Whither Sequential?: Rethinking Parallel Execution for Future Multicore Processors

Guri Sohi
(with Matt Allen, Srinath Sridharan, Gagan Gupta, Hongil Yoon)
University of Wisconsin-Madison
Outline

• Lessons from microprocessor evolution
• Multicore generation and its implications
• **Sequential programming; parallel execution**
  – Dataflow-style parallel execution
  – Managing perils of excessive parallelism
• Concluding thoughts
Beyond pipelining to ILP

• Late 1980s to mid 1990s
• Search for “post RISC” architecture
  – More accurately, instruction processing model
• Desire to do more than one instruction per cycle—exploit ILP
• VLIW/EPIC
• Out-of-order (OOO) superscalar
VLIW/EPIC School

• Descendants of HPC computing experience (array processors)
• Search for independence (by compiler)
• Express independence in static program
  • Take program/algorith parallelism and mold it to given execution schedule for exploiting parallelism
• Strive for “efficiency”
  – Static scheduling to “saturate” a resource
VLIW/EPIC School

• Creating effective parallel representations (statically) introduces several problems
  – Predication
  – Statically scheduling loads
  – Exception handling
  – Recovery code

• Lots of research addressing these problems
OOO Superscalar

• Not from HPC school
  – Non-scientific influence important (e.g., branch prediction)

• No static search/representation for independence
  – Arguably statically representing dependent operations (e.g., accumulator) would help
    • Improvements to superscalar were in this direction
OOO Superscalar

- Create dynamic parallel execution from sequential static representation
  - dynamic dependence information accurate
  - execution schedule flexible
- Parallelism in application/algorithm required but program representation is sequential
- None of the problems associated with trying to create a parallel representation statically
VLIW/Superscalar

• Superscalar not as “efficient” as VLIW
  – Has all this extra hardware for dynamic dataflow execution
  – Hard to “saturate” a resource like VLIW

• But provides natural (sequential) interface for program generator

• Much more adaptable to run time uncertainties
  – E.g., resources changing dynamically
Lessons from VLIW/Superscalar

• Wisdom from HPC is natural to apply, but is this a good idea?
• Hardware “efficiency” may not be the best notion of “efficiency”
• Parallel execution can be achieved even without a parallel representation
Lessons from VLIW/Superscalar

• Dataflow execution is much more flexible and adaptable than control flow execution
  – E.g., new forms of speculation easily added
  – E.g., resource architecture easily changed

• (Hardware) overheads for achieving dynamic dataflow execution worth the effort
On to Multicore

Parallel execution hardware is everywhere

*What are we going to do?*
What Are We Going to Do?

• Leverage knowledge from 4+ decade experience in parallel processing
  – Mostly in scientific/HPC arena
  – Create a program with parallelism expressed statically

• Teach students about parallel programming
  – Eventually statically parallel programs will be pervasive and universal

• Have expert programmers write parallel libraries

• Etc, etc
Lots of Experience with Parallelism

• Models, languages, hardware
• Main issues: expressing, exposing, handling
• Examples: Actors, Shared memory, Message passing, Dataflow, Task Execution systems, Futures, Concurrent OO, Inspector-Executor, Speculative systems, Thread-level speculation, SIMD, Vectors, Jade, Linda, Cilk, Multilisp, Id, PVM, MPI, OpenMP, Charm, .................

• Parallel Programming vs. Parallel Execution
Canonical Parallel Execution Model

A: Analyze program to *identify independence* in program
   – independent portions executed in parallel

B: Create static representation of independence
   – *synchronization* to satisfy independence assumption

C: Dynamic parallel execution unwinds as per static representation
   – potential consequences due to static assumptions
What We Desire

• Parallel execution
  – Regardless of what the static program is.

• Achieve this without significant rethinking in common programming practices/styles
Control and Data-Driven

• Parallelism is due to operations on disjoint sets of data

• Static representation typically control-driven
  – Most if not all practical programming languages
  – Need to ensure that execution is on disjoint data
    • Use synchronization to ensure
    • But nature of data revealed dynamically
  – Potentially obscures application parallelism
    • Conflates parallelism with execution schedule
Control and Data-Driven

• Data-driven focuses on data dependence
  – Naturally separates operations on disjoint data
  – Can be easily derived from total (sequential) order

• Remember VLIW (control driven and parallel) and OOO superscalar (data-driven from sequential)

