Towards a compiler/runtime synergy to predict the scalability of parallel loops

*Georgios Chatzopoulos[1], Kornilios Kourtis[2], Nectarios Koziris[1], and **Georgios Goumas**[1]*

1 School of Electrical and Computer Engineering
National Technical University of Athens, Greece
{gchatz,nkoziris,goumas}@cslab.ece.ntua.gr

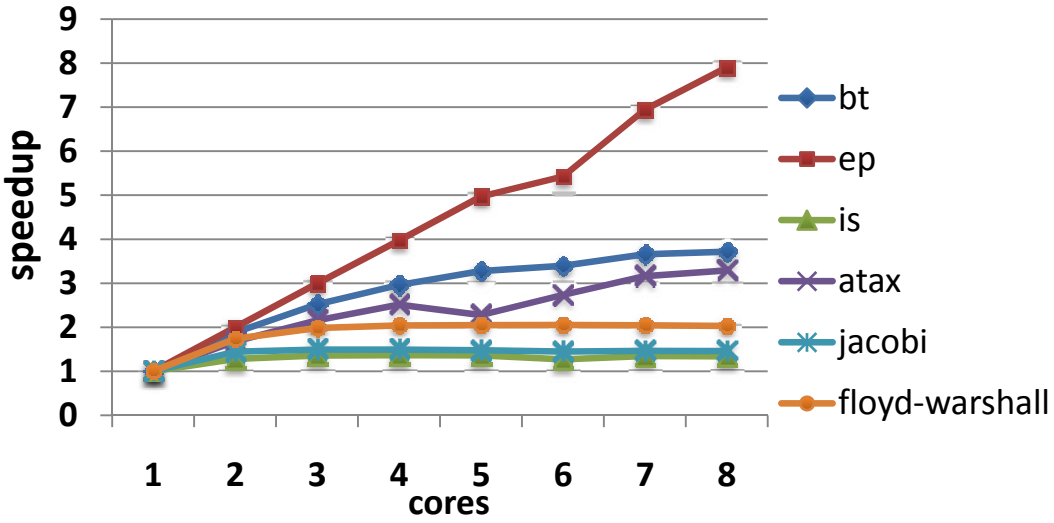
2 Department of Computer Science
ETH, Zurich, Switzerland
kkourt@inf.ethz.ch

Motivation for this work

- Large classes of multithreaded applications do not scale well in CMPs due to memory bandwidth limitations

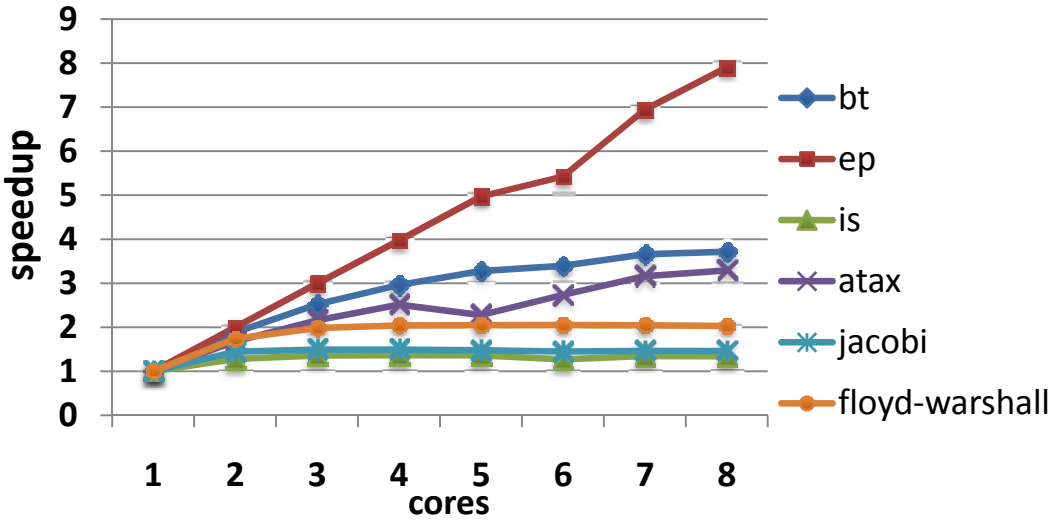
Motivation for this work

- Large classes of multithreaded applications do not scale well in CMPs due to memory bandwidth limitations
- **bt**, **ep**, **is** taken from NAS parallel benchmarks, **atax**, **jacobi**, **floyd-warshall** taken from Polybench (platform: 8-core Sandy Bridge)



Motivation for this work

- Large classes of multithreaded applications do not scale well in CMPs due to memory bandwidth limitations
- **bt**, **ep**, **is** taken from NAS parallel benchmarks, **atax**, **jacobi**, **floyd-warshall** taken from Polybench (platform: 8-core Sandy Bridge)



- Huge fights with memory-bandwidth limitations for SpMV (=Sparse Matrix-Vector multiplication) in our previous works, e.g. [KourtisPPoPP11], [KarakasisTPDS13]

Motivation for this work

- What about emerging workloads for CMPs?

Motivation for this work

- What about emerging workloads for CMPs?
- Quoted from C. Bienia, S. Kumar, J. P. Singh and K. Li. **The PARSEC Benchmark Suite: Characterization and Architectural Implications**, *Technical Report TR-811-08, Princeton University, January 2008:*

"Since many PARSEC workloads have high bandwidth requirements and working sets which exceed conventional caches by far, ***off-chip bandwidth will be their most severe limitation of performance****."*

Motivation for this work

Why do we really care?

- Non-scalable applications running on the full system are a clear resource waste. Badly utilized cores could alternatively be:
  - Switched off to save energy
  - Assigned to another application
- We need resource-aware systems for CMPs, e.g.
  - Resource/power/contention-aware **schedulers**
  - **Compilation frameworks** and **runtime systems**
- Key component of such a system is a **resource utilization predictor**
  - Capable of predicting with acceptable accuracy the resource utilization of an application
  - In our case: “best” number of cores (in terms of speedup, efficiency, EDP, etc)
  - In this paper we take one step back and try to predict maximum scalability

Defining the problem

- **Platforms:** CMPs with Uniform Memory Access (no NUMA)
- **Programming constructs:** Parallel loops
- **Objective:** Predict the **scalability**, i.e. **maximum speedup** of each parallel loop region of an application.
  - Deciding upon the **best number of cores** is left for future work
- **Disclaimer:** We do not work on scalability limitations due to:
  - serial parts of the code
    - these reside outside the parallel regions we work on
  - load imbalance
    - loops are considered balanced
  - task/thread synchronization overhead
    - no synchronization occurs in the loop constructs of our model
    - task/thread orchestration is considered of minimal overhead

Defining the problem

Parallel loop 1

```
#pragma omp parallel for
for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
      C[i] = C[i] + A[i][j] * B[j];
```

Parallel loop 2

```
#pragma omp parallel for
for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
      C[i] = C[i] + A[j][i] * B[j];
```

Defining the problem

Parallel loop 1

```
#pragma omp parallel for
for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
      C[i] = C[i] + A[i][j] * B[j];
```

Parallel loop 2

```
#pragma omp parallel for
for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
      C[i] = C[i] + A[j][i] * B[j];
```

Any bets on the scalability of these loops?
Good, fair or bad scalability?
Any differences between them?

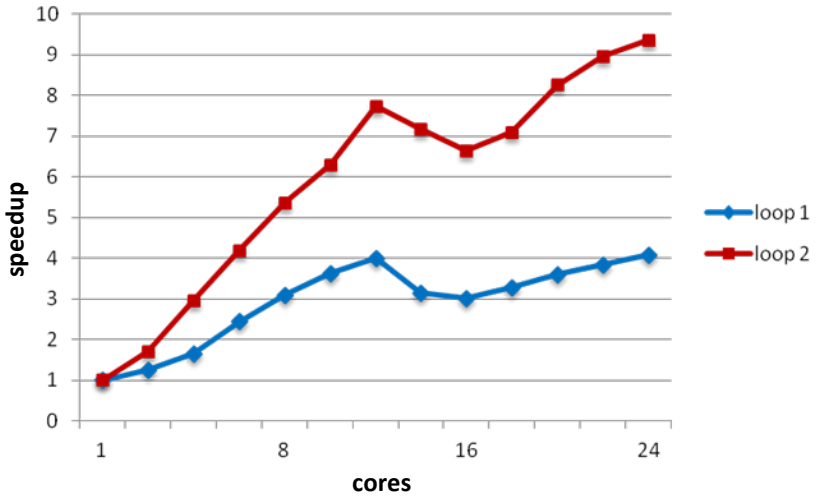
Defining the problem

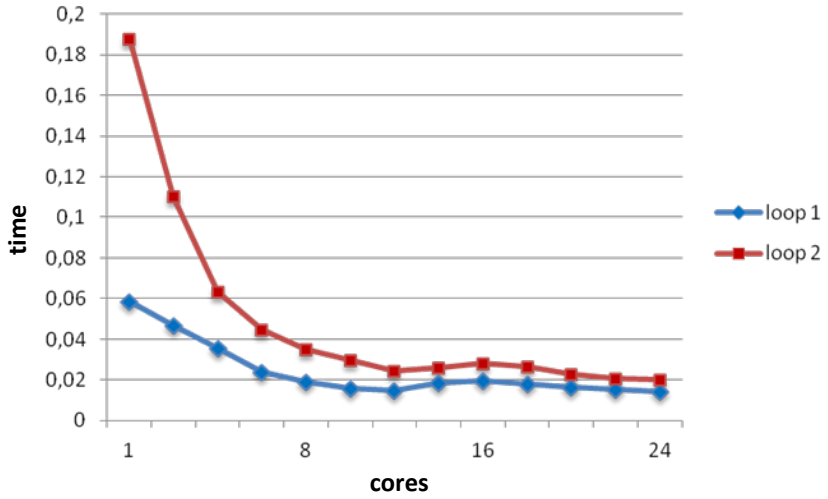
Parallel loop 1

```
#pragma omp parallel for
for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
      C[i] = C[i] + A[i][j] * B[j];
```

Parallel loop 2

```
#pragma omp parallel for
for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
      C[i] = C[i] + A[j][i] * B[j];
```





24-core Intel Dunnington SMP

Core idea

What we learn from the roofline prediction model [williamsComACM2009]

