ZG-machine에서 기억 장소 재활용 체계의 영향
(Effect of Garbage Collection in the ZG-machine)

우 균 † 한 태 숙 ††
(Gyun Woo) (Taisook Han)

요 약 ZG-machine은 태그옮김이라는 간단한 부호화 기법을 채택한 공간 효율적인 G-machine이다. 기억 장소 재활용 체계 없이 실험한 이전 실험에서 ZG-machine은 G-machine과 비교하여 30%의 힙 공간을 절약할 수 있었고 수행 시간 부담은 6%를 넘지 않았었다. 이 논문에서는 ZG-machine에 기억 장소 재활용 체계를 장착하여 추가로 실험한 결과를 설명한다. 결과에 따르면, G-machine과 비교할 때, ZG-machine의 수행 시간은 34% 증가하였지만 최소 힙 사용량은 평균 34% 감소하였다. 수행 시간 부담이 커진 이유는 기억 장소 재활용 체계때문이다. 그러나 힙 공간을 최소 힙 사용량의 7 배 정도로 늘렸을 경우에 G-machine에 대한 수행 시간 부담은 12%를 넘지 않았다. ZG-machine에서 최소 힙 사용량이 줄어든 특성은 ZG-machine이 내장 체계와 같은 기억 장소가 제한된 응용 분야에 사용될 수 있음을 의미한다. 또한 보다 효율적인 기억 장소 재활용 체계를 개발함으로써 수행 시간은 상당히 줄어들 것으로 예상된다.

Abstract The ZG-machine is a space-efficient G-machine, which exploits a simple encoding method, called tag-forwarding, to compress the heap structure of graphs. Experiments on the ZG-machine without garbage collection shows that the ZG-machine saves 30% of heap space and the run-time overhead is no more than 6% than the G-machine. This paper presents the results of further experiments on the ZG-machine with the garbage collector. As a result, the heap-residency of the ZG-machine decreases by 34% on average although the run-time increases by 34% compared to the G-machine. The high rate of the run-time overhead of the ZG-machine is incurred by the garbage collector. However, when the heap size is 7 times the heap-residency, the run-time overhead of the ZG-machine is no more than 12% compared to the G-machine. With the aspect of reduced heap-residency, the ZG-machine may be useful in memory-restricted environments such as embedded systems. Also, with the development of a more efficient garbage collector, the run-time is expected to decrease significantly.

1. Introduction

Lazy functional languages facilitate fast-prototyping by abstracting the execution flow of programs. Basically, functional programming languages provide more abstraction mechanisms than imperative ones: algebraic data types abstracts away the memory access; higher order functions increase the modularity of the programs. Lazy functional languages further abstract the control flow of programs by evaluating expressions on demand.

Graph reduction is a well-known way to implement lazy functional languages. However, it generally requires a lot of heap space to execute a program. In graph reduction, an expression is represented as a graph and the evaluation of the expression is achieved by transforming the graph. The transformation is performed repeatedly until the result is the simplest form, i.e., the value of the initial expression. Many intermediate graphs con

· 본 연구는 첨단정보기술 연구센터를 통하여 과학재단의 지원을 받았음.
† 정 회 원 : 동아대학교 전기전자컴퓨터공학부 교수
woogyun@daunet.donga.ac.kr
†† 종신회원 : 한국과학기술원 전자전산학과 교수
han@pllab.kaist.ac.kr
논문접수 : 1999년 10월 8일
심사완료 : 2000년 4월 3일

structed during the sequence of transformations are stored in heap, hence graph reduction generally requires a lot of heap space.

Tag-forwarding is a simple encoding method to reduce the heap space required to store the graphs. Tag-forwarding concerns the tagged representation of graphs, where the graph node is composed of a tag and some data fields. The G-machine [1, 2] is a typical example which adopts tagged representation. In tagged representation, the tag of a graph node is generally stored in a whole heap word. This is certainly a waste of space because only a few bits are sufficient to store a tag. Tag-forwarding is a method to utilize the remaining bits of the tag word.

