Managing Heterogeneity by Light-weight Abstraction and Self-Guidance

Rainer Buchty

KIT, Institute of Computer Science & Engineering (ITEC), Chair for Computer Architecture and Parallel Processing
Eberhard Karls Univ. Tübingen, Wilhelm Schickard Institute for Computer Science (WSI), Dept. of Computer Engineering
Motivation

Heterogeneity on the rise

- in the past: “everything is software”
- Application requirements and Technology aspects shift focus
- Revival of heterogenous architectures
  - System architectures
  - Processor architectures
  - Platform FPGAs

(Host + accelerator)
(STI Cell BE)
(Xilinx Virtex)

Status quo

- Multicore architectures for general-purpose use
- Manycore architectures for data-parallel acceleration
- Reconfigurable architectures for dedicated acceleration

Source: Intel Corp.
Motivation (cont’d)

Architectures

- **Thread & task-level parallelism**
  - Multiplication of general-purpose cores
  - Same ISA and (typ.) speed
  - **Examples**: IA32, Tilera

- **Data parallelism**
  - ALU replication, e.g. FP accelerators
  - Host/Master ↔ Accelerator/Slave
  - Host enforces control flow
  - **Examples**: GPU, ClearSpeed

- **Heterogeneous architectures**
  - Heterogeneous CPUs *(Cell BE)*
  - Host+accelerator *(GPU, FPGAs)*

Source: Intel

Source: Clearspeed
Motivation (cont’d)

Example: HTX-based reconfigurable accelerator

- FPGA-based universal accelerator
- Flexible use of FPGA resources by partitioning
  - Dynamic configuration of individual “slots”
  - Focus on use within multitasking/multithreading environments

HTX-based reconfigurable accelerator

Accelerator system

- PC-based host running Linux
- FPGA fabric partitioned into 6 slots
  - Individual accelerator modules
  - Central control via Command & Status Bus
  - Abstract interface in hardware
  - Monitoring facility
- HyperTransport bus interface
  - Memory-mapped I/O
  - DMA-capable accelerators
“If you build it, they will come ...”

... and curse you.

- **Heterogeneous architectures:**
  
  Easy to build but a pain to program

- **Hardware-aware approach**
  
  - Leaving everything to the programmer
  
  - Fine-grain control, but tedious work
  
  - Worst case: several environments, several languages

- **Vendor-specific approaches**
  
  - Dedicated platform-specific environments
  
  - Easing programming, but transition basically means reimplementation

- **Problem-specific approaches**
  
  - Focus on parallelism level

In any case: **heavy impact on source code**
Programmability

Arising problems

1. Collision of principles
   - Parallelization on abstract level
   - Architecture mapping: hardware-aware, specific

2. Complexity aspects
   - Resource sharing in multitasking environments
   - Phase behavior of applications
   - Impact of workload

3. Compatibility aspects
   - Re-programming means re-approval
Providing required abstraction
Step 1: Achieving abstraction

Uniform application description

- Introduce **abstraction layer** for decoupling programmers from hardware
- **Function-level granularity** sufficient
  - Provide individual function implementations
  - Invoke desired implementation (and therefore associated hardware) during run-time
- **Sounds like dynamic linking**
  - Dynamic linking included in any modern OS
  - however: performed only once per function call
- But: **Any-time re-linking required**
  - Dynamic selection of suitable implementation
Abstraction layer

Light-weight run-time layer extension

- Function call is a proxy
- Proxy dynamically mapped to desired implementation
- Flexible mapping-control enabling external guidance
- No measurable impact on run-time

Abstraction layer (cont’d)

Expansion of Task-State Segment (TSS)

- **TSS**: OS’s task management structure
  - TSS handled in software, hence changes possible
  - Slight changes to kernel source required

- **Keep management list with thread**
  - Function mappings individual per thread
  - “Unlimited” implementation alternatives possible
  - Registering of alternatives required
Abstraction layer (cont’d)

Kernel
- `dls_set_fct()`

dls.h
- `dls_struct_ptr`
  - `this: dls_struct*`
  - `next: dls_struct_ptr*`

ProcFS

Control Daemon

Proxy Function
- `long (*fct)(int a, ...)`

`dls_fcts_ptr`: `dls_fct_type*`
- `num_fcts: int`
- `next: dls_struct*`

`long libfct_a(int a, ...)`
- `long libfct_b(int a, ...)`
Software stack

Flexibility and compatibility

- Embrace OS structure
- Spans 4 dedicated system layers
  - Application and library reside in user address space
  - Control daemon decoupled in own address space
  - Kernel address space (hardware access)
  - Hardware
- Interfacing between layers
  - Inter-process communication (IPC) using procfs between user and daemon address space
  - Hardware device drivers between kernel and hardware
- Basic framework open for later extension
Software stack (cont’d)

Application

AMS

Library

IPC

AMS

Control Daemon

Device

AMS

Kernel Driver

Mem.

Accel.

Accel.

Main Memory

User address space

Daemon address space

Kernel address space

Hardware
Dealing with complexity
 Hardware awareness? Application awareness!

- Hardware-aware mapping not enough
  - Tasks competing for resources
  - Applications expose phase behavior
  - Different workloads ↔ different “best” implementations

- Programmer unable to oversee all eventualities
- But even if...
  - Most programming time is spent on implementation selection, not implementation itself
  - Detection of workload, congestion, phase ...

- How to deal with that?
Complexity issues (cont’d)

- **Overcoming complexity by Self-X**
  - Self-awareness: system-state analysis and evaluation
  - Self-adaptation and Self-optimization
  - Self-protection and Self-healing

- **Introduce bio-inspired flexibility**
  - “Sensors and actuators”
  - Communication and control

---

## Cost-aware function migration

### Harnessing the power of Self-X

1. **More than workload balancing required**
   - Selection of most suitable implementation
   - Knowledge about run-time required

2. **Run-time insufficient criterion**
   - Run-time differs with workload of task
   - Different workloads might require different implementations

3. **Off-line training lacks dynamics**
   - Application phases in relation to workload
   - Competition in multitasking systems
   - Dynamic resource availability
Step 2: Achieving cost-awareness

Measuring execution time

- Unobtrusive method required
  - No instrumentation on source-code level
  - Could we move it into abstraction layer?
- Proxy points to function implementation
  - Why not call timer functions before and after as well?
  - Dynamic instrumentation on run-time level
Cost-aware function migration (cont’d)

Dynamic instrumentation

- Light-weight expansion of the abstraction layer
- Proxy function resolves to function list
- Call any amount of fcts. before and after selected implementation
- However: requires caller stack-frame duplication
- Instrumentation costs hidden by pre/post fcts.

Cost-aware function migration (cont’d)

Stack-frame manipulation
Step 3: Cost-awareness and Evaluation

Evaluation and guided execution

- Two-step process
  1. Online-creation of initial classification
  2. Guided execution

- Learning retriggered upon changes / deviations

Example: Time consumption of square-matrix multiplication related to dimensions and acceleration method
Cost-aware function migration (cont’d)

Phase 1: Online learning

- Rate only execution, not start-up time
  - First two executions are not measured
  - Neglect influence of library loading and linking, CUDA kernel invocation, etc.

- Create initial classification
  - Alternate use of implementations
  - 5 runs per implementation
  - Determine cost value from workload size and execution time
Cost-aware function migration (cont’d)

Classification process
Phase 2: Guided execution

- Select implementation based on workload size and associated cost value
- Measure execution time
- Redo classification if too much deviation from expectation

Cost-aware function migration (cont’d)

Adaptation of classes during application runtime in reaction to resource contention
Delivering guidance information
Compatibility issues

So far we achieved...

- (Almost) **compatibility on source-code level**
  - Only registration of functions required
  - No code overloading with implementation selection
  - Approach orthogonal to parallel programming models

- **Compatibility on execution level**
  - Transparent changes to runtime system
  - Legacy software unharmed

- **But what about run-time compatibility?**
  - Classification takes time
  - Application might break due to given constraints!
Compatibility issues (cont’d)

Constraint-based guidance

- Annotate application requirements
  - Throughput, execution speed, accuracy, ...
- Deliver pre-classification of implementations
  - Speed up/avoid initial classification
  - Re-classification eventually done later

- Source-code attribution using pragmas
- Binary-level attribution using additional sections or resource files
- Compatibility achieved on both levels

Fabian Nowak, Rainer Buchty: Providing Guidance Information for Application Mapping on Heterogeneous Parallel Systems; 22nd PARS Workshop, Parsberg, Juni 2009
Providing guidance information

example.c:

```c
/* Attributed implementation */
#pragma DLRS ...
void matrix_mult (matrix *A,
    matrix *B,
    matrix **C) {
...
...
int main () {
    /* Attributed call */
    #pragma DLRS ...
    matrix_mult (A, B, C);
    return 0;
}
```

1. Extract attributes from source code
2. Generate attribute file
3. Bind attributes into binary format

Compatibility with existing tool chain
Evaluate guidance information

Run-time evaluation

- Requirements for function calls
- Implementation performance
- Available HW resources

Fabian Nowak, Mario Kicherer, Rainer Buchty, Wolfgang Karl: Delivering Guidance Information in Heterogeneous Systems, PARS 2010, Hannover, Februar 2010
Summary
# Summary of Features

## Benefits

- **Maximum compatibility**
  - Source-code level (registering, guidance information)
  - Binary level (guidance information)
  - Run-time (performance)

- **Interoperability** with existing approaches
  - Programming models (HW-aware, OpenMP, CUDA)
  - Programming tools (gcc, gdb, ...)

- **Modest expansion** of existing services

→ easy upgrade path from conventional to self-guiding systems
Managing Heterogeneity...

- Attributes
  - Code
    - LOAD r0, arg
    - LOAD r1, arg
    - call fn()

- Code

- Control System
  - Code Layer
  - Run-time Layer

- Library
  - HW Library
  - SW Library

- System/HW Monitor(s)

- Hardware Domains
  - HW Predef.
  - Univ. Binary
  - Hardware Abstraction Layer
  - Heterog. Processing Hardware

- Compiler Domains
  - Impl. #1
  - Impl. #2
  - Impl. #3

- Predefined HW

- Hardware Domains
  - HW Domain
  - Compiler Domain

- Application

- Code
  - Attributes
    - Impl. #1
    - Impl. #2
    - Impl. #3

- Control System

- Application

- Code
  - Attributes
    - Impl. #1
    - Impl. #2
    - Impl. #3
Managing Heterogeneity by Light-weight Abstraction and Self-Guidance

Rainer Buchty

KIT, Institute of Computer Science & Engineering (ITEC), Chair for Computer Architecture and Parallel Processing
Eberhard Karls Univ. Tübingen, Wilhelm Schickard Institute for Computer Science (WSI), Dept. of Computer Engineering
## Function resolution

### Basic overhead

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>avg.</th>
<th>max</th>
<th>Ovhd.</th>
</tr>
</thead>
<tbody>
<tr>
<td>native</td>
<td>21.26s</td>
<td>21.60s</td>
<td>21.91s</td>
<td>–</td>
</tr>
<tr>
<td>GLS</td>
<td>21.26s</td>
<td>21.60s</td>
<td>21.91s</td>
<td>0</td>
</tr>
<tr>
<td>DLS-DL</td>
<td>21.08s</td>
<td>21.54s</td>
<td>21.88s</td>
<td>≈0</td>
</tr>
<tr>
<td>DLS-SL</td>
<td>21.06s</td>
<td>21.57s</td>
<td>21.94s</td>
<td>≈0</td>
</tr>
</tbody>
</table>
Function resolution (cont’d)

**Worst case (no fct. payload, external trigger)**

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>avg.</th>
<th>max</th>
<th>Ovhd.</th>
</tr>
</thead>
<tbody>
<tr>
<td>native</td>
<td>21.26s</td>
<td>21.60s</td>
<td>21.91s</td>
<td>–</td>
</tr>
<tr>
<td>GLS</td>
<td>60.86s</td>
<td>63.22s</td>
<td>65.58s</td>
<td>2.93</td>
</tr>
<tr>
<td>DLS-DL</td>
<td>66.88s</td>
<td>69.60s</td>
<td>72.41s</td>
<td>3.22</td>
</tr>
<tr>
<td>DLS-SL</td>
<td>35.20s</td>
<td>37.20s</td>
<td>39.40s</td>
<td>1.72</td>
</tr>
</tbody>
</table>

**Worst case (no fct. payload, internal trigger)**

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>avg.</th>
<th>max</th>
<th>Ovhd.</th>
</tr>
</thead>
<tbody>
<tr>
<td>native</td>
<td>21.26s</td>
<td>21.60s</td>
<td>21.91s</td>
<td>–</td>
</tr>
<tr>
<td>GLS</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>DLS-DL</td>
<td>47.33s</td>
<td>48.41s</td>
<td>49.35s</td>
<td>2.24</td>
</tr>
<tr>
<td>DLS-SL</td>
<td>21.03s</td>
<td>21.85s</td>
<td>22.66s</td>
<td>1.01</td>
</tr>
</tbody>
</table>
## Function resolution (cont’d)

### TSS overhead (OpenMP baseline)

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>avg.</th>
<th>max</th>
<th>Ovhd.</th>
</tr>
</thead>
<tbody>
<tr>
<td>w/o</td>
<td>24.09s</td>
<td>24.88s</td>
<td>25.84s</td>
<td>–</td>
</tr>
<tr>
<td>DLS-SL</td>
<td>25.88s</td>
<td>26.53s</td>
<td>28.26s</td>
<td>1.06</td>
</tr>
</tbody>
</table>

### TSS overhead (OpenMP stress test)

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>avg.</th>
<th>max</th>
<th>Ovhd.</th>
</tr>
</thead>
<tbody>
<tr>
<td>w/o</td>
<td>24.09s</td>
<td>24.88s</td>
<td>25.84s</td>
<td>–</td>
</tr>
<tr>
<td>DLS-SL</td>
<td>36.32s</td>
<td>38.36s</td>
<td>41.87s</td>
<td>1.54</td>
</tr>
</tbody>
</table>
Function resolution (cont’d)

### Thread-related overhead

<table>
<thead>
<tr>
<th>#Threads</th>
<th>1</th>
<th>5</th>
<th>10</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLS-SL</td>
<td>35.93s</td>
<td>39.38s</td>
<td>38.36s</td>
<td>38.29s</td>
</tr>
</tbody>
</table>

### Thread-related overhead

<table>
<thead>
<tr>
<th>#Functions</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLS-SL</td>
<td>37.28s</td>
<td>38.10s</td>
<td>37.46s</td>
<td>37.76s</td>
</tr>
</tbody>
</table>
### Instrumentation

#### Cost of instrumentation

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Time for $10^6$ iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple fct. call</td>
<td>6 ns</td>
</tr>
<tr>
<td>Basic instrumentation (no payload)</td>
<td>57 ns</td>
</tr>
<tr>
<td><em>Dyninst v6.1 fct. start</em></td>
<td>132 ns</td>
</tr>
<tr>
<td><em>Dyninst v6.1 fct. start/end</em></td>
<td>243 ns</td>
</tr>
<tr>
<td>Instrumentation w/ time measuring</td>
<td>2137 ns</td>
</tr>
</tbody>
</table>

#### Cost of stack-frame manipulation (32-bit args.)

<table>
<thead>
<tr>
<th># of args.</th>
<th>1</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>24</th>
<th>32</th>
<th>48</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>time (ns)</td>
<td>57</td>
<td>57</td>
<td>57</td>
<td>65</td>
<td>73</td>
<td>86</td>
<td>104</td>
<td>120</td>
<td>188</td>
</tr>
</tbody>
</table>
## Self-Guidance

### Sorting application

<table>
<thead>
<tr>
<th></th>
<th>Kernel</th>
<th>Rel. Time</th>
<th>App.</th>
<th>Rel. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLS-RTopt</td>
<td>445μs</td>
<td>1.00</td>
<td>72.4s</td>
<td>1.00</td>
</tr>
<tr>
<td>CPU (ser.)</td>
<td>563μs</td>
<td>1.27</td>
<td>82.8s</td>
<td>1.14</td>
</tr>
<tr>
<td>CPU (par.)</td>
<td>801μs</td>
<td>1.80</td>
<td>107.6s</td>
<td>1.49</td>
</tr>
<tr>
<td>GPU</td>
<td>523μs</td>
<td>1.18</td>
<td>79.2s</td>
<td>1.09</td>
</tr>
</tbody>
</table>

### Matrix multiplication

<table>
<thead>
<tr>
<th></th>
<th>Kernel</th>
<th>Rel. Time</th>
<th>App.</th>
<th>Rel. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLS-RTopt</td>
<td>207μs</td>
<td>1.00</td>
<td>154s</td>
<td>1.00</td>
</tr>
<tr>
<td>CPU (ser.)</td>
<td>1184μs</td>
<td>5.72</td>
<td>247s</td>
<td>1.60</td>
</tr>
<tr>
<td>CPU (par.)</td>
<td>259μs</td>
<td>1.25</td>
<td>159s</td>
<td>1.03</td>
</tr>
<tr>
<td>GPU</td>
<td>285μs</td>
<td>1.38</td>
<td>158s</td>
<td>1.03</td>
</tr>
</tbody>
</table>
## Mersenne Twister

<table>
<thead>
<tr>
<th></th>
<th>Kernel</th>
<th>Rel. Time</th>
<th>App.</th>
<th>Rel. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLS-RTopt</td>
<td>$282\mu s$</td>
<td>1.00</td>
<td>$0.291s$</td>
<td>1.00</td>
</tr>
<tr>
<td>CPU (ser.)</td>
<td>$443\mu s$</td>
<td>1.57</td>
<td>$0.443s$</td>
<td>1.52</td>
</tr>
<tr>
<td>GPU</td>
<td>$302\mu s$</td>
<td>1.07</td>
<td>$0.312s$</td>
<td>1.07</td>
</tr>
</tbody>
</table>
## Performance of Self-Guidance (cont’d)

### Worst-case estimation

<table>
<thead>
<tr>
<th></th>
<th>Kernel (µs)</th>
<th>Rel. Time</th>
<th>App. (s)</th>
<th>Rel. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLS-RTopt</td>
<td>1340</td>
<td>≈0.99</td>
<td>25.4</td>
<td>≈0.99</td>
</tr>
<tr>
<td>CPU (ser.)</td>
<td>1330</td>
<td>1.00</td>
<td>25.3</td>
<td>1.00</td>
</tr>
<tr>
<td>GPU</td>
<td>2900</td>
<td>2.18</td>
<td>40.0</td>
<td>1.58</td>
</tr>
</tbody>
</table>

### Best-case estimation

<table>
<thead>
<tr>
<th></th>
<th>Kernel (µs)</th>
<th>Rel. Time</th>
<th>App. (ms)</th>
<th>Rel. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLS-RTopt</td>
<td>174</td>
<td>1.00</td>
<td>416</td>
<td>1.00</td>
</tr>
<tr>
<td>CPU (ser.)</td>
<td>395</td>
<td>2.27</td>
<td>571</td>
<td>1.37</td>
</tr>
<tr>
<td>GPU</td>
<td>388</td>
<td>2.22</td>
<td>628</td>
<td>1.51</td>
</tr>
</tbody>
</table>
Managing Heterogeneity by Light-weight Abstraction and Self-Guidance

Rainer Buchty