• My view: data-driven models much more powerful and practical than control-driven
  – How to get such a model for future hardware?
Data-driven Parallel Execution

• Would like to achieve data-driven parallel execution

• Do we need new programming languages? No

• Do we need to learn to program in parallel? No

• Do we need to throw out everything we have and start from scratch? No

• How are we going to deal with hardware “uncertainties” in the future?
Hardware Going Forward

• Multiple general-purpose processing cores
  – dynamically specialized
• Some “special-purpose” hardware
  • GPUs, specialized units, accelerators, etc.
• Over-provisioning: pool of available (i.e., powered on) resources might change frequently
  – Now called “dark silicon”
• Frequent errors
  – E.g., due to near threshold operation
Meanwhile

• Software is everywhere and its presence is going to continue to increase
  – Software not “shrink wrap” any more
• Programmer productivity is paramount; code “efficiency” less important?
• Modularity, abstraction, encapsulation, information hiding, etc., become the norm
• Object-oriented programming principles (and languages) becoming ubiquitous
Going Forward

• Programmers are going to continue to express computation in familiar ways: sequential, modular, object-oriented programs, with heavy use of abstraction and encapsulation

• We want them to use a parallel algorithm, but don’t necessarily want them to statically express parallelism

How are we going to make it work?
• Then: abstraction is a friend of software
• Now: abstraction is going to help us use future hardware

View program as a **sequential** representation of abstractions of computations which are going to be executed on a (dynamically) heterogeneous pool of hardware resources in a dataflow manner
What?

• Data-driven parallel execution from statically sequential program
  – Data-centric (dynamic) expression of dependence
    • No expression of independence
  – Determinate, race-free execution
  – No locks, no memory models and no explicit synchronization
  – Easier to write, debug, and maintain
  – Comparable or better performance than conventional parallel models without their drawbacks/pitfalls
How? Big Picture

- Sequential imperative programming language
  - Leverage modern programming principles
  - Provision for worst-case scenarios

- Achieving dataflow parallel execution
Program Decomposition

Static Program

object A
..

while (cond) {
  function F ( )
}
function F’ ( )

Dynamic Sequence

.. 
evaluate cond (PS1)
F1 ( )
evaluate cond (PS2)
F2 ( )
evaluate cond (PS3)
F3 ( )
..
Program Decomposition

Static Program

object A

while (cond) {
  function F ( )
}
function F' ( )

Functions

.. 
evaluate cond (PS1)
F1 ( )
evaluate cond (PS2)
F2 ( )
evaluate cond (PS3)
F3 ( )
..
Program Decomposition

Static Program

object A

while (cond) {
    function F ( )
}
function F' ( )

Program Segments

evaluate cond (PS1)
F1 ( )
evaluate cond (PS2)
F2 ( )
evaluate cond (PS3)
F3 ( )

..
Program Decomposition

- Collection of computations
- Implicit order

Dynamic Sequence

\[
\begin{align*}
&\text{PS1} \\
&\text{F1( )} \\
&\text{PS2} \\
&\text{F2( )} \\
&\text{PS3} \\
&\text{F3( )} \\
&\ldots
\end{align*}
\]
Program Execution

Multicore Processor

Dynamic Sequence

.. 
PS1
F1 ( )
PS2
F2 ( )
PS3
F3 ( )
..
Program Execution

Multicore Processor

Dynamic Sequence

..  
PS1
F1( )
PS2
F2( )
PS3
F3( )
..
Program Execution

Multicore Processor

Dynamic Sequence

.. 
PS1 
F1 ( ) 
PS2 
F2 ( ) 
PS3 
F3 ( ) 
..
Program Execution

Multicore Processor

Dynamic Sequence

PS1
F1 ( )
PS2
F2 ( )
PS3
F3 ( )

..
Program Execution

Multicore Processor

Dynamic Sequence

\[
\ldots
\]

PS1

F1

PS2

F2

PS3

F3

\[
\ldots
\]
Program Execution

Multicore Processor

Dynamic Sequence

..  
PS1  
F1(  )  
PS2  
F2(  )  
PS3  
F3(  )  
..
Program Execution

Multicore Processor

Dynamic Sequence

. .