The space inefficiency incurred by storing a tag in a single heap word was pointed out by Peyton Jones [3, page 325]. For a rationale of this inefficiency, he used the example of the Chalmers Lazy ML implementation, where each tag is implemented by the entry address of the dispatch table to the run-time routines. With this implementation, a comparison operation for jumping to the run-time routine can be saved because the tag does not need to be tested before jumping.

We implemented tag-forwarding on top of the G-machine, resulting a new abstract machine called the ZG-machine. Besides the comparison operations with tags, tag-forwarding requires additional operations. Since tag-forwarding uses encoding, it incurs some operations for decoding. The operations for decoding are imposed on run-time. On the other hand, tag-forwarding has a beneficial effect on run-time. The number of memory references for graphs is reduced because the graphs are highly encoded.

The previous experiments [4] on the ZG-machine showed promising results: the total heap allocation decreased about 30% on average and the run-time increased no more than 6% compared to the G-machine. The experiments were performed using the C translators of the G- and the ZG-machines. However, the previous experiments were performed without garbage collectors.

In this article, we report the results of more concrete experiments on the ZG-machine. To get a more realistic view, we have implemented the garbage collectors of the G- and the ZG-machines. With garbage collectors, we can examine another perspective of the space behavior, i.e., the heap-residency. The heap-residency of a program denotes the minimum size of the heap required to execute the program with the garbage collector equipped.

Though the G-machine is a rather old machine, it is still popular due to its compact definition. There are some variants of the G-machine, for example, Spineless G-machine [5] and Spineless Tagless G-machine [6], each of which generally outperforms the G-machine. However, the G-machine is preferred in some applications [7, 8, 9] because its definition is simpler than those of its variants. The ZG-machine is another variant of the G-machine aiming for a better space behavior.

This article is organized as follows. Section 2 explains the basic idea of tag-forwarding and considers the performance-affecting factors of the ZG-machine. Section 3 describes the implementation of the experimental systems and the experimental results, and discusses the limitations of our experiments. Section 4 reviews related works and Section 5 concludes.

2. Tag-forwarding

Tag-forwarding [4] is a simple encoding method which aims for the encoded graph to be smaller than the original graph. A graph in the G-machine is a linked structure of nodes, each of which has a tag and one or more data fields. All links are implemented by absolute pointers which are stored in the data fields of nodes. If we know that a part of a graph is allocated at the same time, we can treat the part as an allocation unit and allocate it in a sequential addressing space. Then the intra-links between the nodes in the allocation unit can be implemented by relative address pointers. These relative addresses can be computed at compile-time and stored in a smaller heap space

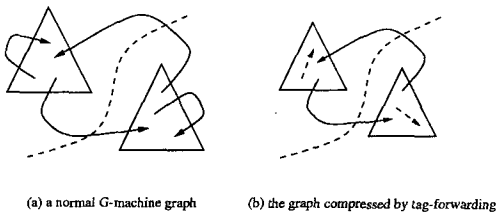

Figure 1: The basic idea of tag-forwarding.

than the absolute counterparts. Tag forwarding is a wayg to merge the relative address and the tag into one heap word.

The basic idea of tag-forwarding is depicted in Figure 1. The triangular chunks denote the detected allocation units of the graphs in the G-machine. In the G-machine, all graph links are wired at run-time as depicted in Figure 1(a). However, the intra-links in a chunk can be wired at compile-time, as depicted by the dashed arrows in the chunks in Figure 1(b). We can reduce the size of the chunks if we forward the tags to the relative pointers and store the tags with the pointers. This procedure is called tag-forwarding. Consequently, the heap can be saved in the amount corresponding to the number of tag words forwarded.

