数理解析研究所講究録
1125 巻 2000 年 1-11

Recursion Removal under Environments with Cache and Garbage Collection

Kazuhiko KAKEHI*
Graduate School of Science and Engineering,
Waseda University

Yoshihiko FUTAMURA†
School of Science and Engineering,
Waseda University

Abstract

Basis of software development is to guarantee productivity and safety. Programming using recursion and/or garbage collection (GC) satisfies these basis, with sacrificing execution time. Recursion removal, which transforms recursive programs into iterative ones, and improvements toward garbage collection tackle this imbalance of advantages and disadvantages. While garbage collection has researched intensively with showing their effects, this is not the case for recursion removal.

This report explains recursion removal brings great refinement toward recursive programs and also ones running with garbage collection under the current computational environment with cache. In our experiments, recursion-removed programs run at most 9.1 times faster than the original at gcc and 5.7 times faster at Common Lisp.

1 Introduction

Good software development requires productivity and safety, that is, what programmers intend should be easily expressed in programming languages and programs should run as intended without errors.

Programming with recursion is said to be easy: easy to write, easy to verify and easy to maintain; such recursively written programs unfortunately run slowly compared with iterative ones in current computer environments. