The Cache Behaviour of Large Lazy Functional Programs on Stock Hardware

Nicholas Nethercote and Alan Mycroft

Computer Laboratory, Cambridge University

http://www.cl.cam.ac.uk/~njn25/

16 June 2002

Overview

- Language and implementation
- Evaluation:
  - Hardware counters (miss counts)
  - Analysis with execution cost model (miss stall times)
  - Simulation (miss locations)
- Improvement: reducing write miss costs with prefetching
- Further work: potential improvements

Language and Implementation

- Haskell: Polymorphically typed, purely functional, *lazy*: expressions only evaluated when needed
- Glasgow Haskell Compiler (GHC): highly-optimising, 'industrial-strength', exotic execution mechanism – everything is a *closure*
- Garbage collector: generational copying collector, flexible
- Benchmarks: 13 "real" programs, non-trivial
- Machine: AMD Athlon 1400 MHz. Worst-case cache latencies: D1 = 12 cycles, L2 = 206 cycles

Glasgow Haskell vs. SML/NJ vs. C

UNIVERSITY OF CAMBRIDGE

Using Athlon hardware counters.

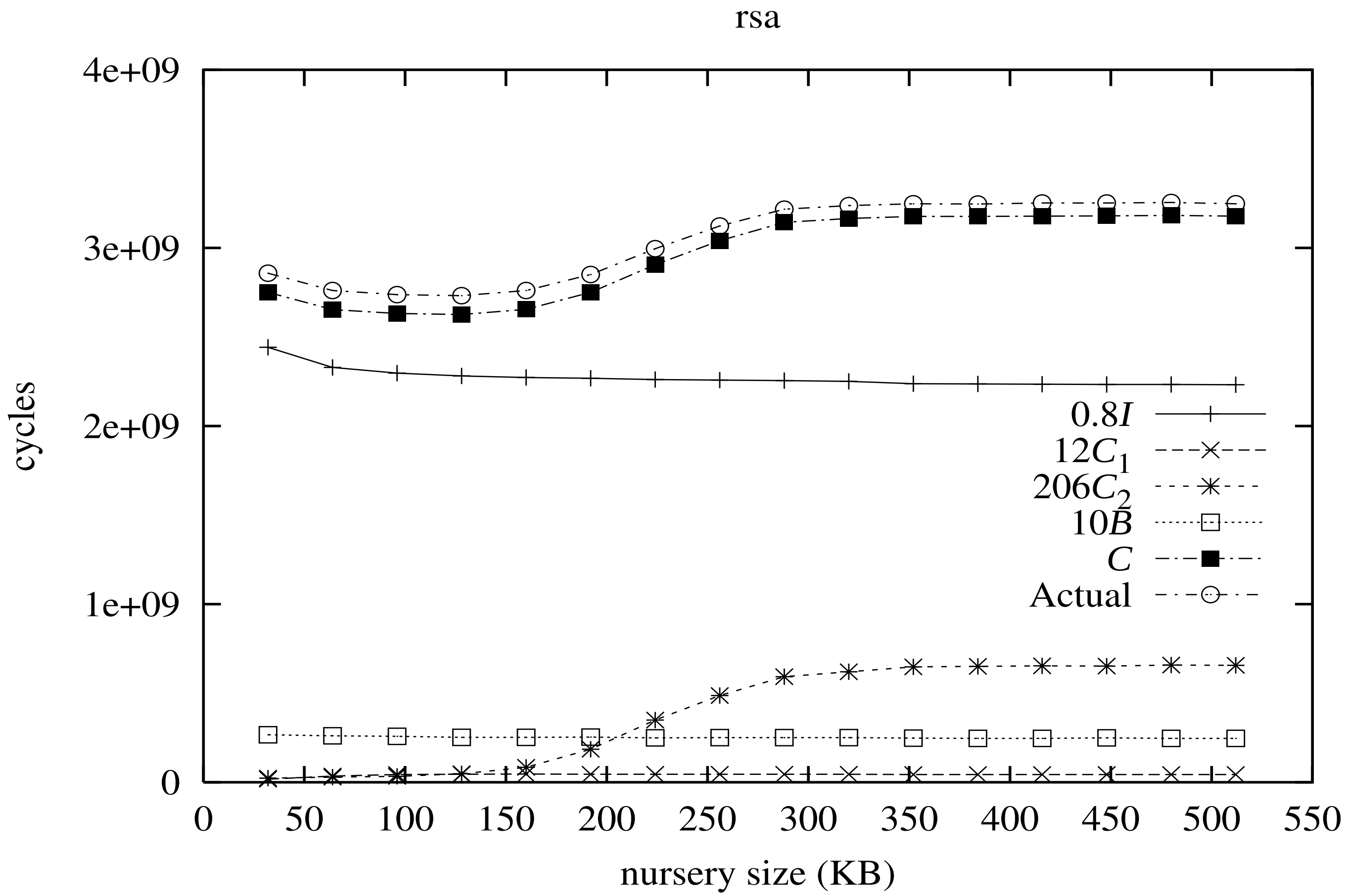
13 Haskell, 5 SML, 5 (comparable) C integer programs:

Language	L2 miss	BrMis	CPI
GHC	1.6	27	2.1
SML/NJ	0.8	10	1.4
C (GCC)	0.3	18	1.3

L2 miss = data cache refills from system / retired instructions * 1000
BrMis = retired branches mispredicted / retired instructions * 1000
CPI = cycles / retired instructions
(geometric means)

Execution Cost Model: Definition

$$C = 0.8I + 12C_1 + 206C_2 + 10B$$



Execution Cost Model: Explanation

What does the simple model show?