Although tag-forwarding can be encoded at compile-time, decoding the tag-forwarded graphs must be performed at run-time. In the tag-forwarded graphs, the tag and the relative address are paired and stored in a heap word. Therefore, some extracting operations are needed to look up the tag or the relative address. Moreover, we need a flag to distinguish absolute address links from relative address links because they are intermixed in the tag-forwarded graphs. The test of the flag has to be performed at run-time, too. Since the tag-forwarding scheme is in fact just an encoding, it inevitably needs some decoding operations that cause run-time overhead.

The encoding overhead of tag-forwarding is mostly imposed on compile-time. However, it also affects the garbage collection. Normally the graphs once constructed need not be reconstructed. However, during the garbage collection, the graphs should be reconstructed to provide a chunk of free space in the heap. The relative address links have to be modified during reconstruction, which incurs some encoding operations of tag-forwarding. The encoding operations during the garbage collection also cause the run-time overhead of tag-forwarding.

Up to now, the factors leading to run-time overhead were discussed; however, tag-forwarding is not always bad for run-time. The number of heap references is reduced by tag-forwarding because the graph size is reduced. The number of memory fetch operations for the heap data is less than that of the G-machine. Most of the additional operations involved in tag-forwarding are performed in a fast pipelined processor and the reduction of memory traffic for the heap data might be more significant than the increase of the additional computation.

The above argument about the run-time overhead of tag-forwarding is rather controversial. To get a realistic view on the performance of the tag-forwarding scheme, some experiments were performed. The main goal of the experiments is to compare the G-machine and the ZG-machine, a modified version of the G-machine performing tag-forwarding. The definition of the ZG-machine is described in [4].

3. Experiments

Normally, the ZG-machine uses less heap space than the G-machine. At most, it uses equal heap space to the G-machine, which is the case when there are no intra-links in the allocation units of the graphs. A heap word will be saved if an intra-link is converted to a relative address and merged with the tag of the node pointed to by the intra-link. Concerning the heap usage, it is apparent that the ZG-machine is more efficient than the G-machine.

Concerning the run-time, the ZG-machine seems slower than the G-machine. The ZG-machine has many factors which cause a delay in run-time. In the ZG-machine, the flag test operations and the

field extracting operations are additionally needed to decode the tag-forwarded graphs. The garbage collector also incurs the burden of encoding. In the ZG-machine, there is one positive factor for run-time: the memory traffic for referencing graphs is reduced.

3.1 The Implementation

Both the G- and the ZG-machines are implemented in the same framework. For a quick implementation, we choose a two-pass implementation: first, the source code, a set of combinator definitions, is translated to the M-code; and second, the M-code is translated to the C program. The M-code [10], used in implementing the Chalmers Lazy ML Compiler, is an abstraction of native machine code which is specialized for translating G-machine code. In our implementation, the M-code is extended to include operations related to tag-forwarding. The translators of the abstract machines generate two different C programs from the same source code. The C programs are compiled and executed to get the statistics.

This two-pass implementation has some benefits. First of all, we can share the code which translates the M-code into the C program. Second, it enables fast prototyping by translating the source program into a C program. Third, it is easier to add the monitoring code for the execution profiles to the C program than to the native machine code. Getting the execution profiles is our main concern in the experiment. The laziness of a source program is implemented by graph construction codes in the target C program.

Besides the additional encoding performed by the garbage collector of the ZG-machine, there is an important difference between the garbage collectors of two machines. We use copying algorithms in implementing the garbage collectors. There are two classical algorithms for copying collectors: one is recursive [11] and the other is non-recursive [12]. The latter is generally faster than the former.

However, we cannot use the non-recursive algorithm for the ZG-machine. In the ZG-machine, since the tag of a graph node is separated from other data fields and forwarded, the garbage collector has to follow the pointers in a depth-first search manner. Hence, the garbage collector of the ZG-machine is implemented using the recursive algorithm.

One of the reason of adopting the recursive algorithm for the garbage collector of the ZG-machine is to compare the machines in a more fair condition. If an additional one-bit flag for primitive data types is used for implementation of the ZG-machine, the non-recursive algorithm can be used for the garbage collector of the ZG-machine. However, adopting the additional flag makes the primitive data type of the ZG-machine cover less range than that of the G-machine. Hence, we use the natural recursive algorithm for the garbage collector of the ZG-machine. In any case, further optimization of the garbage collectors can be considered as a future work.

