Practical Design and Evaluation of Continuation-based Multi-threading Processor

Makoto Amamiya
Kyushu University
Agenda

■ Motivation of Research
  ○ Practical Design of Thread-Level-Parallel Processor
  ○ More Detailed Evaluation

■ Continuation-based Multithreading Model
  ○ Why Continuation Programming?
    • Split-Phase Multithreaded Programming
    • Data-level Parallelism, Stream (Pipelined) Parallelism
    • Event-driven, Demand-driven Computations

■ Fuce Processor Design - Chip-multiprocessor -

■ Performance Evaluation
  ○ Evaluation Parameters: Number of TEUs, Memory Access Speed
  ○ Benchmark Programs: Quicksort, N-Queen

■ Hardware Implementation on FPGA

■ Conclusion

■ Further Work
Introduction

- Limits of ILP
  - Problems of TLP processor
    - Problem of multi-threaded execution (SMT)
    - Overhead of thread scheduling and switching
  - Developing an architecture for TLP
    - Practical Design of TLP Processor

- Continuation-based Multithreading Processor: Fuce (Fusion of Communication and Execution)
  - Continuation-based multithreaded programs
  - Thread scheduling and switching by hardware
  - Chip-multiprocessor
Why Continuation-based Multithreaded Programming?

- Split-phase programming
  - Event-driven computation
  - Demand-driven computation
  - Latency hiding (e.g. memory accesses and IOs)
  - TLP exploitation
    - Data Level Parallelism
    - Pipelined Parallelism

➢ The Fuce processor executes the continuation-based multithreaded programs effectively.
Continuation-based Computation

(a) Simple Continuation

(b) Multiple Continuations
Continuation-based Computation
- Concurrent Multi-threading -

program code

initial trigger

Thread1
th-code1
continue
continue

Thread2
th-code2
continue
continue

Thread3
th-code3
continue

Thread4
th-code4
continue

Thread5
th-code5
continue

Thread6
th-code6
continue

terminate
Continuation-based Computation  
- Interthread Data Passing -

Continuation is *just* notification. 
We (or compiler) need to store data manually before continuation.

Thread 1

```
...  
Pack r1, ins+th3  
; pack the instance  
; and next-thread  
Store r5, x  
Store r7, x+1  
Continue r1
```

Thread 2

```
...  
Pack r1, ins+th3  
Store r5, y  
Store r6, y+1  
Continue r1
```

Data area  
(Inter-thread Bridge Register)

```
<table>
<thead>
<tr>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>x+1</td>
</tr>
<tr>
<td>y</td>
</tr>
<tr>
<td>y+1</td>
</tr>
</tbody>
</table>
```

