Cache Performance of Lazy Functional Programs on Current Hardware

Arbob Ahmad and Henry DeYoung
{adahmad, hdeyoung}@cs.cmu.edu

November 30, 2009

Abstract

Due to garbage collection and language features that preclude stack-based allocation, functional programming languages, such as Haskell, may stress the memory system in unusual ways. In 2002, Nethercote and Mycroft studied the cache behavior of Haskell programs and observed large numbers of L2 cache misses. We reconsider these results in light of recent changes in hardware, especially L2 cache size. Using performance counter measurements and cache simulations, we find a significant decrease in L2 misses and demonstrate that this is primarily due to a larger L2 cache. In addition, we show that simple software prefetching in the copying garbage collector does not further improve performance.

1 Introduction

To cope with increasing complexity of software, programmers require languages with higher-level abstractions. The functional programming paradigm has been proposed as one such means of increasing abstraction. However, functional languages, such as Haskell [16] and Standard ML [12], have very different memory allocation behaviors than conventional imperative languages due to a typical absence of array-based data structures and prevalence of pointer-based data structures, partially applied functions, copying garbage collection, and more intense heap usage.

This difference in allocation behavior means that functional programs may not fit modern architectures as easily as their imperative cousins: architectures assume certain allocation behaviors to improve performance for those cases. For example, cache design and hardware prefetching assumes that temporally proximate memory accesses are often for spatially proximate locations (or, at least, ones related by a regular stride), an assumption that is not necessarily true for functional programs.

In 2002, Nethercote and Mycroft [13] investigated the effect of these differences in allocation behavior on cache performance for programs written in Haskell, using the Glasgow Haskell Compiler (GHC) [17], on an AMD Athlon Model 4 ("Thunderbird") processor. Using performance counters, they found that L2 cache data miss stalls accounted for up to 60% of the execution time. After pinpointing the locations of frequent cache misses by simulation, they inserted software prefetching instructions, and found that this simple optimization improved performance by up to 22%.

However, hardware has changed quite dramatically since these results were reported. In particular, L2 cache sizes have increased significantly and stock processors are now virtually all multicore chips, in contrast with the single core chips used in 2002. Therefore, we would like to determine whether these changes have affected the cache performance of GHC programs and the usefulness of a simple software prefetching scheme to mitigate the cache misses in these programs.

To this end, we reproduced and extended Nethercote and Mycroft's experiments on a modern Intel

Core 2 Quad 6600 processor. We found that L2 cache misses now account for at most 20% in the worst case (Section 3). The difference between this bound and the 60% result reported by Nethercote and Mycroft is primarily due to architectural changes and not changes to GHC (Section 4). In addition, a simple software prefetching scheme is not effective in reducing the number of cache misses (Section 5). These results form the key contribution of this work: evidence that programs written in a functional style can perform reasonably well on current hardware.

We chose to continue the focus on GHC programs for several reasons. First and foremost, it allows us to use Nethercote and Mycroft's results as a basis for comparison. Second, Haskell is a language with lazy evaluation. Because of its indirect execution behavior, lazy functional languages make even heavier use of the memory system than their eager counterparts. Finally, Haskell has been used in commercial projects [1], making its cache behavior of more than simply academic interest.

We close this introduction with a brief overview of the most closely related work.

Related Work. In addition to the aforementioned work by Nethercote and Mycroft, several other research groups have studied the cache behavior of functional programming languages, albeit from slightly different angles.

In 1992, Koopman *et al.* [8] examined the effect of cache properties on combinator graph reduction, a technique for implementing functional programming languages with lazy evaluation (e.g., Haskell). By performing sensitivity analysis on the cache size, block size, associativity, and write policy (write-through, write-back, or write no-allocate), they found that a write-back cache with moderate block size and subblock placement was best because of an unusually high proportion of writes in the reductions.

In 1995, Gonçalves and Appel [6] studied cache performance of ML programs compiled by version 0.93 of the Standard ML of New Jersey (SML/NJ) compiler. In contrast with combinator graph reduction, ML uses eager, rather than lazy, evaluation. Despite this fundamental difference, the conclusions were quite similar to those of Koopman *et al.*: Using a MIPS instruction-level simulator, Gonçalves and Appel determined that caches which allocate, but do not fetch, on write misses yielded the best performance. They also found that over half of all reads were to recently allocated data, meaning that it is even more important to minimize the write miss penalty since the subsequent reads often prevent write misses from being hidden. These simulation results were then confirmed by direct experimentation on a DEC3000 Alpha machine.

Concurrently with the work by Gonçalves and Appel, Diwan *et al.* [5] also simulated memory-system performance of SML/NJ programs. In contrast, however, Diwan *et al.* simulated the full memory system, including write buffers and page-mode writes, to get a truly accurate picture of performance. In addition, they rejected the emphasis in earlier work on the miss rate as a performance metric, and instead relied on the number of cycles per instruction. Again, the recommendation was for a cache with a write allocate policy and subblock placement with a one word subblock size.

None of these quite capture our goals. Only two groups studied languages with lazy evaluation [13, 8], whereas the others looked at a language with eager evaluation [6, 5]. Most importantly, all of these are quite dated with respect to the hardware used, making it difficult to extrapolate for conclusions about *current* cache performance.