PS1

F1 ( )

PS2

F2 ( )

PS3

F3 ( )

. .
Program Execution

- Implicit order
  - Hindrance
  + Exploitable

Dynamic Sequence

.. 
F1( )
PS1
F2( )
PS2
F3( )
PS3
..
object A

while (cond) {
    function F ( )
}
function F' ( )

“Instruction”
object A
..

while (cond) {
    function F ( ..interface.. )
}

function F’ ( )
object A

while (cond) {
  function F ({wr_set} {rd_set})
}

function F’ ( )
object A

..

while (cond) {
    function F ({wr_set} {rd_set})
}

function F' ( )
Achieving Dataflow Parallel Execution

- Our view of a program
  - Computations
  - Level of abstraction

- Classical dataflow machines
  - Tokens: data dependence
  - Resource management
Proposed Model
- Level of abstraction
- Imperative programs can mutate data

Token protocol
- Data dependence

Program order
object A

while (cond) {
    function F ( {wr_set} {rd_set} )
} 
function F' ( )
object A

while (cond) {
    function F ({wr_set} {rd_set})
}
function F’ ( )

F1 {B, C} {A}
F2 {D} {A}
F3 {A, E} {I}
F4 {B} {D}
F5 {B} {D}
F6 {G} {H}
object A
...
while (cond) {
    function F ({wr_set} {rd_set})
}
function F’ ({{}} {{}})

F1 {B, C} {A}
F2 {D} {A}
F3 {A, E} {I}
F4 {B} {D}
F5 {B} {D}
F6 {G} {H}
F’ {??} {??}
Data Dependence

F1 \{B, C\} \{A\}
F2 \{D\} \{A\}
F3 \{A, E\} \{I\}
F4 \{B\} \{D\}
F5 \{B\} \{D\}
F6 \{G\} \{H\}
F’ \{?\} \{?\}
Prototype

- Runtime library (C++ currently)
- Implements token protocol
- Achieves parallel execution
Composing Programs

Sequential Program

Object A
.. 

while (cond) {
    function F: ( .. )
}

Function F': {??} {??}

Program in Proposed Model

Object A
.. 

while (cond) {
    }

F'( .. );
Composing Programs

Sequential Program

Object A
.. 

while (cond) {
    function F: ( .. )
}

Function F’: {??} {??}

Program in Proposed Model

Object A : Token
.. 

while (cond) {
    
}

F'( .. );
Composing Programs

Sequential Program

Object A

.. 

while (cond) {
    function F: ( .. )
}

Function F’: {??} {??}

Program in Proposed Model

Object A : Token

.. 

while (cond) {
    df_execute ( &F);
}

} 

F’( .. );
Sequential Program

Object A
..

while (cond) {
    function F: ( .. )
}

Function F’: {??} {??}

Program in Proposed Model

Object A : Token
..

while (cond) {
    df_execute (wr_set, rd_set, &F);
}

df_end ( );
F'( .. );
Parallel Execution Mechanics

Application

Runtime
  Interface
  Deques
  Task-stealing Schedulers

Multicore Processor
Evaluation

- Ease of programming (qualitative)
- Performance & overheads
- Setup
  - Microbenchmarks & benchmarks
  - 3 stock multicore machines
bzip2

Sequential

..

..

while (!EOF) {

    block = new block_t (InFile, size);

    compress (block)
    file_write (OpFile, block);

}
for (; ; ) { 
  pthread_mutex_lock(fifo->mut);
  while (fifo->empty) { 
    if (allDone == 1) { 
      pthread_mutex_unlock(fifo->mut);
      return (NULL);
    } 
    GetSystemTime(&systemtime);
    SystemTimeToFileTime(&systemtime, (FILETIME*)filetime);
    waitTimer.tv_sec = filetime.QuadPart / 10000000;
    waitTimer.tv_nsec = filetime.QuadPart -
      (LONGLONG)(waitTimer.tv_sec * 10000000) * 10;
    waitTimer.tv_sec++;
   pret = pthread_cond_timedwait(fifo->notEmpty, 
      fifo->mut, &waitTimer);
    FileData = queueDel(fifo, &inSize, &blockNum);
    pthread_mutex_unlock(fifo->mut);
    pret = pthread_cond_signal(fifo->notFull);
    outSize = (int)((inSize*1.01)+600);
    pthread_mutex_lock(MemMutex);
    CompressedData = new char[outSize];
    pthread_mutex_unlock(MemMutex)
    if (CompressedData == NULL) {
      while ((currBlock < NumBlocks) | | (allDone == 0)) { 
        pthread_mutex_lock(OutMutex);
        if ((OutputBuffer.size() == 0) | |
          (OutputBuffer[currBlock].bufSize < 1) | |
          (OutputBuffer[currBlock].buf == NULL)) { 
          pthread_mutex_unlock(OutMutex);
          usleep(50000);
          continue;
        } else
          pthread_mutex_unlock(OutMutex);
      ret = write(hOutfile, OutputBuffer[currBlock].buf,
        OutputBuffer[currBlock].bufSize);
      CompressedSize += ret;
      pthread_mutex_lock(MemMutex);
      if (OutputBuffer[currBlock].buf != NULL) { 
        delete [] OutputBuffer[currBlock].buf;
        NumBufferedBlocks--;
      }
      pthread_mutex_unlock(MemMutex);
      pthread_mutex_unlock(OutMutex);
      currBlock++;
    } 
  pthread_mutex_unlock(MemMutex);
  pthread_mutex_unlock(OutMutex); 
  } 
}
bzip2: Pthread
bzip2: Proposed Model