Instance i

```
| ... |
| ... |
```

```
| ... |
| ... |
```

Thread 3

```
| Load x, r1 |
| Load x+1, r2 |
| Load y, r3 |
| Load y+1, r3 |
| ... |
```

```
| Load x, r1 |
| Load x+1, r2 |
| Load y, r3 |
| Load y+1, r3 |
| ... |
```

```
| Load x, r1 |
| Load x+1, r2 |
| Load y, r3 |
| Load y+1, r3 |
| ... |
```

```
| Load x, r1 |
| Load x+1, r2 |
| Load y, r3 |
| Load y+1, r3 |
| ... |
```

```
| Load x, r1 |
| Load x+1, r2 |
| Load y, r3 |
| Load y+1, r3 |
| ... |
```

```
| Load x, r1 |
| Load x+1, r2 |
| Load y, r3 |
| Load y+1, r3 |
| ... |
```

```
| Load x, r1 |
| Load x+1, r2 |
| Load y, r3 |
| Load y+1, r3 |
| ... |
```

```
| Load x, r1 |
| Load x+1, r2 |
| Load y, r3 |
| Load y+1, r3 |
| ... |
```

```
| Load x, r1 |
| Load x+1, r2 |
| Load y, r3 |
| Load y+1, r3 |
| ... |
```

```
| Load x, r1 |
| Load x+1, r2 |
| Load y, r3 |
| Load y+1, r3 |
| ... |
```

```
| Load x, r1 |
| Load x+1, r2 |
| Load y, r3 |
| Load y+1, r3 |
| ... |
```

```
| Load x, r1 |
| Load x+1, r2 |
| Load y, r3 |
| Load y+1, r3 |
| ... |
```

```
| Load x, r1 |
| Load x+1, r2 |
| Load y, r3 |
| Load y+1, r3 |
| ... |
```

```
| Load x, r1 |
| Load x+1, r2 |
| Load y, r3 |
| Load y+1, r3 |
| ... |
```

```
| Load x, r1 |
| Load x+1, r2 |
| Load y, r3 |
| Load y+1, r3 |
| ... |
```

```
| Load x, r1 |
| Load x+1, r2 |
| Load y, r3 |
| Load y+1, r3 |
| ... |
```

```
| Load x, r1 |
| Load x+1, r2 |
| Load y, r3 |
| Load y+1, r3 |
| ... |
```

```
| Load x, r1 |
| Load x+1, r2 |
| Load y, r3 |
| Load y+1, r3 |
| ... |
```

```
| Load x, r1 |
| Load x+1, r2 |
| Load y, r3 |
| Load y+1, r3 |
| ... |
```

```
| Load x, r1 |
| Load x+1, r2 |
| Load y, r3 |
| Load y+1, r3 |
| ... |
```

```
| Load x, r1 |
| Load x+1, r2 |
| Load y, r3 |
| Load y+1, r3 |
| ... |
```

```
| Load x, r1 |
| Load x+1, r2 |
| Load y, r3 |
| Load y+1, r3 |
| ... |
```

```
| Load x, r1 |
| Load x+1, r2 |
| Load y, r3 |
| Load y+1, r3 |
| ... |
```

```
| Load x, r1 |
| Load x+1, r2 |
| Load y, r3 |
| Load y+1, r3 |
| ... |
```

```
| Load x, r1 |
| Load x+1, r2 |
| Load y, r3 |
| Load y+1, r3 |
| ... |
```

```
| Load x, r1 |
| Load x+1, r2 |
| Load y, r3 |
| Load y+1, r3 |
| ... |
```
Continuation-based Computation
- Function Invocation -

Special instructions for function call

**newins**: acquire a free ACM instance and assign the ACM instance number to destination register.

**newda**: allocate data area for a ACM instance (this is *macro* instruction) actually, this instruction is replaced by function call of dynamic memory allocator which could be slow depending on allocation algorithms
Continuation-based Computation
- Return form Invoked Function -

Special instruction for function return

**delins**: release an ACM instance from ACM

Function Call and Return are programmed by *split-phase manner*.
### Code Example

\[
\text{sum}(n) = \begin{cases} 
0 & \text{if } n == 0 \\
n + \text{sum}(n - 1) & \text{else}
\end{cases}
\]

<table>
<thead>
<tr>
<th>Sum: th1:</th>
<th>th2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>la r5, insData</td>
<td>la r5, insData</td>
</tr>
<tr>
<td>lw r6, 8(r5)</td>
<td>lw r6, 8(r5)</td>
</tr>
<tr>
<td>la r9, th2</td>
<td>lw r7, 16(r5)</td>
</tr>
<tr>
<td>lw r10, 4(r5)</td>
<td>lw r8, 12(r5)</td>
</tr>
<tr>
<td>sub r6, r6, 1</td>
<td>lw r9, 4(r5)</td>
</tr>
<tr>
<td>bne r6, r0, else</td>
<td>add r7, r6, r7</td>
</tr>
<tr>
<td>newins r7, Sum</td>
<td>sw r7, 0(r8)</td>
</tr>
<tr>
<td>newda r8, r7, Size</td>
<td>cont r9</td>
</tr>
<tr>
<td>and r7, r7, th1</td>
<td>thfin</td>
</tr>
<tr>
<td>add r11, r5, 16</td>
<td></td>
</tr>
<tr>
<td>sw r9, 4(r8)</td>
<td></td>
</tr>
<tr>
<td>sw r6, 8(r8)</td>
<td></td>
</tr>
<tr>
<td>sw r11, 12(r8)</td>
<td></td>
</tr>
<tr>
<td>cont r7</td>
<td></td>
</tr>
<tr>
<td>thfin</td>
<td></td>
</tr>
<tr>
<td>else:</td>
<td></td>
</tr>
<tr>
<td>sw r0, 12(r5)</td>
<td></td>
</tr>
<tr>
<td>cont r10</td>
<td></td>
</tr>
<tr>
<td>thfin</td>
<td></td>
</tr>
</tbody>
</table>