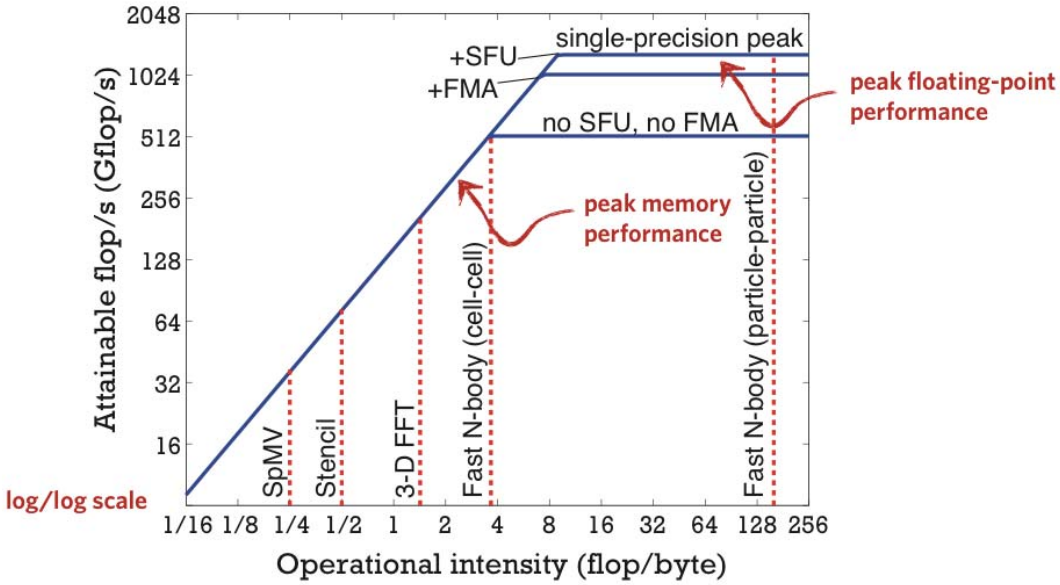


Image taken from http://lorenabarba.com/news/new-paper-published-cise-journal/

- Key metric for the performance of memory-bound applications: **flop/byte ratio**
- Applications with:
  - High flop/byte ratio perform close to machine peak
  - Low flop/byte ratio are memory-bound

Core idea

Scalability of multithreaded applications

- **Rule of thumb:** Parallel loops with activity inside the chip are expected to scale well (put small pressure on the scarce memory link)
- We need a way to **quantify** on-chip (good) and off-chip (bad) activity
- Try to collect as much information as possible at **compile time** to characterize the loop bodies
- Augment this with **runtime** information to get the full picture of the execution profile
- We do not care to be extremely precise, actually we need to **classify** loop regions **as bad performing, fair performing or good performing**
- Promote **simplicity** (sacrificing accuracy and applicability) in order to validate the potential of the idea

A synergistic scalability predictor

In a nutshell

- **Offline:** train a prediction model based on a quantification of on-chip and off-chip traffic
- **At compile-time:**
  - Classify operations as arithmetic or memory
  - Count and **score** arithmetic operations
    - Different scores are assigned for addition/multiplication/division
  - Classify memory references as **streaming** or **latency** (explained later on)
  - Calculate **reuse distance** of memory references as a function of data structure size (unknown at compile time)
  - Conditionally classify memory references as hits (on-chip traffic) or misses (off-chip traffic)
- **At runtime:**
  - Classify memory references as on-chip or off-chip taking into account the data structure size known at runtime
  - Finalize on-chip and off-chip scores
  - Apply the scores to the model and get the prediction

Algorithmic model

What we handle

```
for (i0 = L0; i0 < U0; i0+=c0) do
  CS1
  for (i1 = L1; i1 < U1; i1+=c1) do
    CS2
    for (in = Ln; in < Un; in+=cn) do
       CSn
    …
  …
```

Algorithmic model

What we handle

```
for (i0 = L0; i0 < U0; i0+=c0) do
  CS1
  for (i1 = L1; i1 < U1; i1+=c1) do
    CS2
    for (in = Ln; in < Un; in+=cn) do
       CSn
    …
  …
```

- Imperfectly nested loops with bounds in canonical form (e.g. as required by the OpenMP standard)

Algorithmic model

What we handle

```
for (i0 = L0; i0 < U0; i0+=c0) do
  CS1
  for (i1 = L1; i1 < U1; i1+=c1) do
    CS2
    for (in = Ln; in < Un; in+=cn) do
      CSn
    …
  …
```

- Imperfectly nested loops with bounds in canonical form (e.g. as required by the OpenMP standard)
- Loop bodies are compound statements that can include:
  - other loop nests (recursive definition)
  - arithmetic operations and memory references on regular data structures

Algorithmic model

What we handle

```
for (i0 = L0; i0 < U0; i0+=c0) do
  CS1
  for (i1 = L1; i1 < U1; i1+=c1) do
    CS2
    for (in = Ln; in < Un; in+=cn) do
       CSn
    …
  …
```

- Imperfectly nested loops with bounds in canonical form (e.g. as required by the OpenMP standard)
- Loop bodies are compound statements that can include:
  - other loop nests (recursive definition)
  - arithmetic operations and memory references on regular data structures
- Any **single** loop in the nest can be parallelized
- Legality of parallelization and insertion of parallelization primitives are left to the programmer

Predictor details

On-chip to off-chip ratio (S_r)

- We need to quantify on-chip and off-chip activity
- On-chip (good) activity is expressed by score S_g as:
  - $S_g = \sum_{o \in O_{on}} n_o \cdot w_o$
  - where n_o is the number of operations of type and w_o the operation's weight
- Off-chip (bad) activity is expressed in the same way by score S_b as:

$$S_b = \sum_{o \in O_{off}} n_o \cdot w_o$$

- Key predictor metric: on-chip to off-chip ratio $\boldsymbol{S_r = S_g / S_b}$

Predictor details

On-chip to off-chip ratio (S_r)

- We need to quantify on-chip and off-chip activity
- On-chip (good) activity is expressed by score S_g as:
  - $S_g = \sum_{o \in O_{on}} n_o \cdot w_o$
  - where n_o is the number of operations of type and w_o the operation's weight
- Off-chip (bad) activity is expressed in the same way by score S_b as:

$$S_b = \sum_{o \in O_{off}} n_o \cdot w_o$$

- Key predictor metric: on-chip to off-chip ratio $\boldsymbol{S_r = S_g / S_b}$

operation	weight
addition/substraction	1
multiplication	2
division	4
cache hit	3

on-chip operations and their weights

	streaming	latency
read	2	1
write	3	1

off-chip operations and their weights

for Dunnington and Sandy Bridge

Predictor details

Classification of memory accesses

- Calculate the **reuse distance** (number of iterations before the element is accessed again) for each array reference
- If the reuse distance is smaller than the aggregate Last Level Cache (LLC) size
  - The reference is considered a hit
- Else
  - The reference is considered a miss with probability: $p = \begin{cases} 0, ws \le cs \\ \frac{ws - cs}{ws} \end{cases}$

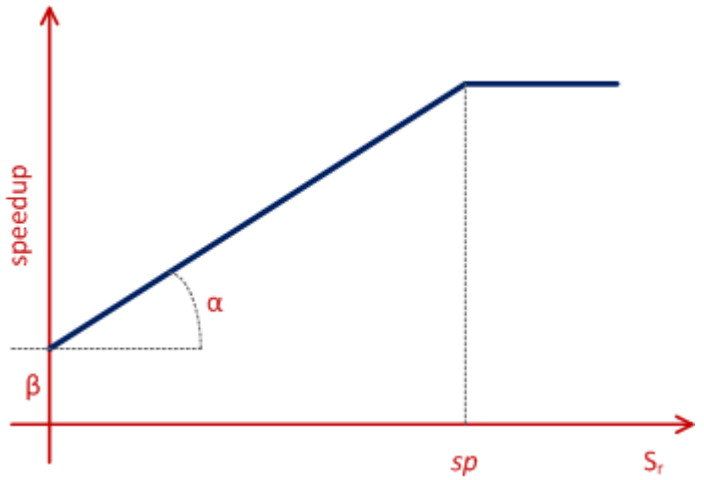
    where *ws* is the working set size and *cs* the LLC size
  - In this case we calculate the expected value of the operation's weight, i.e. we multiply the probability with the weight
- If the fastest changing dimension of the array is indexed using the innermost loop index, we characterize a miss as **streaming**, otherwise we characterize it as a **latency** miss.

Predictor details

Prediction approach

- Speedup (σ) is considered a piecewise function of S_r as follows:

$$\sigma(S_r) = \begin{cases} a \cdot S_r + \beta, S_r \leq sp \\ \sigma_{\max}, S_r > sp \end{cases}$$



- α, β and sp are platform specific
- we use a training set to find sp and calculate α and β with linear regression

Example

Polybench adi benchmark, line 72

```
for (t = 0; t < _PB_TSTEPS; t++) {
#pragma omp parallel for
  for (i1 = 0; i1 < _PB_N; i1++)
    for (i2 = 1; i2 < _PB_N; i2++) {
      X[i1][i2] = X[i1][i2] - X[i1][i2-1]
                  * A[i1][i2] / B[i1][i2-1];
      B[i1][i2] = B[i1][i2] – A[i1][i2]
                  * A[i1][i2] / B[i1][i2-1];
    }
}
```

Example

Polybench adi benchmark, line 72

```
for (t = 0; t < _PB_TSTEPS; t++) {
#pragma omp parallel for
  for (i1 = 0; i1 < _PB_N; i1++)
    for (i2 = 1; i2 < _PB_N; i2++) {
      X[i1][i2] = X[i1][i2] - X[i1][i2-1]
                  * A[i1][i2] / B[i1][i2-1];
      B[i1][i2] = B[i1][i2] – A[i1][i2]
                  * A[i1][i2] / B[i1][i2-1];
    }
}
```

Assume the code is executed:

- On a machine with 64MiB LLC size (*cs* = 64 * 1024 * 1024)
- With data type double (8 bytes)
- With _PB_N = 4000 (known at runtime)

Thus:

- Working set size is *ws* = 3 * 8 * 4000 * 4000
  (arrays, doubles, array dimension)
- miss probability $p_{miss} = (ws - cs) / ws = 0.825$
- hit probability $p_{hit} = 1 - p_{miss} = 0.175$

Example

Polybench adi benchmark, line 72

```
for (t = 0; t < _PB_TSTEPS; t++) {
#pragma omp parallel for
  for (i1 = 0; i1 < _PB_N; i1++)
    for (i2 = 1; i2 < _PB_N; i2++) {
      X[i1][i2] = X[i1][i2] - X[i1][i2-1]
                  * A[i1][i2] / B[i1][i2-1];
      B[i1][i2] = B[i1][i2] – A[i1][i2]
                  * A[i1][i2] / B[i1][i2-1];
    }
}
```

