Cache Behaviour of Lazy Functional Programs

(working paper)

Koen Langendoen Dirk-Jan Agterkamp

University of Amsterdam,
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
e-mail: *koen@fwi.uva.nl*

Abstract. To deepen our quantitative understanding of the performance of lazy evaluation, we have studied the cache behaviour of a benchmark of functional programs. The compiler, based on the G-machine style of graph reduction, has been modified to insert monitoring code into the executable that records instruction and data references at run time. The resulting address trace is used to drive a cache simulator that computes statistics like miss rates and traffic ratios.

A number of experiments with different cache parameters (size, associativity, etc.) shows that the benchmark programs have a strong *spatial* locality in their memory references. This is caused by the heap allocation strategy that allocates nodes by advancing a pointer through the heap, generating new addresses. Therefore the initialisation of new heap nodes results in cache misses, which dominate performance. Comparisons with results of other functional language implementations confirm this behaviour

1 Introduction

Recently, compilers for lazy functional languages have become capable of generating object code whose quality approaches that of imperative languages. For example, in [Smetsers91] it is reported that several CLEAN programs execute within a factor three of the time needed by the corresponding C programs. The remarkable progress in performance of lazy languages can be largely attributed to strictness analysis, which yields information about expressions that may be evaluated eagerly in advance. The replacement of call-by-need with call-by-value for strict arguments saves the unnecessary construction/invocation of delayed computations in the heap and consequently improves performance. Many other compiler optimisations focus on reducing the usage of graph nodes as well, but still the memory consumption of functional programs is orders of magnitude greater than their imperative counterparts.

To further improve state-of-the-art implementations it is vital to have detailed knowledge about the memory reference behaviour of functional programs. It is no longer possible to judge the effectiveness of some specific compiler optimisation by, for example, measuring "the nfib rating" or counting the number of claimed heap cells as in [Hartel91b]. Such coarse measures do not provide enough information to achieve maximal performance on today's computing systems that heavily depend on the effectiveness of their cache memories. Several research groups have realised the need for refined performance measures as can be seen from recent publications about advanced profiling tools for lazy functional languages [Sansom92, Runciman92].

Not only compiler writers need quantitative feed-back to improve their implementations, but also hardware designers and system architects. The RISC revolution has shown that measurements of the behaviour of real applications are necessary to reveal the most common operations that need to be executed as fast as possible. Although it is unlikely that special hardware will be produced to support functional language implementations given the small market place and commercial failures in the past (e.g., NORMA [Scheevel86]), it is viable to compose a computing system out of standard chips selected to match the requirements of functional programs as best as possible. Since functional programs are memory bound, the choice of a specific cache configuration has a strong effect on the overall performance. This paper studies the cache behaviour of lazy functional programs, and discusses the design trade-offs of various cache parameters.

The performance of cache memories in a functional programming environment has been measured before in [Poon85, Koopman92], but these studies are based on interpretive execution (SECD machine code [Landin64] and combinator reduction [Turner79]). The measurements reported in this paper are gathered from benchmark programs compiled to native machine code. The compiler uses an abstract machine model similar to the G-machine style of graph reduction [Johnsson84]. Since the compiled code uses much less (temporary) heap space than the interpretative versions, we expect differences in cache behaviour. Section 4 contains a comparison of the two implementation techniques, and also addresses the differences with measurements taken from imperative programming languages.

2 Trace driven simulation

To measure the memory reference behaviour of lazy functional programs, we have made use of a cache simulator that is driven by an address trace and computes statistics like miss rate and traffic ratios. The address traces have been obtained by modifying the code generator of our compiler to insert monitoring code into the executable to record all memory references at run time. Various aspects of our trace driven simulation approach, including the benchmark programs, are elaborated on in the following sections.

2.1 Compiled graph reduction

The FAST compiler [Hartel91a], which has been developed at Southampton University in the UK, translates lazy functional programs into super-combinators and outputs code for an abstract graph reduction machine similar to the G-machine used in the LML compiler from Chalmers University [Johnsson84]. The FCG code generator [Langendoen92] maps the abstract machine code onto low level assembly instructions and controls register allocation, code scheduling, etc. The combined FAST/FCG compiler generates code whose quality compares well with other state-of-the-art compilers for lazy functional languages: it runs 2 to 3 times faster than LML code, and equals code of the CLEAN compiler.

Part of the success is the strictness analyser employed in the FAST compiler. It is used to translate the default call-by-need strategy into the more efficient call-by-value for strict arguments to avoid interpreting the graph as much as possible. Boxing analysis is another important means to improve the code quality. It is used to determine which objects can be allocated efficiently on the call stack or in registers (in unboxed form), and which are to be stored in the heap (in boxed form).

