

# Update on Prefetching

Zdeněk Dvořák

SuSE CR

Novell

# Importance of Prefetching

- memory latencies are too large

Architecture	L1 miss	L2 miss
x86_64	13 cycles	> 100 cycles
ia64	5 – 10 cycles	> 15 cycles

- problematic for scheduling

# Overview of the current state

- implemented on GIMPLE
- one loop at a time
- local reuse analysis
- unrolling to exploit self-reuse
- prefetch distance takes size of the loop into account

# Implementation overview

```
for (i = 0; i < max; i++)  
    use(a[255]);           (0)  
    use(a[i]);           (1)  
    use(a[i + 64]);      (2)  
    use(a[16*i]);        (3)  
    use(a[187*i]);       (4)  
    use(a[187*i + 50]);  (5)
```

# Implementation overview

```
for (i = 0; i < max; i++)  
    use(a[255]);           (0)  
    use(a[i]);           (1)  
    use(a[i + 64]);      (2)  
    use(a[16*i]);        (3)  
    use(a[187*i]);       (4)  
    use(a[187*i + 50]);  (5)
```

Analyze memory references:

Groups:

- (0) base a, step 0
- (1), (2) base a, step 1
- (3) base a, step 16
- (4), (5) base a, step 187

# Implementation overview

```
for (i = 0; i < max; i++)  
    use(a[255]);           (0)  
    use(a[i]);            (1)  
    use(a[i + 64]);       (2)  
    use(a[16*i]);         (3)  
    use(a[187*i]);        (4)  
    use(a[187*i + 50]);   (5)
```

## Reuse analysis:

- (0) self-reuse  $\rightarrow$  prefetch\_before = 1
- (1) reuse of (2)  $\rightarrow$  prefetch\_before = 64, self-reuse  $\rightarrow$  prefetch\_mod = 64
- (2) self-reuse  $\rightarrow$  prefetch\_mod = 64
- (3) self-reuse  $\rightarrow$  prefetch\_mod = 4
- (4),(5) no reuse – (4) reuses (5) with probability only 7/32.

# Implementation overview

```
for (i = 0; i < max; i++)  
    use(a[255]);           (0)  
    use(a[i]);           (1)  
    use(a[i + 64]);      (2)  
    use(a[16*i]);        (3)  
    use(a[187*i]);       (4)  
    use(a[187*i + 50]);  (5)
```

Determine prefetch distance and unroll factor:

Distance:

- memory latency/iteration time – both unknown
- guess  $cst/\#$  of loop insns used instead

Unroll factor:

- 64 from reuse analysis
- $PARAM\_MAX\_UNROLLED\_INSNS/\#$  of insns

# Implementation overview

```
for (i = 0; i < max; i++)  
    use(a[255]);           (0)  
    use(a[i]);            (1)  
    use(a[i + 64]);       (2)  
    use(a[16*i]);         (3)  
    use(a[187*i]);        (4)  
    use(a[187*i + 50]);   (5)
```

Unroll and issue the prefetches:

- Number of prefetches bounded by `SIMULTANEOUS_PREFETCHES`
- No prefetches for (0), (1).
- No prefetches before loop.

# Miscellaneous Improvements

- tuning of unroll factor heuristics
- number of iteration estimates
- command-line parameters
  - prefetch-latency
  - simultaneous-prefetches
  - l1-cache-size
  - l1-cache-line-size
- autodetection of parameters (x86)
- usage of profile feedback

# Reuse Analysis

- simulating cache behavior
- # of cache lines  $\leq$   
# of misses between accesses  $\rightarrow$  no reuse
- may be less for caches with limited associativity
- scope
  - one loop
  - loopnest
  - cross-loop
  - interprocedural

# Loopnest Reuse Analysis

```
for (i)
  for (j)
    use(a[i][j]);      (0)
    use(a[i+3][j]);   (1)
    use(b[j][2*i]);   (2)
    use(c[j]);        (3)
```