2 Experimental Methodology

2.1 Architecture

For all experiments and simulations in this work, we used an off-the-shelf Intel Core 2 Quad machine running Fedora 7 with kernel version 2.6.31.5. The detailed properties of its clock rate and caches are given in Table 1. With the exception of replace times, these were determined by examining `/proc/cpuinfo` and the `/sys/devices/system/cpu` directory. Worst-case replace times were estimated as the modes of 5 trials using the Calibrator v0.9e [10] micro-benchmark with a clock rate of 2400MHz and

Intel Core 2 Quad Q6600	
Clock Rate	1.6GHz to 2.4GHz with dynamic frequency scaling
D1 Cache	32KB, 64B line size, 8-way associative, per core, 11 cycle replace time
I1 Cache	32KB, 64B line size, 8-way associative, per core
L2 Cache	4MB, 64B line size, 16-way associative, per 2 cores, 179 cycle replace time

Table 1: Properties of the architecture used

AMD Athlon Model 4 ("Thunderbird")	
Clock Rate	1.4GHz
D1 Cache	64KB, 64B line size, 2-way associative, 12 cycle replace time
I1 Cache	64KB, 64B line size, 2-way associative
L2 Cache	256KB, 64B line size, 8-way associative, 206 cycle replace time

Table 2: Properties of the architecture used by Nethercote and Mycroft

memory usage of 512MB.

It is important to note that our architecture differs from the one used 8 years ago by Nethercote and Mycroft; the properties of their architecture are summarized in Table 2. Most significant is the disparity in L2 cache size: rather than a 256KB L2 cache, we now have two 4MB L2 caches! Even once we acknowledge that each of our L2 caches is shared among 2 cores, this is still a dramatic difference. We return to this point in Section 4.

2.2 Benchmarks

Following Nethercote and Mycroft [13], we made use of Haskell's `nofib` benchmark suite [15] in our experiments. `Nofib` contains numerous benchmarks ranging from toy, "imaginary" programs to "real" programs originally designed for non-benchmark purposes. To employ a different and slightly larger series of tests, we chose to extend the set of benchmarks used by Nethercote and Mycroft with `maillist`, `pic`, and `reptile` (which are also from the `nofib` suite). The benchmarks that we used are given in Table 3; all were taken from the "real" subset of `nofib` and were compiled with GHC 6.10.1 using the default youngest generation, or nursery, size (256KB), initial heap size (0MB), and number of generations (2). They are single-threaded programs, which means that the switch to multiple cores should have little direct effect on their performance.

Program	Description
`anna`	Frontier-based strictness analyzer
`cacheprof`	x86 assembly code annotator
`compress2`	LZW text compression
`compress`	Text compression
`fulsom`	Solid modeler
`gamteb`	Monte Carlo photon transport
`hidden`	PostScript polygon renderer
`hpg`	Random Haskell program generator
`infer`	Hindley-Milner type inference
`maillist`	Mailing list generator
`parser`	Partial Haskell parser
`pic`	Particle in a cell
`reptile`	Escher tiling
`rsa`	RSA encryption
`symalg`	Symbolic algebra

Table 3: `nofib` benchmarks used in experiments

The `nofib` benchmark inputs were modified so that each run (without profiling or instrumentation) took a wall clock time as measured by `/usr/bin/time` of between 2 and 3 seconds. (These bounds were set by Nethercote and Mycroft so that cache simulation would not take an unbearable amount of time.) To make the comparison with their work precise and fair, we contacted Nicholas Nethercote to determine what method they used for creating inputs. Since he could not recall, we adopted the following scheme: In most cases, we simply replicated the input file provided with `nofib` until the running time was within the bounds. Because the inputs to `fulsom`, `gamteb`, `hpg`, and `pic` were simply a set of parameters, their values were manually adjusted until the 2–3 second running time was achieved.

This scheme for generating inputs is admittedly

quite naïve. It is possible that replicated inputs might affect the cache behavior in ways not seen in genuine usage. However, we believe that any such effects are likely small since the `nofib` benchmarks do not appear to use memoization.

2.3 Performance Counter Software

Modern processors are equipped with special hardware performance counters that can track events such as clock cycles and cache misses. To keep overhead low and minimize the effect of profiling on the running program, these counters are not measured directly. Instead, a sample is taken every N^{th} event, where N is the sample rate chosen by the user, and the instruction responsible for that event is reported. Despite this, we can recover (an estimate of) the total number of events by multiplying the number of samples and the sample rate.

Using this method, we would like to reproduce the performance counter experiments of Nethercote and Mycroft to measure the total number of cache miss events, etc. However, because the Rabbit performance counters library [7] that they used does not support the Core 2 architecture of our machine, we had to rely on a different software package for measuring the performance counters.

A first attempt was to use OProfile [9]. This was initially attractive because it is available on facilitized machines. However, after using it for preliminary tests, we found that OProfile is geared toward long-term, full system profiling. This means that its indirect workflow and dependence on root privileges was cumbersome for our purpose of profiling shorter, but many, benchmarks. An even more critical problem with OProfile was a series of strange results on toy C programs. Because of reports of its bugginess [11], we decided to move away from OProfile. In hindsight, these strange results were likely due to poor assumptions about the clock and sample rates (see below) and not due to OProfile, but this was not recognized at the time.

