Code transformations for energy efficiency; a decoupled access-execute approach

Work performed at Uppsala University

Konstantinos Koukos

November 2016
The big goal

- Better exploit the potential of the memory

- Exploit the energy savings potential of HW

Higher performance & better energy efficiency

Save energy without performance degradation

Outline

Transforming code to better exploit the memory system for energy & performance

- Decoupled access-execute (DAE) idea
  - Manual code transformation
  - Applied on task-based scientific codes
    

- DAE Compiler (LLVM passes)
  - For task-based scientific codes
    
    Jimborean et al. Fix the code. Don't tweak the hardware: A new compiler approach to Voltage–Frequency scaling, CGO-2014

- DAE Compiler for GP codes
  - Support GP codes and multi-versioning
    
    Koukos et al. Multiversioned decoupled access-execute: The key to energy-efficient compilation of general-purpose programs, CC-2016

*GP: General-Purpose
Motivation

Energy ≈ Time $C_{eff} f V^2$

- Power management $\rightarrow$ DVFS
  - Provides: Quadratic power improvement for at most linear performance slowdown
- Dennard scaling break down
  - Range of $V$ scaling is severely reduced
    - Because of leakage increase at low voltages
- DVFS $\rightarrow$ DFS
- To make DVFS more efficient:
  - Exploit the non-linear relationship of $f$ scaling to performance


*DVFS: Dynamic Voltage-Frequency Scaling*
Background

**Memory Bound**

<table>
<thead>
<tr>
<th>Maximum Frequency</th>
<th>Low Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>X Stall (MEM)</td>
<td></td>
</tr>
</tbody>
</table>

**Compute Bound**

<table>
<thead>
<tr>
<th>Maximum Frequency</th>
<th>Low Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Miss penalty

Energy waste

Real programs

\[ f_{\text{max}} \quad f_{\text{min}} \quad f_{\text{max}} \quad f_{\text{min}} \quad f_{\text{max}} \quad f_{\text{min}} \quad f_{\text{max}} \]

Which is the optimal DVFS-setting?

Impossible to DVFS this fast

How do we solve this problem?

Is this the best we can do?
Optimizing Execution - DAE

**Coupled Execution**

| $f_{\text{max}}$ | $f_{\text{min}}$ | $f_{\text{max}}$ | $f_{\text{min}}$ | $f_{\text{max}}$ | $f_{\text{min}}$ | $f_{\text{max}}$ |

**Decouple** the compute (**execute**) and memory (**access**)

- **Access**: prefetches the data to the L1 cache
- **Execute**: using the data from the L1 cache

**Access** at low frequency and **Execute** at high frequency

**Decoupled Execution (DAE)**
**Optimizing Execution - DAE**

**Coupled Execution**

- **Decouple** the compute (execute) and memory (access)
  - **Access**: prefetches the data to the L1 cache
  - **Execute**: using the data from the L1 cache
- **Access at low frequency** and **Execute at high frequency**

**Decoupled Execution (DAE)**

![Graph showing coupled and decoupled execution](image)
How do we implement DAE?

- Parallel workloads (scientific codes)
  - Utilize all cores to do useful work

- Task-based programming model. Why?
  - Schedule tasks independently
  - Control task size
  - Easy to convert to DAE!

- How? *Split* each task into *access* phase and *execute* phase
How do we implement DAE?

- Size the tasks to *fit in the* private cache (e.g., L1)

**Access phase: prefetches the data**
- Remove arithmetic computation
- Keep (replicate) address calculation
- Turn loads into prefetches: *improves MLP*
- Remove stores (no side effects)

**Execute phase: Original (unmodified) task**
- Scheduled to run on same core right after Access

**DVFS Access and Execute independently**

*MLP: Memory-Level Parallelism*
Evaluation Methodology

**Profile:**
- Time
- IPC

on real HW for each $f$

**Estimate Access / Execute Energy for each $f,V$**

**Power Model**

**Verify**

**Estimate Power Model Accuracy**

Now we can model the benefits of per-core DVFS on real hardware!