Significant effort has been undertaken to limit space usage. For example, variable length nodes (VAPs) are used to hold spines of binary application nodes and user-defined data structures. To support the efficient allocation of such variable length nodes, a two-space copying garbage collector [Cheney70] is used to compact the heap whenever the application runs out of free space. Each variable length node begins with a one-word header that contains several fields to describe the length, type, etc. of the node. Part of the type tag information is encoded in the least significant bits of the pointer to the node so that the graph reducer can often determine that a node is in head normal form without dereferencing the pointer. Details about this space efficient data representation scheme can be found in [Langendoen92].

The FCG code generator does not generate native assembly code directly, but (ab)uses C as a sophisticated assembly language for two reasons: 1) portability, 2) register allocation and code optimisations. In contrast to other functional language compilers that use the "generate-C" method [Peyton Jones91, Schulte91], FCG does not use the standard C function call mechanism. Instead it generates code that maintains an explicit call stack to bring all pointers under control of the two-space copying garbage collector. Thus FCG generates one large `main()` function that encompasses the code of all functions defined by the user in the original source program. This set-up automatically causes the C compiler to produce globally optimised code, for example, frequently accessed pointers like the stack- and heap-pointer are allocated in registers.

2.2 Address trace generation

The FCG code generator generates low level (assembly like) C-code and explicitly generates statements to load/store data objects one word at the time. Examination of the machine code generated by the C compiler confirmed that there is a direct relation between the C-statements and the underlying machine instructions. Because of this direct mapping it is possible to have the code generator annotate the C-code output with function calls that record the *data* references in a trace file at runtime. The tracing overhead does not have any effect on the behaviour of the functional program because the tracing instructions themselves are not traced.

Functions for tracing *instruction* fetches can not be inserted directly by the FCG code generator since the C compiler controls the generation of the final machine code. Therefore a small analysis program is used that first compiles the C-assembly to machine code, locates the basic blocks, counts their lengths, and then annotates the original C-assembly with function calls to trace the instruction fetches. This analysis is possible because the FCG code generator unambiguously defines the basic blocks with C-labels and C-gotos.

A functional program can thus be especially compiled to include code for address trace generation. To capture all memory references made on behalf of the user program, the object code is linked with a runtime support system that has been annotated by hand with trace functions. It is important to collect those references too, since the two-space copying garbage collector changes the location of the graph nodes being accessed by the user program. In addition the garbage collector may take a significant amount of the total execution time if the available heap space is small, so its behaviour must not be neglected.

2.3 Cache simulator

We have used the existing dineroIII cache simulator [Hill85], which models an associative cache that is arranged as a collection of sets holding a number of cache blocks (also known as cache lines), see Figure 1. The address of a memory location uniquely defines the set in which the memory word might reside by a hashing scheme. An associative search of all blocks in the set is performed to find the block that holds the word. If the referenced word is not in the cache, a cache miss occurs: the block that holds the word is fetched from main memory, while some block is purged from that referenced set to create room for the new block.

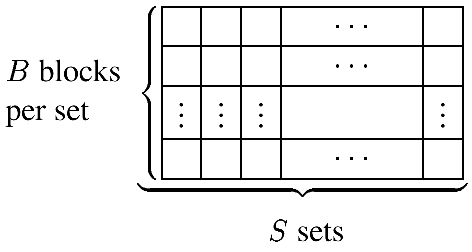


Figure 1: Cache organisation.

The dineroIII simulator can model a wide variety of cache designs by means of command line parameters that select the appropriate cache size, block size, associativity, replacement policy, etc. When processing an address trace, the simulator counts the number of instructions, data references (both reads and writes), words written back to main memory, etc. Finally the simulator computes the miss rate and traffic ratio of an application, which indicate how successful a specific cache configuration is at reducing access time (low miss ratio) and the main-memory bandwidth consumption (low traffic ratio).

The interpretation of the simulated miss rate and traffic ratios requires some care since other factors influence the execution time as well. For example, when increasing the cache size the cache cycle time increases also because of physical limitations. In general a decrease in miss rate or traffic ratio indicates an increase in performance.

However, since the cycle time increases with discrete steps it can be more efficient to use a smaller cache despite the lower miss rate of a larger, but slower, cache.

2.4 Application benchmark

The cache simulator is driven by traces extracted from a benchmark of lazy functional programs that consists of three familiar toy programs (queens, qsort, and prime) and two medium sized programs (comp-lab and 15-puzzle). Some characteristics of the benchmark programs are listed in Figure 2. The first two columns give the size of the text and data segment of the executable, without tracing code, for a SPARC processor. Note that even the toy programs have large object files because of the inclusion of standard library functions to handle I/O etc.