Determine distance vectors

(1) (0)  $\langle 3, 0 \rangle$   
(3) (3)  $\langle 1, 0 \rangle$

# Loopnest Reuse Analysis

```
for (i)
  for (j)
    use(a[i][j]);      (0)
    use(a[i+3][j]);   (1)
    use(b[j][2*i]);   (2)
    use(c[j]);        (3)
```

## Add self-reuse distance vectors

(1)	(0)	$\langle 3, 0 \rangle$
(3)	(3)	$\langle 1, 0 \rangle$
(0)	(0)	$\langle 0, 1 \rangle$ , prob. 94%
(1)	(1)	$\langle 0, 1 \rangle$ , prob. 94%
(2)	(2)	$\langle 1, 0 \rangle$ , prob. 88%
(3)	(3)	$\langle 0, 1 \rangle$ , prob. 94%

# Loopnest Reuse Analysis

```
for (i)
  for (j)
    use(a[i][j]);      (0)
    use(a[i+3][j]);   (1)
    use(b[j][2*i]);   (2)
    use(c[j]);        (3)
```

# of cache lines touched in loops

for (j) ... 4  
for (i) ...  $4n$

# Loopnest Reuse Analysis

```
for (i)
  for (j)
    use(a[i][j]);      (0)
    use(a[i+3][j]);   (1)
    use(b[j][2*i]);   (2)
    use(c[j]);        (3)
```

# of cache lines touched in loops

for (j) ... 4

for (i) ...  $4n$

**Wrong** – ignores reuses

# Loopnest Reuse Analysis

```
for (i)
  for (j)
    use(a[i][j]);      (0)
    use(a[i+3][j]);   (1)
    use(b[j][2*i]);   (2)
    use(c[j]);        (3)
```

# of new cache lines touched

for (j) ... 0.1875

for (i) ...  $0.1875n$

# Loopnest Reuse Analysis

```
for (i)
  for (j)
    use(a[i][j]);      (0)
    use(a[i+3][j]);   (1)
    use(b[j][2*i]);   (2)
    use(c[j]);        (3)
```

# of new cache lines touched

for (j) ...  $0.1875$

for (i) ...  $0.1875n$

**Also wrong** – ignores cache size

# Loopnest Reuse Analysis

```
for (i)
  for (j)
    use(a[i][j]);      (0)
    use(a[i+3][j]);   (1)
    use(b[j][2*i]);   (2)
    use(c[j]);        (3)
```

# of new cache lines touched

for (j) ... 0.1875

for (i) ...  $0.1875n$

**Also wrong** – ignores cache size

**Solution** – iterate with the next phase

# Loopnest Reuse Analysis

```
for (i)
  for (j)
    use(a[i][j]);      (0)
    use(a[i+3][j]);   (1)
    use(b[j][2*i]);   (2)
    use(c[j]);        (3)
```

Are reuses realized?

distance · # of touched lines < cache size → **yes**

# Cross-loop Reuse Analysis

```
for (i, j) a[i][j];      (0)
if (...)
    for (i, j) b[i][j];  (1)
    for (i, j) a[i][j];  (2)
else
    for (i, j) c[i][j];  (3)
for (i, j) b[i][j];      (4)
```

Algorithm by K. Cooper, K. Kennedy, N. McIntosh:

Cross-loop Reuse Analysis and its Application to Cache Optimizations

# Cross-loop Reuse Analysis

```
for (i, j) a[i][j];      (0)
if (...)
    for (i, j) b[i][j];  (1)
    for (i, j) a[i][j];  (2)
else
    for (i, j) c[i][j];  (3)
for (i, j) b[i][j];      (4)
```

## Representation of memory regions

```
for (i = 0; i <= n; i++)      (0)
    for (j = n; j >= 0; j-)    (1)
        use(a[i][j]);

base:      &a
index 0:   0 ...n, loop (0), stride 4n
index 1:   n ...0, loop (1), stride 4
```

