Performance issues in emulated shared memory computing

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

SaS Division Seminar
March 05, 2007
Linköping, Sweden
Contents

Part 1
Introduction
Background
Multi-core systems
Alternatives
Emulated shared memory
Realization attempts
Performance issues

Part 2
ILP-TLP co-exploitation
ILP-TLP co-exploitation
Superpipelining
Chained execution
Scheduling algorithm
Example
Evaluation
Speedup

Part 3
Language primitives
Language primitives
Solutions
CRCW model
Active memory
Dropping properties
Fast mode
Experimentation
Primitive execution times

Part 4
Active memory
Active memory
Multioperations
Evaluation
Part 1: Introduction — Background

Superscalar processors scale poorly with respect to number of functional units (FU):

- Large amounts of ILP is **hard to extract**
- The silicon **area** taken by certain resources (e.g. issue logic, forwarding network, ROB) **increases quadratically** as the number of FUs increases
- The practical **limit of power** consumption/heat dissipation has been **achieved** and slows down the clock frequency development both in high-performance and low-power embedded systems
- A single processor **can not exploit control parallelism** (which is available e.g. in a simple maximum find problem)
Processor manufacturer’s solution: Multi-core systems

Processor manufacturers have switched to multi-core or multiprocessor systems on chip (MP-SOC) engines utilizing the symmetrical multiprocessing (SMP) paradigm or even some fancier models like message passing (MP), shared memory (SM).

It is expected that the number of cores per chip will fast increase to a level in which only a fraction of the total computational power can be allocated for a single computational task using plain SMP.

=> Something more efficient than plain SMP is required, very likely including major architectural innovations!
SMP alternatives

SMP alternatives, e.g. MP, NUMA, and CC-NUMA, have severe limitations in ease of programming:

- locality-aware partitioning of data and mapping of functionality
- slow synchronization/asynchronous operation
- explicit communication

and/or performance for fine-grained general purpose parallel computing.

Will parallel computing be available to masses of users/programmers that are fixated to sequential-only paradigm? What about the next generation? Teaching parallelism should be started now!

Possible data movements in parallel merge sort. Too bad for cache coherent systems!
Emulated shared memory computing

An emulated shared memory machine provides ideal (e.g. synchronous) shared memory view to a programmer although the underlying parallel architecture is typically distributed.

Emulated shared memory computing is an promising attempt to solve the programmability problems, but so far realizing the full potential of them has turned out to be very difficult.

Main issues:
- Scalability (number of processors)
- Efficiency (performance, performance/area, power)
- Generality (irregular problems, granularity)
Attempts to realize emulated shared memory computing

<table>
<thead>
<tr>
<th>Machine</th>
<th>Year</th>
<th>Processor</th>
<th>Network</th>
<th>MT</th>
<th>Caches</th>
<th>Multiprefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ultracomputer</td>
<td>1979-</td>
<td>AMD 29050 15 MHz</td>
<td>omega</td>
<td>SW</td>
<td>non-coherent</td>
<td>combining</td>
</tr>
<tr>
<td>IBM RP3</td>
<td>1981-</td>
<td>ROMP 2.5 MHz</td>
<td>omega</td>
<td>no</td>
<td>coherent by SW</td>
<td>combining</td>
</tr>
<tr>
<td>BBN Butterfly</td>
<td>1985-</td>
<td>Motorola 68020</td>
<td>butterfly</td>
<td>no</td>
<td>no</td>
<td>serializing</td>
</tr>
<tr>
<td>HEP</td>
<td>1980-</td>
<td>custom 10 MHz</td>
<td>butterfly</td>
<td>HW,16</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Cray MTA</td>
<td>1990-</td>
<td>custom 300 MHz</td>
<td>3D mesh</td>
<td>HW,128</td>
<td>no</td>
<td>serializing</td>
</tr>
<tr>
<td>SB-PRAM</td>
<td>1990-</td>
<td>custom 8 MHz</td>
<td>butterfly</td>
<td>HW, 32</td>
<td>no</td>
<td>combining</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>On-going projects</th>
<th>Year</th>
<th>Processor</th>
<th>Network</th>
<th>MT</th>
<th>Caches</th>
<th>Multiprefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMT framework</td>
<td>1997-</td>
<td>XMT (unimpl.)</td>
<td>?</td>
<td>HW</td>
<td>yes?</td>
<td>?</td>
</tr>
<tr>
<td>Eclipse</td>
<td>2002-</td>
<td>MTAC (unimpl.)</td>
<td>2D Sparse mesh/multi mesh</td>
<td>HW</td>
<td>no</td>
<td>combining serializing hybrid</td>
</tr>
</tbody>
</table>
Performance issues