Sequential

.. ..
.. while (!EOF) {
    block = new block_t (InFile, size);
    compress (block)
    file_write (OpFile, block);
}

Proposed Model

.. ..
.. while (!EOF) {
    block = new block_t (InFile, size);
}

..
bzip2: Proposed Model

Sequential

..  
..  
while (!EOF) {

    block = new block_t (InFile, size);

    compress (block)
    file_write (OpFile, block);
}

..  

Proposed Model

..  

opset.insert (OpFile);
while (!EOF) {

    block = new block_t (InFile, size);
    blset.insert (block);
    df_execute (blset, &compress);
    df_execute (opset, blset, &file_write);
}

..  
## Results

<table>
<thead>
<tr>
<th>Benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>barneshut</td>
</tr>
<tr>
<td>barneshut-LG</td>
</tr>
<tr>
<td>blackscholes</td>
</tr>
<tr>
<td>blackscholes-LG</td>
</tr>
<tr>
<td>bzip2</td>
</tr>
<tr>
<td>bzip2-LG</td>
</tr>
<tr>
<td>dedup</td>
</tr>
<tr>
<td>histogram</td>
</tr>
<tr>
<td>reverse_index</td>
</tr>
<tr>
<td>microbenchmarks</td>
</tr>
</tbody>
</table>

- **Machines**
  - 4-core/8-thread Nehalem Core i7
  - 16-core AMD Opteron 8350
  - 32-core AMD Opteron 8356

- **Input sizes**
  - Small, Medium, Large
Performance Evaluation
Results: Competitive Performance

![Bar chart showing competitive performance across different machines.]

- **Harmonic Mean (LG)**
  - Y-axis: Speedup
  - X-axis: Machines
  - Comparisons:
    - PT vs. DF
    - Machines: 8x Nehalem, 16x Barcelona, 32x Barcelona
How? Summary

• Serialize computations to same object
  – Enforce dependence

• Do not look for/represent independence
  – Falls out as an effect of enforcing dependence

• Updates to given state in same order as in sequential program
  – Determinate
  – No races
  – If sequential correct; parallel execution is correct (same input)
Environment Going Forward

• Multicore processors everywhere
• Won’t know (dynamic) capabilities of hardware or operating environment when writing code
  – Different machines may have different hardware and software resources
  – Different programs may be running in a multiprogrammed environment
• How much parallelism should we have?
  – Too much for dynamic situation can be problematic!
void parallel_memcpyp(void* dest, void* src, size_t length)
{
    char* d = (char*) dest;
    char* s = (char*) src;
    size_t chunks = length / NUM_THREADS;
    parallel_for ( i=0; i < chunks; i++)
    {
        for(j=i*chunks; j<(i+1)*chunks;j++)
        {
            dest[j] = src[j];
        }
    }
}
Parallelism and Resources

- Same input data on three different machines
- Performance degrades as more threads added

![Graph showing speedup with respect to sequential execution for different numbers of threads and configurations.](image)
Parallelism and Resources

• Point of degradation is different on different machines

![Graph showing speedup with varying number of threads for different configurations.](image-url)
Parallelism and Resources

- Different input data on the same machine (4X4-B)
- Performance degrades as more threads added

![Graph showing speedup with respect to sequential execution for different numbers of threads for Small Trad and Large Trad.]
Parallelism and Resources

- Point of degradation is different for different input sizes

![Graph showing speedup with respect to sequential execution for different numbers of threads]

- Speedup wrt sequential execution
- Num_threads
- Small Trad
- Large Trad
Why performance degradation?