The input parameters of the applications are selected such that all execution times are around 3 seconds on a SUN 4/260 processor. The amount of heap space for each application has been chosen such that the two-space copying garbage collector is invoked three times. This assures that most of the execution time is spent in graph reduction, not in collecting garbage.

The amount of stack space required by each application does not show surprises, but the number of claimed words in the heap reveals remarkable differences. The *qsort* application

program	description	text (Kb)	data (Kb)	stack (words)	heap (Mb)	heap-usage (words)	live-data (words)
qsort 1000	Sorts a list of 1000 random numbers with the quicksort algorithm.	32	44	3000	5.0	2,265,507	6,180
prime 900	Uses the sieve of Eratosthenes to compute the 900th prime number.	32	44	3603	4.5	2,120,722	4,104
queens 10	A divide and conquer solution to the 10 queens problem [Langendoen91].	32	44	781	0.6	286,467	94
comp-lab	An image processing application that labels all four-connected pixels into objects [Stout87, Embrechts90].	56	44	375	2.3	974,513	43,797
15-puzzle	A branch and bound program to solve the 15-puzzle. The iterative deepening search strategy is used [Glas92].	64	44	3681	3.0	1,467,412	1,329

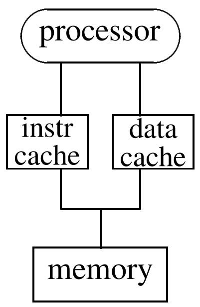
Figure 2: Benchmark characteristics.

allocates new heap cells much faster than the *queens* program. This is caused by the nature of both programs. The quicksort function creates two new lists at each invocation, while the queens function repreatedly extends a (shared) partial solution with a new valid position. The last column lists the maximum amount of live data present after a run of the two-space copying garbage collector. Because of the low number of garbage collects (3), this measure only provides a rough indication for the upper bound of the working set of the applications. The difference in sharing of data structures by the application is also reflected in the memory reference behaviour listed in Figure 3.

Both the *queens* and *comp-lab* programs read considerably more words from the heap than they write, which shows that a number of heap cells is shared because they are read several times. The large consumers of heap space, on the other hand, write more values to the heap than they read. For all programs the number of writes into the heap exceeds the number of allocated words (Figure 2). This is caused by the updates of suspension nodes that hold lazy computations.

program	text	global		stack		heap	
	#instr	#read	#write	#read	#write	#read	#write
qsort 1000	38,442,785 (84.3%)	1,151 (0%)	145 (0%)	1,029,330 (2.3%)	1,027,204 (2.3%)	2,308,579 (5.1%)	2,808,344 (6.2%)
prime 900	38,286,604 (81.7%)	166 (0%)	145 (0%)	1,722,262 (3.7%)	1,720,173 (3.7%)	2,154,387 (4.6%)	2,994,369 (6.4%)
queens 10	74,216,523 (83.6%)	120 (0%)	116 (0%)	5,374,417 (6.1%)	6,030,064 (6.8%)	2,782,680 (3.1%)	372,569 (0.4%)
comp-lab	45,317,893 (84.5%)	11,265 (0%)	141 (0%)	1,753,394 (3.3%)	2,109,840 (3.9%)	2,803,194 (5.2%)	1,634,006 (3.0%)
15-puzzle	52,359,565 (83.4%)	191,944 (0.3%)	149 (0%)	2,926,298 (4.7%)	2,903,912 (4.6%)	2,324,368 (3.7%)	2,104,069 (3.3%)

Figure 3: Memory reference statistics; absolute numbers and percentages.



program	miss rate	traffic ratio
qsort 1000	4.80	0.761
prime 900	4.48	0.929
queens 10	0.28	0.049
comp-lab	5.07	0.771
15-puzzle	1.98	0.328

Figure 4: Base system configuration; miss rate and traffic ratio for data references of benchmark applications.

Despite the difference in the number of executed instructions, all applications issue roughly one data reference every four instructions. This matches well with the numbers for imperative languages such as those reported for C programs in [Hennessy90]. The number of stack read and write operations are about equal, which shows that our FAST/FCG compiler succeeds in saving only live variables across function calls. The number of data references to globally allocated data such as constant strings is negligible. Thus the cache behaviour of the lazy functional applications is mainly determined by the stack and heap references.

3 Cache behaviour

The simulation experiments described in this section only take the data references of the benchmark programs into account. The instruction references can safely be neglected since we assume that the processor is equipped with separate instruction and data caches (Figure 4), hence, nearly all instruction references will be hits because of the good spatial locality in the small text segments. The system configuration in Figure 4 has been used as a base for our experiments, and contains separate instruction and data caches of 64 Kb each (32 byte blocks, direct mapped, copy-back, and fetch on demand).