06.2.7  
Makoto Amamiya @ Kyushu University
We use **Tail Recursion** as a Loop tail recursive function call is efficiently executed in comparison with normal function call. Because we don’t need to allocate new instances and data area, we can always re-use the same instance and data area.

Nomal loops in a thread are of course supported. However, in this case, one thread could occupy a Thread Execution Unit long time....
Continuation-based Computation
- Iteration vs. Tail Recursion -

(a) iteration in a thread

(b) tail recursion

some instance

thread
L: add...
  bne L, cont r9

return thread

....

thread

instances of tail recursive function

thread
  beqz r5, L, rcont ...
  L: store r7...
  cont ... delins ...
  delda ...
Continuation-based Computation
- Critical Section -
Two Methods of Critical Section Control

1. **Lock-Unlock Operation**
   - **Merit:**
     - Continues from any number of threads
   - **Demerit:**
     - Busy-wait
     - Reactivation of Self-thread

2. **Demand-driven Programming**
   - **Ack and Nack Handshaking**
   - **Merit:**
     - No need of Lock Operation
   - **Demerit:**
     - Continues from predefined of threads,
       Overhead of Demand signaling (?)
Parallelism Control

Problem: Overflow of ACM

- Inhibit the Explosive Instance Generations

✓ Two modes of Instance Generation: Parallel Mode and Serial Mode

- \textit{act-ins} register: number of current active ACM Blocks
- Mode switching: controlled by thread program

\textbf{If} \textit{act-ins} < threshold \textbf{then} Parallel mode \textbf{else} Serial mode

• Program Code is quite simple (next slide)
Parallelism Control

Threaded Code

- $p$
- $p_1$
- $p_2$

**act-ins**: the number of current active instances
**$k$**: threshold of act-ins

**Hardware registers**

- act-ins

```
call $p_1$
If act-ins < $k$ then
  call $p_2$
else
  continue th2;
```

```
call p1;
If act-ins < $k$ then
  call p2
else
  continue th2;
```

```
call p2;
continue th3
```

```
call p;
```

```
call p;
If act-ins < $k$ then
  call $p_2$
else
  continue th2;
```

```
call p; continue th3
```

```
return
```

return
The Fuce Processor

- Unify inner-computation (inner-events) and communication (outer-events) as threads.
  - Every program (user program and OS) is decomposed into multiple threads (split-phase)
    - Thread level parallelism (TLP)
    - Execute multiple threads concurrently
    - No pre-emptive thread execution
  - Thread Activation Controller (TAC)
  - Multiple Thread Execution Units (Multiple TEUs)
Fuce Processor Overview

Diagram showing the architecture of the Fuce processor, including components like the Communication Controller, I/O Controller, Preload Unit, I-Cache, Execution Unit, Memory Access Controller, Thread Activation Controller, Thread Execution Controller (TEX), Main Memory, and various other subsystems and controllers.
Fuce Processor - Overview

Thread Execution Unit (TEU)

Load/Store Unit

D-Cache

Main memory

Thread Activation Controller (TAC)

Activation Control Memory (ACM)
Thread Execution Unit (TEU)

- **Thread Execution Unit**
  - simple pipeline architecture
    - register-to-register operation
    - pre-loaded thread context

- **Preload Unit**
  - Loading operation only
    - Set up thread context before execution.
    - pre-loading instructions are put in the forepart of thread body
Fuce Processor - TEUs -

Ready-Thread Queue

TEU Dispatch Unit
(Allocate Thread Execution Unit;
Trigger Thread Execution Unit.)