At the recommendation of GHC's Simon Marlow [11], we then moved to using the Perf suite [2], built-in to the Linux 2.6.31 kernel, for measuring the performance counters. We still experienced strange results for toy C programs under Perf. However, we eventually realized that we were using a too small sample rate, which Perf was silently clamping at some unknown, preset minimum. Hence, the value obtained by multiplying the number of samples and the assumed sample rate was a wildly inaccurate representation of the total number of events. (In addition, we were incorrectly assuming that the clock rate was the minimum DVFS clock rate, as was reported in `/proc/cpuinfo`.) After correcting our mistake, we found that Perf gave much more reasonable results.

2.4 Cache Simulation Software

In choosing the cache simulator for our experiments, we again followed Nethercote and Mycroft and used Valgrind's [14] Cachegrind tool [13]. Conveniently, Cachegrind was already integrated into the `nofib` test script.

As pointed out by Nethercote and Mycroft, Cachegrind does have several drawbacks. First, it can only simulate program-level cache accesses. It is not possible to account for accesses due to kernel behavior and TLB misses, for example. Second, it does not model hardware prefetching in any form. This can affect cache miss predictions. Third, Cachegrind uses only virtual addresses, rather than the physical addresses that would actually be used in the memory system.

3 Cache Performance of GHC Programs

To measure the baseline cache behavior of GHC programs, we carried out a series of performance counter measurements on our Intel Core 2 Quad 6600 machine, using Perf to sample the clock cycle, instruction, and L1 and L2 cache events. Specifically, for each event, we performed 5 runs each of the 15 `nofib` benchmarks described earlier. The sample rate for each event was chosen as the power of 10 that yielded between 1000 and 10000 samples for that event on all benchmarks. To ensure that the sample rate was not being silently clamped at another value, we varied the sample rate by a factor of 10 in each direction

and verified that the number of samples scaled by (approximately) the corresponding factor of 10.

Using this raw data, we estimated the total number of events for each run by multiplying the number of samples and sample rate. The arithmetic mean number of events, for each event and benchmark, is given in Table 4, along with the number of cycles per instruction (CPI) and cache miss rates as derived metrics. The formulas used to derive these values are also presented in Table 4; monospace font indicates a Perf event name.

3.1 Discussion

As a primary performance metric, we observe that the CPI values are not unreasonable: with the notable exception of the `maillist`, most of the 15 benchmarks have CPI values between 0.5 and 0.75. Most of the benchmarks also have quite low L2 cache miss rates, on the order of 0.1% or 1.0%; only `compress2` and `pic` break this trend. Even the D1 cache miss rates are reasonable, with only `hidden` falling above 10% (and then only slightly).

These results are in stark contrast with the measurements taken by Nethercote and Mycroft. Their experiments yielded a mean CPI of 1.88 for the twelve benchmarks shared with our work, whereas we observe a mean CPI of 0.71. Similarly, they report a mean L2 cache miss rate of 30.9%, whereas we found a mean L2 miss rate of 1.14%.

At this point, we are obligated to make two disclaimers. First, Nethercote and Mycroft performed all of their experiments using per benchmark garbage collector parameters—nursery size, initial heap size, and number of generations—observed to be optimal for that benchmark. We chose not to follow this approach since we are more interested in drawing generally applicable conclusions rather than specific information about the manually tuned performance of these particular benchmarks. However, because this kind of manual tuning could only improve performance, our comparisons with their results remain valid.

Second, Nethercote and Mycroft were able to measure the number of *data* L2 cache misses using the Athlon's events; we could only measure the combined instruction and data L2 cache misses on the Intel Core 2 Quad 6600. However, in our experiments, I1 cache misses accounted for only 7.38% of all L1 cache misses. Given this, the data L2 miss rate could not be more than 1.23%, and so, the large difference between Nethercote and Mycroft's and our L2 cache miss rates still stands.

Having made these disclaimers, we now return to comparisons of our results with those of Nethercote and Mycroft. To expose another distinction between current and 2002 hardware, we can consider the fraction of execution time due to L2 cache miss stalls. Figure 1 shows that a dramatic difference has occurred over the past seven years. Whereas Nethercote and Mycroft found that L2 miss stalls accounted for between 1% and 60% of the execution time (with most in the 10%–30% range), we observe much lower numbers: L2 miss stalls account for between 0.1% and 41% of the execution time (with most in the 4%–8% range).

All of our results point to a significant improvement in cache behavior of GHC programs when comparing current hardware with that of 2002. The next section examines the extent to which a change in cache sizes effected this improvement.

4 Effect of Cache Sizes on Performance Improvement

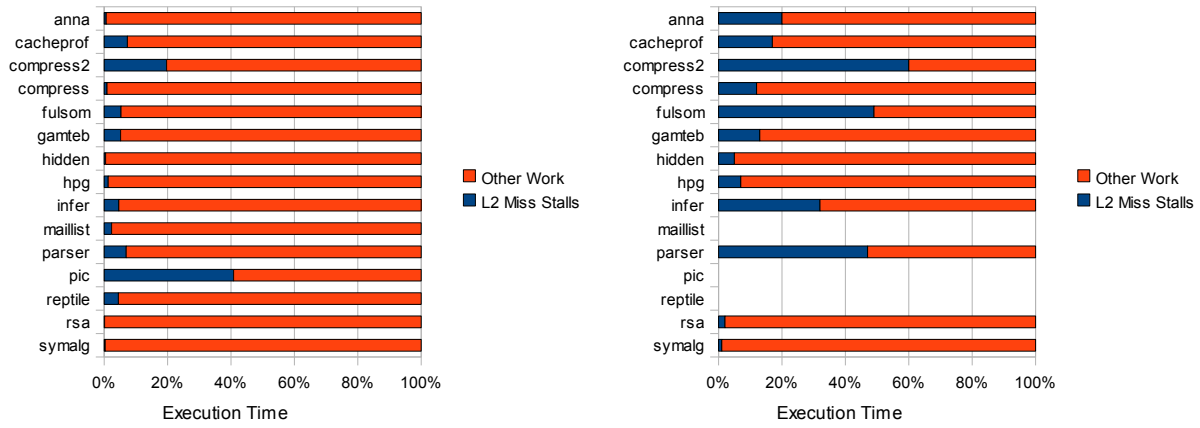
As pointed out earlier, there is a large difference in L2 cache size between Nethercote and Mycroft's AMD Athlon Model 4 machine and our Intel Core 2 Quad 6600 machine: 256KB to 4MB (per 2 cores), respectively. In addition, the L1 caches shrank from 64KB to 32KB. However, many other variables are at play in these experiments, most notably the versions of GHC and `nofib`. It is natural to ask to what extent the change in cache sizes was responsible for the performance improvement observed in the previous section.

Ideally, we would quantify the effect of cache sizes on the observed performance improvement by changing *only* the cache sizes and keeping other parameters, such as the versions of GHC and `nofib`, con-

				D1 Cache			L2 Cache		
Benchmark	Cycles	Instrs	CPI	Refs	Misses	Miss Rate	Refs	Misses	Miss Rate
`anna`	7225M	9858M	0.73	4399M	178M	4.05%	108M	252K	0.23%
`cacheprof`	6488M	8944M	0.73	4109M	326M	7.95%	165M	2656K	1.61%
`compress2`	5884M	8205M	0.72	3905M	283M	7.24%	116M	6466K	5.55%
`compress`	5915M	10295M	0.57	5671M	542M	9.56%	203M	294K	0.14%
`fulsom`	4927M	5286M	0.93	2821M	233M	8.25%	114M	1452K	1.28%
`gamteb`	6373M	8122M	0.78	3927M	222M	5.65%	192M	1852K	0.96%
`hidden`	5904M	7997M	0.74	4040M	422M	10.45%	142M	144K	0.10%
`hpg`	5005M	5067M	0.99	2488M	183M	7.37%	277M	350K	0.13%
`infer`	6076M	11105M	0.55	5467M	340M	6.22%	228M	1578K	0.69%
`maillist`	6627M	4230M	1.57	2220M	196M	8.85%	403M	868K	0.22%
`parser`	5916M	8453M	0.70	3774M	210M	5.56%	111M	2293K	2.07%
`pic`	6642M	7180M	0.92	3142M	162M	5.14%	95M	15138K	15.87%
`reptile`	6187M	9040M	0.68	4273M	335M	7.85%	137M	1554K	1.13%
`rsa`	6154M	10620M	0.58	3765M	73M	1.93%	35M	46K	0.13%
`symalg`	6713M	13441M	0.50	2577M	24M	0.91%	16M	129K	0.82%

Cycles:	`cycles`
Instrs:	`instructions`
CPI:	Cycles / Instrs
D1 Refs:	`L1-dcache-loads` + `L1-dcache-stores`
D1 Misses:	`L1-dcache-load-misses` + `L1-dcache-store-misses`
D1 Miss Rate:	D1 Misses / D1 Refs
L2 Refs:	`cache-references`
L2 Misses:	`cache-misses`
L2 Miss Rate:	L2 Misses / L2 Refs

Table 4: Mean total cycle, instruction, D1 cache and L2 cache events and derived metrics



(a) Intel Core 2 Quad 6600

(b) AMD Athlon Model 4 (from [13])

Figure 1: Fraction of execution time due to L2 data miss stalls.

stant. Unfortunately, we do not have access to a machine that matches the Athlon's cache sizes but is otherwise identical to our Core 2 architecture and compilation environment. For this reason, we must rely on simulation via Cachegrind.

First, we describe simulations using cache settings that match the Intel Core 2 Quad 6600. We show that these simulated results serve as a reasonable, though certainly not perfect, approximation to the performance counter measurements from Section 3. (The simulations have an added benefit of functioning as a sanity check on the performance counter results.)

Second, we simulate an AMD Athlon Model 4 machine's cache behavior in Cachegrind. We then contrast Cachegrind's predictions for the Core 2 and Athlon. Only because we will have shown that the simulated Core 2 approximates the actual Core 2 can we use the simulated Core 2 and simulated Athlon to draw conclusions about the *actual* cache behavior of these differing architectures.