# Cross-loop Reuse Analysis

```
for (i, j) a[i][j];      (0)
if (...)
    for (i, j) b[i][j];  (1)
    for (i, j) a[i][j];  (2)
else
    for (i, j) c[i][j];  (3)
for (i, j) b[i][j];      (4)
```

**Algorithm** – after processing subloops,

- process acyclic CFG in topological order
- compute cache contents, with ages
- dealing with join points

# Cross-loop Reuse Analysis

```
for (i, j) a[i][j];      (0)
if (...)
    for (i, j) b[i][j];  (1)
    for (i, j) a[i][j];  (2)
else
    for (i, j) c[i][j];  (3)
for (i, j) b[i][j];      (4)
```

**Algorithm** – after processing subloops,

- process acyclic CFG in topological order
  - compute cache contents, with **ages**
    - age – number of cache misses after the access
    - remove references with age above cache size
    - keep only younger for duplicated regions
- after (2): a age 0, b age  $n^2$ ; after (3): a age  $n^2$ , c age 0
- dealing with join points

# Cross-loop Reuse Analysis

```
for (i, j) a[i][j];      (0)
if (...)
  for (i, j) b[i][j];    (1)
  for (i, j) a[i][j];    (2)
else
  for (i, j) c[i][j];    (3)
for (i, j) b[i][j];      (4)
```

**Algorithm** – after processing subloops,

- process acyclic CFG in topological order
- compute cache contents, with ages  
after (2): a age 0, b age  $n^2$ ; after (3): a age  $n^2$ , c age 0
- dealing with join points: **conservative**
  - keep common regions, with older age
  - before (4): a age  $n^2$

# Cross-loop Reuse Analysis

```
for (i, j) a[i][j];      (0)
if (...)
    for (i, j) b[i][j];  (1)
    for (i, j) a[i][j];  (2)
else
    for (i, j) c[i][j];  (3)
for (i, j) b[i][j];      (4)
```

**Algorithm** – after processing subloops,

- process acyclic CFG in topological order
- compute cache contents, with ages  
after (2): a age 0, b age  $n^2$ ; after (3): a age  $n^2$ , c age 0
- dealing with join points: **keep more probable branch**
  - simpler
  - handles guarded loops correctly  
if (n > 0) {i = 0; do {...; i++} while (i < n);}

# Reuse Exploitation

- not prefetching reused accesses

```
for (i) use(a[i]);  prefetch
```

```
for (i) use(a[i]);  do not prefetch
```

- loop peeling to exploit reuse
- temporality hints (prefetches, loads)
- streaming stores
- other optimizations (loop interchange, ...)

# Reuse Exploitation

- not prefetching reused accesses
- loop peeling to exploit reuse

```
for (i) for (j)
    use(a[i][j]);
    use(a[i+1][j]);
```

peel + prefetch only the 1st iteration of i loop

- temporality hints (prefetches, loads)
- streaming stores
- other optimizations (loop interchange, ...)

# Reuse Exploitation

- not prefetching reused accesses
- loop peeling to exploit reuse
- temporality hints (prefetches, loads)
- streaming stores

```
for (i = 0; i < LOT; i++)
```

```
    a[i] =
```

```
        b[i];
```

use streaming store

use nt load/prefetch

- other optimizations (loop interchange, ...)

# Reuse Exploitation

- not prefetching reused accesses
- loop peeling to exploit reuse
- temporality hints (prefetches, loads)
- streaming stores
- other optimizations (loop interchange, ...)

# TODO

- prefetching outside of loops
- prefetching first iterations of loops
- interaction with hardware prefetcher
- better support for target-specific constraints

# ia64 Specific Issues

tuning (?fixing) parameters

Dealing with

- temporality hints for loads
- different size of L1 and L2 cache lines
- floating-point operations bypass L1 cache
  - prefetch floating-point data only to L2
  - integer and fp should not share cache line
  - different temporality hints

# Questions

