Discovering Cache Partitioning Optimizations for the K Computer

Swann Perarnau, Mitsuhisa Sato

RIKEN AICS, Programming Environment Research Team
University of Tsukuba

APPLC'13, Shenzhen, China

The K Computer

The K architecture

Organization

6D network.

Over 700,000 processors.

→ 80,000 compute nodes.

→ 800 racks.

Compute Node

1 CPU: SPARC64VIIIfx.

8 cores.

16 GB shared memory.

6MB L2 shared cache.

The K architecture

Organization

6D network.

Over 700,000 processors.

→ 80,000 compute nodes.

→ 800 racks.

Compute Node

1 CPU: SPARC64VIIIfx.

8 cores.

16 GB shared memory.

6MB L2 shared cache.

Optimizing single node performance matters.

Cache Optimization

Locality is a critical performance factor in shared memory.

On the K Computer

Hardware cache partitioning mechanism.
$\rightarrow$ Isolate thrashing accesses from *useful* data.
$\rightarrow$ Favor data fitting in cache against others.

Implementation

Sector cache: instruction-based, only 2 sectors.

Our Work

Issues

The sector cache is hard to use :

- Very low level API.
- Requires good knowledge of code locality.
- Finding the best partitioning is not obvious.

Our goal

Assess the applicability of the sector cache:

- Design a locality analysis tool for it.
- Optimize HPC applications.

Outline

Sector Cache

Hardware Cache Partitioning

The cache can be split in two sectors.
Accesses to one sector cannot evict memory from the other.
Special instructions `sxar1,sxar2` to configure/use it.

How it works

Sectors are a split of each associative set of the cache.
$\rightarrow$ 11 available sizes.

Operation

1. Specify size of each sector
2. Use instruction to tag a load into one sector.
3. Hardware keeps track of the sizes of each sector.
4. If space is needed, eviction is an LRU inside a sector.

Instruction Level

Instruction :	Sector :
`load 0x10`	*s0*
`sxar 1`	
`load 0x20`	*s1*
`load 0x30`	*s0*
`sxar2 1 1`	
`load 0x10`	*s1*
`load 0x20`	*s1*
`load 0x10`	*s0*

User API

Compiler Hints

Over a code region, tag an array to be in sector 1.

```
double myarray[NSIZE];
double otherarray[NSIZE];

void mywork(void)
{
        int i;
        double sum = 0;
#pragma statement cache_sector_size 1 11
#pragma statement cache_subsector_assign myarray
        for(i = 2; i < NSIZE-2; i++)
        {
                // myarray in sector 1
                sum += myarray[i-2] + myarray[i-1] +
                       myarray[i] + myarray[i+1] +
                       myarray[i+2] + otherarray[i];
        }
}
```

Difficulties

- Optimization must be decided at compile time.
- No automatic detection of optimization points.
- Impact of sector cache configuration on performance not obvious.

Our goal

- Analyze structures locality.
- Suggest valid sector cache configurations.

Outline

Overview

3 Analysis Phases

- Trace and identify memory accesses to a data structure.
- Measure locality of those accesses.
- Predict sector cache performance.

How ?

- Binary instrumentation and debugging information.
- Reuse distances.
- Model sectors as smaller independent caches.

Outline

This Step

Principle

Use debugging information to discover each data structure location.
Trace memory accesses using binary instrumentation (Pin).

Operation

- User provide a structure name and scope.
- Tool reads DWARF debugging information.
- Location of structure is saved for runtime use.
- One instrumented run traces accesses
- Each access is mapped to a structure's location.

Structure identification

User information

- Data structure name.
- Scope: enclosing function or compilation unit.

Finding a structure location

DWARF contains beginning address and location expression.
Location expression is a stack automata using machine registers.
$\rightarrow$ Save expression to use at runtime.

Binary Instrumentation

A Pin Tool

Execute a specific code every time a memory access instruction is executed.
Only works on x86/amd64: analysis done outside of the K Computer.

