The Cache Behaviour of Large Lazy Functional Programs on Stock Hardware

Nicholas Nethercote
Computer Laboratory
Cambridge University
United Kingdom
njn25@cam.ac.uk

Alan Mycroft
Computer Laboratory
Cambridge University
United Kingdom
am@cl.cam.ac.uk

ABSTRACT

Lazy functional programs behave differently from imperative programs and these differences extend to cache behaviour. We use hardware counters and a simple yet accurate execution cost model to analyse some large Haskell programs on the x86 architecture. The programs do not interact well with modern processors—L2 cache data miss stalls and branch misprediction stalls account for up to 60% and 32% of execution time respectively. Moreover, the program code exhibits little exploitable instruction-level parallelism.

We then use simulation to pinpoint cache misses at the instruction level. With this information we apply prefetching to minimise the cost of write misses, speeding up Haskell programs by up to 22%. We conclude with more ideas for changing the Glasgow Haskell Compiler and its garbage collector to improve the cache performance of large programs.

Categories and Subject Descriptors

B.3.3 [**Memory Structures**]: Design Styles—*Cache memories*; B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; D.3.2 [**Language Classifications**]: Applicative (functional) languages—*Haskell*; D.3.4 [**Programming Languages**]: Processors—*compilers, memory management (garbage collection)*

General Terms

Experimentation, languages, measurement, performance

Keywords

Cache measurement, cache simulation, hardware counters, branch misprediction, Haskell, Glasgow Haskell Compiler

1. INTRODUCTION

Cache misses are expensive. An L2 cache miss on a modern personal computer can waste hundreds of CPU cycles. While techniques for optimising the cache performance of array-based programs are mature and well-known, the situation is murkier for general purpose programs.

Declarative languages are possible beneficiaries from cache optimisations. Programs written in declarative languages are often substantially slower than equivalent programs written in lower-level languages that are "closer to the machine". However, given the increasing complexity of computers, being "closer to the machine" can be bad, since most programmers will not understand enough about the hardware to extract good performance from it. Better decisions might be made by a compiler and/or runtime system, thanks to characteristics of declarative languages such as strong static typing, a lack of (or limited) mutable data structures, and greater flexibility in data representation and organisation.

In this paper, we investigate the cache behaviour of programs written in the lazy functional language Haskell, using the Glasgow Haskell Compiler (GHC). Modifying garbage collection parameters to examine the effect on cache behaviour, we found that a simple execution cost model is sufficient to determine that programs spend up to 60% of their time waiting for L2 cache data miss stalls. We also found that up to 32% of program time is taken up by branch misprediction stalls, together the two stall types account for up to 67% of program time, and that the generated code exhibits little exploitable instruction-level parallelism. In response to this, we present detailed simulation-based measurements to precisely identify the locations of L2 cache data misses, and use prefetching to speed up programs by up to 22% by reducing the cost of cache write misses.

This work represents the first detailed measurement of the cache behaviour of realistic, highly optimised, lazy functional programs. It shows the execution times of GHC programs[1] can be predicted accurately with a simple cost execution model, that the impact of L2 cache misses and branch mispredictions is significant, and that GHC programs interact quite poorly with aggressive modern processors. Execution techniques supporting lazy evaluation should be reconsidered; they do not behave well on modern architectures and there is much room for hardware-targeted optimisations to improve their performance. It also shows how prefetching can minimise the cost of write misses in systems using copying garbage collection.

Section 2 describes the language Haskell and the GHC

[1]We will use the phrase "GHC programs" to denote implementation-specific behaviour of Haskell programs compiled with GHC.

MSP 2002 Berlin, Germany

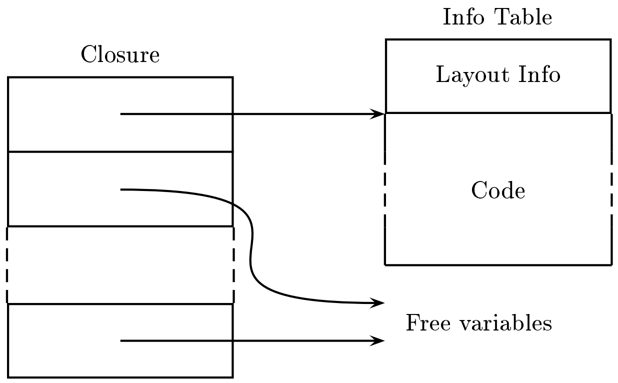


Figure 1: Closure and info table

implementation, the suite of benchmark programs we measured, and the machine we used. Sections 3 and 4 analyse the programs using information from hardware counters. Section 5 introduces a simple execution cost model to aid the analysis. Section 6 uses simulation to pinpoint cache miss locations, and shows the results of prefetching. Section 7 discusses related work. Section 8 discusses further work and concludes.

2. SYSTEM CHARACTERISTICS

2.1 Language and Implementation

Haskell is a polymorphically typed, lazy, purely functional programming language widely used as a testbed for research. The Glasgow Haskell Compiler (GHC) [7] is a highly optimising "industrial-strength" compiler for Haskell. The compiler is itself written in Haskell; its runtime system is written in C. Although its distribution contains an interpreter, GHC is a true compiler designed primarily for creating standalone programs. The optimisations it performs include full laziness, deforestation, let floating, beta reduction, lambda lifting and strictness optimisations; it also supports unboxed values [17]. It is widely considered to be the fastest implementation of a lazy functional language [9]. Because it is highly optimising, it is not a soft target. This is important, since optimising non-optimised systems is always less of a challenge than optimising optimised systems.

2.2 The STG Machine

To understand the causes of cache misses requires some understanding of GHC's execution mechanism, the spineless tagless G-machine (STG machine).

Programs consist of two main kinds of objects: evaluated values (function and data values), and as-yet unevaluated suspensions (called *thunks*). All objects are represented uniformly, as *closures* (Figure 1). Some closures are static, and some are allocated on the heap dynamically. Function values are represented by a pointer to a static *info table*—which contains code for the function and some layout information used during garbage collection—plus a *payload* of (pointers to) the values of any free variables. Thunks have the same representation, but when a thunk is evaluated its must update its own value (so that it is not evaluated more than once). The representation of data values is best explained by the example of a cons cell: a pointer to an info table containing code that simply returns immediately, followed by a payload of a head pointer and tail pointer. For more details see [16].

Most closures are 1–4 words which means that 4–16 can fit within each 64 byte cache block typical of recent x86 processors. This is enough for the effect of spatial locality to be important. All info tables have two or three words of layout information, and most have 1–30 instructions.

This execution mechanism is necessarily quite exotic in order to support lazy evaulation, and stresses processors in unusual ways, as we shall see in Sections 4 and 5.

2.3 Benchmark Suite

Twelve of the the benchmark programs tested come from the "real" part of the `nofib` suite [15] of Haskell programs. These programs were all written to perform an actual task; most are a reasonable size, and none have trivial input or behaviour. This is important—small and large programs have different cache behaviour, and we want to optimise large and realistic programs. The other program tested was GHC itself, compiling a large module with and without optimisation. The benchmark programs are described in Figure 2, and their sizes in lines of code (minus blanks and comments) are given. Inputs were chosen so each program ran for about 2–3 seconds, except for the `ghc` benchmarks, which were substantially longer. This is long enough to realistically stress the cache, but short enough that the programs ran for reasonable times when simulated (see Section 6).