Instead of presenting an exhaustive search of all possible parameters of the dineroIII cache simulator, we have varied just one cache parameter at the time for simplicity. In addition we only present results of the most "interesting" parameters; previous research has shown that some cache parameters have an optimal setting that consistently outperforms the alternatives. For example, the replacement policy should be LRU [Smith82], and the write-allocate policy should be selected over write-no-allocate [Koopman92]. In the coming experiments we have varied the cache size, the block size (i.e. line size), the update policy, the associativity, and the fetch policy of the standard cache as defined in Figure 4.

3.1 Cache size

In our first experiment, we look at the total size of the cache since it has the strongest influence on performance of all cache parameters [Przybylski90]. We have varied the cache size in the standard configuration from 1 Kbyte to 256 Kbyte. The results of the benchmark programs in Figure 5 show that the miss rate and traffic ratios strictly decrease when the cache is enlarged, but that large differences exist between the individual applications.

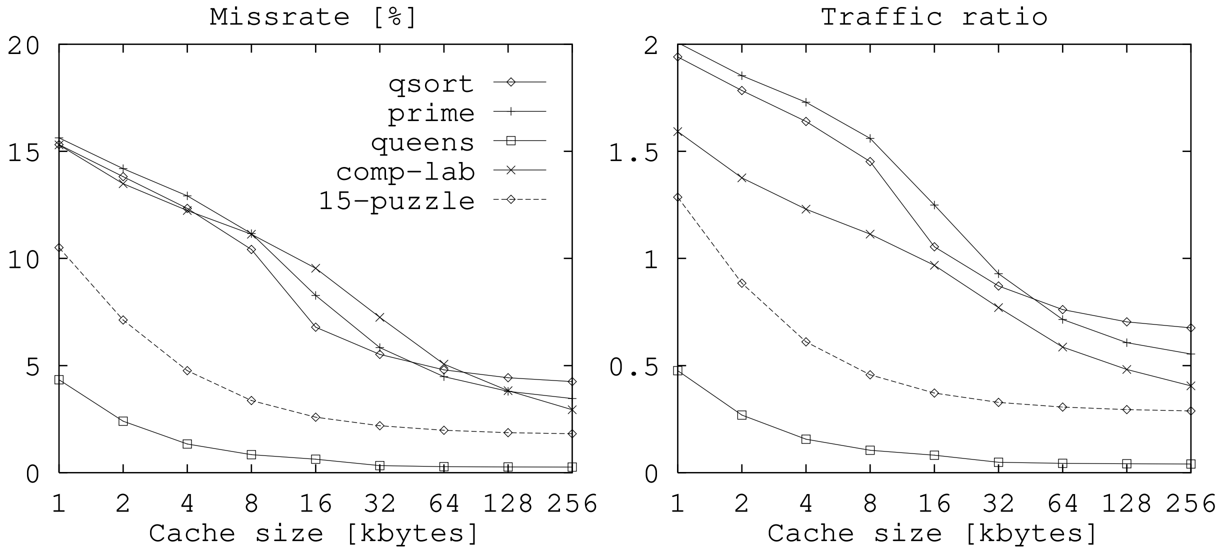


Figure 5: Miss rate and traffic ratio for various cache sizes.

The 15-puzzle and queens program show rapidly decreasing miss rate curves. The height of the curve is essentially determined by the rate of allocating nodes in the heap since the initialization of a fresh node always causes a miss in the cache; nodes are allocated by incrementing the free-space pointer in the heap, so new nodes are always allocated on fresh memory addresses (although the copying collector recycles memory space, the reused nodes have been purged from the cache by nodes allocated more recently in the other semi space).

The qsort, prime, and comp-lab programs do not show a sharp decrease in miss rate until the cache size exceeds 8 Kbyte. This is caused by the larger working sets of these programs (cf. the column live data in Figure 2) in combination with the allocation strategy for new heap nodes. The sequential allocation of heap nodes causes a cyclic stride through the direct mapped cache purging all "old" values including the working set, hence, it has to be re-fetched for each stride. Therefore the cache can not really exploit the temporal locality of the working set until those values stay long enough in the cache to service several references. The larger the working set, the larger the cache has to be to survive the allocation strides. Note that only comp-lab will benefit considerably from very large caches since it is the only program that has not reached its asymptotic minimum miss rate.

The traffic ratios of the benchmark programs have curves whose shape matches the corresponding miss rates. A traffic ratio of less than 1 means that the cache succeeds in reducing main memory bandwidth consumption since the number of data words transported between the memory and cache is smaller than the number of memory references issued by the processor. Unfortunately, the benchmark programs except queens show high traffic ratios, even for large caches, which severely limits their performance. The traffic ratios are higher than for imperative programs since often useless data (garbage) is fetched from or stored in main memory.