3.2 Experimental Results

The experiment concerns mainly two points. One is how much heap space can be saved by tag-forwarding and the other is how much the run-time overhead will be. There are two factors concerning the heap space: the total heap allocation and the heap-residency. The total heap allocation is the sum of all heap words allocated during an execution. If sufficient memory up to this size is provided, the garbage collector will never be invoked. The heap-residency is the minimum size of the heap memory required to execute the program with the garbage collector equipped. If the heap memory provided is less than the heap-residency, the program will run out of space and cannot be completed.

For the actual meaning of the heap-residency, let us first consider the meaning of the residency of a program. The residency of a program at a particular moment is the size of the graph at that moment [3, page 403]. We can think of the maximal residency of a program as the largest residency value attained during the execution of the program. For a given program, the heap-residency denotes the actual heap size for the maximal

residency.

As benchmark programs, five small programs were selected from the imaginary subset of the nofib benchmark suite of Haskell programs [13][1]. Since the full Haskell language is not implemented in the experimental system, we had to translate programs by hand from Haskell into the source language of the experimental system. Hence, small programs are selected. The selected programs can be briefly described as following:

- exp: Computation of 3 powered by 8 where the numbers are encoded by data structures.
- nfib: The nfib function, which is similar to the fib function returning a Fibonacci number, with the argument 30.
- primes: The first 300 primes using Eratho-stene's sieve.
- queens: The number of solutions of the '10 queens' problem, which is an extended version of the '8 queens' problem.
- tak: The tak function with arguments 24, 16, and 8.

Though the exp function just computes 3^8, it generates a lot of live graphs during run-time because the integers are represented by data constructors in the program, and furthermore the graphs for them are hardly destroyed until the program ends. Nfib and queens generate a lot of pending applications because these extensively recursive functions are computed lazily. Primes and queens are using list data structures extensively, but the size of live graphs generated by these programs seem to be small because the live list data structures are destroyed rapidly since they perform a lot of backtracking.

GNU C compiler version 2.7.2.3 was used to get the executable from the C programs generated by the experimental system. The executables were timed on a SUN SPARC 20 UNIX system. We used the C library function times, taking the sum of user and system times as the total execution time.

Table 1 The space efficiency of the ZG-machine

(a) Heap-residency

(unit: KB)

	exp	nfib	primes	queens	tak	average
G-machine	463	8	310	420	43	—
ZG-machine	309	6	180	259	29	—
inc. rate	-33%	-25%	-42%	-38%	-33%	-34%

(b) Total heap allocation

(unit: KB)

	exp	nfib	primes	queens	tak	average
G-machine	118,248	182,027	2,010	121,474	193,877	—
ZG-machine	86,731	126,645	1,551	92,046	134,443	—
inc. rate	-27%	-30%	-23%	-24%	-31%	-27%

Table 1 shows the heap space behavior of the abstract machines for the benchmark programs. Table 1(a) shows the heap-residency and Table 1(b) shows the total heap allocation. In both cases, the statistics were measured in kilobytes(KB), hence the error is bound within 1 KB. The increasing rates of the ZG-machine values relative to the G-machine values are also shown in Table 1. A negative value of the increasing rate means that space is saved.

According to the results on Table 1, the ZG-machine always requires less heap space than the G-machine, for both the heap-residency and the total heap allocation. For each machine, the heap-residency varied greatly depending on the programs. For the test programs, the heap-residency of the G-machine varied from 8 KB to 463 KB, and that of the ZG-machine varied from 6 KB to 309 KB. The total heap allocation also varied greatly depending on the programs. However, the increasing rate was rather constant. For all test programs, the heap-residency of the ZG-machine decreased by 34% on average and the total heap allocation also decreased by 27% on average.