- **Performance**: Measured on real HW
- **Energy**: Modeled $\rightarrow$ verified on real HW
Understanding DAE (case-study)

**EVALUATION**

<table>
<thead>
<tr>
<th></th>
<th>Coupled</th>
<th>Decoupled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (sec)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Energy (Joule)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CG RunTime Profile -4 Threads-**

- Access Time
- Overhead Time
- Execute Time

**CG Energy Profile -4 Threads-**

- Access Energy
- Execute Energy

**Modeling zero-latency, per-core DVFS**

$\text{f}_{\text{min}} \rightarrow \text{f}_{\text{max}}$

Access: $\text{f}_{\text{min}}$

Execute: $\text{f}_{\text{min}} \rightarrow \text{f}_{\text{max}}$
Understanding DAE (case-study)

Performance: "unaffected"

Energy: 25% reduced

Slightly Faster and 25% Lower Energy
Evaluation: CAE vs DAE

Performance (normalized to original at $f_{\text{max}}$)

- No Slowdown
- Slightly faster
- Turning off HW prefetcher does not affect DAE
- 25% Lower EDP

Zero-latency, per-core DVFS

Significant EDP improvement at high performance

*EDP: Energy-Delay Product
$EDP = \text{Time} \times \text{Energy}$
Evaluation: Optimizing DAE

- For some applications we can choose better frequencies!
  - Access phase: Complex calculations
    - Slightly higher \( f \) than min is optimal
  - Execute phase: Too many / irregular writes
    - Slightly lower \( f \) than max is optimal

- Optimal $\rightarrow$ “OptEDP” policy
  - Use power and performance models [3] to predict \( f, V \) that has the optimal EDP and set this at runtime

Evaluation: OptEDP DAE

**Performance** (normalized to original at $f_{\text{max}}$)

- Slight performance decrease
- 29% Lower EDP

**Zero-latency, per-core DVFS**

Slight EDP improvement. Naïve approach is already performing very well!
Evaluation: Future trends

- Very promising to improve:
  - Performance
  - Energy efficiency

With DVFS overhead, per-core DVFS

% of Savings for a Given DVFS Transition Latency

ACPI - DVFS
Conclusions & Future Work

- **Without** fast, per-core DVFS:
  - DAE Improves the MLP $\rightarrow$ performance of memory-bound applications and therefore, their EDP.

- **With** fast, per-core DVFS:
  - DAE provides significant Energy / EDP improvement without performance degradation.

Can we automatically transform codes to DAE? …
DAE Compiler

Compiler → automatically generates the access-phase

Automatically generated access-phase must be:

- **Lean** (not too many instructions)
- **Accurate** (prefetch the right data)
  - Derive the access patterns from the execute code and optimize their prefetch

- 1. **Affine**
  → polyhedral model
- 2. **Non-affine**
  → skeleton with optimizations

*Jimborean et al. Fix the code. Don't tweak the hardware: A new compiler approach to Voltage–Frequency scaling, CGO-2014*
Affine-Code Handling in DAE

```c
for ( i = 0; i < N; i++)
    for ( j = i+1; j < N; j++){
        A[j][i]2 /= A[i][i]1;
        for ( k = i+1; k < N; k++)
    }

Access Phase

Solution: Polyhedral Model [4,5]

for ( i = 0; i < N; i++)
    for ( j = 0; j < N; j++)
        prefetch A[i][j];

Should prefetch each element once
```

Non-Affine-Code Handling in DAE

1. Clone the original task
2. Keep only instructions involved in:
   - Memory address computation
   - The control-flow graph (CFG)
3. Replace load with prefetch instructions
   - Discard store instructions
4. Apply heuristics → lightweight access-phase
Non-Affine-Code Handling in DAE

**Execute:**

```c
for ( i = 0; i < N; i++)
    for ( j = i+1; j < N; j++){
        if (cond)
            A[j][i] += A[i][j];
        A[i][j] -= N;
    }
```