3.2 Block size

To increase efficiency caches usually do not fetch a single word from memory at a time, but fetch a block consisting of several words. This prefetching of data is effective if the application shows spatial locality in its memory references, that is if the application references nearby memory addresses in a short period of time. This is often the case, for example, when data structures like arrays are tarversed. The spatial locality in the benchmark programs stems from:

- the behaviour of the stack;
- the sequential allocation of new nodes in the heap;
- the evaluation of suspensions, which involves fetching consecutive arguments from the heap.

Figure 6 shows the benchmark results for block sizes ranging from 4 bytes (one word) to 2 Kbytes.

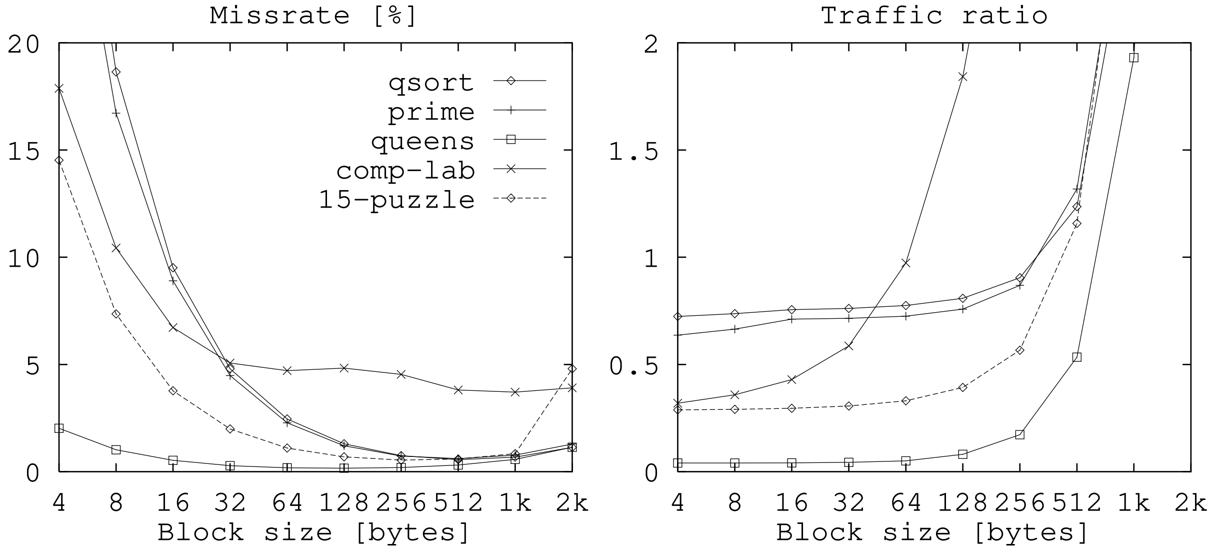


Figure 6: Miss rate and traffic ratio for various block sizes.

Increasing the block size reduces the miss rate, since the prefetched data is likely to be used in the future, but reduces the number of sets in the cache as well. Fewer sets means more clashes of unrelated memory addresses (e.g., stack and heap) purging each other from the cache, which increases the miss rate. These contrary effects are clearly visible for the benchmark programs: initially the miss rate decreases, reaches a minimum at ca. 512 bytes, and then starts to rise again.

For overall performance a block size of 512 bytes is not the right choice since the traffic ratios of all but the queens program exceed 1.0. Note that the traffic ratio curves start off nearly flat, which indicates that most prefetched data is referenced before being purged from the cache. Once the block size exceeds 128 bytes, the traffic ratio increases sharply showing that mostly useless data is transferred between the cache and memory. The comp-lab program behaves worst of all because of its large working set.

The choice of a 32 or 64 byte block size gives reasonably low miss rates, while keeping the traffic ratio acceptably low.

3.3 Update policy

When the processor writes a data value to a location that does not reside in the cache, the data can either be written directly to memory (write-through) or buffered in the cache until the block is subsequently purged from the cache when servicing another miss (copy-back). The write-through policy ensures that main memory is always consistent with the cache contents, but at the expense of a main memory access for each write. The copy-back policy decreases traffic since multiple writes within a cache block (i.e. the initialisation of a new heap node) are written to main memory in one transfer.

When a cache detects a write miss and employs the write-through policy, it can either just write the data to memory or buffer it in the cache as well to service future references. The latter write-allocate strategy has superior performance for lazy functional programs since many nodes have a short life time and are likely to be referenced while the just written data is still in the cache. The data in [Koopman92] confirms this behaviour. We have simulated the copy-back and write-through policy with write-allocate, which only differ in the traffic ratio as reported in Figure 7.

program	write-through	copy-back	ratio
qsort 1000	0.919	0.761	1.21
prime 900	0.907	0.716	1.27
queens 10	0.463	0.044	10.38
comp-lab	0.856	0.586	1.46
15-puzzle	0.638	0.306	2.09

Figure 7: Traffic ratios for different update policies.