4.1 Simulated Intel Core 2 Quad 6600

To simulate the Intel Core 2 Quad 6600 machine's cache behavior, each benchmark was run once under Cachegrind with the appropriate parameters taken from Table 1. Rather than giving the raw results from these simulations, we choose to present in Table 5 the ratios of Cachegrind predictions to the performance counter measurements. Note that there are no entries in this table for the number of cycles. This is because Cachegrind simulates only the cache behavior, not the full system, and, consequently, cannot report a predicted execution time.

One might expect that Cachegrind and Perf would report *exactly* the same number of instructions because, in principle, Cachegrind sees every instruction in the binary. In fact, our experiments and simulations show that Cachegrind predicts nearly all or slightly more instructions than Perf measures (see Instrs column in Table 5); `cacheprof` and `hpg` are noticeable outliers to this trend. These deviations from the expected ratio of 100.00% can be explained in two ways.

First, in contrast with other dynamic binary instrumentation frameworks which typically use a more direct "copy-and-annotate" method, Valgrind disassembles the binary into a RISC-like intermediate representation (IR), instruments it with analysis code using a tool plug-in, e.g., Cachegrind, and resynthesizes a binary from the instrumented IR [14]. The RISC-like IR is likely to contain slightly more instructions than the underlying CISC assembly code. Since Cachegrind only has access to the IR, this may account for the small deviations above 100.00% that were observed.

Second, the small deviations (averaging only 0.845%, excluding `hpg`) below 100.00% may be due to Cachegrind's inability to simulate the kernel and other programs' instructions. Because Perf's measurements are made for the system as a whole, these additional sources of instructions may explain why Cachegrind reports fewer instructions, even though it is using a larger RISC-like IR instruction stream.[1]

Cachegrind predicts between 51.7% and 96.3% of the D1 cache references. With the exception of `maillist` (which they did not test), these values are all comparable to the 71.6%–96.8% range of values observed by Nethercote and Mycroft. As they point out, the fact that Cachegrind does not predict all of the measured D1 cache references can likely be attributed to Cachegrind's inability to simulate non-program-level accesses. A similar story can be told for L2 cache references: Cachegrind predicts between 30.0% and 89.8% of the L2 cache references measured by Perf.

The ratios of Cachegrind predictions to Perf measurements for L2 cache misses is somewhat troublesome. There is a very large range of values: from 19.6% to 682.1%. We conjecture that these unusual ratios may be due to Cachegrind's lack of simulation support for hardware prefetching. By not supporting prefetching, Cachegrind may incorrectly count as misses those accesses which would have been satisfied by prefetched blocks. If prefetching is performing well for a benchmark, then Cachegrind would overcount the number of misses. To test this hypothesis, we attempted to turn off hardware prefetching on our machine via BIOS settings and the manufacturer spe-

[1] Another, weaker explanation is that Perf may be undercounting the number of instructions by using a too infrequent sample rate.

		D1 Cache		L2 Cache	
Benchmark	Instrs	Refs	Misses	Refs	Misses
`anna`	100.08%	87.12%	51.33%	88.05%	88.49%
`cacheprof`	107.43%	93.46%	35.35%	78.11%	452.26%
`compress2`	98.49%	89.98%	33.69%	83.26%	427.22%
`compress`	100.63%	91.59%	29.01%	79.49%	19.57%
`fulsom`	98.76%	79.77%	28.16%	72.12%	653.11%
`gamteb`	99.54%	87.58%	33.65%	68.98%	425.92%
`hidden`	101.41%	86.56%	25.79%	85.54%	35.53%
`hpg`	84.45%	74.35%	34.54%	30.04%	347.63%
`infer`	100.26%	91.97%	58.27%	88.32%	682.10%
`maillist`	64.90%	51.73%	18.33%	29.19%	10.50%
`parser`	102.91%	88.12%	41.84%	86.83%	593.31%
`pic`	98.73%	88.26%	48.68%	83.48%	178.50%
`reptile`	100.25%	88.24%	33.22%	85.63%	888.67%
`rsa`	100.80%	86.73%	38.86%	89.76%	90.54%
`symalg`	99.83%	96.33%	44.71%	70.78%	46.76%

Table 5: Ratios of Cachegrind predictions to performance counter measurements

cific register. Unfortunately, we were unsuccessful in getting this to work. Perhaps a rigorous evaluation of this hypothesis would serve as an interesting opportunity for future work.

Thus, Cachegrind serves as a reasonable, though certainly not perfect, approximation to actual cache behavior on our Intel Core 2 Quad 6600 machine. Once again, this permits us to use a simulated Intel Core 2 Quad 6600 and a simulated AMD Athlon Model 4 to draw conclusions about *actual* cache behavior of these differing architectures. We now turn to this problem.