Preload Unit

Thread Execution Unist

I-Caches

0 1 2 3 4

m-1 m

n-1 n

Thread Execution Unit

Register Files

Load/Store Control Unit

MMU

D-Caches

Register file
TAC (Thread Activation Controller)

- Controls inter-thread synchronization
  - continuations and mutual exclusions.
    - ACM (Activation Control Memory)
    - TBR (Inter-thread Bridge Register)
    - RTQ (Ready-Thread Queue)
    - TDU (Thread Dispatch Unit)
Fuce Processor - ACM -

Activation Control Memory (ACM)

Instance number

Base address

Sync-count  Fan-in  Code-entry

Lock-bit

Thread entry

decrement sync-count;
if sync-count=0 then
enqueue to Ready-Thread Queue;

Ready-Thread-Queue

To TEX

From TEU

Instance  Thread-entry
TAC and ACM

<table>
<thead>
<tr>
<th>base-address</th>
<th>lock-bit</th>
<th>sync-count</th>
<th>fan-in</th>
<th>code-entry</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- base-address: pointer to a thread data
- lock-bit: tag bit for mutual exclusion
- sync-count: waiting continuation
- fan-in: fan-in value of thread
- code-entry: pointer to thread code

Instance | Thread-entry
---|---
Select ACM entry
Select Thread entry of Instance

TAC
- Activation Control Memory (ACM)
- Thread Dispatch Unit
- Ready-Thread-Queue
Inter-thread Data Passing via Inter-thread Bridge Register

register-file number = instance name
(= ACM block number)
displacement = allocated by compiler

instance $i$

Thread-a
write

Thread-b
read

Register file of instance $i$

ACM Block Entry (for Instance)

one-to-one mapping

Base address
Sync-count
Fan-in

32 registers/file * 1024 instances

06.2.7
Makoto Amamiya @ Kyushu University
Why Continuation-based Multithread Programming? (revisit)

- Split-phase programming
  - Event-driven computation
  - Demand-driven computation
  - Latency hiding (e.g. memory accesses and IOs)
  - TLP exploitation
    - Data Level Parallelism
    - Pipeline Parallelism

➢ Fuce processor executes the continuation-based multithreaded programs effectively.

06.2.7

Makoto Amamiya @ Kyushu University
Data parallel execution

Execute multiple threads in data-parallel

- Exploit TLP for data-parallel programs (Inter-/Intra-Instance level)
Thread pipelining (Stream Processing)

- execute multiple threads in pipelined fashion
- exploit TLP for sequential programs (Inter-/Intra-Instance level)

![Diagram of Thread Pipelining]

```
loop
Thread A
```

```
loop
Thread B
```

```
loop
Thread C
```

```
loop
Thread D
```

---

continue-with data
(data stream)
Two Methods of Critical Section Control (Semaphore)

1. Lock-Unlock Operation
   - **Merit:** Continues from any number of threads
   - **Demerit:** Busy-wait
     Reactivation of Self-thread

2. Demand-driven Programming
   - Ack and Nack Handshaking
   - **Merit:** No need of Lock Operation
   - **Demerit:** Continues from predefined of threads, Overhead of Demand signaling (?)
Demand-driven Computation (Stream Processing)

- execute multiple threads in pipelined fashion (overhead of sending demand signal will be hidden)
- exploit TLP for sequential programs (Inter-/Intra-Instance level)
Performance Evaluation

- The Fuce simulator implemented with VHDL
  - Maximum of 8 TEUs
  - No D-caches
  - Various memory access speeds

- Benchmark programs:
  - well-known method vs. stream processing (thread pipelining)
    - Prime numbers (Eratosthenes’s Sieve)
      - Stream processing; Thread pipelining (Pipeline Parallelism)
    - FFT: Data parallelism, Stream processing; Pipeline Parallelism
    - Quick sort, Merge sort
      - Stream processing; Thread pipelining (Pipeline Parallelism)
      - Instance-level Parallelism (Well-known method)
    - N-Queen
      - Instance-level Parallelism (Well-known method)

- Simulation parameters
  - Number of TEUs
  - Memory access latency
  - Data size
Prime numbers (Eratosthenes’s sieve)

- Inter-thread data transfer with memory buffer
  - problem is memory access conflict and memory access latency

- Inter-thread direct data transfer without buffer
Prime numbers (Eratosthenes’s sieve) clock cycles vs. memory access latency

