CERE: LLVM based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization

Chadi Akel, P. de Oliveira Castro, M. Popov, E. Petit, W. Jalby

University of Versailles – Exascale Computing Research

EuroLLVM 2016
Motivation

- Finding best application parameters is a costly iterative process

- **Codelet Extractor and REplayer**
  - Break an application into standalone codelets
  - Make costly analysis affordable:
    - Focus on single regions instead of whole applications
    - Run a single representative by clustering similar codelets
Extract codelets as standalone microbenchmarks.

C or Fortran application

- capture machine state

```
for (i = 0; i < N; i ++) {
    for (j = 0; j < N; j ++) {
        a[i][j] += b[i]*c[j];
    }
}
```

codelet wrapper

- state dump

IR

```
for (i = 0; i < N; i ++) {
    for (j = 0; j < N; j ++) {
        a[i][j] += b[i]*c[j];
    }
}
```

codelets can be recompiled and run independently.
CERE Workflow

Applications → LLVM IR Region outlining → Working set and cache capture → Region Capture

Change: number of threads, affinity, runtime parameters → Warmup + Replay → Generate codelets wrapper → Codelet Replay

Retarget for: different architecture, different optimizations

CERE can extract codelets from:
- Hot Loops
- OpenMP non-nested parallel regions [Popov et al. 2015]
Extracting and Replaying Codelets
- Faithful
- Retargetable

Applications
- Architecture selection
- Compiler flags tuning
- Scalability prediction

Demo
- Capture and replay in NAS BT
- Simple flag replay for NAS FT

Conclusion
Capturing codelets at Intermediate Representation

- **Faithful**: behaves similarly to the original region
- **Retargetable**: modify runtime and compilation parameters

![Diagram showing Faithful and Retargetable on the same assembly vs different assembly]

- Same assembly: similar to original region
- Hard to retarget (compiler, ISA)
- Costly to support various ISA

- Not assembly: [Akel et al. 2013]
- Easy to retarget
- Costly to support various languages

- LLVM Intermediate Representation is a good tradeoff
Faithful capture

- Required for **semantically accurate** replay:
  - Register state
  - Memory state
  - OS state: locks, file descriptors, sockets
- No support for OS state except for locks. CERE captures fully from userland: no kernel modules required.

- Required for **performance accurate** replay:
  - Preserve code generation
  - Cache state
  - NUMA ownership
  - Other warmup state (eg. branch predictor)
Faithful capture: memory

Capture access at page granularity: coarse but fast

- region to capture
  - protect static and currently allocated process memory (/proc/self/maps)
  - intercept memory allocation functions with LD_PRELOAD
    1. allocate memory
    2. protect memory and return to user program
- memory allocation
  - a = malloc(256);
- memory access
  - a[i]++;
  - segmentation fault handler
    1. dump accessed memory to disk
    2. unlock accessed page and return to user program

- Small dump footprint: only touched pages are saved
- Warmup cache: replay trace of most recently touched pages
- NUMA: detect first touch of each page
Outline

Extracting and Replaying Codelets
  Faithful
  Retargetable

Applications
  Architecture selection
  Compiler flags tuning
  Scalability prediction

Demo
  Capture and replay in NAS BT
  Simple flag replay for NAS FT

Conclusion
Selecting Representative Codelets

- **Key Idea:** Applications have redundancies
  - Same codelet called multiple times
  - Codelets sharing similar performance signatures
  - Detect redundancies and keep only one representative

Figure: SPEC tonto make_f@shell2.F90:1133 execution trace. 90% of NAS codelets can be reduced to four or less representatives.
Step A: Perform static and dynamic analysis on a reference architecture to capture codelet's feature vectors.

Step B: Using the proximity between feature vectors we cluster similar codelets and select one representative per cluster.

Step C: CERE extracts the representatives as standalone codelets. A model extrapolates full benchmark results.