Program	Description	lines
`anna`	Frontier-based strictness analyser	5740
`cacheprof`	x86 assembly code annotator	1489
`compress`	LZW text compression	403
`compress2`	Text compression	147
`fulsom`	Solid modeller	857
`gamteb`	Monte Carlo photon transport	510
`hidden`	PostScript polygon renderer	362
`hpg`	Random Haskell program generator	761
`infer`	Hindley-Milner type inference	561
`parser`	Partial Haskell parser	932
`rsa`	RSA file encryptor	48
`symalg`	Symbolic algebra program	831
`ghc`	GHC, no optimisation	78950
`ghc -O`	GHC, with `-O` optimisation	78950

Figure 2: Haskell program descriptions

GHC can compile via C, or use its x86 native code generator; the distinction is unimportant for us, as programs compiled by the two routes have extremely similar cache behaviour. All programs were compiled with a recent development version of GHC (derived from v5.02.2), via C using GCC 3.0.4, using the `-O` optimisation flag. For all experiments, they were run with a stack size of 10MB,[2] and with context-switching turned off (as is sensible for single-threaded programs).

2.4 Machine Characteristics

The machine used for the experiments was an AMD Athlon, running Red Hat Linux 7.1, kernel version 2.4.7—a typical

[2]The stack limit is set to abort execution in the case of accidental infinite recursion; it has no effect on cache behaviour.

modern system, described in Figure 3. The information in the first part of the table was gathered from AMD documentation [1] and the results of the CPUID instruction.

Architecture	AMD K7, model 4
Clock speed	1400 MHz
I1 cache	64KB, 64B lines, 2-way
D1 cache	64KB, 64B lines, 2-way, write-allocate, write-back, 2 64-bit ports, LRU
L2 unified cache	256KB, 64B lines, 8-way, on-die, exclusive (contains only victim blocks)
System bus	Pair of unidirectional 13-bit address and control channels; bidirectional, 64-bit, 200 MHz data bus
Write buffer	4-entry, 64-byte
D1 replace time	12 cycles
L2 replace time	206 cycles

Figure 3: Athlon characteristics

Note that the I1 cache does some prefetching—upon a miss both the current block and following block are read into the cache. Also, the L2 cache is 8-way according to the CPUID instruction's result, not 16-way as claimed in [1].

The cache replace times in the second part of the table were found using Calibrator v0.9e [13], a micro-benchmark which performs multiple dependent array accesses with various "stride" lengths to estimate worst-case D1 and L2 cache latencies. Of course, a 206 cycle L2 replace time does not imply that each L2 miss will cause a 206 cycle stall; in real programs, out-of-order execution can hide cache latencies somewhat. Nonetheless, we will shortly see the usefulness of these figures.

3. VARYING GC PARAMETERS

The cache behaviour of programs using garbage collection can vary considerably with the collector's parameters. Before we can properly analyse the chosen Haskell programs, we need to find the best garbage collector configuration for each one.

GHC's garbage collector is a highly flexible generational copying collector with multiple generations each containing multiple steps; the number of steps and generations is configurable at runtime. The first step of the first generation is the allocation area, or *nursery*. The starting heap size can also be specified, and the heap will then grow and shrink as necessary, guided by heuristics. It also treats large objects separately, to avoid copying them.

We tried individually varying the collector's nursery size, the initial heap size, and the number of generations to determine their effect on program cache behaviour. This served three uses: firstly, it allowed us to find the best garbage collector configuration for each program; secondly, it provided interesting data about the behaviour of the garbage collector; and thirdly, it gave us the insight needed to formulate the simple execution cost model described in Section 5.

3.1 Nursery Size

The most revealing results were seen when changing the size of the allocation area. The default nursery size is 256KB; we varied it from 32KB–512KB for each program. The results are shown in Figures 4 and 5.

Consider first the graph for `compress`, in Figure 4. The "Actual" line gives the number of cycles, reported by Rabbit [10], which provides control over the Athlon's hardware performance counters [1]. (We will explain the other lines on the graphs in Section 5.) The line dips to minimum around 160KB, before rising again as the nursery size increases. This is due to two competing effects that arise as the nursery size increases: firstly, fewer garbage collections are performed, so the instruction count decreases; secondly, the amount of memory touched grows, so the number of cache misses increases.

The second effect becomes significant as the nursery size approaches 256KB, the size of the L2 cache. This matches Wilson, Lam and Moher's advice to fit the youngest generation of a generational collector within the cache [19]. The effect starts before the 256KB point is reached because the Athlon's L2 cache is unified, containing both data and code. This graph shape is seen for most of the programs.[3]

3.2 Initial Heap Size

The second parameter we varied was the initial heap size. Unlike when changing nursery size, there is no clean explanation for the four broad trends observed as the initial heap size was increased from 0–64MB: relatively flat graphs, with some changes, but no general trend (e.g. `cacheprof`); graphs with a sudden jump at the start, but flat afterwards (e.g. `hpg`); graphs with a downward slope (e.g. `compress2`); graphs with an upward slope (e.g. `rsa`).

These representative examples are shown in Figure 6. No one initial heap size is best, although the default of zero gave good results for most programs.