How to exploit instruction-level parallelism (ILP) and thread-level parallelism (TLP) at the same time?

How to implement control and synchronization primitives efficiently for high-level parallel languages?

How to employ active memory for processing of concurrent operations efficiently?

Partial solutions exist but they are not simple

Existence of integrated solution is still a bit blurred but the research community (including us) is working on it.
Part 2: ILP-TLP co-exploitation

- Current superscalar processors use out-of-order execution to eliminate data dependencies during execution, ruining the lock-step synchronicity of emulated shared memory machines.

- Current emulated shared memory machines do not exploit available ILP efficiently.

- In addition to intrathread dependencies present in singlethreaded execution, TLP execution introduces also interthread dependencies.

Try to compress execution of threads!
Possible solutions

Use parallel slackness of TLP execution (instructions belonging to different threads are independent within a step of execution) to reschedule execution of instructions so that ILP can be exploited easily and efficiently.

This can happen in two forms:
- **Hazard-free superpipelining** (i.e. overlap execution of instructions belonging to different threads not to the same thread)
- **Chained execution** (i.e. executing multiple instructions sequentially during a single step of execution rather than doing it in parallel)

We will describe an algorithm to apply this kind of chaining to a sequentially scheduled threads and provide a short evaluation of these forms of exploiting ILP in a TLP machine.
Hazard-free superpipelining

**Superpipelining** = deep pipelining in which also the actual execution stages are pipelined

By organizing execution so that threads are executed in a $L_S$-stage superpipelined manner one can achieve a speedup of $L_S$ because there will not be any pipelined hazards as long as the number of threads is higher than the latency of the pipelined instruction.

This kind of superpipelining can be realized by a special TLP hardware called **MTAC processor**.
Chained execution

**Execution** of subinstructions (at instruction level) is *organized* as a **sequential chain** rather than in parallel (as in ordinary ILP processors)

In a **singlethreaded** processor this kind of organization **would cause** a lot of pipeline **hazards** delaying execution, but if **interthread** pipelining is used **no hazards** occur since threads are independent within a step of execution

The **ordering** of functional units (FU) in the chain (in MTAC) is selected **according to** the average ordering of instructions in a **basic block**:  
- Two thirds of the **ALUs** in the beginning of the chain  
- The **memory units** and the rest of the **ALUs** in the middle  
- The **compare unit** and the **sequencer** in the end
ILP-TLP scheduling algorithm

An algorithm to reschedule subinstructions of VLIW instructions so that chained type of ILP is maximized within single basic blocks:

Going beyond the borders of basic blocks is possible, but interthread dependencies may then cannibalize speedups.

This algorithm works only with single memory unit processors, but it can be extended to multiple units. (Then we lose the ability to execute fully sequential code, because in MTAC style machines memory units must be organized in parallel due to synchronization efficiency reasons).