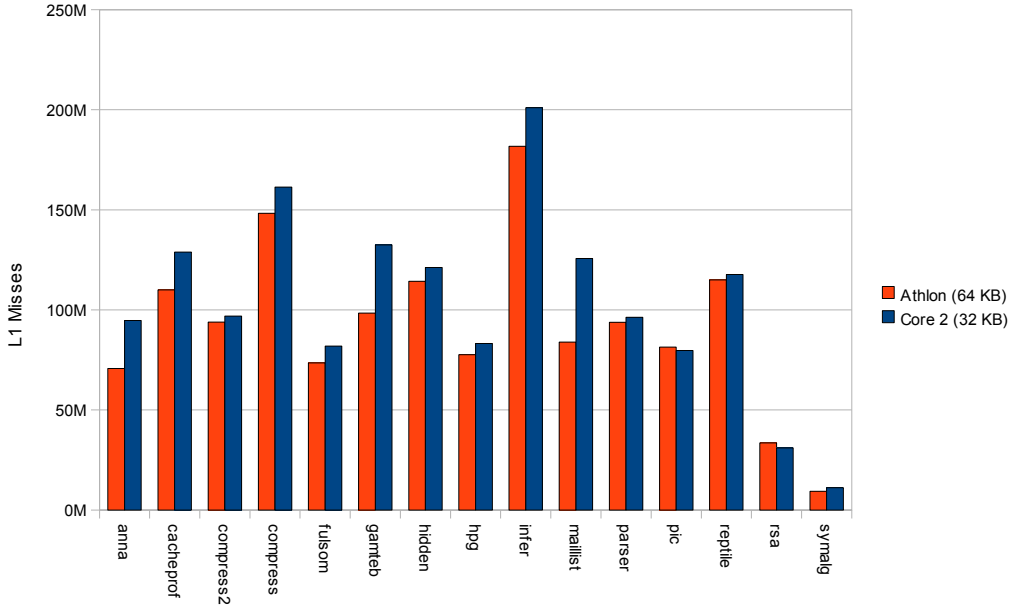
4.2 Comparison with Simulated AMD Athlon Model 4

Next, we repeated the Cachegrind simulations with settings matching Nethercote and Mycroft's AMD Athlon Model 4 machine (see Table 2). Again, each of the 15 benchmarks was run once. A comparison of the simulated Core 2 and simulated Athlon cache misses is given in Figure 2.

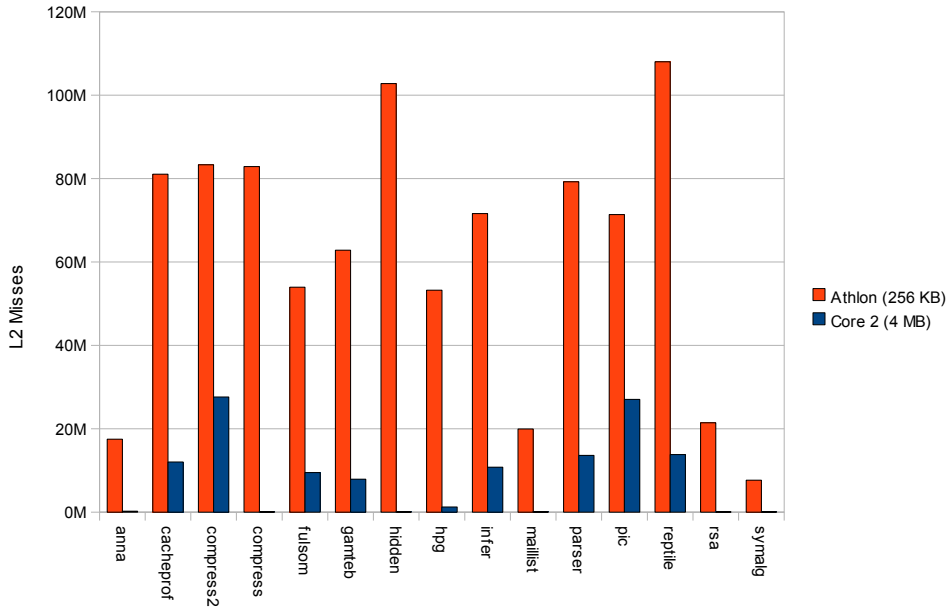
As expected for smaller L1 caches, the L1 cache misses were more numerous for most benchmarks (`pic` and `rsa` being the sole exceptions) under the Core 2's 32KB L1 caches than the Athlon's 64KB L1 caches. However, the difference is not especially striking. Moreover, if L1 cache size was the determining factor for performance, then Figure 2a would suggest that performance should be worse under the Core 2 machine than the Athlon machine. This is clearly contradicted by the results from Section 3; there must be another explanation for the improvement we observed.

In contrast with L1 cache misses, the difference between the Athlon's 256KB L2 cache and the Core 2's 4MB L2 cache appears to have an extremely large effect on the number of L2 cache misses: the number of misses decreases by a factor of at least 2.5 in all 15 benchmarks, with most decreasing by at least a factor of 5. If the versions of GHC and `nofib` used were the primary factors responsible for the performance improvement observed in Section 3, then we would expect Figure 2b to show roughly equal numbers of L2 misses for the simulated Core 2 and Athlon.

As a result, we conclude that the change in L2 cache size is responsible for most of the performance improvement. In the context of Scheme programs,



(a) Combined I1 and D1 cache misses



(b) L2 cache misses

Figure 2: Cachegrind predictions for L1 and L2 cache misses under Core 2 and Athlon cache configurations.