3.3 Number of Generations

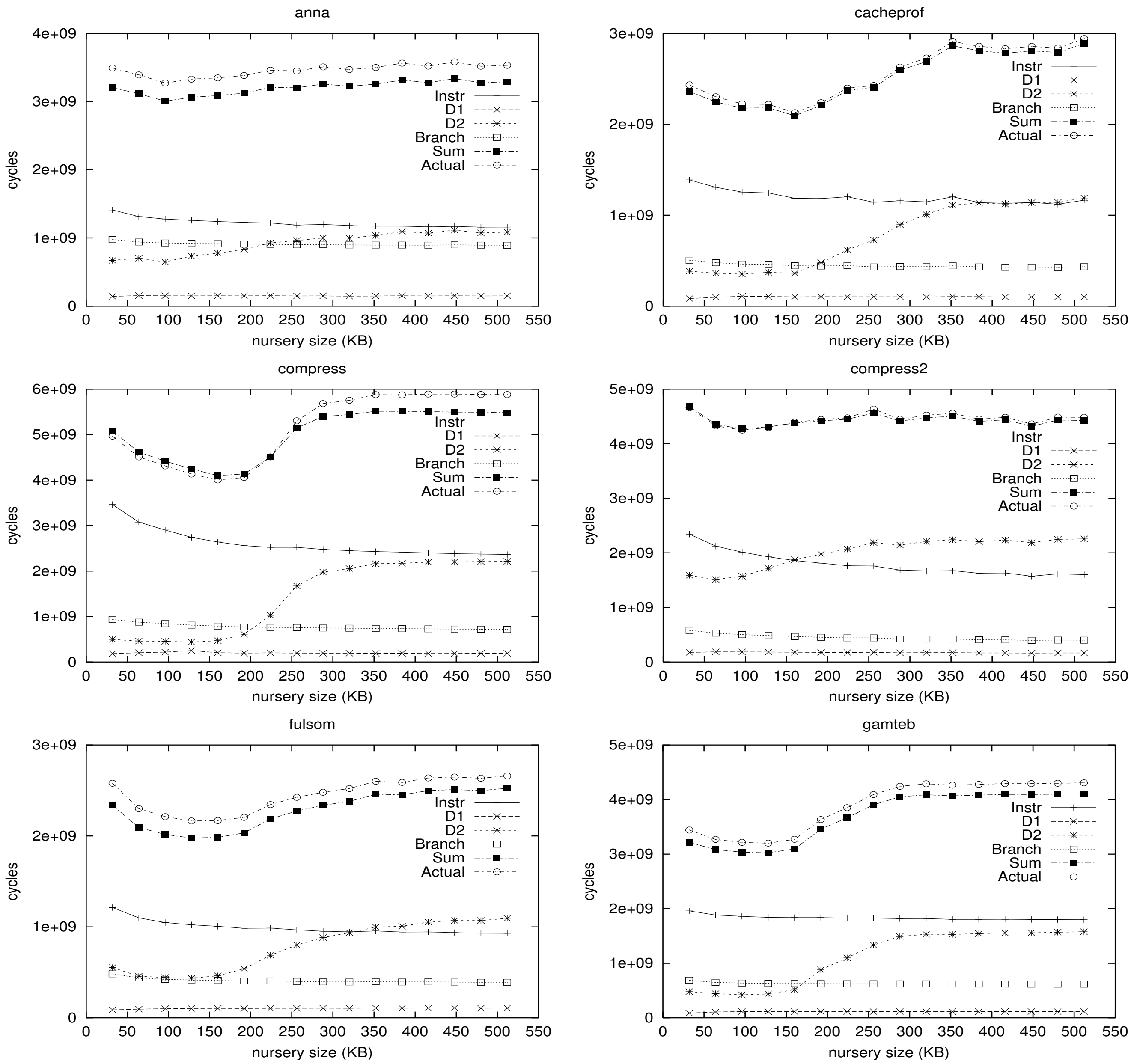
The third parameter we varied was the number of generations, from one (giving a standard two-space copying collector) to six. Having one generation gave easily the worst results, two gave the best, and three to six were marginally worse than two. The only programs to buck this trend were `compress` (one generation gave the best result, and the graph sloped upwards) and `infer` (the graph was a very shallow 'V' shape, with a minimum at three generations). The default choice of two generations was clearly the best.

4. PROGRAM ANALYSIS

To count the number of cache misses occurring during program execution, we again used Rabbit and the Athlon's hardware performance counters. The Athlon has 23 documented measurable events, and four counters. Although Rabbit can use sampling to give approximate results for all events in a single program run, we did not use this facility in order to obtain exact event counts.[4] Instead we divided the events into six event sets. Each program was run five times per event set, thirty times in total. The performance counters measure all events taking place on the CPU, so the event counts from the fastest of the five executions were chosen to minimise the interference of other processes and the operating system. In all cases the program measured was running for at least 99.5% of the elapsed time.

[3] One can see the two effects clearly in the "Instr" and "D2" lines; we will return to this point in detail in Section 5.

[4] Although according to [1], "the performance counters are not guaranteed to be fully accurate".



Instr:	0.8 × Retired instructions (event 0xc0)
D1:	12 × Data cache refills from L2 (event 0x42)
D2:	206 × Data cache refills from system (event 0x43)
Branch:	10 × Retired branches mispredicted (event 0xc5)
Sum:	Instr + D1 + D2 + Branch
Actual:	Actual cycle count

Figure 4: Effects of varying nursery size (I)

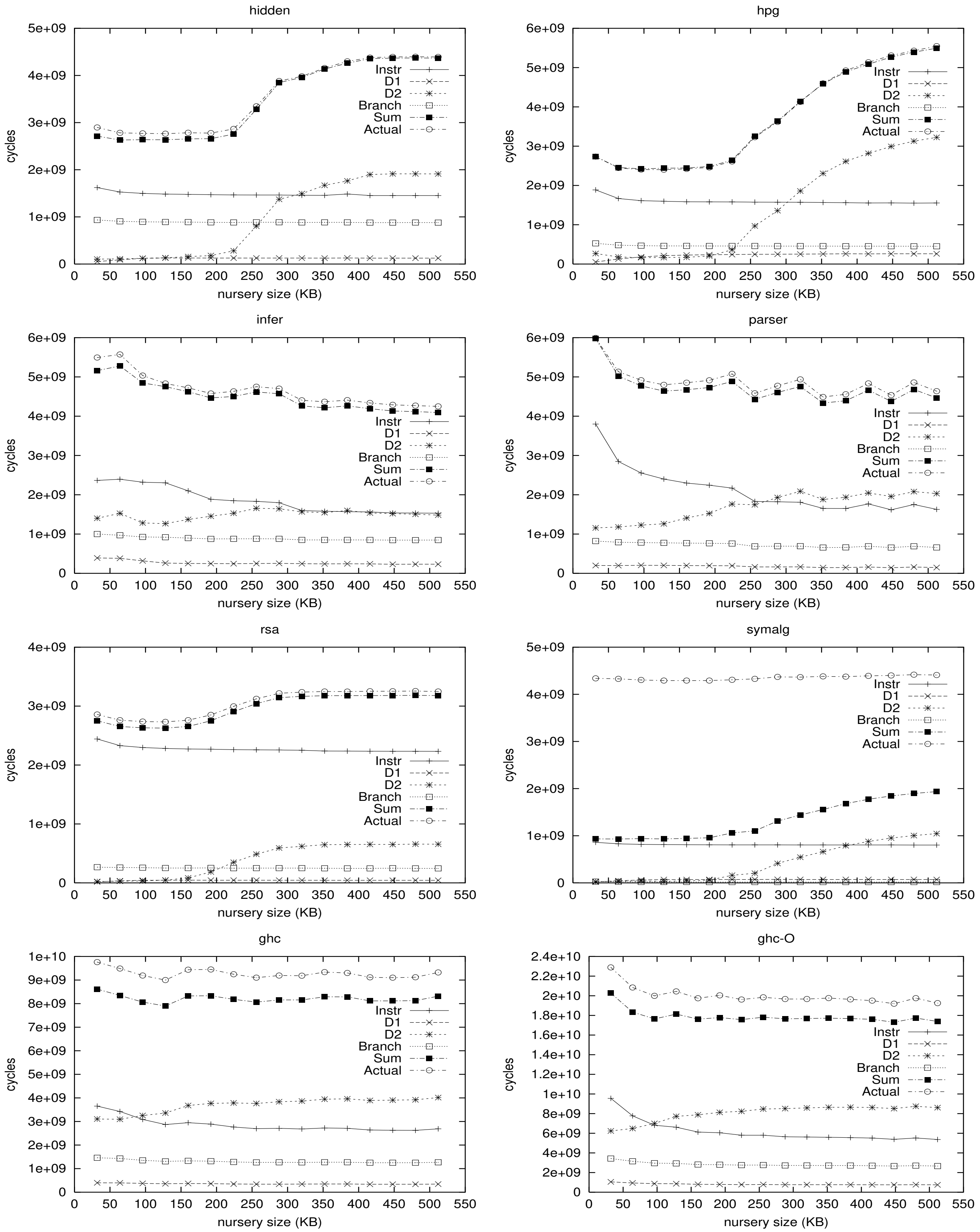


Figure 5: Effects of varying nursery size (II)