FOR each basic block $B$ in program $P$ DO
FOR each free subinstruction slot $S$ in $B$ DO
REPEAT
- SEARCH the next subinstruction $I$ that has the same class as $S$ from the current basic block
- IF subinstruction $I$ was found AND the predecessors of $I$ are scheduled to be executed before $S$ THEN release the slot occupying $I$ and assign $I$ to $S$
UNTIL the $S$ is filled OR the basic block boundary is reached
DELETE empty instructions in $B$
Example - block copy (single processor view)

**Compiled (DLX)**

L0: lw r4,0(r1)
addi r1,r1,#4
sw 0(r2),r4
slt r4,r1,r3
bnez r4,L0
addi r2,r2,#4

**Translated (ECLIPSE)**

Iteration 1
- LD
- ADD
- ST
- SLT
- BNEZ
- ADD

Iteration 2
- LD
- ADD
- ST
- SLT
- BNEZ
- ADD

Iteration 3
- LD
- ADD
- ST
- SLT
- BNEZ
- ADD

Iteration 4
- LD
- ADD
- ST
- SLT
- BNEZ
- ADD

**Virtual ILP Optimized (ECLIPSE)**

Iteration 1
- LD
- ADD
- ST
- BNEZ
- ADD

Iteration 2
- LD
- ADD
- ST
- BNEZ
- ADD

Iteration 3
- LD
- ADD
- ST
- BNEZ
- ADD

Speedup 267%

12 threads x iteration 1

Speedup 800%, 300%

Single issue multithreaded

Chained multithreaded
We measured the execution time of 9 simple benchmarks and a set of randomly chosen specint92 basic blocks before and after applying the proposed scheduling algorithm.

We used a typical 5-stage pipelined RISC processor with 4 FUs (DLX processor) as a baseline machine in our comparison.

In order to investigate the effect of the number of functional units we applied the algorithm for three MTAC configurations with 4, 6 and 10 FUs.
Evaluation—speedup provided by chaining

The average speedup with the simple benchmark suite was 230%, 386% and 496% for T5, T7 and T11 processors.

In the random specint test the obtained speedup was 248%, 401% and 453% for T5, T7 and T11 processors.

We also calculated the maximum achievable speedup with any basic block scheduling algorithm for the benchmark suite:

- The achieved speedups were 230%, 386% and 496%—the same as with our algorithm within the measurement accuracy 1%.

According to our performance-area-power model the silicon area overhead of using chaining is typically less than 2%.
Evaluation—speedup provided by a reference machine with parallel FUs

For comparison purposes we applied the same algorithm to an architecture that is similar than MTAC but FUs are organized in parallel:

- The average speedups were 157%, 175% and 175% for 4, 6 and 10 unit processors

These numbers show the (limited) amount of ILP present within the basic blocks of the (general purpose) code.
Evaluation—speedup achievable by superpipelining

In order to evaluate the effect of superpipelining we measured the relative speedups gained with superpipelining of 2 to 8 stages in respect to non-superpipelined machine.

It seems that the speedup is somewhat proportional to the degree of superpipelining, but sometimes increasing the number of stages it by one will not give any speedup due to quantization effects.

Due to already maximal power/area figures increasing the clock frequency may not be feasible!
Part 3: Implementation of high-level parallel language primitives

Emulated shared memory machines require efficient and easy-to-use parallel language. Essential features include e.g.

- Thread group management
- Splitting a group hierarchically to subgroups
- Support for shared and private variables
- Synchronization over control structures having unbalanced execution paths
- Support for strong operations e.g. CRCW, multioperations, multiprefixes

In the following, we consider an emulated shared memory machine realizing the EREW+ model (Eclipse MP-SOC architecture) and related high-level parallel language (e) implementation for it as a baseline.
Possible solution: Architectural support strong operations + fast barriers

Add architectural techniques for strong operations and provide support for them at language level

1. Support for the CRCW model
   - Step caches

2. Support for active memory operations
   - Step caches + scratchpads + active memory

3. Implementation of an experimental fast mode
   - Step caches + scratchpads + active memory
   - Drop some structural features from e
   - Fast barrier mechanism
Support for CRCW model

Rewrite e-language construct support code for the CRCW model:

- **Transform** complex limited active memory based **concurrent accesses** on a top of the EREW model to **real CRCW accesses**
- **Reduce** the number of **implicit support variables** by eliminating the internally used subgroup concept
- **Modify** the thread **group frames** used in subgroup creation accordingly
- **Rewrite** the **run time library** eRunLib to reflect better the CRCW functionality.

```plaintext
SPREAD SCALAR a_ TO ARRAY b_:
int a_; // A shared variable
int b_[size]; // A shared array of integers
int tmp_[size]; // Thread-wise copies of a_ EREW

// EREW version:
int i;
// Spread a_ to tmp_ with a logarithmic algorithm
if_ (_thread_id==0 , tmp_[0]=a_; )
for (i=1; i<_number_of_threads; i<<=1)
  if_ (_thread_id-i>=0 ,
       tmp_[_thread_id]=tmp_[_thread_id-i]; );
b_[_thread_id]+=tmp_[_thread_id]

// CRCW version:
b_[_thread_id]+=a_;
```
Support for active memory operations

Add four primitives to the e-language:

prefix(p,OP,m,c) Perform a two instruction arbitrary multiprefix operation OP for components c in memory location m. The results are returned in p.

class_prefix(p,OP,m,c) Perform a single instruction multiprefix operation OP for at most O(sqrt T) components c in memory location m. The results are returned in p.

multi(OP,m,c) Perform a two instruction arbitrary multioperation OP for components c in memory location m.

fast_multi(OP,m,c) Perform a single instruction multioperation OP for at most O(square root T) components c in memory location m.

COMPUTE A SUM OF ARRAY a_ to sum_:

```c
int sum_; // A shared variable
int a_[size]; // A shared array of integers

// EREW version—logarithmic algorithm for sum
for_ ( i=1 , i<_number_of_threads , i<<=1 ,
    if (_thread_id-i>=0)
        a_[_thread_id] += a_[_thread_id-i]; );
sum_=a_[_number_of_threads-1]
```

// Active memory version

```c
//—just call the constant time sum primitive:
multi(MADD,&sum_,a_[_thread_id]);
```
What about altering the language, e.g. dropping some properties?

The high-level language implementations for emulated shared memory machines feature high parallel primitive execution time overheads.

The switch from EREW to CRCW is only a partial solution.

To further cut overheads, we implemented an experimental fast mode for simple non-structural programs that provides higher performance but limited feature set:

<table>
<thead>
<tr>
<th>Primitive</th>
<th>gcc on DLX</th>
<th>ec on Eclipse</th>
<th>fcc on SB-PRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>barrier</td>
<td>-</td>
<td>32</td>
<td>18</td>
</tr>
<tr>
<td>private synch if-else</td>
<td>4-5</td>
<td>94</td>
<td>50</td>
</tr>
<tr>
<td>private synch while</td>
<td>3</td>
<td>125</td>
<td>33</td>
</tr>
<tr>
<td>private synch for</td>
<td>5</td>
<td>124</td>
<td>33</td>
</tr>
</tbody>
</table>
Implementing experimental fast mode for e compiler

Limitations:

- **Overlapped** execution of e **constructs** employing subgroup creation or barrier synchronization by multiple thread groups is **not allowed**
- **Shared** variables local to functions are **not supported**
- Subgroup specific thread **numbering won’t be passed automatically** across subroutine borders
- The number of **simultaneously** active **barriers** is **limited** by the underlying architecture

```c
SORT src_ IN PARALLEL IN O(1) TIME
int src_[N]; // Array to be sorted
int tgt_[N]; // Rank for each elem

Flexible mode code:
void rank() // Parallel rank funct.
{
    int i = _thread_id >> logn;
    int j = _thread_id & (N-1);
    fast_multi(MADD,&tgt_[j],src_[i]<src_[j]);
}

if_(_thread_id<N2 rank());
if_(_thread_id<N, src_[tgt_[_thread_id]]=src_[_thread_id]);

// Fast mode code:
int i = _thread_id >> logn;
int j = _thread_id & (N-1);
if_(_thread_id<N2,
    fast_multi(MADD,&tgt_[j],src_[i]<src_[j]));
if_(_thread_id<N,
    src_[target_[_thread_id]]=src_[_thread_id]);
```
## Experimentation—Configurations

