Hardware Acceleration for Just-In-Time Compilation on Heterogeneous Embedded Systems

A. Carbon, Y. Lhuillier, H.-P. Charles

CEA LIST
DACLE division
Embedded Computing & Embedded Software Laboratories
France

Contact: alexandre.carbon@cea.fr

24th IEEE International Conference on Application-specific Systems, Architectures and Processors
June 5-7 2013, Washington D.C., USA.
Outline

- Context: Virtualization & JIT emergence
- JIT optimization opportunities (based on the LLVM framework)
- Hardware accelerator proposal
- Experiments & results
- Conclusion
Parallelism emergence
- Parallelism emergence
- Heterogeneity development in *computing systems*
  - ILT/TLP-based multi-cores and wide SIMD GPUs
Parallelism emergence

Heterogeneity development in *embedded systems*

- Heterogeneous asymmetric many-core processors (AMP) emergence

---

- PPE: a major concern of embedded systems designers
- Code deployment on such architectures
  - High development cost to efficiently target one AMP
  - Code portability has become a major issue
Emergence of virtualization abstraction layers

- First mention in 60’s (IBM 360/67)
- Java Virtual Machines, CLI, LLVM => Virtual Machines (VMs) development
Context: Virtualization emergence

- Emergence of virtualization abstraction layers
  - Based initially on interpretation
  - Suffer from considerable performance overheads
Emergence of virtualization abstraction layers

- Coupling today interpretation and Just-In-Time compilation
Context: Just-In-Time emergence

- JIT compilation: widely used in GPP
- Performance & consumption overheads in embedded

- 2 kinds of existing optimizations to reduce JIT impact
  - Software optimizations
  - System design specialization
    - Specialized dedicated resources
    - Additional standard dedicated resources
  - JIT compilation complexity limits performance gains
    - Pointer-based algorithms

→ Proposing tuned hardware associated to these dedicated resources to manage JIT compilation algorithms

---

1 Ting Cao et al. « The yin and yang of power and performance for asymmetric hardware and managed software », ISCA ’12.
Outline

- Context: Virtualization & JIT emergence
- JIT optimization opportunities (based on the LLVM framework)
- Hardware accelerator proposal
- Experiments & results
- Conclusion
LLVM framework

- LLVM bytecode compiler (LLC): used in many projects
- Profiling: identifying LLC’s most critical parts
  - Experiments on a ARM Cortex-A5 model

- Associative array management & dynamic memory allocation: on average 24% of LLC execution time
LLVM: existing software optimizations

- New abstract-data-types (optimized versions of STL C++ ADT)
  - Provide [multi]map & multi[set] abstract data types
  - Hash table implementations rather than sorted-tree

- STL C++: still used when performance is not a key issue

- Specialized allocators (eg. RecycleAllocator)
  - Keeps track of recently deallocated objects to reuse them
  - Avoiding frequent allocations and deallocations

- Despite these software optimizations, associative arrays & memory allocation still prevail
Our goal: proposing an alternative acceleration of associative arrays & dynamic memory allocation

Mean: standardization of LLC
- Proposing a solution based on standard libraries for reuse
- Using only STL C++ library for [multi]map & multi[set] ADT
- C’s memory allocation standard library (dlmalloc-based)

Transferring optimizations to a hardware accelerator

Solution portability: accelerator reuse
- Benefit to all pointer-based algorithms using massively associative arrays and dynamic memory allocation
Outline

- Context: Virtualization & JIT emergence
- JIT optimization opportunities (based on the LLVM framework)
- Hardware accelerator proposal
- Experiments & results
- Conclusion
Current implementation of associative arrays & memory allocation

- Standard C++ `Map & Set` libraries
  - Using Red-Black Tree representation
    - Binary tree with coloring property

- C’s memory allocator: `dlmalloc`
  - Using associative arrays to associate data sizes with free memory chunks
  - Using hash table & double linked-lists

Systematic usage of RB-Trees

- Proposing an implementation of `dlmalloc` using RB-Trees
- Modifying the allocator without modifying user interface
Hardware acceleration description

- New RB-Tree node structure: held in 128-bits
  - Digest key in 31-bits, color in the last bit, preserving the sorting order

![Diagram showing initial and proposed RB-Tree node structures](image)

- Proposing hardware accelerated instructions
  - Specialized instructions for RB-Tree management functions
  - Accelerating traversals, key look-ups, balanced insertion and removal
15 new instructions over 400 in ARM ISA

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Function</th>
<th>Used by</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBTINC Rd, Rm</td>
<td>increment</td>
<td>map::iterator, set::iterator</td>
</tr>
<tr>
<td>RBTDEC Rd, Rm</td>
<td>decrement</td>
<td>map::iterator, set::iterator</td>
</tr>
<tr>
<td>RBTLOW Rd, Rn, Rm</td>
<td>lower bound</td>
<td>map::lower_bound, set::lower_bound, malloc, realloc</td>
</tr>
<tr>
<td>RBTUP Rd, Rn, Rm</td>
<td>upper_bound</td>
<td>map::upper_bound, set::upper_bound</td>
</tr>
<tr>
<td>RBTDEL Rn, Rm</td>
<td>rebalance for erase</td>
<td>map::erase, set::erase, malloc, realloc, free</td>
</tr>
<tr>
<td>RBTINS &lt;L</td>
<td>R&gt; Rn, Rm, Rs</td>
<td>insert &amp; rebalance</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- Read at most 3 registers, write at most 1 register
- Multi-cycles instructions: hiding iterative computations
New Cortex-A5 pipeline & HW accelerator proposal

Full FSM size estimation: 161 states, 223 transitions
Outline

- Context: Virtualization & JIT emergence
- JIT optimization opportunities (based on the LLVM framework)
- Hardware accelerator proposal
- Experiments & results
- Conclusion
Instrumented ARM Cortex-A5 ISS with a cache simulator

Speedups obtained for SW and HW accelerated versions of LLC

- **SW**: 29% of gain, **HW**: 50% / LLC standardized version
- **15%** of gain for HW acc / SW acc
Experiments & Results

- Evolution of time spent in memory allocation & associative array management (relative to total execution time)

![Graph showing evolution of time spent in memory allocation & associative array management](image)

- From **41%** to **24%** (SW) & **12%** (HW)
- Raw speedup on memory allocation & associative arrays: \(\approx 5x\)
Outline

- Context: Virtualization & JIT emergence
- JIT optimization opportunities (based on the LLVM framework)
- Hardware accelerator proposal
- Experiments & results
- Conclusion
Conclusion

- Interest of dedicated resources for virtualization services
- Limited gains for JIT compilation with SW optimizations
  - Impact of dynamic memory allocation & associative array management on execution time
- Proposing tuned HW for JIT compilation, coupled to dedicated resources
- HW accelerator hidden behind standard libraries
  - Valuable for all pointer-based algorithms
  - ISA extension for RB-Tree management functions
  - 15% of gain comparing to SW opt in LLVM code gen
  - ≈5x raw speedup for memory allocation & associative array management

- Next acceleration opportunities: instruction graph handling
Thank you

Questions?