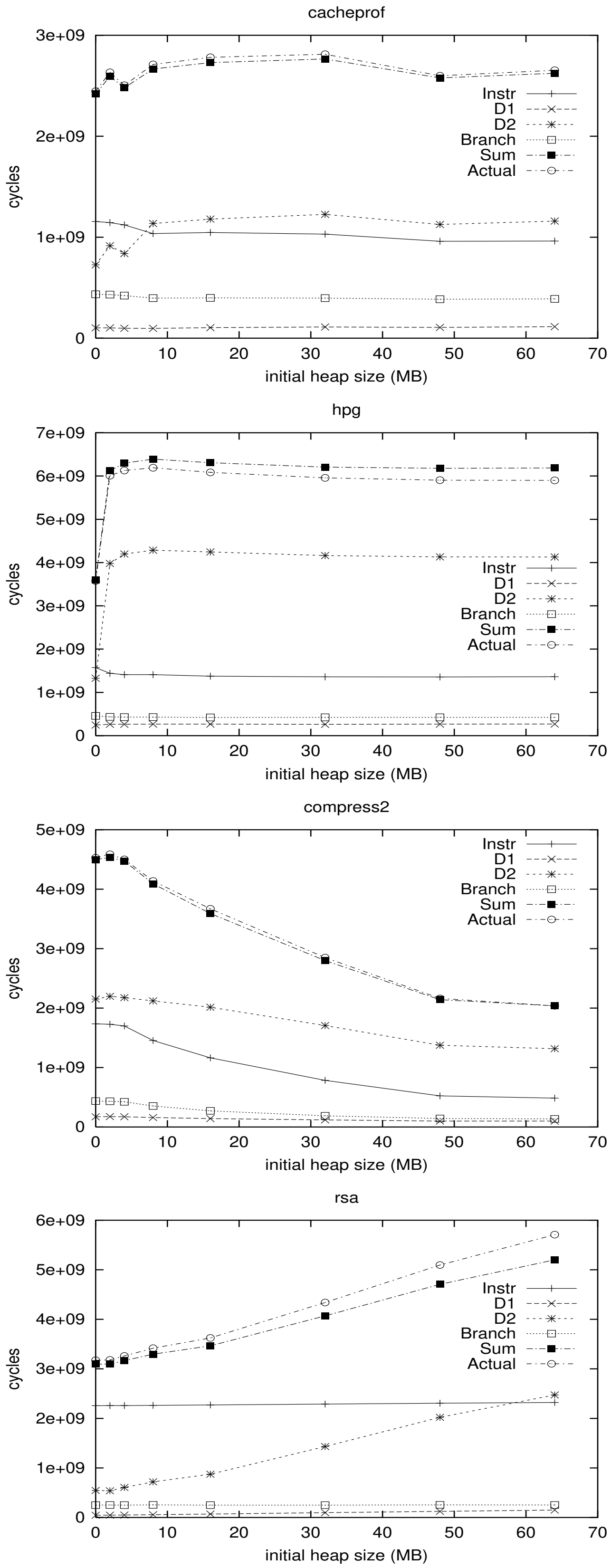


Figure 6: Effects of varying initial heap size

The GHC programs were run using the garbage collector configuration that gave the fastest times, found by varying the nursery size from 32KB–512KB in 32KB increments and varying the initial heap size from 0MB–32MB in 4MB increments (the number of generations was always two).

To provide a comparison with the GHC programs, five SML/NJ (v110.0.7) programs and five C programs[5] were measured as well. They are described in Figure 7.[6]

SML Program	Description
`count-graphs`	Graph manipulation
`logic`	Simple Prolog-like interpreter
`ray`	Ray tracer
`simple`	Spherical fluid dynamics simulation
`tsp`	Travelling salesperson
C Program	
`gcc`	C compiler, v3.0.4
`gzip`	LZ77 compression, v1.3
`latex`	LaTeX2ε/TeXv3.14159 typesetter
`perlparse`	Assembly code parser (in Perl v5.6.0)
`vpr`	FPGA placement and routing tool

Figure 7: SML and C program descriptions

The results for all programs are given in Figure 8, which contains a key explaining the columns. The execution figures (columns 4–5, 9–12) were obtained with the hardware counters and the garbage collector figures (columns 6–8) with the `-s` runtime option.

The immediate conclusions are that GHC programs and SML/NJ programs are memory intensive, with higher instruction / memory access ratios than the C programs. This is because they both use copying garbage collection, and allocate memory furiously (58–287MB per second of mutator time for the Haskell programs, even higher for SML/NJ).

But the CPI figures are most compelling—the GHC programs range from 1.0–4.3, with most in the 1.5–3.0 range, compared to 1.2–1.6 for SML/NJ programs, and 1.1–1.5 for the C programs. We will see in Section 5.3 that the GHC programs' high L2 data miss and branch misprediction rates account for much of this difference.

5. AN EXECUTION COST MODEL

To understand and explain the results found in the previous section, we used a simple execution cost model to determine where processor time is going in the GHC programs.

5.1 The Model

The model takes into account only the four largest components of execution time: instruction execution, stalls due to L1 and L2 cache data misses, and stalls due to branch mispredictions. The model is:

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

where I is the number of instructions executed, C_1 and C_2 are the number of D1 cache misses and L2 cache data misses respectively, and B is the number of mispredicted branches.

[5]The program `perlparse` is written in Perl, but the Perl interpreter is written in C.

[6]Two of the C programs are used in the SPEC benchmarks; the inputs we used were not from the SPEC benchmarks, but chosen to give similar running times to the Haskell programs.

GHC program	Nurs.	Heap_0	Instr	Mem	GC	Alloc'd	Copied	L2	Sys	BrMis	CPI
`anna`	96KB	0MB	1598M	86%	23%	133MB	35MB	7.9	2.0	58	2.1
`cacheprof`	160KB	0MB	1484M	72%	33%	222MB	33MB	5.7	1.2	30	1.4
`compress`	160KB	0MB	3301M	83%	30%	532MB	65MB	5.2	0.8	24	1.2
`compress2`	96KB	32MB	972M	73%	46%	169MB	42MB	10.1	8.4	20	2.9
`fulsom`	288KB	28MB	717M	82%	28%	183MB	10MB	9.7	6.9	40	2.9
`gamteb`	128KB	0MB	2301M	73%	17%	350MB	25MB	4.1	0.9	27	1.4
`hidden`	128KB	0MB	1857M	85%	3%	535MB	2MB	5.8	0.4	48	1.5
`hpg`	96KB	0MB	2018M	69%	19%	533MB	23MB	7.5	0.4	23	1.2
`infer`	512KB	32MB	1523M	90%	29%	109MB	35MB	15.7	4.0	59	2.6
`parser`	288KB	32MB	1517M	75%	34%	287MB	41MB	7.2	5.9	38	2.6
`rsa`	128KB	0MB	2853M	52%	6%	188MB	3MB	1.3	0.1	9	1.0
`symalg`	192KB	0MB	1013M	49%	1%	410MB	1MB	6.0	0.3	2	4.3
`ghc`	448KB	32MB	2232M	78%	17%	480MB	32MB	9.8	6.7	46	3.1
`ghc -O`	320KB	32MB	4580M	79%	23%	922MB	98MB	11.5	7.7	49	3.4
SML/NJ program											
`count-graphs`	256KB	512KB	9598M	78%	?	3122MB	14MB	6.0	0.3	10	1.2
`logic`	256KB	512KB	1528M	80%	?	534MB	14MB	11.6	1.2	14	1.6
`ray`	256KB	512KB	1003M	64%	?	540MB	347KB	13.2	0.9	12	1.5
`simple`	256KB	512KB	920M	75%	?	269MB	157KB	8.4	1.1	9	1.3
`tsp`	256KB	512KB	1943M	59%	?	25MB	3MB	8.5	1.1	7	1.5
C program											
`gcc`	–	–	2250M	59%	–	–	–	3.6	1.6	21	1.5
`gzip`	–	–	3189M	46%	–	–	–	13.1	0.3	10	1.1
`latex`	–	–	1741M	69%	–	–	–	2.7	0.4	17	1.2
`perlparse`	–	–	1778M	67%	–	–	–	3.7	0.1	27	1.4
`vpr`	–	–	1958M	70%	–	–	–	5.3	0.2	14	1.3

