Research Trends in Hardware/Software Codesign of Embedded Operating Systems for FPGAs

Vincent J. Mooney III
http://codesign.ece.gatech.edu
http://www.crest.gatech.edu

Associate Professor, School of Electrical and Computer Engineering
Adjunct Associate Professor, College of Computing
Co-Director, Center for Research on Embedded Systems and Technology
Georgia Institute of Technology
Atlanta, Georgia, USA
Outline

• Trends

• Custom RTOS Hardware IP
  • System-on-a-Chip Dynamic Memory Management Unit (SoCDMMU)
  • The δ Hardware/Software RTOS Generation Framework

• Compilation for Reverse Execution & Debugging

• Conclusion
Moore’s Prediction (“Law”)

- **Si atom ~ 2Å**
  - .09u = 90nm = 900Å  July 2004
    - \( \Rightarrow \sim 450 \text{ atoms, } 90^2 = 8100 \)
  - .065u = 65nm = 650Å  Jan 2006
    - \( \Rightarrow \sim 325 \text{ atoms, } 65^2 = 4225 \)
  - .045u = 45nm = 450Å  July 2007
    - \( \Rightarrow \sim 225 \text{ atoms, } 45^2 = 2025 \)
  - .032u = 32nm = 320Å  Jan 2009
    - \( \Rightarrow \sim 160 \text{ atoms, } 32^2 = 1024 \)
  - .022u = 22nm = 220Å  July 2010
    - \( \Rightarrow \sim 110 \text{ atoms, } 22^2 = 488 \)
  - .016u = 16nm = 160Å  Jan 2012
    - \( \Rightarrow \sim 80 \text{ atoms, } 16^2 = 256 \)
  - .011u = 11nm = 110Å  July 2013
    - \( \Rightarrow \sim 55 \text{ atoms, } 11^2 = 121 \)
  - .008u = 8nm = 80Å  Jan 2015
    - \( \Rightarrow \sim 40 \text{ atoms, } 8^2 = 64 \)
  - .006u = 6nm = 60Å  July 2016
    - \( \Rightarrow \sim 30 \text{ atoms, } 6^2 = 36 \)
  - .004u = 4nm = 40Å  Jan 2018
    - \( \Rightarrow \sim 20 \text{ atoms, } 4^2 = 16 \)
  - .003u = 3nm = 30Å  July 2019
    - \( \Rightarrow \sim 15 \text{ atoms, } 3^2 = 9 \)
  - .002u = 2nm = 20Å  Jan 2020
    - \( \Rightarrow \sim 10 \text{ atoms, } 2^2 = 4 \)
Power Consumption

- Power consumption of VLSI is a fundamental problem of mobile devices as well as high-performance computers
  - Limited operation (battery life)
  - Heat
  - Operation cost
- Power = dynamic + static
  - Dynamic power more than 90% of total power (0.18u tech. and above)
  - Static nearly equal to dynamic power for latest processes (.65u)
- Dynamic power reduction:
  - Technology scaling
  - Frequency scaling
  - Voltage scaling

Cost of latest Silicon Processes

- Mask sets for 90nm: over $1 million (U.S. $)
- NRE costs $10 to $100 million U.S.
  - Verification costs can exceed design costs
  - Embedded software costs may exceed hardware costs
Increasing Cost of Customization*

Example: Design with 80 M transistors in 100 nm technology

Estimated Cost - $85 M - $90 M

- Cost and Risk rising to unacceptable levels

- Top cost drivers
  - Verification (40%)
  - Architecture Design (26%)
  - Embedded Software Design
    - 1400 man months (SW)
    - 1150 man months (HW)
  - HW/SW integration

12 – 18 months

Yearly Chip Design Starts

- Roughly 8,000 in 1996
- Peaked at approx. 12,000 in 2000
- Recent years approx. 3,000 per year
- However, comparing, say, 2005 to 2000, number of EDA tool licenses purchased roughly the same
  - Larger team per chip design start
- System level chip design 2005
  - 33.5% U.S.
  - 26.1% Japan
  - 9.9% Taiwan, 5.8% Germany, 5.4% China, 5.3% Korea
Digital Silicon CMOS VLSI Trends