<table>
<thead>
<tr>
<th>Configuration</th>
<th>P</th>
<th>Model</th>
<th>Fast mode</th>
<th>Active memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>E4</td>
<td>4</td>
<td>EREW</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>E16</td>
<td>16</td>
<td>EREW</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>E64</td>
<td>64</td>
<td>EREW</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>C4</td>
<td>4</td>
<td>CRCW</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>C16</td>
<td>16</td>
<td>CRCW</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>C64</td>
<td>64</td>
<td>CRCW</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>F4</td>
<td>4</td>
<td>CRCW</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>F16</td>
<td>16</td>
<td>CRCW</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>F64</td>
<td>64</td>
<td>CRCW</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>C4+</td>
<td>4</td>
<td>CRCW</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>C16+</td>
<td>16</td>
<td>CRCW</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>C64+</td>
<td>64</td>
<td>CRCW</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>F4+</td>
<td>4</td>
<td>CRCW</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>F16+</td>
<td>16</td>
<td>CRCW</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>F64+</td>
<td>64</td>
<td>CRCW</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Baseline EREW**

**CRCW**

**Fast mode CRCW**

**Multioperation CRCW**

**Fast mode + multiop + CRCW**
## Experimentation—Benchmark programs

<table>
<thead>
<tr>
<th>Name</th>
<th>N</th>
<th>EREW</th>
<th>E</th>
<th>P</th>
<th>W</th>
<th>MCRCW</th>
<th>E</th>
<th>P=W</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>barrier</td>
<td>T</td>
<td></td>
<td>log N</td>
<td>N</td>
<td>N log N</td>
<td>1</td>
<td>N</td>
<td>Commit a barrier synchronization for a set of N threads</td>
<td></td>
</tr>
<tr>
<td>fft*</td>
<td>64</td>
<td></td>
<td>log N</td>
<td>N</td>
<td>N log N</td>
<td>1</td>
<td>$N^2$</td>
<td>Perform a 64-point complex Fourier transform</td>
<td></td>
</tr>
<tr>
<td>max</td>
<td>T</td>
<td></td>
<td>log N</td>
<td>N</td>
<td>N log N</td>
<td>1</td>
<td>N</td>
<td>Find the maximum of a table of N words</td>
<td></td>
</tr>
<tr>
<td>mmul*</td>
<td>16</td>
<td></td>
<td>$N^2$</td>
<td>$N^3$</td>
<td>$N^3$</td>
<td>1</td>
<td>$N^3$</td>
<td>Compute the product of two 16-element matrixes</td>
<td></td>
</tr>
<tr>
<td>sort* °</td>
<td>64</td>
<td></td>
<td>log N</td>
<td>N</td>
<td>N log N</td>
<td>1</td>
<td>$N^2$</td>
<td>Sort a table of 64 integers</td>
<td></td>
</tr>
<tr>
<td>sum</td>
<td>T</td>
<td></td>
<td>log N</td>
<td>N</td>
<td>N log N</td>
<td>1</td>
<td>N</td>
<td>Compute the sum of an array of N integers</td>
<td></td>
</tr>
</tbody>
</table>

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).

* Brute force algorithm, not work optimal

° Randomized algorithm
Relative performance (top) and code size (bottom) of MP-SOC configurations with respect to corresponding E4, E16, and E64 EREW configurations.

### Average speedups related baseline EREW MP-SOC

- Baseline EREW: 1
- CRCW: 1.47
- Fast mode CRCW: 4.28
- Multioperation CRCW: 47.12
- Fast mode multioperation CRCW: 73.13
Performance of an F64+ MP-SOC utilizing fast mode e with respect to that of a single 5-stage pipelined DLX processor system utilizing sequential c.