[Oliveira Castro et al. 2014]
Figure: Benchmarking NAS serial on three architectures

- **real**: speedup when benchmarking original applications
- **predicted**: speedup predicted with representative codelets
- **CERE 31× cheaper** than running the full benchmarks.
Autotuning LLVM middle-end optimizations

- LLVM middle-end offers more than 50 optimization passes.
- Codelet replay enable per-region fast optimization tuning.

![Graph showing performance improvement](image)

**Figure**: NAS SP ysolve codelet. 1000 schedules of random passes combinations explored based on O3 passes.

CERE **149× cheaper** than running the full benchmark ( **27× cheaper** when tuning codelets covering 75% of SP)
Fast Scalability Benchmarking with OpenMP Codelets

![Bar chart showing runtime cycles for varying thread numbers on Sandy Bridge](chart.png)

<table>
<thead>
<tr>
<th></th>
<th>Core2</th>
<th>Nehalem</th>
<th>Sandy Bridge</th>
<th>Ivy Bridge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>98.2%</td>
<td>97.1%</td>
<td>92.6%</td>
<td>97.2%</td>
</tr>
<tr>
<td>Acceleration</td>
<td>× 25.2</td>
<td>× 27.4</td>
<td>× 23.7</td>
<td>× 23.7</td>
</tr>
</tbody>
</table>

**Figure**: Varying thread number at replay in SP and average results over OMP NAS [Popov et al. 2015]
Outline

Extracting and Replaying Codelets
  Faithful
  Retargetable

Applications
  Architecture selection
  Compiler flags tuning
  Scalability prediction

Demo
  Capture and replay in NAS BT
  Simple flag replay for NAS FT

Conclusion
Conclusion

- CERE breaks an application into faithful and retargetable codelets
- Piece-wise autotuning:
  - Different architecture
  - Compiler optimizations
  - Scalability
  - Other exploration costly analysis?
- Limitations:
  - No support for codelets performing IO (OS state not captured)
  - Cannot explore source-level optimizations
  - Tied to LLVM
- Full accuracy reports on NAS and SPEC’06 FP available at benchmark-subsetting.github.io/cere/#Reports
https://benchmark-subsetting.github.io/cere/
distributed under the LGPLv3

Thanks for your attention!
Akel, Chadi et al. (2013). “Is Source-code Isolation Viable for Performance Characterization?” In: 42nd International Conference on Parallel Processing Workshops. IEEE.


Retargetable replay: register state

- **Issue:** Register state is non-portable between architectures.

- **Solution:** capture at a function call boundary
  - No shared state through registers except function arguments
  - Get arguments directly through portable IR code

- Register agnostic capture

- Portable across Atom, Core 2, Haswell, Ivybridge, Nehalem, Sandybridge

- Preliminary portability tests between x86 and ARM 32 bits
Retargetable replay: outlining regions

Step 1: Outline the region to capture using \textit{CodeExtractor} pass

\begin{verbatim}
original:
%0 = load i32* %i, align 4
%1 = load i32* %s.addr, align 4
%cmp = icmp slt i32 %0, %1
br i1 %cmp, ; \textit{loop branch here}
label %for.body,
label %for.exitStub ...
\end{verbatim}

\begin{verbatim}
define internal void @outlined(
i32* %i, i32* %s.addr,
i32** %a.addr) {
  call void @start_capture(i32* %i,
i32* %s.addr, i32** %a.addr)
  %0 = load i32* %i, align 4
  ...
  ret void
}
original:
call void @outlined(i32* %i,
i32* %s.addr, i32** %a.addr)
\end{verbatim}

Step 2: Call \textit{start\_capture} just after the function call

Step 3: At replay, reinlining and variable cloning [Liao et al. 2010] steps ensure that the compilation context is close to original
Comparison to other Code Isolating tools