Now, let us investigate the run-time overhead. Table 2 shows the run-time behavior of the ZG-machine. For all benchmark programs, the run-time of each machine was measured with

1) We do not insist that the experiment is performed on the nofib benchmark suite, since the entire real subset of nofib is not covered by the experiment.

Table 2 The run-time overhead of the ZG-machine

(unit: 10ms)

multiplier		1	2	3	4	5	6	7
exp	GM	6221	3839	3578	3490	3464	3462	3476
	ZGM	8872	4722	4286	4099	3950	3934	3852
	inc. rate	43%	23%	20%	17%	14%	14%	11%
nfib	GM	14997	5109	4378	4095	3937	3834	3771
	ZGM	17479	5823	4788	4423	4216	4099	4021
	inc. rate	17%	14%	9%	8%	7%	7%	7%
primes	GM	273	88	74	71	71	70	69
	ZGM	351	111	87	83	81	78	77
	inc. rate	29%	26%	18%	17%	14%	11%	12%
queens	GM	12041	4647	4231	4080	4021	4022	4007
	ZGM	16625	5351	4717	4484	4354	4285	4258
	inc. rate	38%	15%	11%	10%	8%	7%	6%
tak	GM	6389	4006	3521	3358	3219	3163	3126
	ZGM	9116	4836	3998	3622	3448	3317	3241
	inc. rate	43%	21%	14%	8%	7%	5%	4%
average inc. rate		34%	20%	14%	12%	10%	9%	8%

different heap sizes. For each execution, the heap size was set to the value of the heap-residency multiplied by the number shown in the top row of the table. For each program, the increasing rate of the execution time of the ZG-machine compared to that of the G-machine is also shown. According to the results, the ZG-machine is much slower than the G-machine when the heap constraint is severe. When the heap is set exactly to the heap residency, the average increasing rate reaches up to 34%. However, when the heap constraint is relieved, the run-time overhead of the ZG-machine is reduced. Typically, when the heap memory is set to seven times the heap-residency, the run-time overhead of the ZG-machine is less than 12% for all test programs.

The garbage collector of the ZG-machine seems to be responsible for the high increasing rate of the run-time when the heap constraint is severe. When the heap is set to the heap-residency, the numberof calls to the garbage collector is maximal, hence the garbage collector greatly affects the overall