Nurs: Nursery size (KB)
Heap_0: Initial heap size (MB)
Instr: Retired instructions (Athlon event 0xc0)
Mem: Data cache accesses / Retired instructions (0x40 / 0xc0)
GC: Garbage collector time %
Alloc'd: Megabytes allocated on heap
Copied: Megabytes copied during garbage collection
L2: Data cache refills from L2 / Retired instructions × 1000 (0x42 / 0xc0 × 1000)
Sys: Data cache refills from system / Retired instructions × 1000 (0x43 / 0xc0 × 1000)
BrMis: Retired branches mispredicted / Retired instructions × 1000 (0xc5 / 0xc0 × 1000)
CPI: Cycles / instruction retired (cycles / 0xc0)

Figure 8: Program characteristics

The constants were chosen for the following reasons: 12 and 206 for the cache misses because they are the worst-case numbers reported by Calibrator; 10 for branch mispredictions because page 208 of [1] says "In the event of a mispredict, the minimum penalty is ten cycles"; and 0.8 for unstalled instructions quite arbitrarily (a three-way machine with multiple functional units could retire unstalled instructions faster than this).

5.2 Justification

One would expect that this model is far too simple for the Athlon, an aggressive, out-of-order, three-way superscalar processor with nine functional units. And yet, it is surprisingly accurate. Returning to the graphs in Figures 4–6, the four components are shown as the "Instr", "D1", "D2" and "Branch" lines. The "Sum" line is their sum.

Consider the results from varying nursery size in Figures 4 and 5. For eleven of the fourteen programs, the model gives a correctly shaped graph that is almost spot on the real value (e.g. `hpg`) or underestimates by a small constant factor (e.g. `fulsom`). For `compress` the model becomes a little less accurate as the nursery size increase; for `hidden` it becomes more accurate. The graphs in Figure 6 also show an impressive match between "Sum" and "Actual" values.

Only for `symalg` in Figure 5 are the predictions badly wrong. This is because 8.3% of its instructions are `div` instructions—from the GNU multi-precision library which GHC uses to implement infinite precision integers—which take 42 cycles on the Athlon ([1] p. 270). If we assume 0.8 cycles per unstalled instruction for the remaining 91.7%, the average cycles per unstalled instruction jumps to 4.3; if we took this into account the "Sum" line would be much closer to the "Actual" line, although its shape would still be wrong.

What can we deduce from this surprising accuracy?

1. The L2 cache data stall times are believable. If the penalty of 206 cycles was an overestimate or underestimate, the shape of the "Sum" and "Actual" lines would not match so well for the programs in which in the number of L2 misses changes a lot while the other

components do not change very much (e.g. `gamteb`, `hpg`, `rsa`). Little if any useful work is being done during L2 data miss stalls.

2. A similar argument can be made for the figure of 0.8 cycles per unstalled instruction, else the shape of the "Sum" and "Actual" lines would not match so closely for the programs in which the number of instructions executed changes a lot (e.g. `parser`, `compress2` in Figure 6). This component covers instruction execution time plus anything else proportional to the number of instructions, such as instruction fetch stalls.

3. The branch misprediction times should be an accurate lower bound, assuming ten cycles is the minimum misprediction penalty. This component may be underestimated; it is hard to tell due to the the flatness of the "Branch" lines.

4. The D1 cache stall times may be wrong. Unlike the "Branch" case, there is no minimum penalty for a D1 miss. The "D1" line is not likely to be an underestimate, since the 12 cycle figure used is a worst-case latency, the D1 cache has two ports and the processor may be able to do some useful work when an D1 cache miss occurs. Either way, D1 stall times have little impact on overall execution times.

5. For those programs in which the "Sum" line falls short of the "Actual" line, there is a constant "everything else" component that is independent of the number of instructions and L2 cache misses. This could include any underestimation of branch misprediction penalty.

The inaccurate graphs are also worth considering. For `compress`, the model overestimates the cost of an unstalled instruction, underestimates the cost of an L2 cache data miss, or possibly both. For `hidden`, the model overestimates the cost of an L2 cache data miss. For `symalg`, once the cost of the `div` instructions is factored in, the model greatly overestimates the cost of an L2 cache data miss; as a counter-example of a program where the processor can mask the cost of an L2 miss by doing other useful work, it emphasises how little the processor can do for the more normal programs.

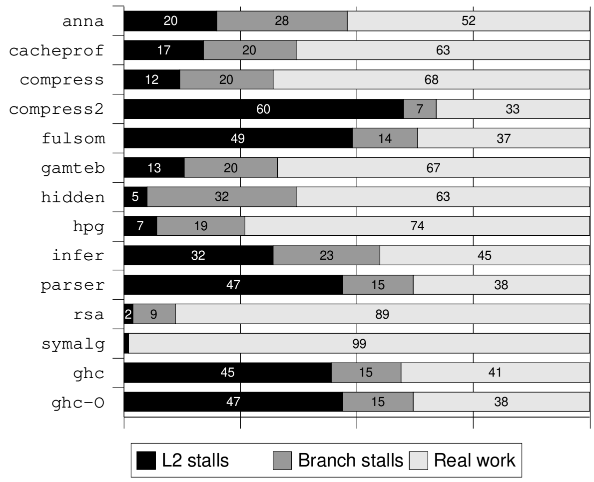
The interested reader may care to re-inspect Figures 4–6 to see how the interaction between instruction counts and cache misses affect program speed for the different garbage collector configurations.

5.3 Using the Model

The model is far from perfect. However, we believe it is accurate enough that we can state with confidence the proportion of execution time taken up by L2 miss and branch misprediction stalls—the stalls that we have found to be significant for GHC programs—just from the counts provided by the hardware counters.

Figure 9 shows the proportion of execution times taken up by L2 miss and branch misprediction stalls for the GHC programs. The numbers were deduced from the execution cost model, using the same optimal nursery and initial heap sizes as in Figure 8.

L2 cache data stalls account for 1–60% of execution time. Not surprisingly, the programs that use more memory tend to have worse cache behaviour. Branch misprediction stalls account for 0–32% of execution time. Together the two kinds



L2 stalls %: 206 × Data cache refills from system / Cycles

Branch stalls %: 10 × Retired branches mispredicted / Cycles

Real work %: 100 − L2 stalls % − Branch stalls %

Figure 9: Execution time lost to hardware stalls

of stall account for 1–67% of time, with most programs in the range 30–60%.