Work on first statement:

Assume the code is executed:

- On a machine with 64MiB LLC size (*cs* = 64 * 1024 * 1024)
- With data type double (8 bytes)
- With _PB_N = 4000 (known at runtime)

Thus:

- Working set size is *ws* = 3 * 8 * 4000 * 4000
  (arrays, doubles, array dimension)
- miss probability $p_{miss} = (ws - cs) / ws = 0.825$
- hit probability $p_{hit} = 1 - p_{miss} = 0.175$

Example

Polybench adi benchmark, line 72

```
for (t = 0; t < _PB_TSTEPS; t++) {
#pragma omp parallel for
  for (i1 = 0; i1 < _PB_N; i1++)
    for (i2 = 1; i2 < _PB_N; i2++) {
      X[i1][i2] = X[i1][i2] - X[i1][i2-1]
                * A[i1][i2] / B[i1][i2-1];
      B[i1][i2] = B[i1][i2] – A[i1][i2]
                * A[i1][i2] / B[i1][i2-1];
    }
}
```

Work on first statement:

1 substraction (weight = 1)

1 mulplication (weight = 2)

1 division (weight = 4)

$S_{g1} = 1 + 2 + 4 + \ldots$ (on chip memory accesses)

Assume the code is executed:

- On a machine with 64MiB LLC size (*cs* = 64 * 1024 * 1024)
- With data type double (8 bytes)
- With _PB_N = 4000 (known at runtime)

Thus:

- Working set size is *ws* = 3 (arrays) * 8 (doubles) * 4000 * 4000 (array dimension)
- miss probability $p_{miss} = (ws - cs) / ws = 0.825$
- hit probability $p_{hit} = 1 - p_{miss} = 0.175$

Example

Polybench adi benchmark, line 72

```
for (t = 0; t < _PB_TSTEPS; t++) {
#pragma omp parallel for
  for (i1 = 0; i1 < _PB_N; i1++)
    for (i2 = 1; i2 < _PB_N; i2++) {
      X[i1][i2] = X[i1][i2] - X[i1][i2-1]
                  * A[i1][i2] / B[i1][i2-1];
      B[i1][i2] = B[i1][i2] – A[i1][i2]
                  * A[i1][i2] / B[i1][i2-1];
    }
}
```

Assume the code is executed:

- On a machine with 64MiB LLC size (*cs* = 64 * 1024 * 1024)
- With data type double (8 bytes)
- With _PB_N = 4000 (known at runtime)

Thus:

- Working set size is *ws* = 3 * 8 * 4000 * 4000
  (arrays, doubles, array dimension)
- miss probability $p_{miss} = (ws - cs) / ws = 0.825$
- hit probability $p_{hit} = 1 - p_{miss} = 0.175$

Work on first statement:

1 substraction (weight = 1)

1 mulplication (weight = 2)

1 division (weight = 4)

`X[i1][i2-1]` is a hit since reuse distance from `X[i1][i2]` = 1 (weight = 3)

S_{g1} = 7 + 3 + (rest of on-chip memory references)

Example

Polybench adi benchmark, line 72

```
for (t = 0; t < _PB_TSTEPS; t++) {
#pragma omp parallel for
  for (i1 = 0; i1 < _PB_N; i1++)
    for (i2 = 1; i2 < _PB_N; i2++) {
      X[i1][i2] = X[i1][i2] - X[i1][i2-1]
                  * A[i1][i2] / B[i1][i2-1];
      B[i1][i2] = B[i1][i2] – A[i1][i2]
                  * A[i1][i2] / B[i1][i2-1];
    }
}
```

Assume the code is executed:

- On a machine with 64MiB LLC size (*cs* = 64 * 1024 * 1024)
- With data type double (8 bytes)
- With _PB_N = 4000 (known at runtime)

Thus:

- Working set size is *ws* = 3 (arrays) * 8 (doubles) * 4000 * 4000 (array dimension)
- miss probability $p_{miss} = (ws - cs) / ws = 0.825$
- hit probability $p_{hit} = 1 - p_{miss} = 0.175$

Work on first statement:

1 substraction (weight = 1)

1 mulplication (weight = 2)

1 division (weight = 4)

`X[i1][i2-1]` is a hit since reuse distance from `X[i1][i2]`= 1 (weight = 3)

`X[i1][i2]` (twice), `A[i1][i2]` and **`B[i1][i2-1]`** are read hits with probability 0.175

$S_{g1} = 10 + 4 * 3 * 0.175 = 12.1$

(4: references, 3: weight, 0.175: probability)

Example

Polybench adi benchmark, line 72

```
for (t = 0; t < _PB_TSTEPS; t++) {
#pragma omp parallel for
  for (i1 = 0; i1 < _PB_N; i1++)
    for (i2 = 1; i2 < _PB_N; i2++) {
      X[i1][i2] = X[i1][i2] - X[i1][i2-1]
                  * A[i1][i2] / B[i1][i2-1];
      B[i1][i2] = B[i1][i2] – A[i1][i2]
                  * A[i1][i2] / B[i1][i2-1];
    }
}
```

Assume the code is executed:

- On a machine with 64MiB LLC size (*cs* = 64 * 1024 * 1024)
- With data type double (8 bytes)
- With _PB_N = 4000 (known at runtime)

Thus:

- Working set size is *ws* = 3 (arrays) * 8 (doubles) * 4000 * 4000 (array dimension)
- miss probability $p_{miss} = (ws - cs) / ws = 0.825$
- hit probability $p_{hit} = 1 - p_{miss} = 0.175$

Work on first statement:

$S_{g1} = 12.1$

X[i1][i2], A[i1][i2] and **B[i1][i2-1]** are **streaming read misses** with probability 0.825

$S_{b1} = 3$ (references) $*$ 2 (weight) $*$ 0.825 (probability) $= 4.95$ + (write misses)

Example

Polybench adi benchmark, line 72

```
for (t = 0; t < _PB_TSTEPS; t++) {
#pragma omp parallel for
  for (i1 = 0; i1 < _PB_N; i1++)
    for (i2 = 1; i2 < _PB_N; i2++) {
      X[i1][i2] = X[i1][i2] - X[i1][i2-1]
                  * A[i1][i2] / B[i1][i2-1];
      B[i1][i2] = B[i1][i2] – A[i1][i2]
                  * A[i1][i2] / B[i1][i2-1];
    }
}
```

Assume the code is executed:

- On a machine with 64MiB LLC size (*cs* = 64 * 1024 * 1024)
- With data type double (8 bytes)
- With _PB_N = 4000 (known at runtime)

Thus:

- Working set size is *ws* = 3 * 8 * 4000 * 4000
  (arrays, doubles, array dimension)
- miss probability $p_{miss} = (ws - cs) / ws = 0.825$
- hit probability $p_{hit} = 1 - p_{miss} = 0.175$

Work on first statement:

S_{g1} = **12.1**

`X[i1][i2], A[i1][i2]` and `B[i1][i2-1]` are **streaming read misses** with probability 0.825

`X[i1][i2]` is a **streaming write miss** with probability 0.825

S_{b1} = 4.95 + 3 * 0.825 = **7.425**
(weight, probability)

Example

Polybench adi benchmark, line 72

```
for (t = 0; t < _PB_TSTEPS; t++) {
#pragma omp parallel for
  for (i1 = 0; i1 < _PB_N; i1++)
    for (i2 = 1; i2 < _PB_N; i2++) {
      X[i1][i2] = X[i1][i2] - X[i1][i2-1]
                  * A[i1][i2] / B[i1][i2-1];
      B[i1][i2] = B[i1][i2] – A[i1][i2]
                  * A[i1][i2] / B[i1][i2-1];
    }
}
```

Assume the code is executed:

- On a machine with 64MiB LLC size (*cs* = 64 * 1024 * 1024)
- With data type double (8 bytes)
- With _PB_N = 4000 (known at runtime)

Thus:

- Working set size is *ws* = 3 * 8 * 4000 * 4000
  (arrays, doubles, array dimension)
- miss probability $p_{miss} = (ws - cs) / ws = 0.825$
- hit probability $p_{hit} = 1 - p_{miss} = 0.175$

Work on second statement:

In an similar way we get:

S_{g2} = **19.52**

S_{b2} = **2.47**

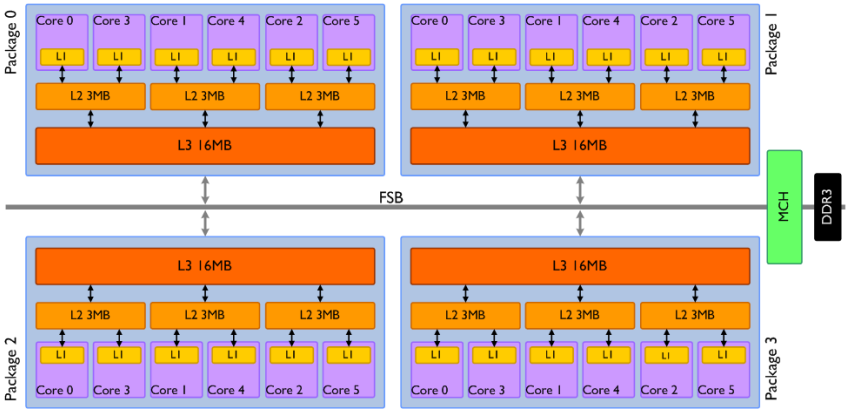
Note that in this case references in the rhs are all hits due to reuse from previous statement

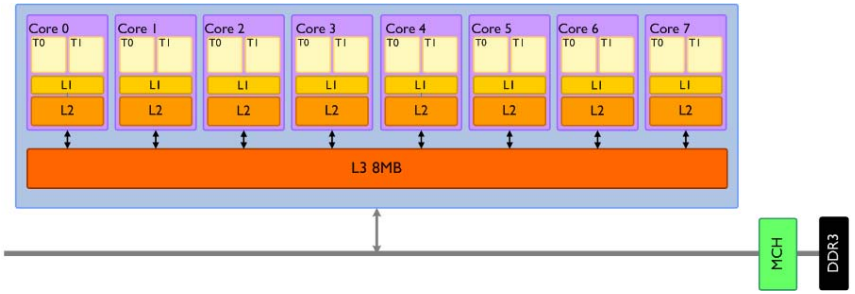
Overall for the loop:

$S_r = (S_{g1} + S_{g2}) / (S_{b1} + S_{b2}) = 3.19$

Experimental evaluation

Setup: architectures





24-core Dunnington
4 packages, 6 cores per package
16MB LLC per package, 64Mb aggregate
Linux kernel 3.7.10
gcc version 4.6.3
programs compiled with –O3

8-core Sandy Bridge
1 package, 8 cores per package
8MB LLC
Linux kernel 3.7.10
gcc version 4.6.3
programs compiled with –O3

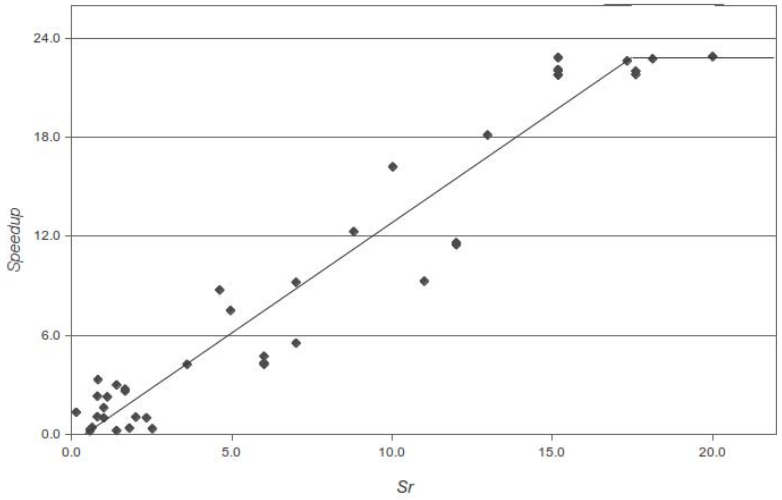
Experimental evaluation

Setup: benchmarks and data sets

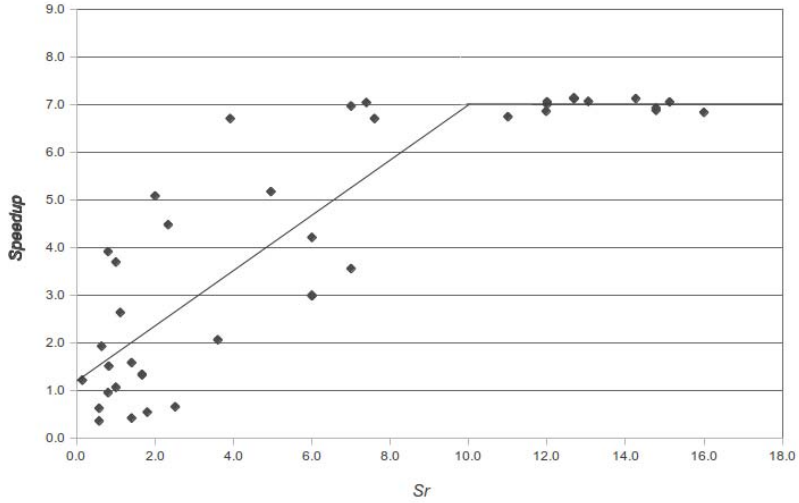
- Loops collected from the Polybench suite:
  http://www.cse.ohio-state.edu/~pouchet/software/polybench/
- Initially 60 loops:
  - Discarded 9 with trivial workloads (e.g. initialization loops)
  - 5 loops violated the restrictions of our model
  - 6 loops could not be parallelized
  - Finally worked on 40 loops
    - 11 as training set
    - 29 as testing set
- Two data sets per architecture (small / large)
  - Small: almost equal to the size of the (aggregate) LLC
  - Large: 2-5 times larger than the LCC (depending on the arch and benchmark)

Experimental evaluation

Correlation of S_r with Speedup



Dunnington

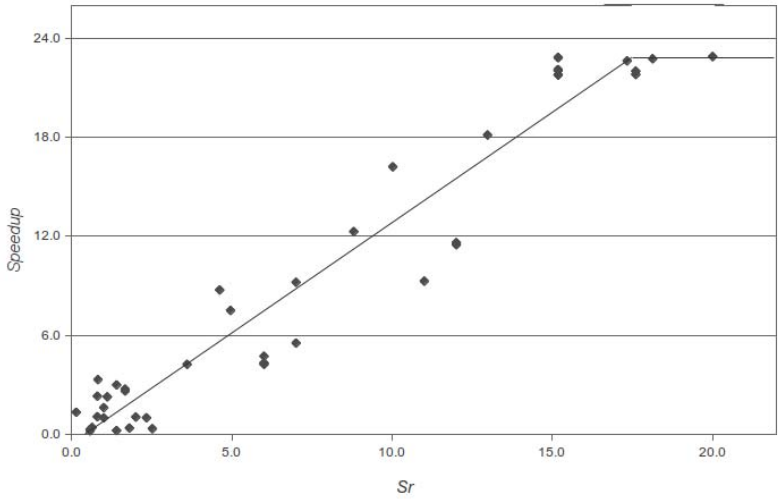


Sandy Bridge

National Technical University of Athens
CSLab

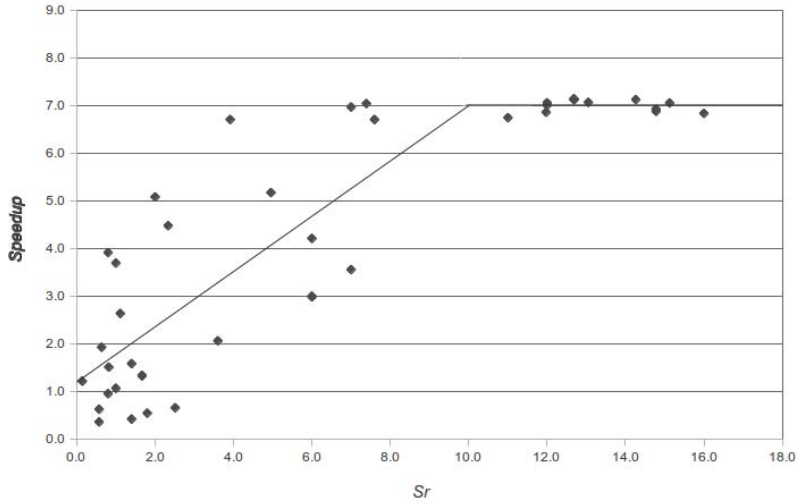
Experimental evaluation

Correlation of S_r with Speedup



Dunnington

95.7% correlation



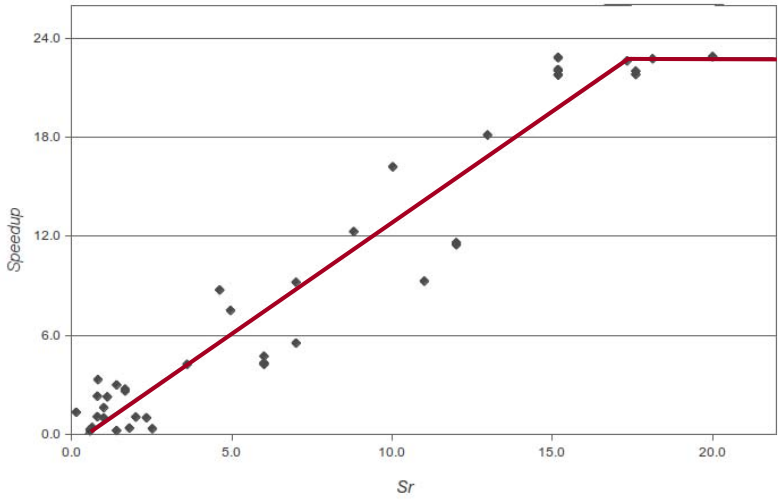
Sandy Bridge

84.1% correlation

National Technical University of Athens
CSLab

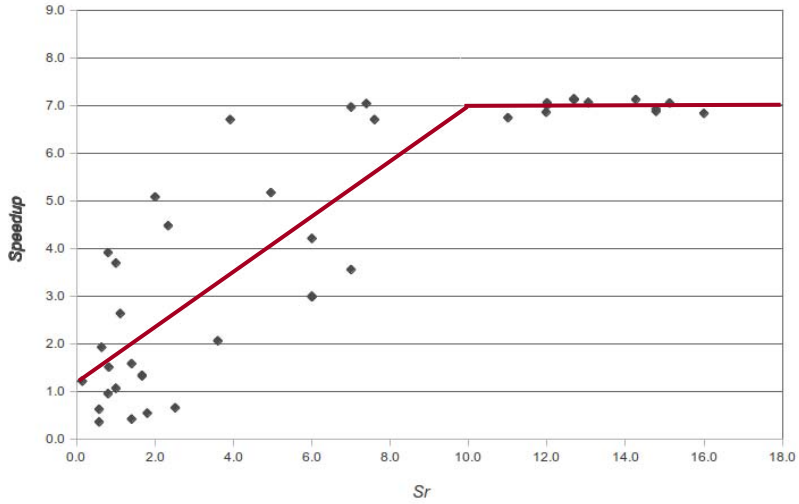
Experimental evaluation

Correlation of S_r with Speedup



Dunnington

95.7% correlation



Sandy Bridge

84.1% correlation

National Technical University of Athens
CSLab

Experimental evaluation

Prediction accuracy

We use as prediction error the metric:

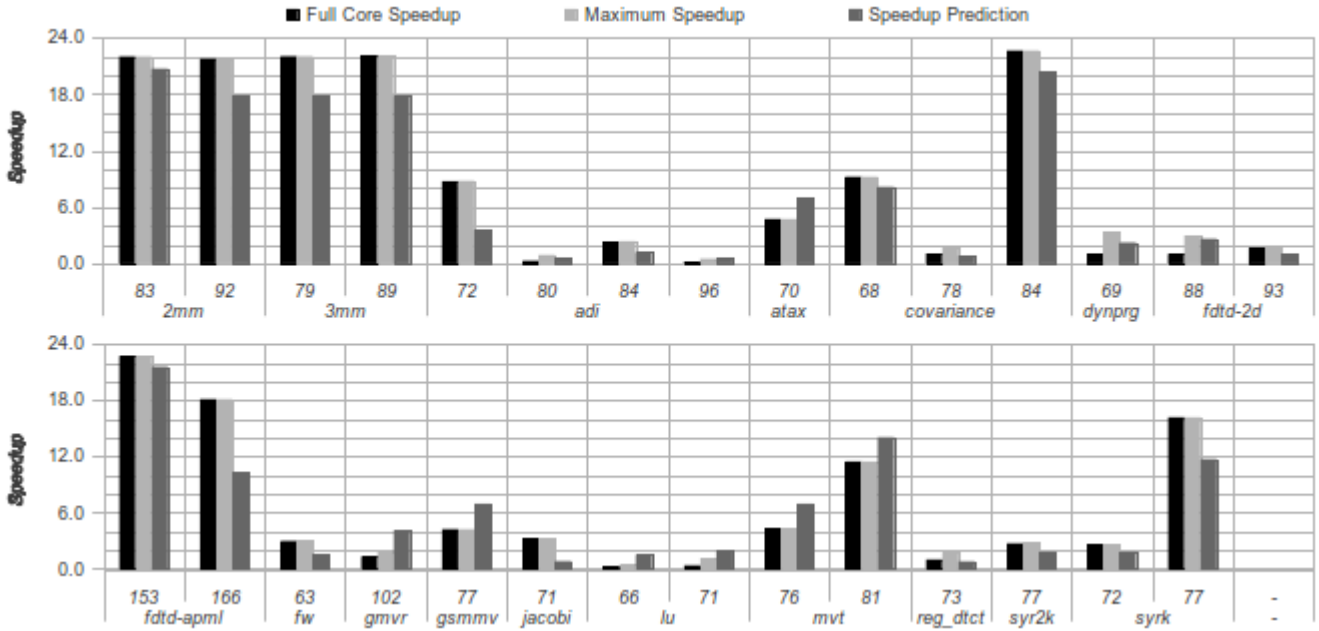
$$\frac{\left| predicted\ speedup - \max speedup \right|}{number\ of\ cores}$$

expressed in %

We argue that a prediction error <25% is acceptable for a scalability predictor as it would successfully classify a loop as bad, fair or good performing

Experimental evaluation

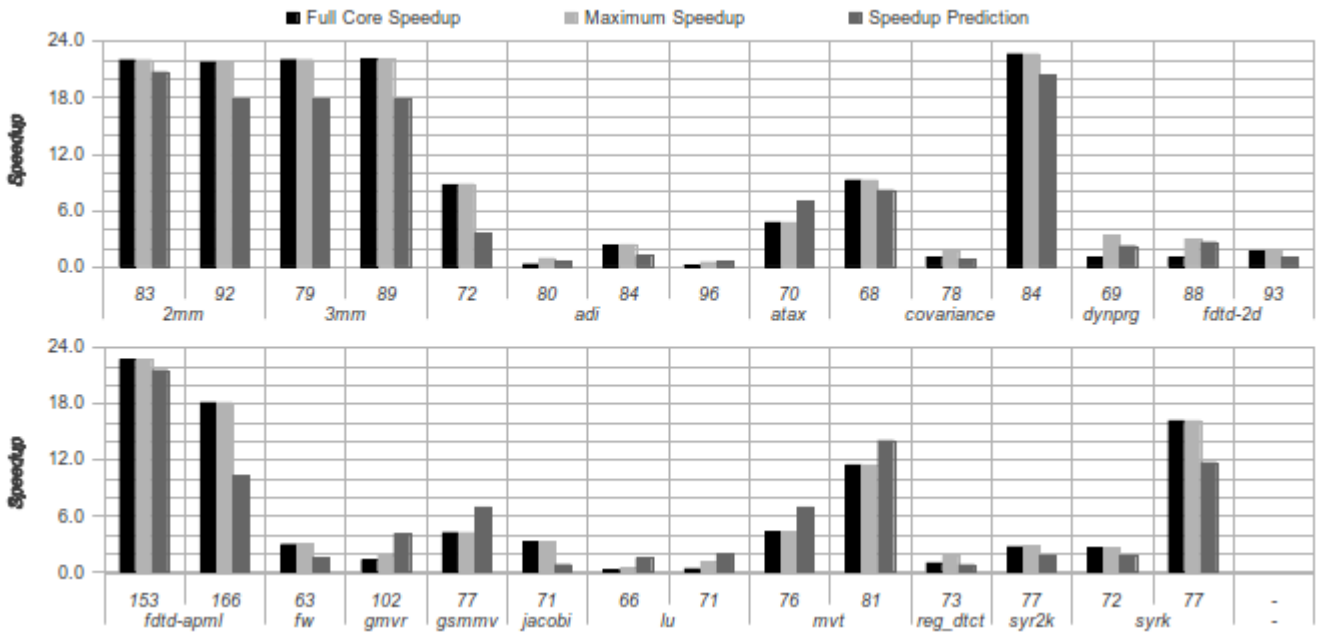
Prediction accuracy: large data set / Dunnington