### Average speedups related baseline scalar DLX

<table>
<thead>
<tr>
<th>Description</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline scalar DLX</td>
<td>1</td>
</tr>
<tr>
<td>Fast mode multiop CRCW, (P=64, F=5)</td>
<td>50.03</td>
</tr>
<tr>
<td>Fast mode multiop CRCW, (P=64, F=11)</td>
<td>87.16</td>
</tr>
</tbody>
</table>

* Brute force algorithm, not work optimal
## Primitive execution time

<table>
<thead>
<tr>
<th>Primitive</th>
<th>gcc on DLX</th>
<th>ec on Eclipse</th>
<th>fcc on SB-PRAM*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>EREW+</td>
<td>MCRCW</td>
</tr>
<tr>
<td>barrier**</td>
<td>-</td>
<td>32</td>
<td>10</td>
</tr>
<tr>
<td>private synch if-else</td>
<td>4-5***</td>
<td>94</td>
<td>34</td>
</tr>
<tr>
<td>private synch while</td>
<td>3</td>
<td>125</td>
<td>32</td>
</tr>
<tr>
<td>private synch for</td>
<td>5</td>
<td>124</td>
<td>32</td>
</tr>
</tbody>
</table>

* according to “Practical PRAM programming” book
** barrier delay can be eliminated if execution paths can be balanced for all architectures
*** 1-2 on Eclipse thread
Part 4: Efficient active memory technique to implement multioperations

Active memory = memory that can perform some computation

Active memory is used in emulated shared memory machines to implement multi(prefix)operations.

Example: Sum the data sent by all threads in a memory location

Most existing solutions do not provide multioperations and those that do, like SB-PRAM, consume at least two steps per operation and require a quite complex sorting network + combining network with buffers for original references. Can we do it faster?
Possible solution: Combining/serializing hybrid

Combine references step-wisely at processor level and process them sequentially at memory modules. Limitations:

- Provides associative multioperations and arbitrary ordered multiprefixes that can be used e.g. for renumbering threads and compaction
- Works currently for fast SRAM memories presuming that

\[ \text{Latency} + \#\text{Processors} \ll \#\text{Threads} \]

We will outline the implementation in the guest lecture of “Advanced Parallel Programming” course after this seminar!

<table>
<thead>
<tr>
<th>Access model</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>EREW</td>
<td>1 step</td>
</tr>
<tr>
<td>CRCW</td>
<td>1 step</td>
</tr>
<tr>
<td>Multi(prefix) operations</td>
<td>1-2 steps</td>
</tr>
</tbody>
</table>

Provides faster EREW and CRCW simulation times than Ranade’s or SB-PRAM’s simulation taking always at least 2 steps!
Evaluation results

According to our (over 100 parameter) performance-area-power model, the area overhead of the MCRCW realization is less than 1% for typical configurations.

Assuming

- roughly $\frac{2}{3}$ of memory references are reads
- 30% of memory reads in general purpose code are dependent on the other instruction so that extra delay cannot be hidden [Hennessy90]

we estimate that SB-PRAM algorithm runs about 20% slower than the MCRCW algorithm excluding full multiopeations and ordered multiprefix cases.
Conclusions

We have described a number of performance issues related to emulated shared memory computing

- ILP-TLP co-exploitation
- Efficient implementation high-level parallel language primitives
- Efficient use of active memory for processing of concurrent operations

We have presented some promising (non-trivial) techniques to address these issues

- Hazard free superpipelining and chaining
- Architectural support for strong models and fast barrier as well as an experimental fast mode
- Active memory via combining serializing hybrid technique

Nevertheless, a number of issues remain open (how to remove the limitations of above techniques).

-------- Academic and/or industrial interest/support/partnership/funding welcome --------