Without memory buffer

With memory buffer
Prime numbers (Eratosthenes’s sieve)

Comparison between with-memory-buffer and without-memory-buffer

- **memory access latency = 20**
  - (Normalized with with-buffer and preload-on)

- **memory access latency = 40**
  - (Normalized with with-buffer and preload-on)

- **memory access latency = 90**
  - (Normalized with with-buffer and preload-on)

Graphs showing the comparison with different memory access latencies and buffer preload configurations.
Prime numbers (Eratosthenes’s sieve) 
Speed-up ratio

without memory buffer (normalized with preload-on and latency = 90)
Prime numbers (Eratosthenes’s sieve)

Speed-up ratio
with memory buffer (normalized with preload-on and latency = 90)
FFT Program

Well-known

Thread Pipelining
FFT
clock cycles vs. number of TEUs

Data size: 2048 elements
Memory access latency: 20 clocks

Memory access latency: 60 clocks
Quick Sort Program

Well-known method

Thread pipelining (stream)
Quick Sort

clock cycles vs. data size  (8 TEUs)

![Graph showing clock cycles vs. data size for Quick Sort with different latency settings.](image)
Quick Sort
Speed-up ratio (data size = 7000)
Quick Sort
Speed-up ratio (data size = 7000)

Speed up ratio (based on each 1 PE)

number of TEUs

- Pipelining Qsort latency = 20
- Pipelining Qsort latency = 60
- Pipelining Qsort latency = 100
- Well Known Qsort latency = 20
- Well Known Qsort latency = 60
- Well Known Qsort latency = 100
Quick Sort
IPC vs. data size  (8 TEUs)
Merge Sort Program

Well-known

Thread Pipelining
Merge Sort
clock cycles vs. number of TEUs

(data size = 2048)
Merge Sort
Speed-up ratio (data size = 2048)
Event-driven vs. Demand-driven Merge Sort (data size = 2048)

Memory access latency: 20 clocks

Memory access latency: 60 clocks
N-Queen

Data Level Parallelism

Try(1,1)  Try(2,1)  X
1

Try(1,2)  Try(2,2)  X
2

Try(1,3)  Try(2,3)
3

Try(1,N)  Try(2,N)  .  .  .
N

Try(1,1)  Try(3,1)  X
1

Try(1,2)  Try(3,2)  X
2

Try(1,3)  Try(3,3)  X
3

Try(1,N)  Try(3,N)  .  .  .
N

Try(1,1)  Try(4,1)  X
1

Try(1,2)  Try(4,2)
2

Try(1,3)  Try(4,3)  X
3

Try(1,N)  Try(4,N)  .  .  .
N

Try(1,1)  Try(5,1)  X
1

Try(1,2)  Try(5,2)
2

Try(1,3)  Try(5,3)  X
3

Try(1,N)  Try(5,N)  .  .  .
N

Try(1,1)  Try(6,1)  X
1

Try(1,2)  Try(6,2)
2

Try(1,3)  Try(6,3)  X
3

Try(1,N)  Try(6,N)  .  .  .
N

Try(1,1)  Try(7,1)  X
1

Try(1,2)  Try(7,2)
2

Try(1,3)  Try(7,3)  X
3

Try(1,N)  Try(7,N)  .  .  .
N
N-Queen
clock cycles vs. Number of Queens (8 TEUs)
N-Queen (N=8)
speed-up vs. number of TEUs

![Graph showing speed-up vs. number of TEUs for different memory access latencies.](image-url)
N-Queen

clock cycles vs. memory access latency

![Graph showing the relationship between clock cycles and memory access latency for different N-Queens.](image-url)
N-Queen (N=8)  
IPC vs. memory access latency

<table>
<thead>
<tr>
<th>Preload on</th>
<th>Preload off</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 TEUs</td>
<td>6 TEUs</td>
</tr>
<tr>
<td>6 TEUs</td>
<td>4 TEUs</td>
</tr>
<tr>
<td>4 TEUs</td>
<td>2 TEUs</td>
</tr>
<tr>
<td>2 TEUs</td>
<td>1 TEU</td>
</tr>
</tbody>
</table>