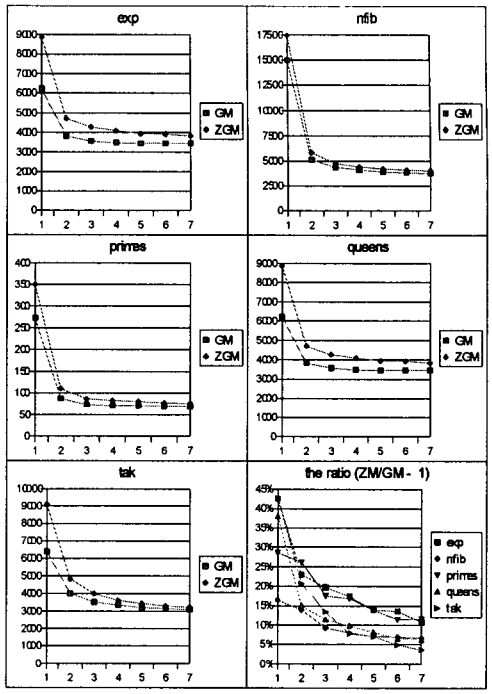

Fig. 2 The execution time *vs* heap size

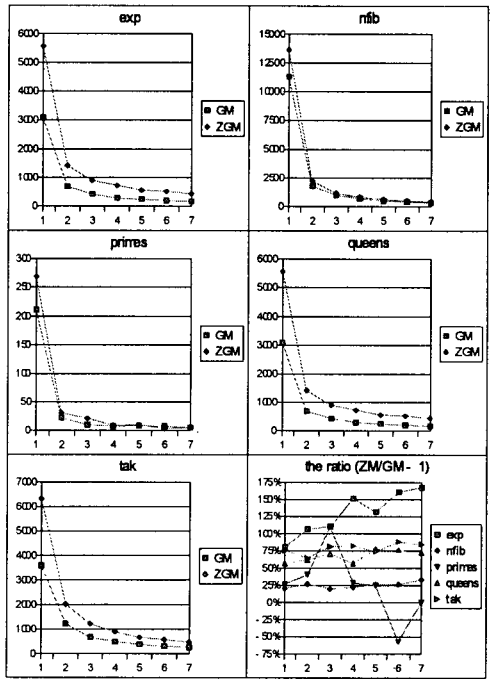

Fig. 3 The garbage collection time *vs* heap size

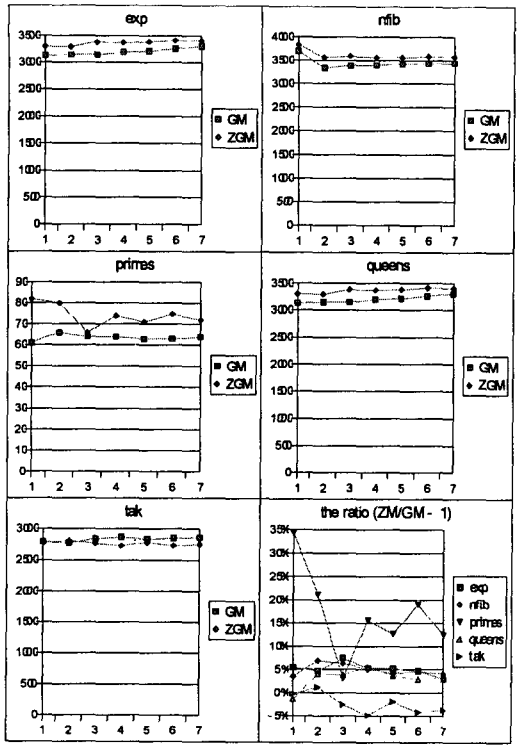

Fig. 4 graph reduction time vs heap size

run-time. Figure 2, 3 and 4 support this interpretation. Figure 2 depicts the overall run-time; Figure 3 and Figure 4 depict the garbage collection time and the graph reduction time, respectively. Figure 2 and Figure 3 show the same decreasing behavior. However, Figure 4 shows that the graph reduction time almost remains constant independent to heap size. This means that the garbage collection overhead dominates the overall run-time overhead of the ZG-machine.

The overhead of the copying collector is proportional to the size of live graphs. According to the experimental results, exp and tak show the worst performance when the heap constraint is severe. As mentioned above, since exp generates a lot of live graphs which burden the garbage collector. Nfib and tak are both generating a lot of pending applications. However, the size of the graphs for pending application for tak is bigger than that for nfib because tak has four recursive calls and nfib has only two. Therefore, the garbage collection time for tak is higher than that of nfib. Though primes and queens are extensively using data constructors like exp, the size of live graphs for them seems not so big compared to exp because these programs perform a lot of back-tracking. Hence, the garbage collection overhead of these programs is less than that of exp.

To make the ZG-machine practical, the garbage collection overhead of the ZG-machine should be relieved. As mentioned before, the garbage collector of the ZG-machine has the burden of encoding and was implemented using a recursive algorithm. A recursive algorithm has more call/return overhead than a non-recursive algorithm. We are currently working on developing an efficient algorithm for garbage collector of the ZG-machine.

When generating the results shown in Table 2, the heap size was set as the multiple of the heap-residency, but the heap-residency of the ZG-machine was less than that of the G-machine. To compare the machines in the same condition, the execution time should be measured with a heap of the same size. Table 3 shows the result when the heap is set to 1 and 2 megabytes(MB). The increasing rate of the execution time of the ZG-machine is on average 8% when the heap is of 1 MB, and 5% when 2 MB. The run-time overhead of the ZG-machine is tolerable in these cases.