- L2 data cache misses not 'hidden'
- Unstalled instructions execute slowly (0.8 CPI)

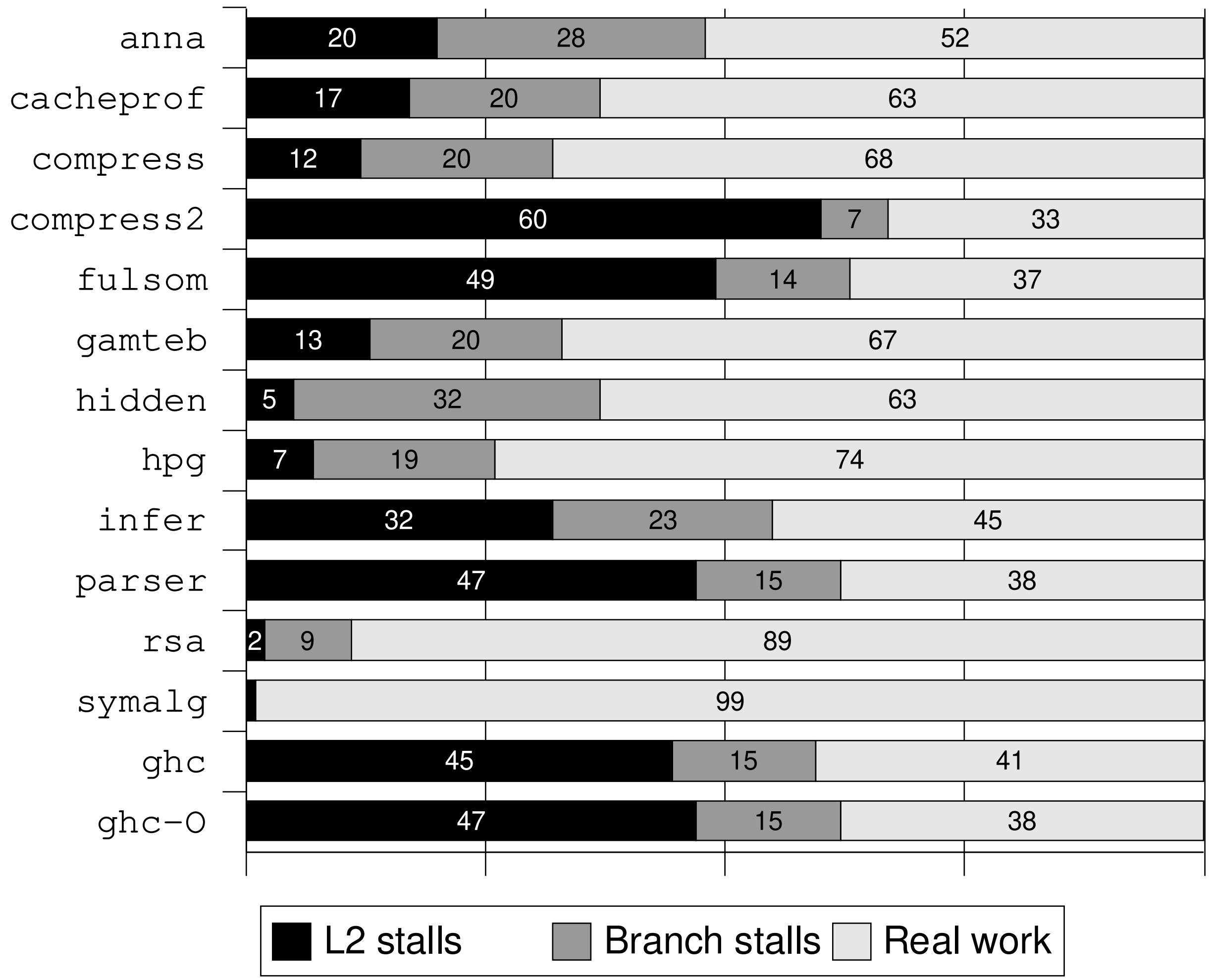
Why?

Emitted code has frequent dependent loads (heap $\leftrightarrow$ stack):

- Little exploitable instruction-level parallelism
- Little work to do during cache misses

GHC programs turn an Athlon into a 486 with a high clock speed!

Execution Cost Model: Use



Simulation: Method

Execution-driven simulation using *Cachegrind*. Annotates source with cache misses line-by-line (C or asm).

Not perfect, ignores:

- Kernel activity
- Cache accesses upon TLB misses
- Virtual-to-physical address translation

But accurate enough to be useful.