Instrumentation:

Limited to either a function scope or a range of source code lines.
$\rightarrow$ improves the speed of the instrumented run.
Logging of dynamic memory allocation function calls.

Supported types

Stack and global arrays using location expression.
Dynamic structures identified recursively (*very slow*).
No support for pointer function arguments.

Outline

Purpose

Goal

Understand what happens if we push a specific structure into sector 1.

How ?

Measure the locality of this structure if it was alone in the sector.

What to measure ?

Special version of reuse distance for a set of memory accesses.

Reuse Distance

Definition

For a memory access : number of unique memory locations touched after the previous access to the same location.

Reuse Distance

Definition

For a memory access : number of unique memory locations touched after the previous access to the same location.

Access :

```
load 0x10
load 0x20
load 0x30
load 0x10
load 0x20
load 0x10
load 0x30
```

Reuse Distance

Definition

For a memory access : number of unique memory locations touched after the previous access to the same location.

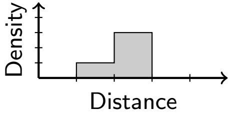
Access :	Distance :
`load 0x10`	∞
`load 0x20`	∞
`load 0x30`	∞
`load 0x10`	2
`load 0x20`	2
`load 0x10`	1
`load 0x30`	2

Reuse Distance

Definition

For a memory access : number of unique memory locations touched after the previous access to the same location.

Access :	Distance :
`load 0x10`	∞
`load 0x20`	∞
`load 0x30`	∞
`load 0x10`	2
`load 0x20`	2
`load 0x10`	1
`load 0x30`	2



Conditional Reuse Distance

For each structure s

Only consider memory reference m iff it satisfies a condition.
Two kinds of conditions here:

- m is an access to s (*isolated reuse*).
- m is not an access to s.

Implementation

Fastest sequential reuse distance algorithm.
Could be parallelized for better performance. Each CRD is saved as an histogram per structure.

Tracing algorithm

Reuse distance algorithm

A hash map from address to timestamp.
A balanced binary tree ordered by timestamp, saving addresses.
Each node of the tree maintain a count of its left and right children.
Reuse distance is a simple tree traversal ($O(\log M)$).

Optimizations

Consider two addresses in the same cache line as the same location.
Only maintain information for the amount of addresses the cache can contain.

Outline

Principle

Approximate cache requirements using the reuse distance histogram.

Operation

1. For each structure:
2. For each sector cache configuration:
3. Compute cache misses triggered by structure isolation.
4. Find best configuration among all.

Cache model

Assume a fully associative cache, perfect LRU.

Reuse distance is the number of unique locations the program accessed between two accesses to the same location.
$\rightarrow$ corresponds to the number of **cache lines** fetched from memory.
$\rightarrow$ if more lines are fetched that the cache size, a cache miss is triggered.

For the sector cache

Modeled as two caches of specific sizes.
Only accesses inside a sector matter to predict cache misses.

For each structure

Isolated reuse histogram gives approximation of sector 1 cache misses.
Other histogram gives cache misses in sector 0.

Outline

Experimental setup

Validation

Analyze and optimize toy application.

- A single memory access pattern.
- Known locality requirements.
  $\rightarrow$ Validate analysis.
- Test all possible optimizations.
  $\rightarrow$ Validate optimization.