Wilson *et al.* [18] have shown that good cache performance of copying generational garbage collection depends on the relative sizes of the youngest generation and (second-level) cache. If the cache is too small to hold the youngest generation, then capacity misses are likely when new data are allocated. Based on these observations, we believe that this may explain why the larger L2 cache improved performance: the default 256KB youngest generation just barely fits the 256KB L2 cache, but easily fits the 4MB L2 cache. A next step, which we leave for future work, is to formally verify this claim by running experiments with varying youngest generation sizes.

5 Adding Software Prefetching to the GHC Runtime System

Nethercote and Mycroft were able to mitigate L2 cache misses by inserting software prefetches to anticipate key memory accesses. This simple method yielded a 22% improvement in performance for their experiments. Given the reduced number of L2 misses that we observed for GHC programs on current hardware, it is natural to ask whether a similar software prefetching scheme would still provide benefits.

Using the profiling data collected by Perf during our performance counting measurements, we were able to select candidate accesses for prefetching. (To a lesser extent, Cachegrind's simulation-based profiling was also useful.) Consistent with the observations of Nethercote and Mycroft, we found that, for all benchmarks except `hidden` and `pic`, a function named `evacuate()` is the single highest source of L2 cache misses (33.96% on average, in fact). This function, a portion of which is shown in Figure 3, is part of the GHC runtime system's multigenerational copying garbage collection scheme, and is called for every live object in the system. Because GHC uses intensive heap allocation, this translates to an extremely large number of calls to `evacuate()`. Thus, it is a prime candidate in which to add software prefetches.

By further examining the assembly code for `evacuate()` which was annotated by Perf's profiling mechanism, we pinpointed most of the L2 cache

```
REGPARM1 GNUC_ATTR_HOT void
evacuate(StgClosure **p)
{
  StgClosure *q;
  const StgInfoTable *info;
  StgWord tag;

  q = *p;
  // __builtin_prefetch(get_itbl(q));

loop:
  tag = GET_CLOSURE_TAG(q);
  q = UNTAG_CLOSURE(q);

  if (!HEAP_ALLOCED_GC(q)) {
    if (!major_gc) return;

    info = get_itbl(q);
    switch (info->type) {
```

Figure 3: C code fragment in GHC's runtime system responsible for 33.96% of L2 cache misses. A potential prefetch is indicated in the code's comment.

misses as arising from pointer chasing and dereferencing to copy the live objects. Specifically, dereferencing `q` (implicitly via macros) and `info` in the code from Figure 3 proved to be costly.

As an attempt at avoiding L2 cache misses due to dereferencing `info`, we inserted a prefetch for this location using `__builtin_prefetch(get_itbl(q))`, as shown in the commented code in Figure 3. We also inserted prefetches for the memory at location `get_itbl(q)` each time `q` was updated for the next iteration of the "loop" defined by the label `loop`.

Because the `get_itbl` macro dereferences its argument, there is no benefit in prefetching `q` also. Moreover, the closure header pointed to by `q` does not contain the info table directly; it only contains a *pointer* to the info table. For this reason, we cannot prefetch only `q`.

After rebuilding GHC's runtime system to incorporate this prefetching scheme, we repeated our earlier performance counter experiments on the Intel Core 2 Quad 6600 for the number of clock cycles and L2 cache misses. Quite disappointingly, the num-

Benchmark	Speedup	L2 Misses as Percent of Original
`anna`	96%	44.03%
`cacheprof`	93%	106.64%
`compress2`	75%	137.06%
`compress`	94%	31.15%
`fulsom`	93%	99.15%
`gamteb`	94%	103.92%
`hidden`	95%	40.65%
`hpg`	92%	108.50%
`infer`	94%	87.95%
`maillist`	98%	99.15%
`parser`	92%	104.28%
`pic`	90%	130.14%
`reptile`	95%	94.87%
`rsa`	99%	85.91%
`symalg`	99%	87.00%

Table 6: "Speedups" and L2 misses (as a percent of baseline) due to the software prefetching shown in Figure 3.

ber of cycles actually increased. Table 6 shows the "speedups" (or rather, slowdowns) experienced by the 15 benchmarks after adding prefetching; on average, each benchmark slowed down to 93.2% of its speed without prefetching.

We can attempt to analyze these slowdowns by giving in Table 6 the number of L2 misses as a percent of the baseline (cf. Table 4). For the `compress2`, `pic`, `hpg`, `cacheprof`, `parser`, and `gamteb` benchmarks, prefetching has apparently increased the number of L2 misses, possibly because of evictions. Given an increase in L2 misses, it is not surprising that we observe a decrease in performance here.

For the `anna`, `compress`, and `hidden` benchmarks, prefetching does seem to have dramatically decreased the number of L2 misses. However, as seen in Figure 1a, these benchmarks also happen to be ones where L2 misses already represent a *very* small fraction of the total execution time. We therefore conjecture that slowdown occurs here because the overhead of prefetching outweighs its benefits when there are few misses. Since they too already have a small number of L2 misses, this may also be the cause of degraded performance for the `fulsom`, `infer`, `maillist`, `reptile`, `rsa`, and `symalg` benchmarks.