Concentrate on L2 data misses; L1 and instruction misses much less important.

Simulation: Findings

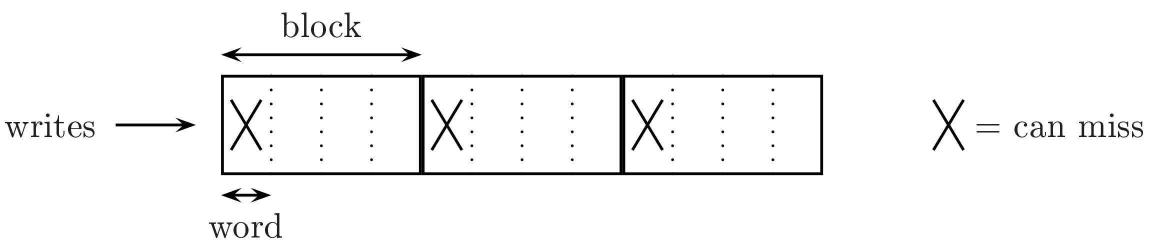
Data read misses, 40–50% of misses, 6 types:

- 3 in-program
- 3 during GC

Data write misses, 50–60% of misses:

- Allocating/initialising new closures
- `copy()` during GC

Sequential writes due to copying GC, only first word can miss:

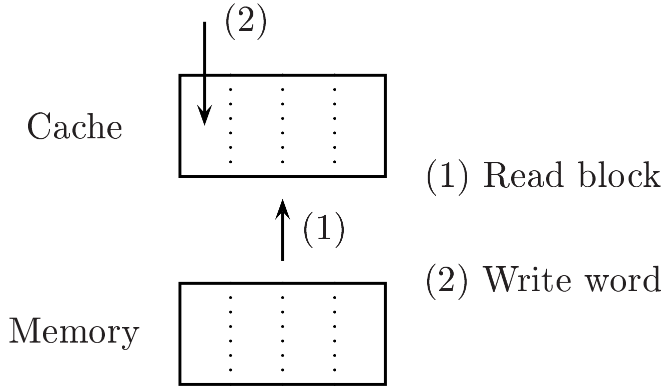


Reducing Write Miss Costs: Prefetching

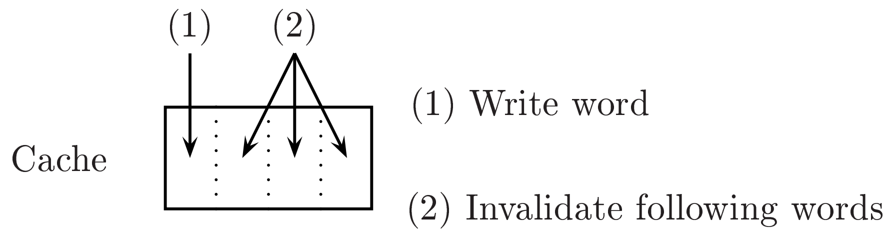
UNIVERSITY OF CAMBRIDGE

Write misses not necessary!

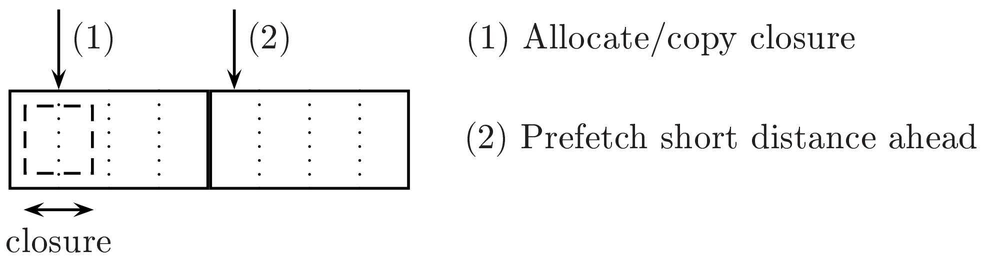
Write-allocate



Better: Write-invalidate



But not possible on x86. Instead, prefetch ahead on allocation/copy.



Reducing Write Miss Costs: Results

UNIVERSITY OF CAMBRIDGE

Program	Speed-up
`anna`	4%
`cacheprof`	5%
`compress`	5%
`compress2`	12%
`fulsom`	17%
`gamteb`	0%
`hidden`	2%
`hpg`	4%
`infer`	9%
`parser`	22%
`rsa`	2%
`symalg`	1%
`ghc`	17%
`ghc -O`	14%

- Code size: + 0.8–1.6%
- Prefetch distance: 64 bytes
- Not just for Haskell!

Conclusions and Further Work

Many cache misses and branch mispredicts cause high CPI.

Also little exploitable instruction-level parallelism.

Write miss costs reduced by prefetching, up to 22% speed-up. Can be used for any program/language using copying GC.

Some room for read miss improvement:

- Improve locality (e.g. data reorganisation at GC)
- Compressing data representation to reduce memory footprints (eg. BIBOP)

Time to redesign GHC's execution mechanism?

Execution Cost Model: Examples

UNIVERSITY OF CAMBRIDGE