<table>
<thead>
<tr>
<th></th>
<th>CERE</th>
<th>Code Isolator</th>
<th>Astex</th>
<th>Codelet Finder</th>
<th>SimPoint</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Support</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Language</td>
<td>C(++) , Fortran, ...</td>
<td>Fortran</td>
<td>C, Fortran</td>
<td>C(++) , Fortran</td>
<td>assembly</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indirections</td>
<td>IR</td>
<td>source</td>
<td>source</td>
<td>source</td>
<td>assembly</td>
</tr>
<tr>
<td></td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><strong>Replay</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simulator</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hardware</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td><strong>Reduction</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Capture size</td>
<td>reduced</td>
<td>reduced</td>
<td>reduced</td>
<td>full</td>
<td>-</td>
</tr>
<tr>
<td>Instances</td>
<td>yes</td>
<td>manual</td>
<td>manual</td>
<td>manual</td>
<td>yes</td>
</tr>
<tr>
<td>Code sign.</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
Figure: The Coverage the percentage of the execution time captured by codelets. The Accurate Replay is the percentage of execution time replayed with an error less than 15%.
Figure: The Coverage is the percentage of the execution time captured by codelets. The Accurate Replay is the percentage of execution time replayed with an error less than 15%.

- low coverage (sphinx3, wrf, povray): < 2000 cycles or IO
- low matching (soplex, calculix, gamess): warmup “bugs”
CERE page capture dump size

Figure: Comparison between the page capture and full dump size on NAS.A benchmarks. CERE page granularity dump only contains the pages accessed by a codelet. Therefore it is much smaller than a full memory dump.
CERE page capture is **coarser but faster**

<table>
<thead>
<tr>
<th></th>
<th>CERE</th>
<th>ATOM 3.25</th>
<th>PIN 1.71</th>
<th>Dyninst 4.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>cg.a</td>
<td>19.4</td>
<td>98.82</td>
<td>222.67</td>
<td>896.86</td>
</tr>
<tr>
<td>ft.a</td>
<td>24.1</td>
<td>44.22</td>
<td>127.64</td>
<td>1054.70</td>
</tr>
<tr>
<td>lu.a</td>
<td>62.4</td>
<td>80.72</td>
<td>153.46</td>
<td>301.4</td>
</tr>
<tr>
<td>mg.a</td>
<td>8.9</td>
<td>107.69</td>
<td>168.61</td>
<td>989.53</td>
</tr>
<tr>
<td>sp.a</td>
<td>73.2</td>
<td>67.56</td>
<td>93.04</td>
<td>203.66</td>
</tr>
</tbody>
</table>

Slowdown of a full capture run against the original application run. (takes into account the cost of writing the memory dumps and logs to disk and of tracing the page accesses during the whole execution.). We compare to the overhead of memory tracing tools as reported by [Gao et al. 2005].
CERE cache warmup

for (i=0; i < size; i++)
a[i] += b[i];

array a[] pages
array b[] pages

memory

FIFO (most recently unprotected)

pages addresses

Reprotect 20

warmup page trace
### Test architectures

<table>
<thead>
<tr>
<th></th>
<th>Atom</th>
<th>Core 2</th>
<th>Nehalem</th>
<th>Sandy Bridge</th>
<th>Ivy Bridge</th>
<th>Haswell</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td>D510</td>
<td>E7500</td>
<td>L5609</td>
<td>E31240</td>
<td>i7-3770</td>
<td>i7-4770</td>
</tr>
<tr>
<td><strong>Frequency (GHz)</strong></td>
<td>1.66</td>
<td>2.93</td>
<td>1.86</td>
<td>3.30</td>
<td>3.40</td>
<td>3.40</td>
</tr>
<tr>
<td><strong>Cores</strong></td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td><strong>L1 cache (KB)</strong></td>
<td>2×56</td>
<td>2×64</td>
<td>4×64</td>
<td>4×64</td>
<td>4×64</td>
<td>4×64</td>
</tr>
<tr>
<td><strong>L2 cache (KB)</strong></td>
<td>2×512</td>
<td>3 MB</td>
<td>4×256</td>
<td>4×256</td>
<td>4×256</td>
<td>4×256</td>
</tr>
<tr>
<td><strong>L3 cache (MB)</strong></td>
<td>-</td>
<td>-</td>
<td>12</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td><strong>Ram (GB)</strong></td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>6</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>