Yesterday (1980s):
- Memory
- Processors
- Gate arrays
- ASICs

Today:
- Memory
- Processors
- Reconfigurable SoC
- Gate arrays
- ASICs

Tomorrow:
- Memory
- Processors
- Platform SoC
- Custom SoC
- Reconfigurable
- Gate arrays / struct. ASIC
- ASICs

≥ 10 products
≤ 9 products
Embedded Systems Market Drivers

- 1950s-1980s: Business Applications
- 1990s-Today: Consumer Electronics

Some observations
- profit margins tight in consumer
- volumes and risk high in consumer
- supply chain as or more important than raw technology
Chip Software Business Models

- Electronics Design Automation
  - Synopsys, Cadence, Mentor, Synplicity
  - Yearly/quarterly subscriptions

- Embedded Operating System Companies
  - Windriver, Montavista, Timesys, Integrity, etc.
  - Upgrades, debuggers, Integrated Development Environment (IDE)
  - Per-seat costs 1/10 of EDA
    - $10K per seat (as compared to $100K for EDA)

- Cooley’s observation
  - FPGA software sells for far less than ASIC software
  - Many FPGA companies give away software (lock-in...)
Failed Reconfigurable Attempts

- ST
- Phillips
- Texas Instruments
- Pilkington
- Atmel
- Samsung
- Motorola
- IBM
- Vantis
- Lattice
- Quicksilver
- Concurrent Logic
- Toshiba
- Chameleon
- Plessey
- Adaptive Silicon
- Dynachip
- Crosspoint
- Lucent
- National Semiconductor
A Few Successes

- Triscend
- Synplicity

Note1:
- Xilinx, Altera: each probably have, as a lower bound, approx. 12 million lines of code for FPGA programming tools

Note2:
- Key reconfigurable patents from 1970s have expired
Comments (probably incorrect) on Trends!

- Reconfigurable like to increase superlinearly (increasing % of market)
  - Tools critical to adoption

- Chips for consumer electronics to drive technology
  - E.g., Cell processor

- Embedded software and operating systems more important and less supported than ever

- Breakthrough platform SoC design flows to determine chip success
  - E.g., will not be Cell because too difficult to program…
Outline

• Trends

• Custom Operating System Hardware IP
  • System-on-a-Chip Dynamic Memory Management Unit (SoCDMMU)
  • The δ Hardware/Software RTOS Generation Framework

• Compilation for Reverse Execution & Debugging

• Conclusion
System-on-a-Chip (SoC)

- This architecture is suitable for embedded multimedia applications, which require great processing power and large volume data management.
SoC

The existence of *global* on-chip memory, arises the need for an efficient way to dynamically allocate it among the PEs.
Problem

- How to deal with the allocation of the large global on-chip memory between the PEs in a dynamic yet deterministic way?
Solution 1

- Custom Memory Configuration (Static)
  - Hardware/Software co-synthesis with memory hierarchies [Wayne Wolf]
  - Matisse [IMEC]
  - Memory synthesis for telecom applications [WUYTACK et Al.], [YKMAN et al.]
Custom Memory Configuration

- **Pros:**
  - Easy
  - Deterministic
- **Cons:**
  - Inefficient memory utilization
  - System modification after implementation is very difficult if not impossible
Solution 2

- Shared memory multiprocessor (Dynamic)
  - Using conventional software memory Allocation/Deallocation techniques (e.g., Sequential Fits, Buddy Systems, etc.)
  - Sharing one heap (using locks)
  - Multiple heaps (one per processor)
Shared memory multiprocessor

- **Pros**
  - Flexible
  - Efficient memory utilization

- **Cons**
  - Worst case execution time is very high and usually not deterministic
Our Solution

- We introduce a new memory management hierarchy, Two-Level Memory Management, for a multiprocessor SoC.

- Two-Level Memory Management combines the best of dynamic memory management techniques (flexibility and efficiency) with the best of static memory allocation techniques (determinism).
Our Solution (2)

- In Two-Level Memory Management, large on-chip memory is managed between the on-chip processors (Level Two)
- Memory assigned to any processor is managed by the operating system running on that particular processor (Level One)
- To manage Level Two, we present the System-on-a-Chip Dynamic Memory Management Unit (SoCDMMU)
Dynamic Memory Management

- **Automatic**
  - Automatically recycles memory that a program will not use again
  - Either as a part of the language or as an extension

- **Manual**
  - The programmer has direct control over when memory is allocated and when memory may be de-allocated (e.g., by using `malloc()` & `free()`)
Memory Allocation
Software Techniques

- Sequential Fits

- First Fit,
- Next Fit,
- Best Fit or
- Worst Fit
Memory Allocation
Software Techniques

- Segregated Free Lists

  1-2

  3

  4

  5-8

  9-16

- Simple Segregated Storage
- Segregated Fit
Memory Allocation

Software Techniques

- Buddy System

- Bitmapped Fits
Memory Allocation

Hardware Techniques

- **Knowlton**
  Binary buddy allocator that can allocate memory blocks whose sizes are a power of 2

- **Puttkamer**
  Hardware buddy allocator (using Shift Register)

- **Chang and Gehringer**
  Modified hardware-based binary buddy system that suffers from the *blind spot* problem

- **Cam et al.**
  Hardware buddy allocator that eliminates the *blind spot* problem in Chang’s allocator

* References are available in the thesis by M. Shalan
Memory Allocation

Hardware Techniques

- Request size is 3
- It searches for 4
  
  [3 rounded to the nearest power of 2]
Memory Model Assumptions

- The global memory is divided into a fixed number of equally sized blocks (e.g., 16KB)
- The global memory allocation done by the SoCDMMU will be referred to as $G_{\text{allocation}}$
- The global memory de-allocation done by the SoCDMMU will be referred to as $G_{\text{deallocation}}$
- The PE can $G_{\text{allocate}}$ one or more than one block.
- Different PEs can issue the $G_{\text{allocation}}/G_{\text{deallocation}}$ commands simultaneously
Memory Model Assumptions (continued)

- Each memory block has one physical address and one or more virtual addresses. The block virtual address may differ from PE to another.
- The block virtual address will be referred to as PE-address.
Two-Level Memory Management

- The SoCDMMU manages the memory between the PEs
- The OS (or custom software) on each PE manages the memory between the processes that run on that PE
- The process requests the memory allocation from the OS or custom software. If there is not enough memory, the OS requests memory allocation from the SoCDMMU
Types of Memory Allocation

- **Exclusive**
  - Only the owner can access it. No other PE can access it.

- **Read/Write**
  - The owner can read/write to it. Other PE’s can read from it if it \(_G\text{ allocated}\) it as read only.

- **Read Only**
  - The PE \(_G\text{ allocates}\) the memory for read only. Other PE \(_G\text{ allocated}\) it as Read/Write.
PE-SoCDMMU Interface

Global On-Chip L2 Memory

Memory Module_1

Memory Module_2

Memory Module_N

PE_1

Cache

PE_2

Cache

......

......

PE_M

Cache

SoCDMMU

Configurable Xbar

control
### SoCDMMU Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Page ID (SW ID)</th>
<th>Size</th>
<th>Virtual Block No.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>G_alloc_ex</code></td>
<td>Page ID (SW ID)</td>
<td>Size</td>
<td>Virtual Block No.</td>
</tr>
<tr>
<td><code>G_alloc_rw</code></td>
<td>Page ID (SW ID)</td>
<td>Size</td>
<td>Virtual Block No.</td>
</tr>
<tr>
<td><code>G_alloc_ro</code></td>
<td>Page ID (SW ID)</td>
<td>n/a</td>
<td>Virtual Block No.</td>
</tr>
<tr>
<td><code>G_dealloc</code></td>
<td>Page ID (SW ID)</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td><code>G_move</code></td>
<td>Page ID (SW ID)</td>
<td>n/a</td>
<td>Virtual Block No.</td>
</tr>
</tbody>
</table>
The SoCDMMU Hardware

Address Converter

Virtual Block no.  Offset

Binary Encoder

0  1  2  3  4  5
G000A  G0000  G0001  G0002  G0003  G0004

Physical G_block no.  Offset

PEi Memory Bus

PEi Memory Bus

PEj Memory Bus

To Global Memory Buses

BASIC SoCDMMU

MUX

Control Unit

Request Scheduler

Command/Statusi
The SoCDMMU Hardware

The Basic SoCDMMU

Basic SoCDMMU
The SoCDMMU Hardware

The Basic SoCDMMU

Allocation Vector

Allocation Table

Allocation Table

MODE | PE | SW ID
---|---|---
Ex | PE 1 | 25
Ex | PE 2 | 11
The SoCDMMU Hardware

The Basic SoCDMMU

Allocation Table

Allocation Vector

Basic SoCDMMU

FPGAworld 13 September 2007

©Vincent J. Mooney III, 2007
The SoCDMMU Hardware

The Basic SoCDMMU

[Diagram of Basic SoCDMMU hardware components: Allocation Unit, Deallocation Unit, Control Unit, Status Register, Command Register, Allocation Table, and MUX.]

The Basic SoCDMMU
The SoCDMMU Hardware

The Basic SoCDMMU

Basic SoCDMMU
The SoCDMMU Hardware

The Allocation Unit

```
1 allocate(size,in[0:n-1]) {
2     for (i:=0 to n-1) {
3         if (in[i]==0 and size>0) {
4             out[i]:=1;
5             size:=size-1;
6         } else out[i]:=0;
7     }
8     if (size>0) return NOT_ENOUGH_MEMORY;
9     else return out;
10 }
```
The SoCDMMU Hardware

The Allocation Unit

```
 0                          0             0             0
1                             1               1               1
0                          0             1             1
21
1               0              0
0
```

```
0 in[n-1]
1
y x
FS b
b_out
out[n-1]
0

0 in[2]
1
0
1
y x
FS b
b_out
out[2]
0
0

0 in[1]
1
0
y x
FS b
b_out
out[1]
0
1

0 in[0]
1
0
y x
FS b
b_out
out[0]
0
1

Size[log_2(n)-1,0]
```
The SoCDMMU Hardware

The Allocation Unit

in[n-1:n-m-2] → 0's Counter → MUX → 1's Selector → MUX → out[n-1:n-m-2]
## The SoCDMMU Hardware

### The Allocation Unit

<table>
<thead>
<tr>
<th></th>
<th>Area (NAND gates)</th>
<th>Worst Delay (ns)</th>
<th>Max. Clock Speed (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimized Allocator</td>
<td>5364</td>
<td>6.6 ns</td>
<td>150 MHz</td>
</tr>
<tr>
<td>Un-optimized Allocator</td>
<td>17930</td>
<td>56.3 ns</td>
<td>17.5 MHz</td>
</tr>
<tr>
<td>Comparison</td>
<td>3.3X</td>
<td>8.5X</td>
<td></td>
</tr>
</tbody>
</table>

- 256 G blocks.
- Synthesized using Synopsys Design Compiler™ and a TSMC 0.25u library from LEDA Systems.
The SoCDMMU Hardware
Execution Times/Synthesis

- Synthesized using the TSMC 0.25u.
- Clock Speed: 300MHz
- Size:
  - ~7500 gates (not including the Allocation Table and Address Converter)
  - Allocation Table: The size of 0.66KB 6T-SRAM
  - Address Converter: The size of 1.22 KB 6T-SRAM

<table>
<thead>
<tr>
<th>Command</th>
<th>Number of Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>G_alloc_ex</td>
<td>4</td>
</tr>
<tr>
<td>G_alloc_rw</td>
<td>4</td>
</tr>
<tr>
<td>G_alloc_ro</td>
<td>3</td>
</tr>
<tr>
<td>G_dealloc</td>
<td>4</td>
</tr>
<tr>
<td>4-Processors WCET</td>
<td>16</td>
</tr>
</tbody>
</table>
Microcontroller Implementation

Microcontroller Roles:

- Stores the allocation Status
- Executes the allocation commands
- Executes the de-allocation commands

- Custom HW: 16 Cycles WCET
-uC: 231 Cycles BCET
Automatic Generation: Motivation

The Design Productivity Gap

CAGR 58%
CAGR 21%

Source: IRIS
To overcome the productivity gap, Intellectual Property (IP) cores should be used in SoC designs.

Also, tools should be used to automatically customize/configure the IPs:
- Processor Generators: Tensilica, ARC Core, etc.
- Memory Compilers: Artisan, Virage, LEDA, etc.

The SoCDMMU as an IP core should be customized before being used in a system different than the one for which it was designed.
DX-Gt Overview
RTOS Support

Introduction

- Conventional memory allocation algorithms (e.g., Buddy-heap) are not suitable for Real-Time systems because they are not deterministic and/or the WCET is high.

- This is mainly because of memory fragmentation and compaction. Also, most allocation algorithms usually use linked lists that have constant search time.

- An RTOS uses a different approach to make the allocation deterministic.
RTOS Support

Introduction

- An RTOS (e.g., uCOS-II, eCOS, VRTXsa, etc.,) usually divides the memory into pools each of which is divided into fixed-sized allocation units and any task can allocate only one unit at a time.
Atalanta Memory Management

Overview

- Atalanta is an open source RTOS developed at GaTech
- Atalanta allows tasks to obtain fixed-sized memory blocks from partitions made of a contiguous memory area
- Allocation and de-allocation of these memory blocks are done in a constant time
- No partition can be created at the run-time
Atalanta Memory Management

API Functions

- *asc_partition_gain*
  - Get free memory block from a partition (non-blocking)

- *asc_partition_seek*
  - Get free memory block from a partition (blocking)

- *asc_partition_free*
  - Free a memory block

- *asc_partition_reference*
  - Get partition information
Atalanta Support for the SocDMMU

Objectives

- Add Dynamic Memory Management to Atalanta
- Use the same Memory Management API Functions
- Keep the Memory Management Deterministic
Atalanta Support for the SocDMMU

Facts

- The SoCDMMU needs to know where the allocated physical memory will be placed in the PE address space.
- The PE address space is much larger than the physical address space (64 MB* vs. 4GB).
- The PE-Address Space Fragmentation can be overcome by:
  - Using the SoCDMUU $G_{move}$ Command (pointers problems)
  - Replicate the physical address space

* A typical global on-chip memory size for billion transistor multiprocessor SoC
Atalanta Support for the SocDMMU

New API New Functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asc_partition_create</td>
<td>Create a partition by requesting memory allocation from the SoCDMMU if necessary.</td>
</tr>
<tr>
<td>asc_partition_delete</td>
<td>Delete a partition and de-allocate memory block if required.</td>
</tr>
<tr>
<td>asc_memory_find</td>
<td>Find a place in the PE address space to which to map the allocated memory.</td>
</tr>
</tbody>
</table>
Comparison to a Fully Shared-Memory Multiprocessor System

Simulation Setup

- Simulation was carried out using Mentor Graphics Co-Verification Environment (CVE), the cycle-accurate XRAY software simulator/debugger and Synopsys VCS Verilog simulator
- ARM SDT was used for software development
Experiment 1

- Global memory of 16MB; Data L1 $ is 64 KB, Instruction L1 $ is 64 KB
- The ARM runs at 150 MHz.
- Accessing the Global Memory costs 5 cycles for the first access.
- A handheld device that utilizes this SoC can be used for OFDM communication as well as other applications (MPEG2 video player).
- Initially the device runs an MPEG2 video player. When the device detects an incoming signal it switches to the OFDM receiver. The switching time (which includes the time for memory management) should be short or the device might lose the incoming message.
Experiment 1

• Sequence of Memory Allocations Required

<table>
<thead>
<tr>
<th>MPEG-2 Player</th>
<th>OFDM Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 Kbytes</td>
<td>34 Kbytes</td>
</tr>
<tr>
<td>500 Kbytes</td>
<td>32 Kbytes</td>
</tr>
<tr>
<td>5 Kbytes</td>
<td>1 Kbytes</td>
</tr>
<tr>
<td>1500 Kbytes</td>
<td>1.5 Kbytes</td>
</tr>
<tr>
<td>1.5 Kbytes</td>
<td>32 Kbytes</td>
</tr>
<tr>
<td>0.5 Kbytes</td>
<td>8 Kbytes</td>
</tr>
<tr>
<td></td>
<td>32 Kbytes</td>
</tr>
</tbody>
</table>
## Experiment 1

Speedup of a single `malloc()`

<table>
<thead>
<tr>
<th></th>
<th>Execution Time (Average Case)</th>
<th>Execution Time (Worst Case)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SDT 2.5 embedded malloc()</strong></td>
<td>106 cycles</td>
<td>559 cycles</td>
</tr>
<tr>
<td><strong>uClib malloc()</strong></td>
<td>222 cycles</td>
<td>1646 cycles</td>
</tr>
<tr>
<td>SoCDMMU allocation</td>
<td>28 cycles</td>
<td>199 cycles</td>
</tr>
<tr>
<td><strong>Speed up over SDT malloc()</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Speed up over uClIBC malloc()</strong></td>
<td>7.92X</td>
<td>8.21X</td>
</tr>
</tbody>
</table>
## Experiment 1

Speedup of a single `free()`

<table>
<thead>
<tr>
<th></th>
<th>Execution Time (Average Case)</th>
<th>Execution Time (Worst Case)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SDT2.5 embedded free()</strong></td>
<td>83 cycles</td>
<td>186 cycles</td>
</tr>
<tr>
<td><strong>uClib free()</strong></td>
<td>208 cycles</td>
<td>796 cycles</td>
</tr>
<tr>
<td><strong>SocDMMU deallocation</strong></td>
<td>14 cycles</td>
<td>28 cycles</td>
</tr>
<tr>
<td><strong>Speed up over SDT free()</strong></td>
<td>5.9X</td>
<td>6.64X</td>
</tr>
<tr>
<td><strong>Speed up over uClibc free()</strong></td>
<td>14.8X</td>
<td>28.42X</td>
</tr>
</tbody>
</table>
## Experiment 1

### Speedup in transition time

<table>
<thead>
<tr>
<th></th>
<th>Using the SOCDMMU</th>
<th>Using SDT <code>malloc()</code> and <code>free()</code></th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Average Case</strong></td>
<td>280 cycles</td>
<td>1240 cycles</td>
<td>4.4X</td>
</tr>
<tr>
<td><strong>Worst Case</strong></td>
<td>1244 cycles</td>
<td>4851 cycles</td>
<td>3.9X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Using the SOCDMMU</th>
<th>Using uClibc <code>malloc()</code> and <code>free()</code></th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Average Case</strong></td>
<td>280 cycles</td>
<td>2593 cycles</td>
<td>9.26X</td>
</tr>
<tr>
<td><strong>Worst Case</strong></td>
<td>1244 cycles</td>
<td>15502 cycles</td>
<td>12.46X</td>
</tr>
</tbody>
</table>
Experiment 2
Speedup in Execution Time

- Same setup used for Experiment 1
- GCC and Glibc were used for development
- 3 kernels from the SPLASH-2 application suite are used
  - Complex 1D FFT (FFT)
  - Integer RADIX sort (RADIX)
  - Blocked LU decomposition (LU)
- They were modified to replace all the static memory allocations by dynamic ones
# Experiment 2

## Speedup in Execution Time

### Glibc malloc() & free()

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>E.T. (Cycles)</th>
<th>Memory Management E. T. (Cycles)</th>
<th>% of E. T. used to Memory Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>LU</td>
<td>318307</td>
<td>31512</td>
<td>9.90%</td>
</tr>
<tr>
<td>FFT</td>
<td>375988</td>
<td>101998</td>
<td>27.13%</td>
</tr>
<tr>
<td>RADIX</td>
<td>694333</td>
<td>141491</td>
<td>20.38%</td>
</tr>
</tbody>
</table>

### Using the SoCDMMU

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>E.T. (Cycles)</th>
<th>Memory Management E. T. (Cycles)</th>
<th>% of E. T. used to Memory Management</th>
<th>% Reduction in Time used to Manage Memory</th>
<th>% Reduction in Benchmark E. T.</th>
</tr>
</thead>
<tbody>
<tr>
<td>LU</td>
<td>288271</td>
<td>1476</td>
<td>0.51%</td>
<td>95.31%</td>
<td>9.44%</td>
</tr>
<tr>
<td>FFT</td>
<td>276941</td>
<td>2951</td>
<td>1.07%</td>
<td>97.10%</td>
<td>26.34%</td>
</tr>
<tr>
<td>RADIX</td>
<td>558347</td>
<td>5505</td>
<td>0.99%</td>
<td>96.10%</td>
<td>19.59%</td>
</tr>
</tbody>
</table>
## Area Estimation of The SoC

<table>
<thead>
<tr>
<th>Element</th>
<th>Number of Transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 ARM9TDMI Cores</td>
<td>4 x 112K = 448K Transistors</td>
</tr>
<tr>
<td>4 L1 Caches (64KB+64KB)</td>
<td>4 x 6.5M = 26M Transistors*</td>
</tr>
<tr>
<td>Global On-Chip Memory (16MB)</td>
<td>134.217M Transistors</td>
</tr>
<tr>
<td>SoCDMMU (w/o memory elements)</td>
<td>30K Transistors</td>
</tr>
<tr>
<td>SoCDMMU Allocation Table</td>
<td>30K Transistors</td>
</tr>
<tr>
<td>SoCDMMU Address Converters (4)</td>
<td>4 x 60K = 240K Transistors</td>
</tr>
<tr>
<td>SoCDMMU (total)</td>
<td>300K Transistors</td>
</tr>
<tr>
<td>SoC (total)</td>
<td>160.965M Transistors</td>
</tr>
<tr>
<td>SoCDMMU w/o memory elements to SoC</td>
<td>0.0186%</td>
</tr>
<tr>
<td>SoCDMMU to SoC (%)</td>
<td>0.186%</td>
</tr>
</tbody>
</table>

* Using dual-port 6T SRAM Cells..
Area Estimation of The SoC

- For this 161 Million transistor chip, the SoCDMMU consumes 300K transistors (0.186% of 161M) and yields a 4-10X speedup in memory allocation/de-allocation.
SoCDMMU Conclusion

- We introduced The Two-Level memory management hierarchy for multiprocessor SoC
- We showed how Level 1 in the hierarchy can be implemented using the SoCDMMU
- We gave a sample hardware implementation of the SoCDMMU
- We introduced DX-Gt to automatically configure/customize the SoCDMU hardware
- We showed how to add the SoCDMMU support to a real-time OS
- Our Experiments show that using the SoCDMMU speeds up the application transition time as well as the application execution time
Outline

- Trends
- Custom Operating System Hardware IP
  - System-on-a-Chip Dynamic Memory Management Unit (SoCDMMU)
  - The $\delta$ Hardware/Software RTOS Generation Framework
- Compilation for Reverse Execution & Debugging
- Conclusion
δ Hardware/Software RTOS Generation Framework and current simulation platform
The $\delta$ Hardware/Software RTOS Generation Framework

A Framework for Automatic Generation of Configuration Files for a Custom Hardware/Software RTOS

<table>
<thead>
<tr>
<th>PE selection</th>
<th>Micellaneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>PE1: PowerPC</td>
<td>Number of cpus: 4</td>
</tr>
<tr>
<td>PE2: PowerPC</td>
<td>Number of tasks: 40</td>
</tr>
<tr>
<td>PE3: PowerPC</td>
<td></td>
</tr>
<tr>
<td>PE4: PowerPC</td>
<td></td>
</tr>
</tbody>
</table>