In summary, the ZG-machine is more efficient than the G-machine space-wise but less efficient

Table 3 The run-time overhead of the ZG-machine with the heap size fixed

(unit: 10ms)

		exp	nfib	primes	queens	tak	average
1 MB	G-machine	3945	3610	75	4465	3461	—
	ZG-machine	5284	3787	81	4671	2999	—
	inc. rate	34%	5%	8%	5%	-13%	8%
2 MB	G-machine	3443	3538	69	3995	2920	—
	ZG-machine	3868	3702	75	4180	2823	
	inc. rate	12%	5%	9%	5%	3%	5%

time-wise. Both the heap-residency and the total heap allocation decrease in the ZG-machine, compared to the G-machine. However, the ZG-machine is slower than the G-machine when the heap constraint is severe. But it is compatible with the G-machine when given a heap of the same size. The garbage collector is responsible for the high rate of the run-time overhead of the ZG-machine in the space-critical cases. To overcome the deficiency of the ZG-machine, an efficient garbage collector is particularly necessary.

3.3 Caveats

Our experiments have a few limitations. First of all, the experiments are not based on real compilers but on C translators. This means that the execution time of the real compiler may differ from that of the experimental results, though the space consumption does not seem to be far from accurate. Secondly, the implementations are not full compilers of lazy functional languages but only concern the back-ends of the compilers. All optimizations and transformations related to the front-end are excluded. In fact, the implementations of both compilers are based on the work by Peyton Jones and Lester [14]. Lacking the front-end of a full compiler, we had to syntactically translate the Haskell benchmark programs by hand. Therefore, we selected small programs.

One might think the benchmark programs are too small to expect any meaningful information from the results. However, they are sufficient for testing the effect of the ZG-machine because the effect of tag-forwarding is proportional to the size of the allocation unit, and larger allocation units can be found more easily in larger programs. Hence, if the ZG-machine is effective on small programs, it is highly expected to be even more effective on larger programs.

4. Related Work

The tag-forwarding method is similar to NORMA [15] in that the words in the graph memory are encoded. A graph node in NORMA is a 64-bit word, which is highly encoded to include all node types. This complicated encoding is supported by specialized hardware, which is different from our approach. There is no special hardware support for tag-forwarding, for it is just a simple encoding.

It seems that there has not been much concern about space efficiency in lazy functional languages. Haydarlow and Hartel introduced thunk-lifting [16], a program transformation which aims to reduce the heap space. Thunk-lifting is based on context analysis, but tag-forwarding is applicable for almost every lazy context. Wakeling applied the dynamic compilation technique to the G-machine and reduced the code size significantly [17] Our approach can be considered complimentary to Wakeling's, for his work concerns the code space and ours concerns the heap space.

Tag-forwarding can be also applied to the variants of the G-machine which are using tags for representing graphs. The Spineless G-machine and the \nuG-machine [18] are the examples. Especially, the Spineless G-machine provides another method to reduce the memory traffic between the reduction engine and the graph store. We can reduce it further by applying tag-forwarding to the Spineless G-machine, but the decoding overhead shall be still remains. To investigate the actual performance of these possible variants of the ZG-machine, further experiments should be performed.

5. Conclusion

This article describes the basic idea and the experimental statistics of the ZG-machine, which requires less heap space than the G-machine. According to the experimental results, the heap-residency of the ZG-machine decreases by 34% and the total heap allocation decreases by 27% on average, compared to the G-machine. When the heap size is set to the heap-residency, the run-time overhead of the ZG-machine is much higher than that of the G-machine; the increasing rate of the run-time reaches up to 34%.

It seems that the garbage collector is responsible

for the high overhead of the run-time when the heap constraint is severe. Currently, the garbage collector of the ZG-machine is less efficient than that of the G-machine. Therefore, the run-time overhead of the ZG-machine decreases rapidly as the heap constraint is relieved. Typically, if the heap size is set to 7 times the heap-residency, the increasing rate of the run-time of the ZG-machine does not reach more than 12%. In any case, developing a fast garbage collector is particularly important for the ZG-machine.