**Access:**

```c
for ( i = 0; i < N; i++)
    for ( j = i+1; j < N; j++)
        prefetch A[i][j];
```

**Problem:**
Complex CFG → heavy access phase.

**Solution:**
Prefetch only the addresses that are guaranteed to be accessed.

**Improvements:**
profile & prefetch the hot path.
Non-Affine-Code Handling in DAE

**Execute:**

```c
for ( i = 0; i < N; i++)
    A[B[i]] -= ...;
```

**Access:**

```c
for ( i = 0; i < N; i++){
    prefetch B[i];
    x = load B[i];
    prefetch A[x];
}
```

**Problem:**
Effective address involve intermediate memory accesses.

**Solution:**
Prefetch only the last indirection.

**Improvements:**
Paper III
Evaluation: DAE Compiler

**Runtime** (normalized to original at $f_{\text{max}}$)

<table>
<thead>
<tr>
<th></th>
<th>LU</th>
<th>Chol.</th>
<th>FFT</th>
<th>LBM</th>
<th>LibQ</th>
<th>Cigar</th>
<th>CG</th>
<th>G.Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAE</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>M-DAE</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>C-DAE</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

**EDP** (normalized to original at $f_{\text{max}}$)

<table>
<thead>
<tr>
<th></th>
<th>LU</th>
<th>Chol.</th>
<th>FFT</th>
<th>LBM</th>
<th>LibQ</th>
<th>Cigar</th>
<th>CG</th>
<th>G.Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAE</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>M-DAE</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>C-DAE</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

**500ns latency, per-core DVFS**

**Automatic DAE provides close to optimal EDP of 25% with less than 4% performance degradation!**

CAE: 15% Per. degradation
M-DAE: <4% Perf. degradation
C-DAE: <4% Perf. degradation

CAE: 26% EDP improvement
M-DAE: 22% EDP improvement
C-DAE: 25% EDP improvement

KTH Royal Institute of Technology
Introduction

Scientific codes
- task-based, parallel

- Affine
- Non-affine

1. Polyhedral
2. Heuristics

→ It is not suitable for general-purpose codes!

Koukos et al. Multiversioned decoupled access-execute: The key to energy-efficient compilation of general-purpose programs, CC-2016
Motivation

Code example (soplex SPEC CPU2006)

```c
for(int i = thedim - 1; i >= 0; i--) {
    int r = row.orig[i];
    int c = col.orig[i];
    Real x = wrk[c] = diag[r] * vec[r];

    vec[r] = 0.0;

    if (x != 0.0) {
        for(int j = u.col.start[c]; j < u.col.start[c] + u.col.len[c]; j++)
            vec[u.col.idx[j]] -= x * u.col.val[j];
    }
}
```

Which accesses to prefetch? Is address computation an overkill or we see benefits?
The *indirection-level* is a metric for heavy-/light-weight *access-phase* generation.

\[
x = a [* b + c -> d [* e ]];
\]

\[
t1 = \text{load } b
t2 = \text{gep } c, 0, \text{index_of_d}
t3 = \text{load } t2
t4 = \text{load } e
t5 = \text{gep } t3, 0, t4
t6 = \text{load } t5
t7 = \text{add } t1, t6
t8 = \text{gep } a, 0, t7
x = \text{load } t8
\]

The *indirection level* of an address A is given by the **number of loads** required to compute A.
Software Multi-Versioning

GP-code example (soplex SPEC CPU2006)

```c
for(int i = thedim - 1; i >= 0; i--) {
    int r = row.orig[i];
    int c = col.orig[i];
    Real x = wrk[c] = diag[r] * vec[r];

    vec[r] = 0.0;

    if (x != 0.0) {
        for(int j = u.col.start[c]; j < u.col.start[c] + u.col.len[c]; j++)
            vec[u.col.idx[j]] -= x * u.col.val[j];
    }
}
```

Initially we generate all the prefetches for the requested indirection level.

For each indirection-level we generate a different access-phase.

Superfluous code removal maintains only the effective ones (not already a load).