• Contention for resources
  – memory bandwidth
  – software locks in user program
  – lock on kernel data-structure
  – size of on-chip caches
  – interconnection bandwidth
  – several other possibilities
Desired result

• Allow less or no performance degradation
How to achieve it?

• Expose parallelism into dynamic computation environment in a controlled manner
  – Establish single point of control introducing computation tasks into environment
  – Detect if parallelism might be excessive
    • e.g., rate of increase in task execution time
    • e.g., rate of temperature increase
  – Control exposure of parallelism by throttling or increasing the number of simultaneous dynamic worker threads (tasks)
Dedicated Environment

Energy

Time

Joules wrt sequential execution

Seconds wrt sequential execution

TBB
FDT
PM
PD

Histogram
Stream
Reverse Index
Hash Join
Barneshut
RE

Histogram
Stream
Reverse Index
Hash Join
Barneshut
RE
Multiprogrammed Environment

![Energy Chart]

- **Scenario 1**
- **Scenario 2**
- **Scenario 3**

**Energy**
- **TBB**
- **PD**

<table>
<thead>
<tr>
<th>Process</th>
<th>Scenario 1</th>
<th>Scenario 2</th>
<th>Scenario 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Histogram</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash Join</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Histogram</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash Join</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Histogram</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash Join</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Multiprogrammed Environment

Time

Seconds

<table>
<thead>
<tr>
<th>Scenario 1</th>
<th>Scenario 2</th>
<th>Scenario 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>Histogram</td>
<td>Stream</td>
</tr>
<tr>
<td>Hash Join</td>
<td>RE</td>
<td>Histogram</td>
</tr>
<tr>
<td>Stream</td>
<td>Hash Join</td>
<td>Stream</td>
</tr>
<tr>
<td>RE</td>
<td>Hash Join</td>
<td>Hash Join</td>
</tr>
<tr>
<td>Histogram</td>
<td>Stream</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RE</td>
<td></td>
</tr>
</tbody>
</table>

TBB
PD
PD variations

Dedicated

Multiprogrammed

- Scenario1
- Scenario2
- Scenario3
Other Issues to Ponder

• Determinate vs. deterministic vs. non-deterministic
• Notion of a precise exception/interrupt
• Future-proofing software
  – Unknown pathologies with locks
  – What else is lurking out there, and how might it be managed when it occurs?
• Given parallelism, how much of a limiter is a sequential representation? Why have parallel?
Summary

• Future general-purpose computing world is going to be parallel everywhere
  – But different from traditional parallel
• Will require new thinking about how to execute programs
• Want programmers to think parallel but program sequential
• Future will likely be sequential programs with dynamic dataflow parallel execution
Questions?
## Benchmarks

<table>
<thead>
<tr>
<th>Program</th>
<th>Source</th>
<th>Original language</th>
<th>Synchronization</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>barnes-hut</td>
<td>Lonestar</td>
<td>C++</td>
<td>barrier</td>
<td>N-body simulation</td>
</tr>
<tr>
<td>black-scholes</td>
<td>PARSE C</td>
<td>C</td>
<td>barrier</td>
<td>financial analysis</td>
</tr>
<tr>
<td>bzip2</td>
<td>pbzip2</td>
<td>C</td>
<td>mutex, condition variables</td>
<td>compression</td>
</tr>
<tr>
<td>dedup</td>
<td>PARSE C</td>
<td>C</td>
<td>mutex, condition variables</td>
<td>enterprise storage</td>
</tr>
<tr>
<td>histogram</td>
<td>Phoenix</td>
<td>C</td>
<td>barrier</td>
<td>image analysis</td>
</tr>
<tr>
<td>reverse index</td>
<td>Phoenix</td>
<td>C</td>
<td>mutex</td>
<td>web indexing</td>
</tr>
</tbody>
</table>
# Hardware configurations

<table>
<thead>
<tr>
<th>µ-arch</th>
<th>AMD Barcelona</th>
<th>Intel Nehalem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Phenom 9850</td>
<td>Opteron 8350</td>
</tr>
<tr>
<td>Sockets</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Cores</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Threads</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Total contexts</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Clock (GHz)</td>
<td>2.5</td>
<td>2.0</td>
</tr>
<tr>
<td>Memory (GB)</td>
<td>4</td>
<td>16</td>
</tr>
</tbody>
</table>