National Technical University of Athens CSLab

Experimental evaluation

Prediction accuracy: large data set / Dunnington



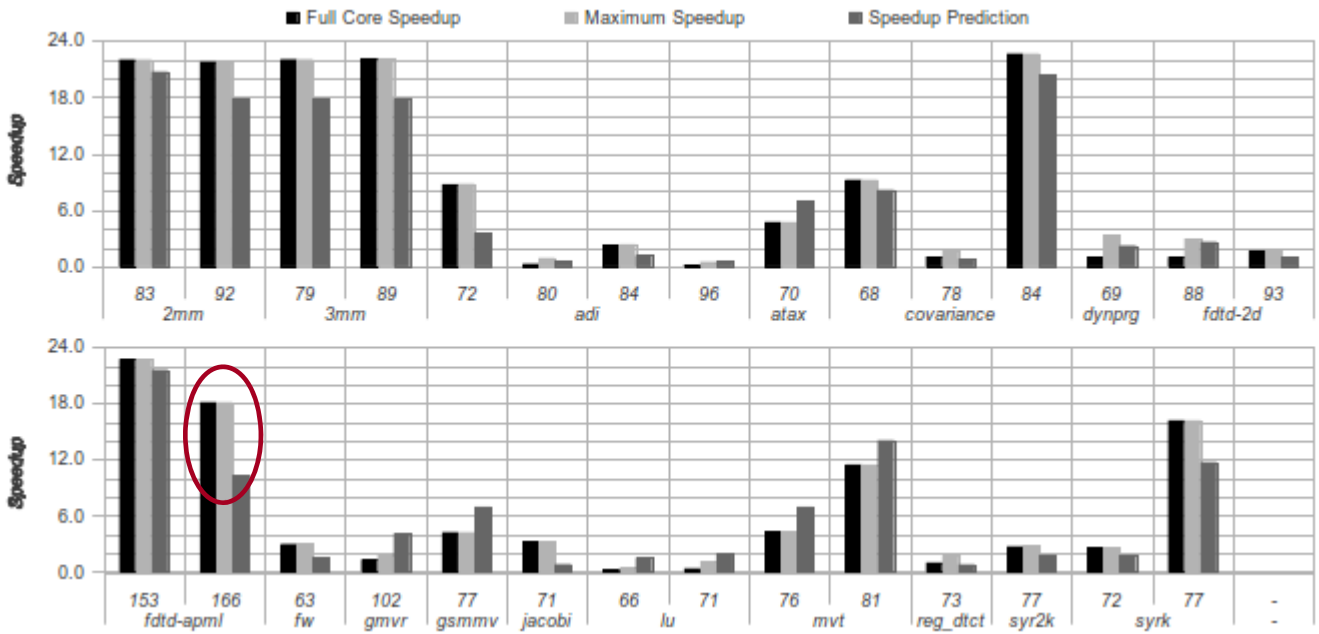
average prediction error: 8.8%

maximum prediction error: 32.7%

National Technical University of Athens CSLab

Experimental evaluation

Prediction accuracy: large data set / Dunnington



prediction error < 25%

28 loops

prediction error > 25%

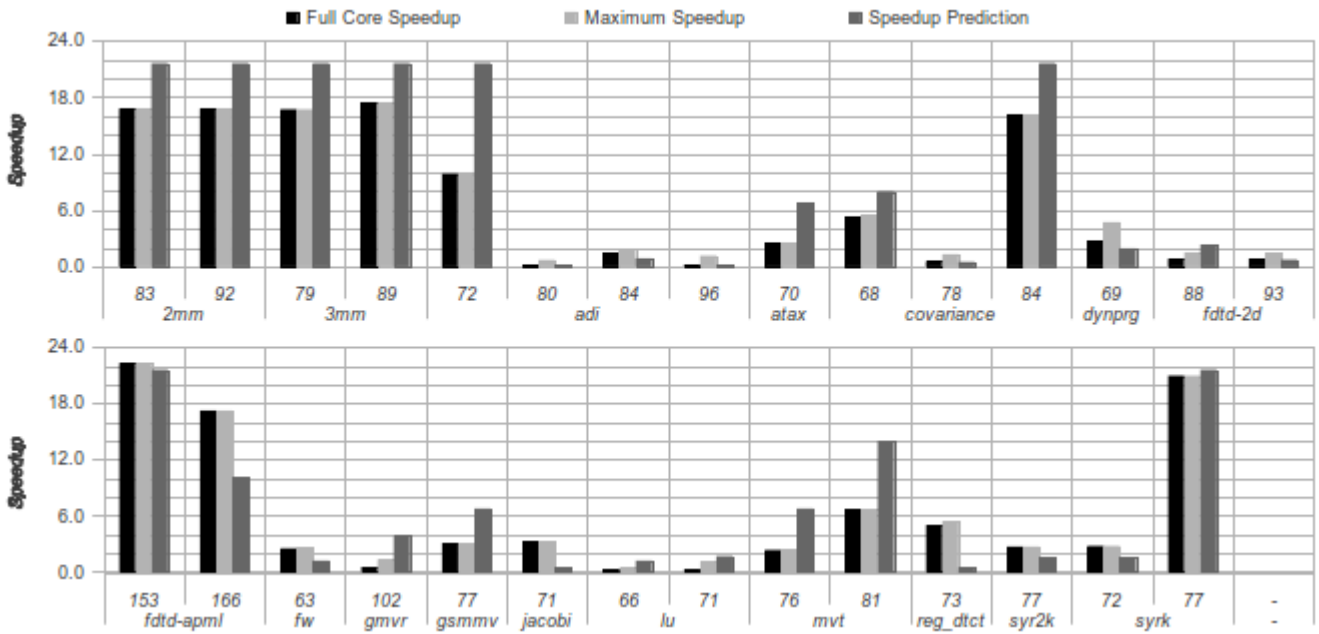
1 loop

average prediction error: 8.8%

maximum prediction error: 32.7%

Experimental evaluation

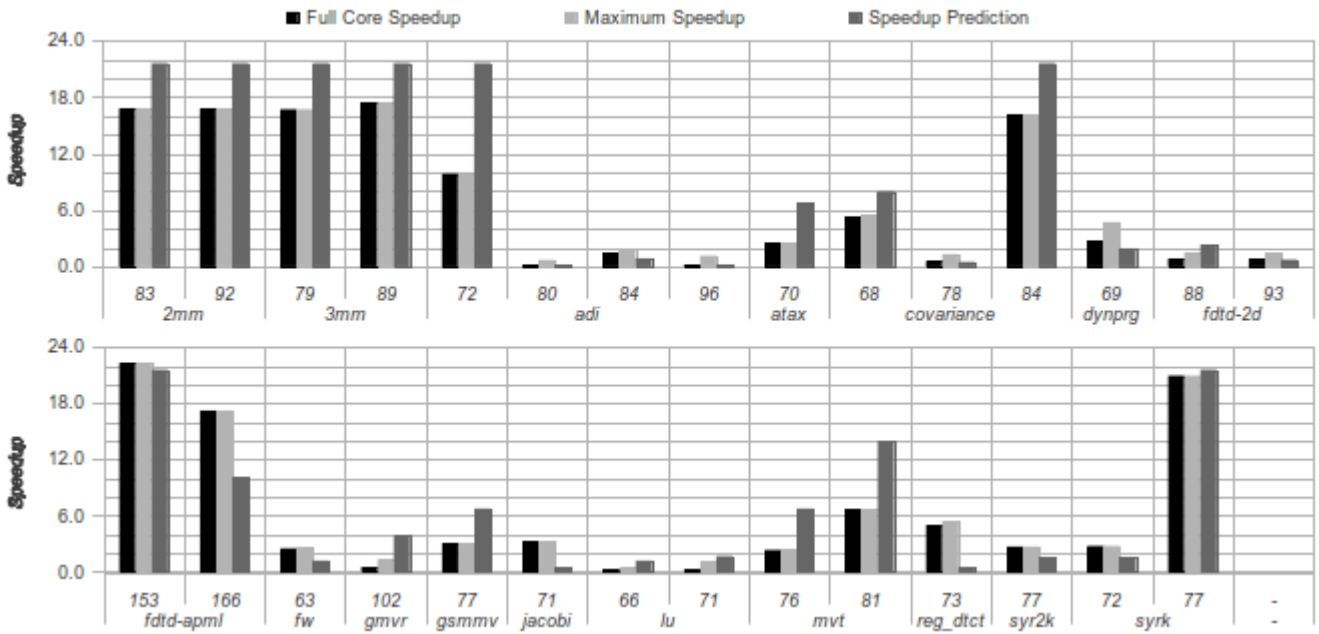
Prediction accuracy: small data set / Dunnington



National Technical University of Athens CSLab

Experimental evaluation

Prediction accuracy: small data set / Dunnington



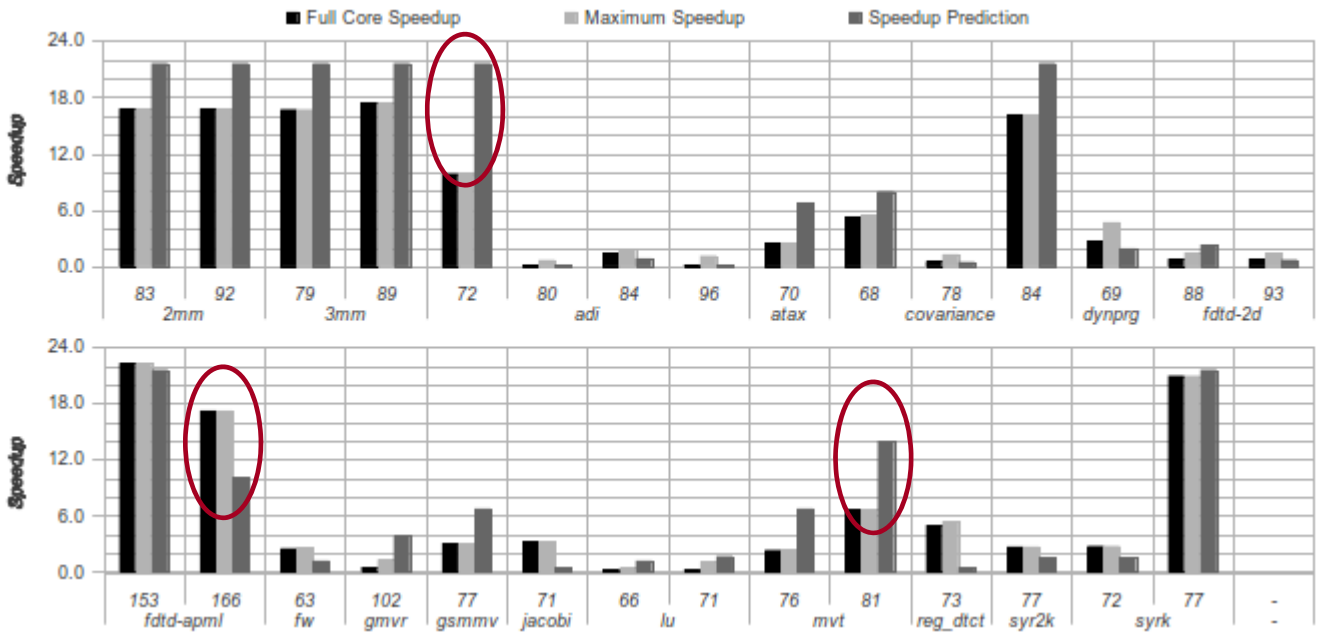
average prediction error: 12.8%

maximum prediction error: 48.7%

National Technical University of Athens CSLab

Experimental evaluation

Prediction accuracy: small data set / Dunnington



prediction error < 25%

26 loops

prediction error > 25%

3 loops

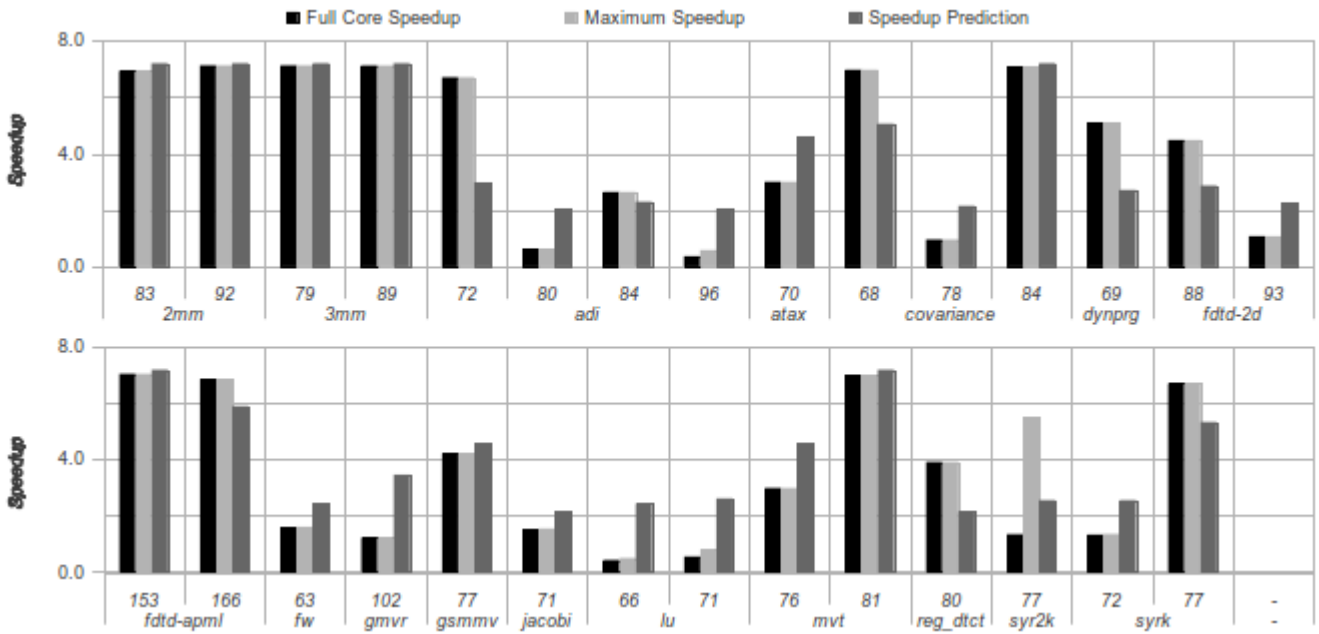
average prediction error: 12.8%

maximum prediction error: 48.7%

National Technical University of Athens CSLab

Experimental evaluation

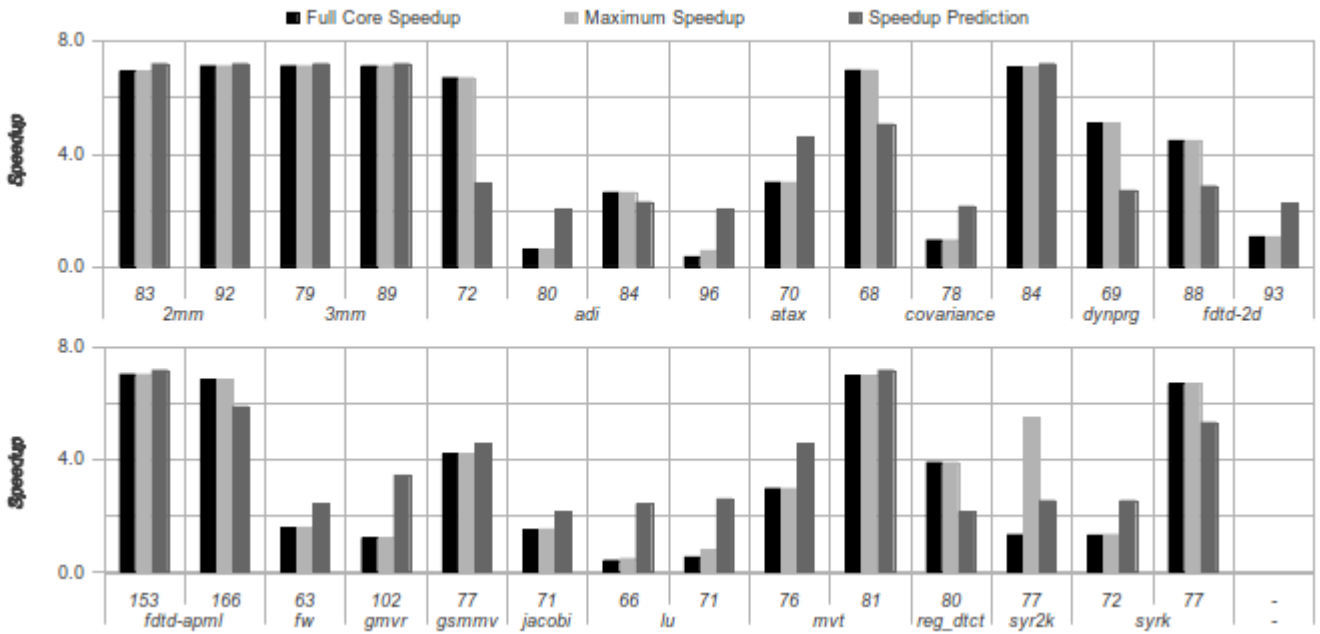
Prediction accuracy: large data set / Sandy Bridge