The results show that copy-back significantly reduces the traffic between the cache and memory. The queens application benefits most because of the relatively large number of stack accesses (see Figure 3); stack writes are always buffered and the cache only copies a stack element back to memory when some heap reference hashes to the same block. The clashes occur infrequently because the heap allocation mechanism cycles through the (large) cache.

3.4 Associativity and replacement policy

The preceding experiments have used a direct mapped cache with one block per set. Increasing the number of blocks in each set (i.e. the associativity) reduces the number of clashes between accesses in different memory regions (i.e. stack and heap). The replacement policy decides which cache block in a set has to be written back to memory when new data has to be fetched to service a cache miss. We have used the LRU (least recently used) replacement policy since it performs better than FIFO (first in, first out) and random; see the results in Figure 8.

The results of the benchmark programs for various associativities are presented in Figure 9.

Apparently only the change from a direct mapped cache to a 2-way associative cache has a significant impact on the miss rate, which is also noted in other cache studies. This is caused by the fact that the stack and heap accesses do not interfere with each other as strongly as in a direct

program	miss rate			traffic ratio		
	LRU	FIFO	random	LRU	FIFO	random
qsort 1000	4.08	4.32	4.41	0.650	0.689	0.702
prime 900	3.13	3.57	3.99	0.500	0.572	0.636
queens 10	0.25	0.25	0.25	0.040	0.040	0.040
comp-lab	3.75	3.82	3.75	0.475	0.481	0.478
15-puzzle	1.81	1.84	1.88	0.287	0.291	0.294

Figure 8: Miss rate and traffic ratio for different replacement policies; associativity = 4.

program	miss rate				traffic ratio			
	1	2	4	8	1	2	4	8
qsort 1000	4.80	4.08	4.08	4.08	0.761	0.650	0.650	0.650
prime 900	4.48	3.13	3.13	3.13	0.929	0.765	0.723	0.729
queens 10	0.28	0.25	0.25	0.25	0.049	0.040	0.040	0.040
comp-lab	5.07	4.35	3.65	3.24	0.771	0.725	0.707	0.697
15-puzzle	1.98	1.81	1.81	1.81	0.328	0.292	0.291	0.291

Figure 9: Miss rate and traffic ratio for various associativities (LRU replacement).

mapped cache. The sequential allocation of heap nodes no longer purges the stack elements in a cyclic fashion since each set holds two words, so the new heap nodes replace the old (garbage) nodes instead of the valuable stack elements as in the case of a direct mapped cache. A further increase in associativity is only beneficial for the comp-lab program, which has a much larger working set than the others. Note that the traffic ratios show a similar effect as the miss rates when the associativity is increased.

4 Discussion and comparison

To place the results of the cache simulations in some perspective, we made a comparison with results of other functional programming environments as well as measurements of imperative programs.

4.1 Functional languages

The cache miss rates and traffic ratios for the TIGRE graph reduction system [Koopman92] can be directly compared with our measurements since they have also simulated the cache behaviour of the heap and stack references for similar cache parameter values. An important difference is that the TIGRE machine is based on interpreting a combinator graph, while our approach uses a compiled graph reduction implementation technique. Furthermore their traces do not include garbage collection, but they do allocate new heap nodes by advancing a free-space pointer through the heap just as in our implementation.

Comparing the results of TIGRE benchmark applications for our base cache of Figure 4 shows that the miss rates and traffic ratios lie in the same range. This is surprising, since we would expect that the TIGRE interpreter would obtain lower miss rates due to the frequent construction of combinator graphs that are immediately reused in subsequent reduction steps.

Just like our implementation, the TIGRE interpreter benefits from a large block size (32 bytes) because of the spatial locality in the address traces of their benchmark programs. The shape of the miss rate and traffic ratio curves for varying block sizes strongly resembles our curves presented in Figure 6. The TIGRE results for the copy-back policy in comparison to write-through show that the copy-back policy roughly halves the traffic ratio. This more or less agrees with our results that indicate a reduction between 18% and 94%. The results for varying the associativity of a cache are the same for both implementations: a 2-way associative cache performs somewhat better than a direct mapped cache, but increasing the associativity further has virtually no effect.