Finally, the "Real work" category accounts for the majority of the time taken by most of these programs. Although L2 cache miss and branch misprediction stalls are a big factor in the high CPI rates of these programs, this "Real work" category should not be ignored; a CPI of around 0.8 for unstalled instructions (as shown by model) is not very good. If that figure could be improved the programs would all run much faster.

5.4 Explanation

Why is the model so accurate? What has happened to the Athlon's aggressive three-way, out-of-order execution?

It is not clear what the exact effects are. But we have some ideas. Firstly, part of the high cache miss rate is due to GHC's intense memory use, thanks to its high allocation rates and use of copying garbage collection. But the cache miss rates seen in Figure 8 compare unfavourably to the SML/NJ programs which use memory in a similar way. It may be that laziness is a large factor. Intuitively, lazy programs may tend to "hang onto" data longer than for strict languages, since program execution does not proceed in a direct fashion.

Secondly, for the high branch misprediction rates, many jumps at the end of closures are indirect, and their target addresses change frequently, so the branch prediction units have little chance of correctly predicting them. GHC uses a technique commonly used to support lazy evaluation, that of jumping into a closure's code to determine if it is evaluated, and returning immediately if so. If GHC, for example, instead used one of the spare two bits on the end of word-aligned pointers to indicate whether a closure has been evaluated, these expensive jumps might be able to be replaced by a better predicted local direct jump in the case where a closure has been evaluated.

Finally, the level of instruction-level parallelism in GHC program code exploitable by modern processors is quite low; a high proportion of instructions touch memory, many of which simply move data between the heap and the stack. Values loaded are usually needed immediately, and while these frequent loads and stores are happening there is little "real" work that can be done by the processor's functional units. This explains the high CPI values for the programs, and also why L2 cache misses stalls are not "masked" at all.

All these factors mean that GHC programs do not interact well with the aggressive modern processors, and that changes to GHC's execution mechanism may improve its performance.

6. SIMULATION

Performance counters are very useful for finding cache miss ratios, and the execution cost model let us determine the proportion of time caused by hardware stalls. But to improve the cache behaviour we need information about where the misses occur. This required the use of software simulation.

6.1 Method

Execution-driven simulation is one popular technique. We used the tool Valgrind [18] as the starting point for our simulation. Valgrind's core is a JIT compiler for x86/Linux programs; it uses a RISC-like intermediate language, which provides an excellent platform for simulation. We extended Valgrind to perform cache profiling, naming the resulting tool "Cachegrind".

Cachegrind instruments each memory referencing instruction with a call to a C function that updates the simulated caches (I1, D1 and unified L2). It also collects cache access and miss counts for each instruction and prints them to file, which allows line-by-line annotation of the program's source files.[7] Programs run around 50 times slower than normal under Cachegrind.

6.2 Shortcomings

Cachegrind's simulation suffers several shortcomings.

- It only measures cache accesses visible at the program level. For example, it does not account for extra cache accesses that occur upon a TLB miss, nor cache accesses that take place under instructions executed speculatively that are later annulled.

- The addresses used for the simulation are virtual; it does not consider virtual-to-physical address mappings at all. The simulated cache state will not exactly match the real cache state. This is hard to avoid; even the extremely rigorous Alpha 21264 simulation of Desikan, Burger and Keckler [5] does not model it.

- It does not model the Athlon's cache semantics exactly. For example, upon an instruction cache miss the Athlon loads the missed block and also prefetches the following block; this is not accounted for.

[7]The "precise event-based sampling" of the Pentium 4 [11] can provide line-level detail. Unfortunately, we do not have access to any Pentium 4 machines and have not been able to try this.

We quantified these differences with a direct comparison of the results from hardware counters and software simulation. Figure 10 shows the results.

Program	Instr	Ref	D1 miss	D2 miss
`anna`	99.4%	71.6%	92.7%	80.2%
`cacheprof`	99.5%	82.6%	92.5%	69.5%
`compress`	99.4%	84.0%	97.0%	64.5%
`compress2`	99.4%	83.3%	94.7%	94.1%
`fulsom`	99.5%	78.3%	89.8%	72.8%
`gamteb`	99.2%	81.8%	91.5%	71.0%
`hidden`	99.2%	74.5%	94.6%	54.3%
`hpg`	97.4%	80.8%	98.1%	38.2%
`infer`	99.0%	76.3%	96.0%	90.6%
`parser`	99.3%	81.1%	89.2%	81.5%
`rsa`	99.4%	91.4%	97.3%	83.1%
`symalg`	99.2%	96.8%	96.8%	45.7%
`ghc`	99.4%	78.6%	87.3%	88.0%
`ghc -O`	100.0%	78.3%	86.6%	81.2%

Figure 10: Ratios between hardware counter and software simulation event counts

Column two gives the ratio of instructions counted by Cachegrind to retired instructions counted by Rabbit (event 0xc0). This is the best comparison of the two techniques, as they are measuring exactly the same event. As expected, Cachegrind gave marginally lower counts than Rabbit, because unlike Rabbit it does not measure other processes and the kernel. Despite this, Cachegrind counted more than 99% of the events counted by Rabbit for all programs except `hpg`, for which it counted 97.4%.

Column three contains the memory reference ratios, where Rabbit measures the number of data cache accesses (event 0x40). Cachegrind falls further short here, by 3–28%. As previously mentioned, this is because some cache accesses are occurring that are not visible at the program level, such as those from TLB misses.

Columns four and five give the D1 cache miss and L2 cache data miss ratios, where Rabbit is measuring the number of data cache refills from L2 and data cache refills from system (Athlon events 0x42, 0x43). Cachegrind underestimates these misses by 3–62%.

With this simulation, we are only aiming for a general picture of where cache misses occur for machines with this kind of setup, rather than matching exactly every miss that occurs for the Athlon. We believe that although Cachegrind's results are not perfect, they give a very good indication of where cache misses are occurring in these programs.

6.3 Annotations

We used Cachegrind to annotate each program line[8] with its number of read and write references and misses, when run with the optimal garbage collector configuration as before.

We concentrated on L2 data misses because data misses are much more frequent than instruction misses, and L2 misses are much more costly than L1 misses. Most data misses are concentrated in certain places; the proportions of L2 data misses in different locations are shown in Figure 11 and explained in the following sections. The first seven parts of each bar are read misses, and marked with "(r)". The

[8]At the assembler level for the compiled GHC code, and the C level for the runtime system.

next three are write misses, marked with "(w)". The last part is the proportion of read and write misses unannotated (the GNU multi-precision library code was not annotated, which explains the high unannotated proportions for `rsa` and `symalg`). We will distinguish between "in-program" misses and those in the runtime system.

6.4 Data Read Miss Locations

Data read misses occur in seven main ways.

1. In-program '`mov`' read misses: occur mostly when reading from the stack, and reading closure fields.

2. In-program '`jmp`' read misses: call/return instructions are never used in the STG machine. All code blocks end with a direct or indirect jump to the code of the next closure to be executed. The first kind of `jmp` that causes many read misses is an indirect jump to another closure's code, of the form `jmp *(%esi)`. Register `%esi` points to a closure C, and this instruction jumps to the code in the info table pointed to by C. If C is not in the cache at that point, a miss occurs.