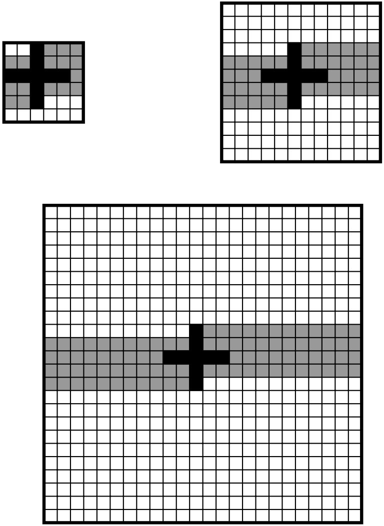
Multigrid Stencil

Stencil

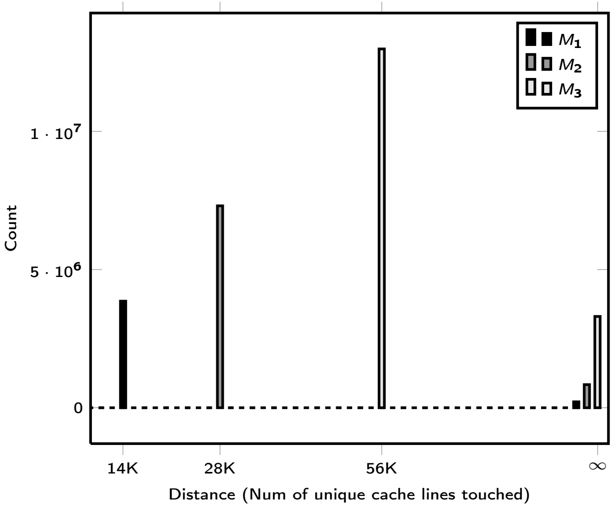
Sum of 9 points over 3 matrices, written to a fourth one.
M_1 4 times smaller than M_2.
M_2 4 times smaller than M_3.
M_r is the same size as M_3.

Cache Requirements

Each matrix requires only 5 of its lines in cache.



Reuse Distances



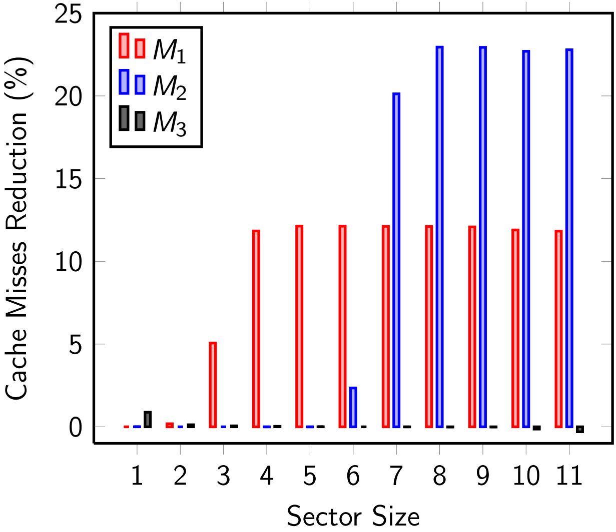
Optimization

Tool's analysis

Our model gives us an optimal setup with M_2 in sector 1 of size 7.

Version	Stencil Miss Rate (%)	Reduction (%)
Unoptimized	2.10	-
$M_2(5,7)$	1.68	20

Full search results



Process

Similar to the toy application.
Additional code analysis for sector sharing.

Benchmark	Function	Isolated Variables	Sector Size	Miss Reduction (%)	Runtime Reduction (%)
CG	conj_grad	p	(1,11)	19	10
LU	ssor	a,b,c,d	(2,10)	48	8
	blts	ldz,ldy,ldx,d		75	10
	buts	d,udx,udy,udz		18	3
	jacld	a,b,c,d		64	14
	jacu	a,b,c,d		57	6

Table: Optimization of NAS Benchmarks.

Outline

Summary

Tool for data structure analysis

Discover its location in virtual memory.
Trace memory access to it during a run.
Compute various types of reuse distances.

Optimizations

Limit analysis to a specific code region.
Find a good sector cache configuration for the region.
Validated on toy application and NPB.

Future Work

Better analysis

Multiple structures in one sector.
Locality across functions.
More Dynamic partitioning.

Better toolset

Parallel analysis.
Faster instrumentation.

Better optimizations

Detect specific locality patterns (streaming).
Automate the source modification.

Thank you for your attention !