National Technical University of Athens CSLab

Experimental evaluation

Prediction accuracy: large data set / Sandy Bridge



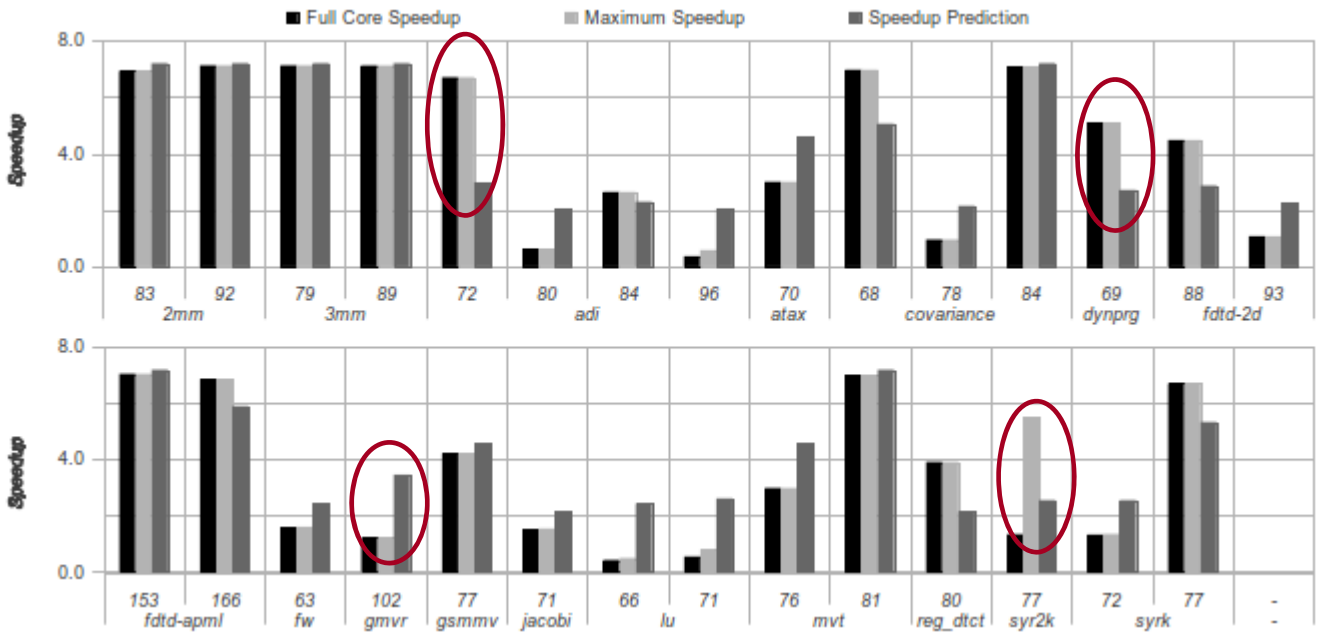
average prediction error: 15.28%

maximum prediction error: 46.66%

National Technical University of Athens CSLab

Experimental evaluation

Prediction accuracy: large data set / Sandy Bridge



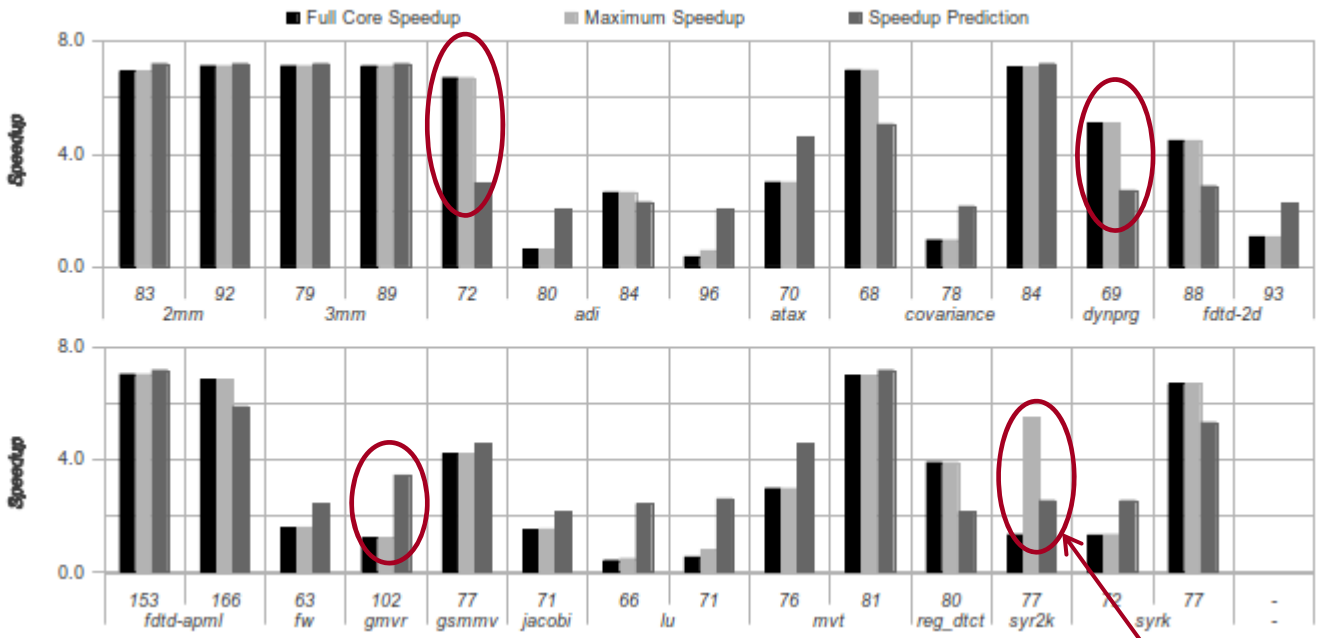
prediction error < 25%
25 loops

prediction error > 25%
4 loops

average prediction error: 15.28%
maximum prediction error: 46.66%

Experimental evaluation

Prediction accuracy: large data set / Sandy Bridge



prediction error < 25%

25 loops

prediction error > 25%

4 loops

average prediction error: 15.28%

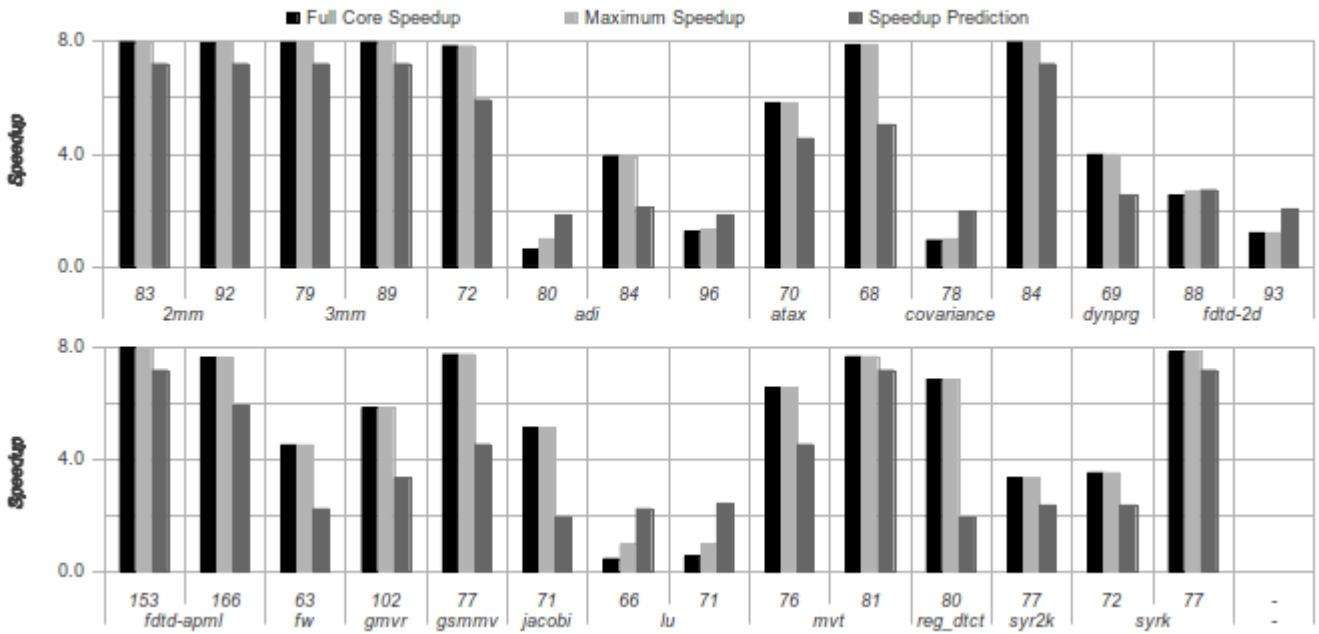
maximum prediction error: 46.66%

speedup prediction is much closer to "full core speedup" in this case

National Technical University of Athens CSLab

Experimental evaluation

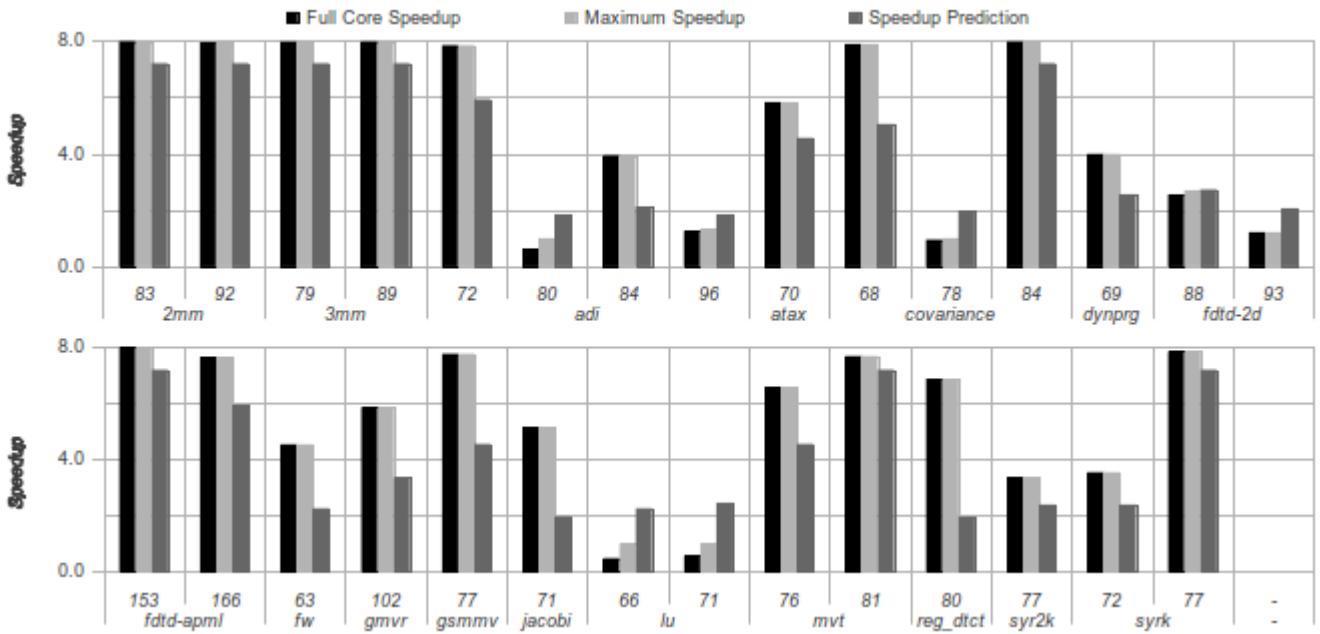
Prediction accuracy: small data set / Sandy Bridge



National Technical University of Athens CSLab

Experimental evaluation

Prediction accuracy: small data set / Sandy Bridge

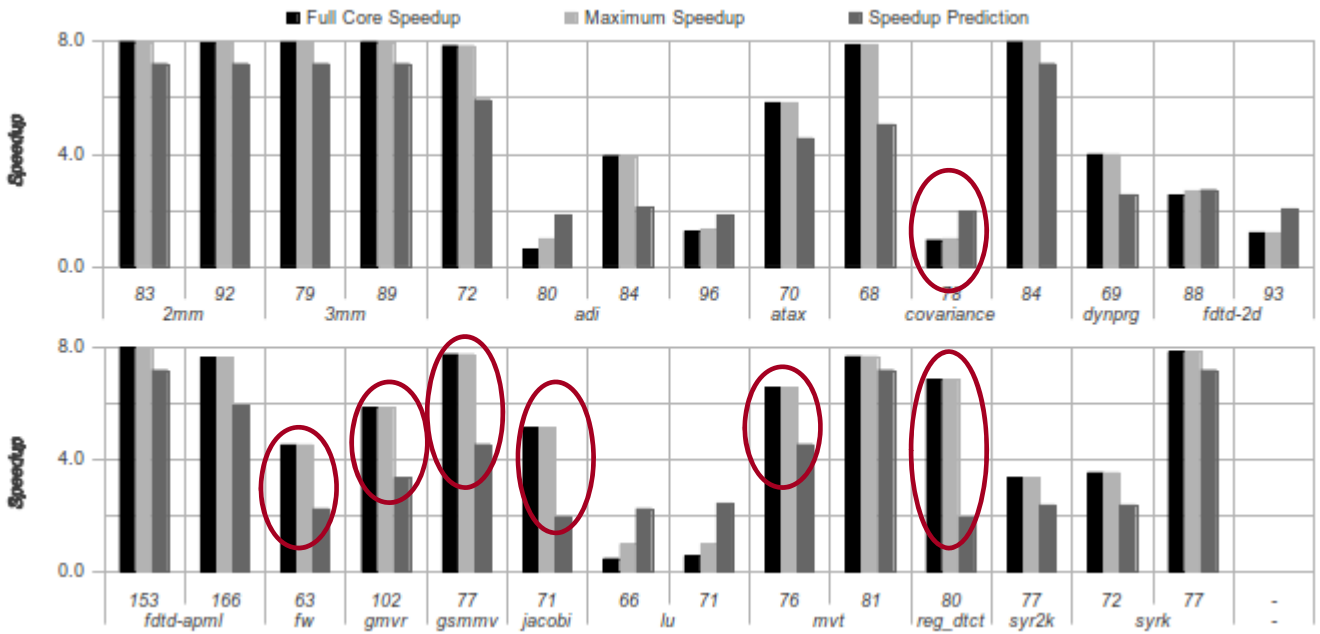


average prediction error: 18.48%

maximum prediction error: 61.5%

Experimental evaluation

Prediction accuracy: small data set / Sandy Bridge



prediction error < 25%

22 loops

prediction error > 25%

7 loops

average prediction error: 18.48%

maximum prediction error: 61.5%

National Technical University of Athens CSLab

Experimental evaluation

Overall remarks

- S_r shows a good correlation with scalability
- Scalability predictor exhibits acceptable accuracy
  - In 116 experiments (39 loops * 2 archs * 2 data sets) 101 predictions had an error <25%
- Our predictor worked better in memory constrained situations (e.g. Dunnington machine and large data set)

Current status, limitations and future work

- Currently, approach is applied by hand
  - Encouraging results urge us to implement a tool
- Limitations in algorithmic model:
  - Do not handle branches and function calls in loop body. Certainly requires more attention
  - Synchronization: approach can be extended to deal with scalability limitations due to synchronization (maybe in an orthogonal way)
- Future work:
  - Enrich hardware model to capture more architectures (e.g. NUMA)
  - Improve prediction accuracy (elaborate more on subtle issues)
  - Predict “best” number of cores
  - Automate parameter selection (e.g. weights) with microbenchmarks

Conclusions

- We took a first step towards predicting an efficient core assignment of multi-threaded applications
- We embraced a simple approach to verify the potential of our method
- We applied a synergistic prediction approach involving compile and runtime submodules
- We base our prediction on the quantification of on-chip and off-chip traffic
- Experimental results on ~30 loops are encouraging regarding the ability of our model to predict bad, fair or good scalability

Questions?

Backup slides

Algorithmic model

What we do not handle

```
for (i0 = L0; i0 < U0; i0+=c0) do
  CS1
  for (i1 = L1; i1 < U1; i1+=c1) do
    CS2
    for (in = Ln; in < Un; in+=cn) do
       CSn
    …
  …
```

Algorithmic model

What we do not handle

```
for (i0 = L0; i0 < U0; i0+=c0) do
  CS1
  for (i1 = L1; i1 < U1; i1+=c1) do
    CS2
    for (in = Ln; in < Un; in+=cn) do
       CSn