32 bits portability test: ARM1176JZF-S on a Raspberry Pi Model B+
### Clustering NR Codelets

#### Codelet
- `toeplz_1`
- `rstrct_29`
- `mprove_8`
- `toeplz_4`
- `realft_4`
- `toeplz_3`
- `svbksb_3`
- `lop_13`
- `toeplz_2`
- `four1_2`
- `tridag_2`
- `tridag_1`
- `ludcmp_4`
- `hqr_15`
- `relax2_26`
- `svdcmp_14`
- `svdcmp_13`
- `hqr_13`
- `hqr_12_sq`
- `jacobi_5`
- `hqr_12`
- `svdcmp_11`
- `elmhes_11`
- `mprove_9`
- `matadd_16`
- `svdcmp_6`
- `elmhes_10`
- `balanc_3`

#### Computation Pattern
- **DP**: 2 simultaneous reductions
- **DP**: MG Laplacian fine to coarse mesh transition
- **MP**: Dense Matrix x vector product
- **DP**: Vector multiply in asc./desc. order
- **DP**: FFT butterfly computation
- **DP**: 3 simultaneous reductions
- **SP**: Dense Matrix x vector product
- **SP**: Laplacian finite difference constant coefficients
- **DP**: Vector multiply element wise in asc./desc. order
- **MP**: First step FFT
- **DP**: First order recurrence
- **DP**: First order recurrence
- **SP**: Dot product over lower half square matrix
- **SP**: Addition on the diagonal elements of a matrix
- **DP**: Red Black Sweeps Laplacian operator
- **DP**: Vector divide element wise
- **DP**: Norm + Vector divide
- **DP**: Sum of the absolute values of a matrix column
- **SP**: Sum of a square matrix
- **SP**: Sum of the upper half of a square matrix
- **SP**: Sum of the lower half of a square matrix
- **DP**: Multiplying a matrix row by a scalar
- **DP**: Linear combination of matrix rows
- **DP**: Substracting a vector with a vector
- **DP**: Sum of two square matrices element wise
- **DP**: Sum of the absolute values of a matrix row
- **DP**: Linear combination of matrix columns
- **DP**: Vector multiply element wise

---

**cut for K = 14**

---

27 / 16
Clustering NR Codelets

<table>
<thead>
<tr>
<th>Codelet</th>
<th>Computation Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>toeplz_1</td>
<td></td>
</tr>
<tr>
<td>rstrct_29</td>
<td></td>
</tr>
<tr>
<td>mprove_8</td>
<td></td>
</tr>
<tr>
<td>toeplz_4</td>
<td></td>
</tr>
<tr>
<td>realft_4</td>
<td></td>
</tr>
<tr>
<td>toeplz_3</td>
<td></td>
</tr>
<tr>
<td>svbkbsb_3</td>
<td></td>
</tr>
<tr>
<td>lop_13</td>
<td></td>
</tr>
<tr>
<td>toeplz_2</td>
<td></td>
</tr>
<tr>
<td>four1_2</td>
<td></td>
</tr>
<tr>
<td>tridag_2</td>
<td></td>
</tr>
<tr>
<td>tridag_1</td>
<td></td>
</tr>
<tr>
<td>ludcmp_4</td>
<td></td>
</tr>
<tr>
<td>hqr_15</td>
<td></td>
</tr>
<tr>
<td>relax2_26</td>
<td></td>
</tr>
<tr>
<td>svdcmp_14</td>
<td>DP: Vector divide element wise</td>
</tr>
<tr>
<td>svdcmp_13</td>
<td>DP: Norm + Vector divide</td>
</tr>
<tr>
<td>hqr_13</td>
<td>DP: Sum of the absolute values of a matrix column</td>
</tr>
<tr>
<td>hqr_12_sq</td>
<td>SP: Sum of a square matrix</td>
</tr>
<tr>
<td>jacobi_5</td>
<td>SP: Sum of the upper half of a square matrix</td>
</tr>
<tr>
<td>hqr_12</td>
<td>SP: Sum of the lower half of a square matrix</td>
</tr>
<tr>
<td>svdcmp_11</td>
<td></td>
</tr>
<tr>
<td>elmhes_11</td>
<td>Reduction Sums</td>
</tr>
<tr>
<td>mprove_9</td>
<td></td>
</tr>
<tr>
<td>matadd_16</td>
<td></td>
</tr>
<tr>
<td>svdcmp_6</td>
<td></td>
</tr>
<tr>
<td>elmhes_10</td>
<td></td>
</tr>
<tr>
<td>balanc_3</td>
<td></td>
</tr>
</tbody>
</table>

