Toward Automated Cache Partitioning for the K Computer

Swann Perarnau, Mitsuhisa Sato

RIKEN AICS, Programming Environment Research Team
University of Tsukuba

The Memory Wall

One access to RAM costs 100 times more than a register access.
$\rightarrow$ cache/locality optimization.

Two classes of methods to improve locality

- Oblivious algorithms.
- Data reorganisation

Cache Partitioning

Principle

Split the cache and distribute application data among partitions.

Advantages

Isolate thrashing accesses from *useful* data.
Favor data fitting in cache against others.

On the K Computer

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

Provide an automated framework to analyze and optimize an application for the sector cache.

- Locality analysis by binary instrumentation.
- Automated Optimization discovery.

Outline

1. Introduction
2. Cache partitioning on the SPARC64 VIIIfx
3. Automated Analysis and Optimization
   - Data Structure Localization
   - Binary Instrumentation
   - Locality Analysis
4. First results
5. Conclusion

The K architecture

Computing Node

1 CPU: SPARC64VIIIfx.

8 cores.

16 GB memory.

L2 shared cache.

Cache

6 MB.

12-way associative.

128 bytes line size.

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.
- No automatic optimizations.
- Impact of sector cache configuration on performance not obvious.

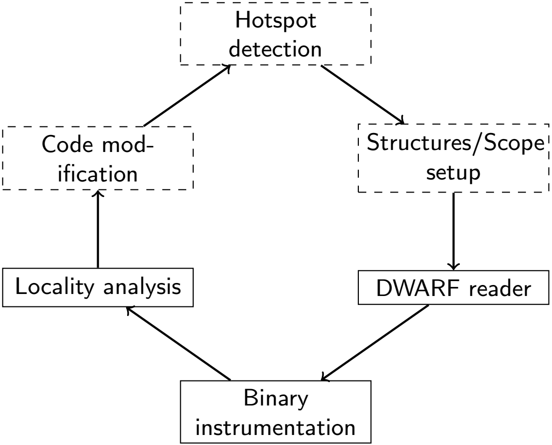
Our goal

Build an automated framework to:

- Detect cache performance hotspots.
- Analyze structures locality on these hotspots.
- Insert API calls to optimize the application.

Outline

Overview


Hotspot detection
Structures/Scope setup
DWARF reader
Binary instrumentation
Locality analysis
Code mod-ification

Overview

DWARF reader

Builds table containing structure location in memory.

Binary instrumentation

Trace memory accesses to each identified structure.

Locality analysis

Use memory trace to predict cache performance of sector configuration.

Outline

This Step

Principle

Use debugging information to discover each data structure location.

Operation

- User provide a structure name and scope.
- Tool reads DWARF debugging information.
- Location of structure is saved for future use.

Structure identification

User information

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

Limitations

Only works for types supported by the sector cache API (arrays).

Debugging Information

DWARF

Standard debugging information format for Linux.
Organised as a tree of all symbols inside the application.

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.

Outline

Purpose

Goal

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

How ?

Use one instrumented run to measure the locality of each data structure.

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.

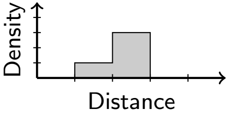
Access :	Distance :
`load 0x10`	∞
`load 0x20`	∞
`load 0x30`	∞
`load 0x10`	2
`load 0x20`	2
`load 0x10`	1
`load 0x30`	2

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



Our implementation

Binary instrumentation with Pin

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 Scope

We limit the instrumentation to either a function scope or a range of source code lines.
$\rightarrow$ improves the speed of the instrumented run.

Reuse Algorithm

Fastest sequential one.
Could be parallelized for better performance.

Tracing algorithm

For each traced structure

Measure its locality if alone.
$\rightarrow$ reuse histogram of accesses to its addresses.
Measure impact of the sector cache on other accesses.
$\rightarrow$ second reuse histogram for all other addresses.

Reuse distance computation

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.

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.

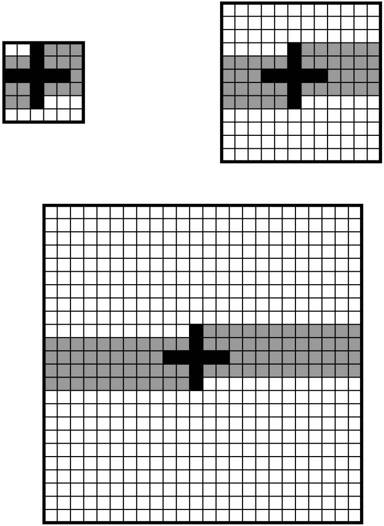
Multigrid Stencil

Stencil

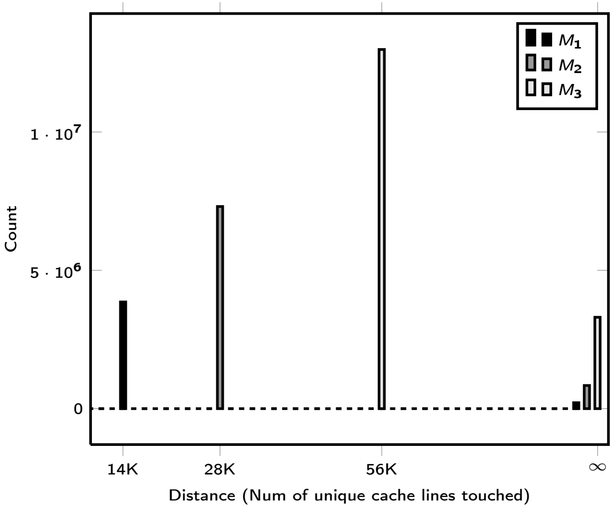
Sum of 9 points over 3 matrices, written to a fourth one.
M_1 4 times smaller than M_2.
M_2 4 times smaller than M_3.
M_r is the same size as M_3.

Cache Requirements

Each matrix requires only 5 of its lines in cache.



Reuse Distances



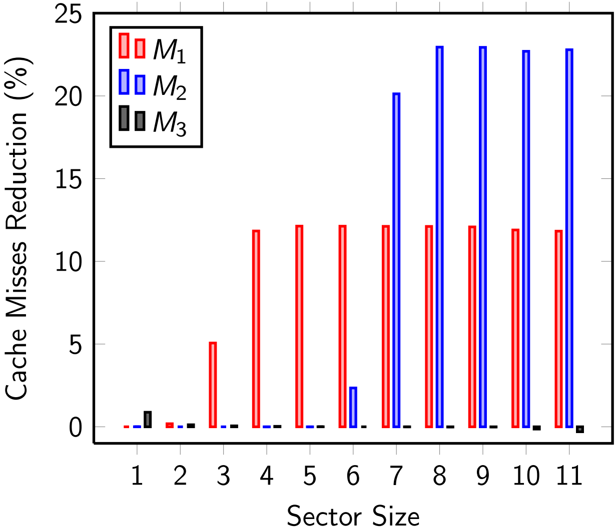
Optimization

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



Outline

Summary

Analysing a data structure

Discover its location in virtual memory.
Trace memory access to it during a run.
Predict its cache behavior.

Optimizing a code

Limit analysis to a specific code region.
Find a good sector cache configuration for the region.

Future Work

Toward automation

Hotspot detection.
Source code analysis.
Code transformation.

Using the framework

Optimize HPC benchmarks (NAS NPB, Spec).
Optimize real applications.

Better optimizations

Multiple structures in sector 1 at the same time.
Detect specific locality patterns (streaming).