Specialized Software Components
- Deadlock Detection
- Memory Management

Hardware Components
- SoCCLC
- SoCDDDU
- SoCDMMU
- RTU

IPC methods
- Semaphore
- Event
- MailBox
- Queue
- Mutual
- Allocation

GUI tool

Compile Stage for each system

User Input

Application

Executable HW file for each

Executable SW file for each

RTOS1
RTOS2
RTOS3
RTOS4
RTOS5
RTOS6

SW RTOS w/ sem
SW RTOS + SoCCLC
SW RTOS w/ deadlock avoid.
SW RTOS + SoCDDDU
SW RTOS + SoCCLC + SoCDDDU
RTU

Simulation in Seamless CVE

VCS

XRAY
Outline

• Trends

• Custom Operating System Hardware IP
  • System-on-a-Chip Dynamic Memory Management Unit (SoCDMMU)
  • The δ Hardware/Software RTOS Generation Framework

• Compilation for Reverse Execution & Debugging

• Conclusion
Generation of a Reverse Program

(a): original program
(b): corresponding MVG
(c): reverse program
(d): reduced reverse program

* See thesis by T. Akgul available from http://codesign.ece.gatech.edu
Experimentation Platform

Host

PC
Windows 2000

Target

MBX860
MPC860 processor
4MB DRAM, 2MB Flash
RTC, four 16-bit timers, watchdog

Background Debug Mode (BDM) Interface
## Memory Usage for State Saving

<table>
<thead>
<tr>
<th></th>
<th>ISS (KB)</th>
<th>ISSDI (KB)</th>
<th>RCG (KB)</th>
<th>ISS / RCG</th>
<th>ISSDI / RCG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection sort (100 inputs)</td>
<td>68.2</td>
<td>46.9</td>
<td>7.5</td>
<td>9X</td>
<td>6.3X</td>
</tr>
<tr>
<td>Selection sort (1000 inputs)</td>
<td>6032</td>
<td>4065</td>
<td>151</td>
<td>40X</td>
<td>27X</td>
</tr>
<tr>
<td>Selection sort (10000 inputs)</td>
<td>593389</td>
<td>397913</td>
<td>7237</td>
<td>82X</td>
<td>55X</td>
</tr>
<tr>
<td>Matrix multiply (4x4)</td>
<td>3.6</td>
<td>2.35</td>
<td>0.17</td>
<td>21X</td>
<td>14X</td>
</tr>
<tr>
<td>Matrix multiply (40x40)</td>
<td>2820</td>
<td>1801</td>
<td>12.6</td>
<td>224X</td>
<td>143X</td>
</tr>
<tr>
<td>Matrix multiply (400x400)</td>
<td>2756883</td>
<td>1755006</td>
<td>1250</td>
<td>2206X</td>
<td>1404X</td>
</tr>
<tr>
<td>ADPCM (32KB input data)</td>
<td>1544</td>
<td>1192</td>
<td>616</td>
<td>2.5X</td>
<td>2X</td>
</tr>
<tr>
<td>ADPCM (64KB input data)</td>
<td>3088</td>
<td>2384</td>
<td>1232</td>
<td>2.5X</td>
<td>2X</td>
</tr>
<tr>
<td>ADPCM (128KB input data)</td>
<td>6175</td>
<td>4767</td>
<td>2464</td>
<td>2.5X</td>
<td>2X</td>
</tr>
<tr>
<td>LZW (1KB input data)</td>
<td>5630</td>
<td>3425</td>
<td>98.4</td>
<td>57X</td>
<td>35X</td>
</tr>
<tr>
<td>LZW (4KB input data)</td>
<td>64970</td>
<td>39163</td>
<td>351</td>
<td>185X</td>
<td>112X</td>
</tr>
<tr>
<td>LZW (16KB input data)</td>
<td>784336</td>
<td>471140</td>
<td>1331</td>
<td>589X</td>
<td>354X</td>
</tr>
</tbody>
</table>
Program Re-execute Approach vs. RCG

- Forward execution
- Reverse execution via RCG

400x400 matrix multiply

Starting point for forward execution
Starting point for reverse execution
Conclusion

- Trends
- SoCDMMU
- Reverse Program Generation
- Have fun at FPGAworld 2007!