3. The second `jmp` that causes read misses, with the form `jmp *-k(%eax)`, is an indirect jumps for a vectored return. `-k` is an offset into a static vector of return addresses (called a *vtbl*) pointed to by register `%eax`. When a data constructor is evaluated, in some circumstances its code returns to the appropriate member of a vector of return addresses rather than returning to a multi-way jump. If the vtbl is not in the cache, a read miss occurs.

4. `evacuate()`: during garbage collection, before a closure is *evacuated* from the old step to the new step its layout must be determined from its info table. This requires two accesses: one of the closure to get its info table pointer and one of the info table layout information. The first access causes around ten times as many misses as the second.

5. `copy()`: `evacuate()` calls `copy()` to do the copying of each closure to the next step. Usually `evacuate()` will have dragged the entire closure into the cache, but if the closure straddles a cache line boundary, copying the second half can cause a read miss.

6. `scavenge*()`: various *scavenge* functions are used to evacuate any closures pointed to by an evacuated closure. When a closure is scavenged it will cause a read miss if it is not already in the cache.

7. Other read misses are scattered about, mostly in the runtime system.

The number of read misses in `evacuate()` is high. This may be caused by putting layout data directly next to code in info tables. Info tables are read-only, so there is never any coherency problems if a single block is placed in both the I1 and D1 caches; however, the split caches are polluted by useless words. In particular, when garbage collecting, the info table of every closure that is evacuated is read. Since the code part of an info table is often much larger than the data part (e.g. 10 or more instructions versus 2 or 3 words of data), this might knock out much data from the D1 cache that would soon be referenced. We plan to try separating the data and code parts of info tables (a form of *structure splitting* [3]). Instruction level miss identification will be invaluable for measuring the effects of this change.

6.5 Data Write Miss Locations

Data write misses occur in three main ways.

1. In-program '`mov`' write misses: most write misses occur when allocating and initialising new closures on the heap. Because of the sequential allocation, most writes hit the cache; misses only occur when a cache line boundary is crossed. Allocation of an N byte closure causes a miss N times out of 64 (the block size is 64 bytes).

2. `copy()`: during garbage collection.

3. Other write misses are in the runtime system, and are only significant for `gamteb`, `rsa` and `symalg`; for these three most of the write misses occur in a function `stgAllocForGMP()` which allocates memory required by the GNU multi-precision library.

This distribution reflects how writes occur—most happen at allocations, as only a fraction of closures survive to be garbage collected.

6.6 Avoiding Data Write Misses

Most write misses are unnecessary. Heap writes are sequential, both when initialising closures, and when copying them during garbage collection. Write misses occur only for the first word in a cache line. There is no need to read the memory block into the D1 cache upon a write miss, as is done in a write-allocate cache; the rest of the line will soon be overwritten. It would be better to write the word directly to the D1 cache and invalidate the rest of the cache line.

This can be achieved by using a write-allocate cache with sub-block placement, as noted by Diwan *et al.* [6]. However, such caches are now a rarity. An equally effective approach would be to use a write-invalidate instruction instead of a normal write. Some architectures have write-invalidate instructions, but unfortunately the x86 is not one of them.

An alternative is to use prefetching. Because writes are sequential it is simple to insert prefetches to ensure memory blocks are in the cache by the time they are written to. We performed some preliminary experiments with the Athlon's `prefetchw` instruction, fetching ahead 64 bytes each time a new closure is allocated or copied by the garbage collector. The changes required were simple, and increased code sizes by only 0.8–1.6%. Figure 12 shows the results: columns 2–4 give the improvement when the prefetching is applied to just the garbage collector, just program allocations, and both. The improvements are quite respectable: programs ran up to 22% faster, and none slowed down when prefetches were added to in-program code and the runtime system.

If a write-invalidate instruction existed that cost no more than a normal write, we can estimate the potential speed-up it would provide by multiplying the proportion of write misses by the proportion of execution time taken up by L2 data cache stalls (from Figure 9), which gives the expected speed-ups shown in column 5 of Figure 12. The prefetching technique—which is applicable to any program using copying garbage collection, not just GHC programs—obtained half or more of this theoretical figure for almost all programs, as shown by column 6 which gives the ratio between

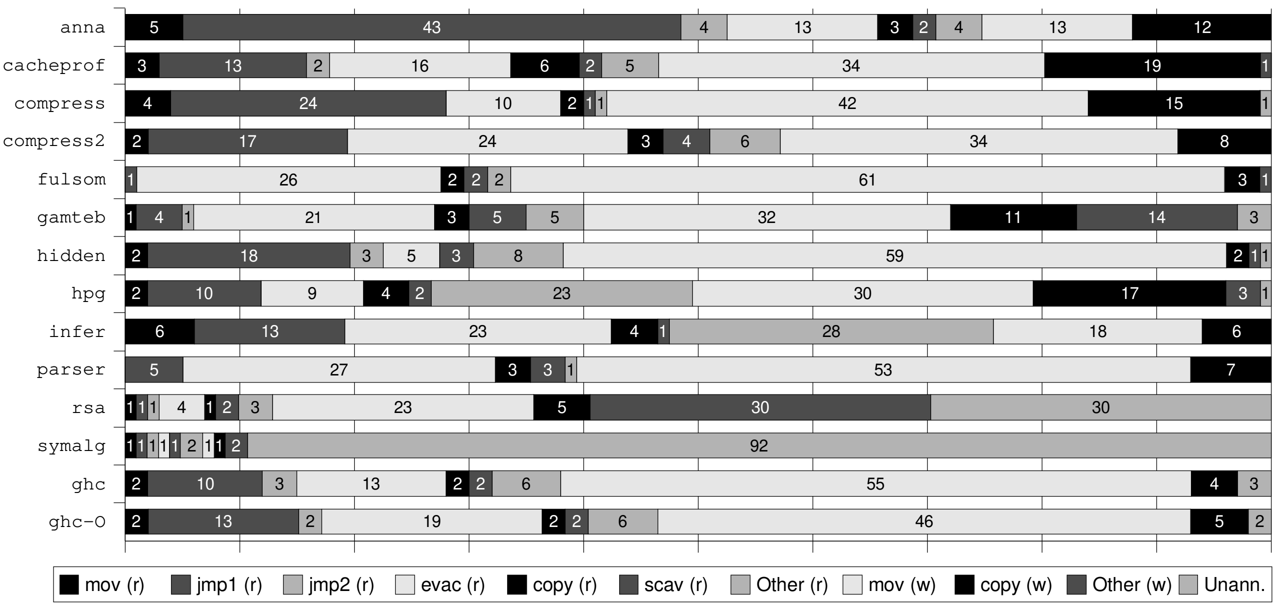


Figure 11: Read and write miss locations

Program	GC	Prog	Both	Theory	Ratio
`anna`	3%	0%	4%	5%	0.8
`cacheprof`	3%	1%	5%	9%	0.6
`compress`	2%	2%	5%	7%	0.7
`compress2`	3%	9%	12%	25%	0.5
`fulsom`	1%	17%	17%	31%	0.6
`gamteb`	0%	-0%	0%	6%	0.0
`hidden`	1%	3%	2%	3%	0.7
`hpg`	4%	1%	4%	3%	1.3
`infer`	3%	8%	9%	8%	1.1
`parser`	1%	19%	22%	28%	0.8
`rsa`	3%	-1%	2%	$< 1\%$	
`symalg`	1%	1%	1%	$< 1\%$	
`ghc`	1%	17%	17%	27%	0.6
`ghc -O`	2%	12%	14%	24%	0.6

Figure 12: Effect of prefetching

theoretical and actual speed-ups. This is pleasing since more prefetches are performed than necessary (one per closure allocated/copied, about six per cacheline), and prefetching increases memory bandwidth requirements.