Memory access latency vs. IPC
Task-Mix
speed-up vs. number of TEUs

\[
\text{speed-up} = \frac{\text{Time(seive) + Time(Q-sort-pipeline) + Time(Q-sort-wellknown)}}{\text{Time(seive + Q-sort-pipeline + Q-sort-wellknown)}}
\]
Performance Evaluation  Part2

- Performance vs. ACM Access Latency
- Effect of Data-Cache
- Effect of ITBR (Inter-thread Bridge Register)
- Effect of Parallelism Control

- Hardware Cost
Sensitivity of ACM Speed to Performance

ACM Access Latency lightly affects Performance.
Effect of D-Cache to Performance (1)

Data-Cache is not so much effective.

![Graph showing the effect of D-Cache on performance]
Effect of D-Cache to Performance(2)

Data-Cache is not so much effective.

![Graph showing the effect of D-Cache on performance](image-url)
Effect of D-Cache to Performance(3)

Data-Cache is not so much effective.
Effect of Inter-thread Bridge Register

Inter-thread Bridge Register is effective.
Effect of Parallelism Control
(Quick-sort well-known algorithm)

Parallelism control is quite effective!
Fuce Hardware Implementation on FPGA Board

- Hardware implementation of Fuce Processor on FPGA Board
  - Fuce Machine Emulator
    - Higher Speed Execution of Benchmark for
      - More Practical Complicated Application
      - OS Construction by Continuation-based Multi-thread Programming
  - Evaluation of Hardware Cost
Fuce Implementation on FPGA
- FPGA Experimental System -

(Accverinos KM-1)
Fuce Implementation on FPGA
- FPGA Function Module -

- Xilinx XC2V6000
  - Virtex-II
- SO-DIMM Slot x2
  - 256MB SDRAM
Fuce Implementation on FPGA
- Allocation of Fuce Modules to FPGA -

Host PC (Linux)

PCI-BUS

RS-232C

FPGA Board
FPGA chip

PCI controller

UART

SDRAM Controller

FUCE Processor

SDRAM Memory (256MB)

FPGA Board
FPGA chip

FUCE Processor 3 MHz

SDRAM Controller 133 MHz

SDRAM (PC133) (256MB) 133 MHz
Fuce Implementation on FPGA
- Construction on FPGA -
Fuce Implementation on FPGA
- Specification of Fuce Processor -

- Number of Thread Execution Units: 8
- Main Memory: 256M Byte
- ACM Size: 4K Byte
- Processor Clock Cycle: 3M Hz
- Memory Access Cycle: 133M Hz
- PCI Bus Cycle: 33M Hz
Hardware Cost

- By VHDL Description
  24Trs. per 1 FPGA gate

  FUCE Processor
  with TAC, 8 TEUs, LS unit
  (without Multiplier, Divider, Floating-point Arithmetic)
  - Logic: 3.5 MTrs.
    (Includes 350 KTrs. for TAC)
  - Memory: 80 MTrs. (1MB SRAM)

- cf.
  Pentium 4: 100 MTrs. (0.09um process, 1MB L2 Cache)
  Power 4: 174 MTrs. (0.18um process, 1.41MB L2 Cache)
Conclusion

- Development of the Fuce processor
  - Continuation-based TLP processor
  - Execute threads by the continuation model
  - Thread scheduling and switching by hardware

- Continuation-based programming
  - Data level parallelism
  - Pipelining parallelism
  - Event-driven, Demand-driven

- Performance Evaluation of the Fuce processor
  - Exploit thread parallelism from poor ILP programs
    - Benchmark: Sieve, FFT, Quick sort, Merge sort, N-queen
Further work

- Fuce emulator on FPGA
  - More practical Fuce processor implementation
  - More precise evaluation data with shorter elapsed time
  - More precise hardware-cost estimation
  - Workbench of custom-chip development

- Evaluation using more sophisticated Benchmarks
  - Task-mixing
  - Spec-Int programs

- Fuce-C language and its compiler
  - Language for continuation-based multithreaded programming

- OS development by continuation-based multi-threaded program
  - OS construction with non pre-emptive thread programs
  - Efficient concurrent process control