Because we prefetched the memory most responsible for L2 cache misses, based on these results, it seems unlikely that other placements of other, more sophisticated prefetches could yield improved performance. Rigorously testing this hypothesis might be an interesting avenue for future work.

6 Conclusion

Using hardware performance counter measurements, we have demonstrated that GHC programs have much more reasonable cache performance, reflected in terms of both cycles per instruction and L2 miss rates, on current hardware than on 2002 hardware. From Cachegrind simulation results, we deduced that the improved cache performance primarily owes to an increase in L2 cache size from 256KB to 4MB. Finally, we have shown that prefetching frequently accessed info tables in fact degrades performance (although it does decrease L2 misses in some cases).

We now close with two suggestions of possible future work and a discussion of the surprises and lessons learned during this project.

Future Work. In the context of the object-oriented language Cecil, Chilimbi and Larus [4] demonstrated the utility of low overhead, real-time profiling of data access patterns in improving cache performance via intelligent data placement during copying garbage collection. It is natural to ask whether similar real-time profiling might benefit functional programs compiled by GHC as well. This approach would require the profiling overhead to be pushed as low as possible since the lightweight closure accesses found in GHC cannot hide a large overhead.

Chilimbi *et al.* [3] used structure reorganization methods, such as splitting structures into frequently and infrequently accessed portions, to great success for Java programs. It might also be useful to perform some kind of structure reorganization for GHC's closures. For example, if applied selectively, removing indirection by placing info tables directly in the closures (i.e., structure merging) might improve locality.

Surprises and Lessons Learned. The most surprising results we encountered were the very large variation between performance counter measurements and Cachegrind predictions of the number of L2 cache misses. We wish that we could have disabled hardware prefetching for further comparison of Cachegrind and performance counters. We experienced first-hand the limitations of simulation, and now more clearly understand the need for both simulation and direct experimentation.

Also, it was unexpected that prefetching info tables in `evacuate()` would actually lead to a decrease in performance; given the smaller number of L2 misses on current hardware, we did not expect large gains, but observing slowdowns was nonetheless somewhat surprising.

By far, the most time consuming part of this project was "debugging" our initial performance counter results (cf. Section 2.3). This easily cost us a week and a half. From this, we re-learned the importance of good documentation and usage troubleshooting tips: OProfile's documentation is rather thin and Perf's is nearly nonexistent. While this lesson was not applicable in our work for this project, we hope that it will serve as a useful reminder when designing software in our research.

Most importantly, we learned how dramatically changes in hardware can affect performance of programs written in unconventional programming paradigms. It is our hope that research into the subtle interactions between architectures and these paradigms will someday free developers from the choice between abstraction and performance.

Acknowledgments

We sincerely wish to thank Simon Marlow and Simon Peyton-Jones for helping us to understand why GHC does not export source-level debugging symbols, Nicholas Nethercote for advice on creating larger `nofib` inputs, and Maynard Johnson and Will Cohen for answering questions about OProfile's sample rates and call graph feature.

References

[1] Haskell in industry. `http://www.haskell.org/haskellwiki/Haskell_in_industry`. Also see *Commercial Users of Functional Programming Workshop (CUFP)* at `http://cufp.galois.com/`.

[2] Performance counters for Linux, v8. `http://lkml.org/lkml/2009/6/6/149`, June 2009.

[3] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI '99)*, pages 13–24, May 1999.

[4] Trishul M. Chilimbi and James R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the International Symposium on Memory Management (ISMM)*, October 1998.

[5] Amer Diwan, David Tarditi, and Eliot Moss. Memory-system performance of programs with intensive heap allocation. *ACM Transactions on Computer Systems*, 13(3):244–273, August 1995.

[6] Marcelo J. R. Gonçalves and Andrew W. Appel. Cache performance of fast-allocating programs. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, pages 293–305, June 1995.

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

[8] Philip J. Koopman, Peter Lee, and Daniel P. Siewiorek. Cache behavior of combinator graph reduction. *ACM Transactions on Programming Languages and Systems*, 14(2):265–297, April 1992.

[9] John Levon and Philippe Elie. OProfile: A system profiler for Linux. `http://oprofile.sourceforge.net/`.

[10] Stefan Manegold and Peter Boncz. Cache-memory and TLB calibration tool: Calibrator v0.9e. `http://www.cwi.nl/~manegold/Calibrator/`.

[11] Simon Marlow. Personal communication, November 2009.

[12] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. The MIT Press, May 1997.

[13] Nicholas Nethercote and Alan Mycroft. The cache behaviour of large lazy functional programs on stock hardware. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance (MSP 2002)*, 2002.

[14] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, pages 89–100, June 2007.

[15] Will Partain. The `nofib` benchmark suite of haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202, 1992.

[16] Simon Peyton-Jones. Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):1–255, 2003.

[17] GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.10.4*, July 2009. Available at `http://www.haskell.org/ghc/docs/latest/users_guide.pdf`. Distribution available at `http://www.haskell.org/ghc/`.

[18] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching considerations for generational garbage collection. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 32–42, 1992.

Distribution of Total Credit

The total credit should be distributed equally among the group members.