Reduced heap-residency in the ZG-machine is an important property. In some computing environments, say for embedded systems, memory is a valuable resource. In this respect, the ZG-machine promotes the possibility of using lazy functional languages in such environments. The ZG-machine can be also applied to mobile computing environments, for a process image becomes compact in the ZG-machine so that it can be moved among processors more quickly. In the usual computing environment for a given program, the ZG-machine can cover larger input instances than the G-machine because the heap-residency is smaller.

References

[1] L. Augustsson, "A Compiler for Lazy ML," In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 218-227, August 1984.

[2] T. Johnsson, "Efficient compilation and lazy evaluation," In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 58-69, June 1984.

[3] S. L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice Hall, 1987.

[4] G. Woo and T. Han, "Compressing the Graphs in G-machine by Tag-Forwarding," *Journal of the Korea Information Science Society*, 26(5):702-712, May 1999.

[5] G. L. Burn, S. L. Peyton Jones, and J. D. Robson, "The spineless G-machine," In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 244-258, July 1988.

[6] S. L. Peyton Jones, "Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-machine," *Journal of Functional Programming*, 2(2):127-202, July 1992.

[7] M. P. Jones, *Hugs 1.3 User Manual — The Haskell User's Gofer System*, August 1996.

[8] D. Wakeling, "A Haskell to Java Virtual Machine Code Compiler," *Proceedings of the 1997 International Workshop on the Implementation of Functional Languages*, pages 39-52, September 1992.

[9] G. Meehan and M. Joy, "Compiling Lazy Functional Programs to Java Bytecode," *Software-Practice and Experience*, 29(7):617-645, June 1999.

[10] Thomas Johnsson, *Compiling Lazy Functional Languages*, PhD thesis, Chalmers Tekniska Högskola, Göteborg, Sweden, January, 1987.

[11] R. R. Fenichel and J. C. Yochelson, "A LISP Garbage Collector for Virtual Memory Computer Systems," *Communications of the ACM*, 12(11):611-612, November 1969.

[12] C. J. Cheney, "A Non-recursive List Compacting Algorithm," *Communications of the ACM*, 13(11):677-678, November 1970.

[13] W. Partain, "The nofib Benchmark Suite of Haskell Programs," In J. Launchbury and P. M. Samson, editors, *Functional Programming, Glasgow*, Workshops in Computing, pages 195-202, Springer Verlag, 1992.

[14] S. L. Peyton Jones and D. R. Lester, *Implementing Functional Languages: a tutorial*, Prentice Hall, 1991.

[15] M. Scheevel, "NORMA: a graph reduction processor," In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 212-219, August 1986.

[16] A. R. Haydarlou and P. H. Hartel, "Thunk Lifting: Reducing Heap Usage in an Implementation of a Lazy Functional Language," *Journal of Functional and Logic Programming*, 1(1):1-24, August, 1995.

[17] D. Wakeling, "The Dynamic Compilation of Lazy Functional Programs," *Journal of Functional Programming*, 8(1):61-81, January 1998.

[18] L. Augustsson and T. Johnsson, "Parallel Graph Reduction with the $\langle \nu, G \rangle$-machine," In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, pages 202-213, 1989.

우 균

1991년 2월 한국과학기술원 전산학 학사. 1993년 2월 한국과학기술원 전산학 석사. 2000년 2월 한국과학기술원 전산학 박사. 현재 동아대학교 전기전자컴퓨터공학부 전임강사. 관심분야는 지연 함수형 언어의 구현과 이를 위한 추상 기계, 프로그램 변환 등임.

한 태 숙

1976년 서울대학교 전자공학과 졸업. 1978년 한국과학기술원 전산학과 졸업. 1990년 Univ. of North Carolina at Chapel Hill 졸업. 현재 한국과학기술원 전자전산학과 부교수. 관심 분야는 프로그래밍 언어론, 함수형 언어임.