    …
  …
```

- Loop bodies do not include conditional or unconditional jumps, function calls, return statements, etc.

Algorithmic model

What we do not handle

```
for (i0 = L0; i0 < U0; i0+=c0) do
  CS1
  for (i1 = L1; i1 < U1; i1+=c1) do
    CS2
    for (in = Ln; in < Un; in+=cn) do
       CSn
    …
  …
```

- Loop bodies do not include conditional or unconditional jumps, function calls, return statements, etc.

- *Much more difficult to handle*
- *Occurs in several cases*
- *Left for future work*

Algorithmic model

What we do not handle

```
for (i0 = L0; i0 < U0; i0+=c0) do
  CS1
  for (i1 = L1; i1 < U1; i1+=c1) do
    CS2
    for (in = Ln; in < Un; in+=cn) do
       CSn
    …
  …
```

- Loop bodies do not include conditional or unconditional jumps, function calls, return statements, etc.
- Loop bodies are free of synchronization primitives (e.g. critical sections)

Algorithmic model

What we do not handle

```
for (i0 = L0; i0 < U0; i0+=c0) do
  CS1
  for (i1 = L1; i1 < U1; i1+=c1) do
    CS2
    for (in = Ln; in < Un; in+=cn) do
      CSn
    …
  …
```

- Loop bodies do not include conditional or unconditional jumps, function calls, return statements, etc.
- Loop bodies are free of synchronization primitives (e.g. critical sections)

- *Current focus is on scalability restrictions due to **memory bandwidth limitations***
- *Future work can augment predictor to handle scalability issues due to synchronization overhead (most probably in an orthogonal way)*

Algorithmic model

What we do not handle

```
for (i0 = L0; i0 < U0; i0+=c0) do
  CS1
  for (i1 = L1; i1 < U1; i1+=c1) do
    CS2
    for (in = Ln; in < Un; in+=cn) do
       CSn
    …
  …
```

- *More difficult to handle*
- *Occurs infrequently in regular, parallel-for loop nests*

- Loop bodies do not include conditional or unconditional jumps, function calls, return statements, etc.
- Loop bodies are free of synchronization primitives (e.g. critical sections)
- A single loop in the nest is parallelized
  - No recursive parallelism

Algorithmic model

What we do not handle

```
for (i0 = L0; i0 < U0; i0+=c0) do
  CS1
  for (i1 = L1; i1 < U1; i1+=c1) do
    CS2
    for (in = Ln; in < Un; in+=cn) do
       CSn
    …
  …
```

- *Actually a dummy restriction (posed to facilitate analysis)*
- *Can easily be handled by straightforward extension of the model*

- Loop bodies do not include conditional or unconditional jumps, function calls, return statements, etc.
- Loop bodies are free of synchronization primitives (e.g. critical sections)
- A single loop in the nest is parallelized
  - No recursive parallelism
  - Does not support multiple parallel loops at the same level

Algorithmic model

What we do not handle

```
for (i0 = L0; i0 < U0; i0+=c0) do
  CS1
  for (i1 = L1; i1 < U1; i1+=c1) do
    CS2
    for (in = Ln; in < Un; in+=cn) do
      CSn
    …
  …
```

- *Sounds like a severe restriction, but is it?*
- *Typically memory-bandwidth bound applications **do** access regular data structures*

- Loop bodies do not include conditional or unconditional jumps, function calls, return statements, etc.
- Loop bodies are free of synchronization primitives (e.g. critical sections)
- A single loop in the nest is parallelized
  - No recursive parallelism
  - Does not support multiple parallel loops at the same level
- Loop bodies involve operations on regular data structures (e.g. arrays)