Similar computation patterns

Long Latency Operations

Reduction Sums
Capturing Architecture Change

**Nehalem (Ref)**
Freq: 1.86 GHz  
LLC: 12 MB

Cluster A: triple-nested high latency operations (div and exp)

LU/erhs.f : 49  
FT/appft.f : 45

BT/rhs.f : 266  
SP/rhs.f : 275

**Core 2**  
→ 2.93 GHz  
→ 3 MB

**Atom**  
→ 1.66 GHz  
→ 1 MB

Cluster B: stencil on five planes (memory bound)

Core 2
50 100 200 400  
+  

Atom
50 100 200 400 800 1500 3000  
+  

Reference  
Target
Same Cluster = Same Speedup

**Nehalem (Ref)**
Freq: 1.86 GHz
LLC: 12 MB

- LU/erhs.f: 49
- FT/appft.f: 45

**Cluster A:** triple-nested high latency operations (div and exp)

- BT/rhs.f: 266
- SP/rhs.f: 275

**Core 2**
→ 2.93 GHz
→ 3 MB

- LU/erhs.f
- FT/appft.f

**Atom**
→ 1.66 GHz
→ 1 MB

**Cluster B:** stencil on five planes (memory bound)
Feature Selection

- Genetic Algorithm: train on Numerical Recipes + Atom + Sandy Bridge
- Validated on NAS + Core 2
- The feature set is still among the best on NAS
Reduction Factor Breakdown

<table>
<thead>
<tr>
<th>Reduction</th>
<th>Total</th>
<th>Reduced invocations</th>
<th>Clustering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atom</td>
<td>44.3</td>
<td>×12</td>
<td>×3.7</td>
</tr>
<tr>
<td>Core 2</td>
<td>24.7</td>
<td>×8.7</td>
<td>×2.8</td>
</tr>
<tr>
<td>Sandy Bridge</td>
<td>22.5</td>
<td>×6.3</td>
<td>×3.6</td>
</tr>
</tbody>
</table>

Table: Benchmarking reduction factor breakdown with 18 representatives.
Profiling Features

Performance counters per codelet

Likwid

4 dynamic features

FLOPS
L2 Bandwidth
L3 Miss Rate
Mem Bandwidth

Maqao

8 static features

Bytes Stored / cycle
Stalls
Estimated IPC
Number of DIV
Number of SD
Pressure in P1
Ratio ADD+SUB/MUL
Vectorization (FP/FP+INT/INT)

Static disassembly and analysis
void main()
{
  #pragma omp parallel
  {
    int p = omp_get_thread_num();
    printf("%d",p);
  }
}

define i32 @main() {
  entry:
  ...
  call @_kmpc_fork_call @.omp_microtask.(...)
  ...
}

define internal void @.omp_microtask.(...) {
  entry:
    %p = alloca i32, align 4
    %call = call i32 @omp_get_thread_num()
    store i32 %call, i32* %p, align 4
    %1 = load i32* %p, align 4
    call @printf(%1)
}