A second study of the cache behaviour of lazy functional languages is reported in [Poon85], which has traced both an SECD machine implementation and a combinator interpreter (similar to the TIGRE machine). It is more difficult to compare those results with the ones in Section 3 since only the references to the heap are taken into account, and our study includes the stack references as well. In addition, only two benchmark programs are used which show similar and smooth behaviour over a range of cache parameters.

A remarkable observation is that the combinator and SECD implementation do not benefit from cache sizes larger than 8 Kbyte. According to [Poon85] this is probably an artifact of the benchmark programs that have only (very) small working sets unlike our comp-lab program for example. The combinator implementation achieves miss rates of 2%, while the SECD machine does not get below a 10% miss rate. Our compiled graph reduction implementation shows miss rates as low as 0.3% for the queens application. The difference is probably caused by the block size parameter that has been fixed to the size of one heap node (8 bytes) for all measurements reported in [Poon85], which makes it impossible to benefit from spatial locality among consecutive heap cells. Based on our results and those of the TIGRE implementation, we reject the assumption that "list processing effectively destroys spatial locality" as stated in [Poon85]. The results for cache associativity and update policy, however, are the same: copy-back reduces traffic ratios and a 2-way associative cache marginally outperforms a direct mapped cache.

4.2 Imperative languages

The cache behaviour of imperative programs has been thoroughly studied, both analytically and empirically [Smith82, Smith87, Hill89, Przybylski90, Hennessy90]. It is, however, rather difficult to compare the results of imperative and functional languages because of the strong dependence on the memory reference behaviour of the application benchmark (i.e. workload). Nevertheless, we can compare the trends in both areas.

In general imperative programs yield lower miss rates than our lazy functional programs. This is caused by the large heap space, which does not fit into the cache, and the sequential allocation strategy for new nodes. The allocation of large numbers of heap nodes is forced by the referential transparency property of lazy functional languages, and results in lots of cache misses because each node is allocated on a new address in the heap. The in-place update policy of imperative language implementations reuses memory locations much faster than the copying garbage collector, hence, the lower misss rates.

The sequential heap allocation mechanism favours large block sizes, while imperative programs favour a rather small block size like 16 or 32 bytes [Przybylski90, Smith87]. The functional programs mainly benefit from spatial locality (prefetching), while the imperative programs also benefit from temporal locality.

Our functional language implementation uses two distinct memory areas: stack and heap. Since the heap is accessed in a sequential way, the functional applications do not benefit from an associative cache with more than two blocks per set. The in-place updates of arrays in imperative programs result in several areas of frequently accessed data, so imperative programs do require higher degrees of associativity to avoid collisions: 4 to 8 as reported in [Smith82].

5 Conclusions

We have measured the cache behaviour of a lazy functional language implementation based on compiled graph reduction. The trace driven cache simulations of the heap and stack references extracted from five benchmark applications show that the performance is largely determined by the cache misses caused by initializing new heap nodes. The heap allocation mechanism advances a free-space pointer sequentially through the available space, handing out new addresses. This results in the heap allocator walking through the cache in a cyclic fashion purging out old (garbage) data before it can be reclaimed by the two-space copying garbage collector.

The heap allocation mechanism benefits from large block sizes and a copy-back update policy since multiple heap cells can be initialised (written) per cache miss. A write-through cache performs considerably less good since each individual write has to go through to main memory. The preference for large blocks differs from imperative programs, but [Koopman92] notes similar behaviour for a functional language implementation based on a combinator interpreter.

Variation of the cache associativity shows that the stack and heap accesses interfer in a direct mapped cache. The cyclic pattern of the heap allocator repeatedly evicts valuable stack elements from the cache. A 2-way associative cache gives lower miss rates than a direct mapped cache. Increasing the associativity above 2 words per set, however, has hardly any effect on the miss rate. Apparently the heap accesses do not collide often, which is probably due to the short life times of most heap nodes. Other functional programming environments report the same behaviour [Koopman92, Poon85], while imperative programs do benefit from a larger cache associativity (4 or 8) [Smith82].

Our functional benchmark applications achieve miss rates between 0.25% and 5.5% for a direct-mapped 64Kbyte cache with 32 byte blocks. These miss rates are larger than for imperative programs in general. Referential transparency essentially causes the difference since it forces the rapid allocation of new heap nodes, while imperative implementations use in-place updates of data structures like arrays. We believe that the performance of lazy functional languages would benefit from a more sophisticated heap management algorithm that reclaims garbage while it is still in the cache.

Acknowledgements

We thank Henk Muller and Rutger Hofman for sharing their expertise, and commenting on a preliminary version of this paper.

References

[Cheney70] C. J. Cheney. A non-recursive list compacting algorithm. *CACM*, 13(11):677–678, 1970.

[Embrechts90] H. Embrechts, D. Roose, and P. Wambacq. Component labelling on a MIMD multiprocessor. prepublished report, Dept. of Comp. Sci., Katholieke Universiteit Leuven, Belgium, 1990.