Indirection Level: 1
for(int i = thedim - 1; i >= 0; i--) {
    int r = row.orig[i];
    int c = col.orig[i];
    Real x = wrk[c] = diag[r] * vec[r];

    vec[r] = 0.0;

    if (x != 0.0) {
        for(int j = u.col.start[c]; j < u.col.start[c] + u.col.len[c]; j++)
            vec[u.col.idx[j]] -= x * u.col.val[j];
    }
}
Software Multi-Versioning

GP-code example (soplex SPEC CPU2006)

```c
for(int i = thedim - 1; i >= 0; i--) {
    int r = row.orig[i];
    int c = col.orig[i];
    Real x = wrk[c] = diag[r] * vec[r];

    vec[r] = 0.0;

    if (x != 0.0) {
        for(int j = u.col.start[c]; j < u.col.start[c] + u.col.len[c]; j++)
            vec[u.col.idx[j]] -= x * u.col.val[j];
    }
}
```

For each indirection-level we generate a different access-phase

Indirection Level: 3
Software Multi-Versioning

GP-code example (soplex SPEC CPU2006)

```c
for(int i = thedim - 1; i >= 0; i--) {
    int r = row.orig[i];
    int c = col.orig[i];
    Real x = wrk[c] = diag[r] * vec[r];

    vec[r] = 0.0;

    if (x != 0.0) {
        for(int j = u.col.start[c]; j < u.col.start[c] + u.col.len[c]; j++)
            vec[u.col.idx[j]] -= x * u.col.val[j];
    }
}
```

For each indirection-level we generate a different access-phase

Indirection Level: 4
Execution overview

Loop execution in slices of \textit{granularity}=10

Compiler. Generates all different access-versions.

Runtime (e.g., [6, 7]). Evaluates each pair of Access-Execute \textbf{at runtime} and \textbf{select} the \textbf{best performing}.

\textbf{SMV-DAE Benefits:}

1. Allows us to optimize statically-unknown overheads.
2. The best performing pair might be different under different system loads (e.g., when sharing LLC and memory bandwidth).

Evaluation

Assuming low-latency, per-core DVFS

- 5% lower execution time
- 25% lower EDP
Evaluation

Huge diversity in energy savings potential (12%-32%)

Same application different system load → different access-version
DAE Recap / Conclusions

- DAE improves the MLP
  - Performance of memory bound applications
- Is a novel execution paradigm to adjust the SW to the DVFS capabilities of the HW
  - Good energy/EDP efficiency for nodes with low-latency, per-core DVFS
- DAE improves the performance and energy efficiency of complex-codes that will remain in the CPU in the heterogeneous era (SMVDAE).
Questions

Thank You!
Towards more efficient execution: a decoupled access-execute approach

Koukos, Konstantinos
Black-Schaffer, David
Spiliopoulos, Vasileios
Kaxiras, Stefanos


Full text can be downloaded from:
UU (author-preprint): http://uu.diva-portal.org
ACM digital library: http://dl.acm.org

Figures and images in slides 4—17 are used under the ACM – author license agreement for educational (non-profitable) purposes.
Slides 18–23 are part of:

*Fix the code. Don't tweak the hardware: A new compiler approach to Voltage–Frequency scaling*

Jimborean, Alexandra
Koukos, Konstantinos
Spiliopoulos, Vasileios
Black-Schaffer, David
Kaxiras, Stefanos


Full text can be downloaded from:
ACM digital library: [http://dl.acm.org](http://dl.acm.org)

Figures and images in slides 18–23 are used under the ACM – author license agreement for educational (non-profitable) purposes.
Slides 24 – 34 are part of:

Multiversioned decoupled access-execute: The key to energy-efficient compilation of general-purpose programs

Koukos, Konstantinos
Ekemark, Per
Zacharopoulos, Georgios
Spiliopoulos, Vasileios
Kaxiras, Stefanos
Jimborean, Alexandra


Full text can be downloaded from:
UU (author-preprint): http://uu.diva-portal.org
ACM digital library: http://dl.acm.org

Figures and images in slides 24—34 are used under the ACM – author license agreement for educational (non-profitable) purposes.