated, ad-hoc language
  - Bend over backwards to support old languages like C/C++
Supporting von Neumann Languages

- Why C (FORTRAN, C++ etc.) became very successful?
  - Abstracted out the differences of von Neumann machines
    - Register set structure
    - Functional units and capabilities
    - Pipeline depth/width
    - Memory/cache organization
  - Directly expose the common properties
    - Single memory image
    - Single control-flow
    - A clear notion of time
  - Can have a very efficient mapping to a von Neumann machine
  - “C is the portable machine language for von Neumann machines”

- Today von Neumann languages are a curse
  - We have squeezed out all the performance out of C
  - We can build more powerful machines
  - But, cannot map C into next generation machines
  - Need better languages with more information for optimization
New Languages for Cool Architectures

- Processor specific languages
  - Not portable

- Increase the burden on programmers
  - Many more tasks for the programmer (parallelism annotations, memory alias annotations)
  - But, no software engineering benefits

- Assembly hacker mentality
  - Worked so hard on putting architectural features
  - Don’t want compilers to squander it away
  - Proof-of-concept done in assembly

- Architects don’t know how to design languages
What Motivates Language Designers

- Primary Motivation → Programmer Productivity
  - Raising the abstraction layer
  - Increasing the expressiveness
  - Facilitating design, development, debugging, maintenance of large complex applications

- Design Considerations
  - Abstraction → Reduce the work programmers have to do
  - Malleablility → Reduce the interdependencies
  - Safety → Use types to prevent runtime errors
  - Portability → Architecture/system independent

- No consideration given for the architecture
  - For them, performance is a non-issue!
Is There a Win-Win Solution

- Languages that increase programmer productivity while making it easier to compile
Example: StreamIt, A spatially-aware Language

- A language for streaming applications
  - Provides high-level stream abstraction
    - Exposes Pipeline Parallelism
  - Improves programmer productivity
- Breaks the von Neumann language barrier
  - Each filter has its own control-flow
  - Each filter has its own address space
  - No global time
  - Explicit data movement between filters
  - Compiler is free to reorganize the computation
Example: Radar Array Front End
Radar Array Front End on Raw

- Blocked on Network
- Executing Instructions
- Pipeline Stall
StreamIt language exposes the data movement
- Graph structure is architecture independent
- Each architecture is different in granularity and topology
  - Communication is exposed to the compiler
- The compiler needs to efficiently bridge the abstraction
  - Map the computation and communication pattern of the program to the tiles, memory and the communication substrate
Bridging the Abstraction layers

StreamIt language exposes the data movement
  - Graph structure is architecture independent

Each architecture is different in granularity and topology
  - Communication is exposed to the compiler

The compiler needs to efficiently bridge the abstraction
  - Map the computation and communication pattern of the program to the tiles, memory and the communication substrate

The StreamIt Compiler
  - Partitioning
  - Placement
  - Scheduling
  - Code generation
Optimized Performance for Radar Array Front End on Raw
Performance

<table>
<thead>
<tr>
<th></th>
<th>MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>C program</td>
<td>240</td>
</tr>
<tr>
<td>1 GHz Pentium III</td>
<td></td>
</tr>
<tr>
<td>C program</td>
<td>19</td>
</tr>
<tr>
<td>420 MHz single tile Raw</td>
<td></td>
</tr>
<tr>
<td>Unoptimized Streamlt</td>
<td>640</td>
</tr>
<tr>
<td>420 MHz 64 tile Raw</td>
<td></td>
</tr>
<tr>
<td>Optimized Streamlt</td>
<td>1,430</td>
</tr>
<tr>
<td>420 MHz 16 tile Raw</td>
<td></td>
</tr>
</tbody>
</table>
### Success Criteria

<table>
<thead>
<tr>
<th>Success Criteria</th>
<th>Effective</th>
<th>Stable</th>
<th>General</th>
<th>Scalable</th>
<th>Simple</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Information available for more optimizations</td>
<td>Much more analyzable</td>
<td>Domain-Specific</td>
<td>No global data structures</td>
<td>Heroic analysis vs. more transformations</td>
</tr>
</tbody>
</table>

**Compiler for:**

- **Von Neumann Languages:**
  - Effective: Green
  - Stable: Yellow
  - General: Green
  - Scalable: Yellow
  - Simple: Green

- **Stream Language:**
  - Effective: Green
  - Stable: Yellow
  - General: Green
  - Scalable: Yellow
  - Simple: Green
Outline

- Introduction
- Overview of Multicores
- Success Criteria for a Compiler
- Data Level Parallelism
- Instruction Level Parallelism
- Language Exposed Parallelism
- Conclusion
## Overview of Success Criteria

<table>
<thead>
<tr>
<th></th>
<th>Data Level Parallelism</th>
<th>Instruction Level Parallelism</th>
<th>Language Exposed Parallelism on Multicore</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Effective</td>
<td></td>
<td>Von Neumann Languages</td>
</tr>
<tr>
<td>2.</td>
<td>Stable</td>
<td></td>
<td>Stream Language</td>
</tr>
<tr>
<td>3.</td>
<td>General</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>Scalable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>Simple</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Can Compilers take on Multicores?

- Success Criteria is Somewhat Mixed
- But....
  - Don’t need to compete with unicores
  - Multicores will be available regardless
- New Opportunities
  - Architectural advances in integrated multicores
  - Domain specific languages
  - Possible compiler support for using multicores for other than parallelism
    - Security Enforcement
    - Program Introspection
    - ISA extensions

http://cag.csail.mit.edu/commit
http://www.determina.com