[Glas92] J. Glas. The parallelization of branch and bound algorithms in a functional programming language. Master's thesis, Dept. of Comp. Sys, Univ. of Amsterdam, 1992.

[Hartel91a] P. H. Hartel, H. W. Glaser, and J. M. Wild. Compilation of functional languages using flow graph analysis. Technical report CSTR 91-03, Dept. of Electr. and Comp. Sci, Univ. of Southampton, UK, 1991.

[Hartel91b] P. H. Hartel, H. W. Glaser, and J. M. Wild. On the benefits of different analyses in the compilation of functional languages. In H. W. Glaser and P. H. Hartel, editors, *Implementation of functional languages on parallel architectures*, pages 123–145, Southampton, UK. CSTR 91-07, Dept. of Electr. and Comp. Sci, Univ. of Southampton, UK, 1991.

[Hennessy90] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, California, USA, 1990.

[Hill85] M. D. Hill. Dineroiii documentation. UNIX man page unpublished, Univ. of California, Berkeley, USA, 1985.

[Hill89] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE transactions on computers*, C-38(12):1612–1630, 1989.

[Johnsson84] T. Johnsson. Efficient compilation of lazy evaluation. In *ACM compiler construction*, pages 58–69, Montréal, Canada. SIGPLAN notices,19(6), 1984.

[Koopman92] P. J. Koopman Jr., D. P. Siewiorek, and P. Lee. Cache behaviour of combinator graph reduction. *ACM Transactions on Programming Language Systems*, 14(2):265–297, 1992.

[Landin64] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[Langendoen92] K. G. Langendoen and P. H. Hartel. FCG: a code generator for lazy functional languages. Technical report CS-92-03, Dept. of Comp. Sys, Univ. of Amsterdam, 1992.

[Langendoen91] K. G. Langendoen and W. G. Vree. Eight queens divided: An experience in parallel functional programming. In J. Darlington and R. Dietrich, editors, *Declarative programming*, pages 101–115, Sasbachwalden, West Germany. Springer-Verlag, 1991.

[Peyton Jones91] S. L. Peyton Jones. The spineless tagless G-machine: a second attempt. In H. W. Glaser and P. H. Hartel, editors, *Implementation of functional languages on parallel architectures*, pages 147–191, Southampton, UK. CSTR 91-07, Dept. of Electr. and Comp. Sci, Univ. of Southampton, UK, 1991.

[Poon85] E. K. Y. Poon and S. L. Peyton Jones. Cache memories in a functional programming environment. In L. Augustsson, R. J. M. Hughes, T. Johnsson, and K. Karlsson, editors, *Implementation of functional languages*, pages 132–150, Aspenäs, Sweden. Programming Methodology group report 17, Dept. of Comp. Sci, Chalmers Univ. of Technology, Göteborg, Sweden, 1985.

[Przybylski90] S. A. Przybylski. *Cache and memory hierarchy design: a performance-directed approach*. Morgan Kaufmann Publishers, Inc., Palo Alto, California, USA, 1990.

[Runciman92] C. Runciman and D. Wakeling. Heap profiling of lazy functional programs. Technical report 172, Dept. of Comp. Sci, Univ. of York, UK, 1992.

[Sansom92] P. M. Sansom and S. L. Peyton Jones. Profiling lazy functional languages (working paper). Internal report 18, Dept. of Comp. Sci, Univ. of Glasgow, Scotland, 1992.

[Scheevel86] M. Scheevel. NORMA: A graph reduction processor. In *Lisp and functional programming*, pages 212–219, Boston, Massachusetts. ACM, 1986.

[Schulte91] W. Schulte and W. Grieskamp. Generating efficient portable code for a strict applicative language. In J. Darlington and R. Dietrich, editors, *Declarative programming*, pages 239–252, Sasbachwalden, West Germany. Springer-Verlag, 1991.

[Smetsers91] S. Smetsers, E. G. J. M. H. Nöcker, J. van Groningen, and M. J. Plasmeijer. Generating efficient code for lazy functional languages. In R. J. M. Hughes, editor, *5th Functional programming languages and computer architecture, LNCS 523*, pages 592–617, Cambridge, Massachusetts. Springer-Verlag, 1991.

[Smith82] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[Smith87] A. J. Smith. Line (block) size choice for CPU cache memories. *IEEE transactions on computers*, C-36(9):1063–1075, 1987.

[Stout87] Q. F. Stout. Supporting divide-and-conquer algorithms for image processing. *J. Parallel and Distributed Computing*, 4(1):95–115, 1987.

[Turner79] D. A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9(1):31–49, 1979.