7. RELATED WORK

Several other works have been published about the cache behaviour of declarative languages, some of which considered the effects of generational copying garbage collection.

Wilson, Lam and Moher measured a bytecode Scheme implementation [19] that used a copying generational garbage collector, and simulated multiple cache configurations. They concluded that fitting the allocation area in cache would help locality greatly.

Koopman, Lee and Siewiorek [12] evaluated various cache configurations for an SK-combinator graph reduction language. Examining very small programs, they found that write-allocate caches gave much better performance than write-no-allocate caches, because most data is referenced almost immediately after allocation.

Diwan, Tarditi and Moss made very detailed simulation of Standard ML programs, including the effects of parts of the memory system usually ignored such as the write buffer and TLB [6]. They compared different write-miss strategies, and found that using sub-block placement cut cache miss rates significantly.

Gonçalves and Appel also made detailed measurements of Standard ML programs [8]. They found the miss rates of SML/NJ programs could be lower than SPEC92 C and Fortran programs. Necula and George also measured SML/NJ programs [14], on a DEC Alpha with performance counters. They found that stalls caused by data cache misses accounted for 24% of execution time.

The main differences between this work and previous work is that we have considered large programs in a lazy language, we have identified cache misses down to the level of individual instructions, we have used prefetching to avoid cache misses, and we have also considered branch mispredictions and instruction-level parallelism.

8. FURTHER WORK AND CONCLUSION

8.1 Data Write Misses

We found that write misses typically account for 50–60% of L2 cache data misses in GHC programs. Using very simple prefetching, we mitigated the cost of cache write misses by around half, improving the speed of GHC programs by up to 22%. With a write-invalidate instruction, we could potentially obtain greater speed-ups.

8.2 Data Read Misses

The remaining data misses are read misses. Firstly, removing GHC's use of code next to data may improve cache behaviour by avoiding polluting the data caches with useless instructions during garbage collection.

After that there are two general ways to reduce data cache misses. The first is to improve program locality, which can reduce read misses. At first, we hoped to use Chilimbi and Larus' technique of using low-overhead real-time profiling information to guide data reorganisation during garbage collection [4]. Unfortunately, closure access in GHC programs is extremely lightweight, and the real-time profiling that worked for Java and Cecil programs would be too expensive. It is not clear whether this technique can be modified for languages with such lightweight data access, e.g. by sampling only a small fraction of closure accesses.

Also, many data accesses in GHC programs are to static structures—some closures, and all info tables and vtbls. Perhaps these structures can be laid out in a way that aids the locality of programs, using static analysis and/or heuristics.

The second general approach to avoiding cache misses (both read and write) is to reduce memory footprints by representing data more compactly. Old techniques invented to avoid page faults could be recycled. For example, a big-bag-of-pages (BIBOP) scheme [2], where the heap is segregated into different areas by type could help; a single info table pointer could be shared between all closures in the page. This would, for example, shrink a cons cell from three words to two. Putting same-typed closures together might also improve locality. The extra costs are that it requires multiple heap pointers, and it requires some kind of test to determine if a closure is in a special page or not. It may also be tricky to determine how closures should be distributed between pages.

8.3 Other Hardware Optimisations

Although this work began as an investigation into the cache behaviour of Haskell, we found that branch misprediction stalls account for up to 32% of program time, and that there is little exploitable instruction-level parallelism exhibited by closure code. GHC's execution mechanism, the STG machine, was designed over ten years ago, when processors were substantially different. It is time to reconsider techniques used in GHC programs (such as always entering a closure, and immediately returning if it is already evaluated) in the light of the changing strengths and weaknesses of modern hardware, as the potential for further significant performance improvements are excellent.

9. ACKNOWLEDGMENTS

Many thanks to: Julian Seward for all his help with Valgrind and GHC, Simon Marlow for help with GHC, Simon Peyton Jones and the anonymous referees for valuable comments on earlier versions of this paper, Don Heller for help with Rabbit, Ian Pratt for his hardware expertise, and Dave Scott for help with SML. The first author gratefully acknowledges the financial support of Trinity College, Cambridge.

10. REFERENCES

[1] Advanced Micro Devices, Inc. AMD Athlon processor x86 code optimization guide, July 2001. `http://www.amd.com`.

[2] H. G. Baker. Optimizing allocation and garbage collection of spaces. In Winston and Brown, editors, *Artificial Intelligence, An MIT Perspective*, volume 2, pages 391–396. MIT Press, 1979.

[3] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *Proceedings of the SIGPLAN '99 Conference on Programming Languages Design and Implementation (PLDI)*, pages 13–24, Atlanta, Georgia, USA, May 1999.

[4] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of ISMM-98*, pages 37–48, Vancouver, Canada, Oct. 1998. ACM Press.

[5] R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of ISCA-28*, pages 266–277, July 2001.

[6] A. Diwan, D. Tarditi, and E. Moss. Memory-system performance of programs with intensive heap allocation. *ACM Transactions on Computer Systems*, 13(3):244–273, Aug. 1995.

[7] The Glasgow Haskell Compiler. `http://www.haskell.org/ghc`.

[8] M. J. R. Gonçalves and A. W. Appel. Cache performance of fast-allocating programs. In *Proceedings of FPCA'95*, pages 293–305, La Jolla, California, USA, June 1995. ACM Press.

[9] P. H. Hartel, *et al.* Benchmarking implementations of functional languages with "Pseudoknot" a float-intensive benchmark. *Journal of Functional Programming*, 6(4):621–655, 1996.

[10] D. Heller. Rabbit: A performance counters library for Intel/AMD processors and Linux. `http://www.scl.ameslab.gov/Projects/Rabbit/`.

[11] Intel. IA-32 Intel architecture software developer's manual, 2001. Order number 245472. `http://www.intel.com`.

[12] P. J. Koopman, Jr., P. Lee, and D. P. Siewiorek. Cache behavior of combinator graph reduction. *ACM Transactions on Programming Languages and Systems*, 14(2):265–297, Apr. 1992.

[13] S. Manegold and P. Boncz. Cache-memory and TLB calibration tool. `http://www.cwi.nl/~manegold/Calibrator/`.

[14] G. C. Necula and L. George. Accounting for the performance of Standard ML on the DEC Alpha. Technical report, AT&T Bell Labs, Murray Hill, New Jersey, USA, Sept. 1994.

[15] W. Partain. The `nofib` benchmark suite of Haskell programs. In J. Launchbury and P. Sansom, editors, *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202, Ayr, Scotland, July 1992. Springer-Verlag.

[16] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, Apr. 1992.

[17] S. L. Peyton Jones and A. L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, Sept. 1998.

[18] J. Seward. Valgrind, an open-source memory debugger for x86-GNU/Linux. `http://developer.kde.org/~jseward/`.

[19] P. R. Wilson, M. S. Lam, and T. G. Moher. Caching considerations for generational garbage collection. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 32–42, San Francisco, California, USA, June 1992.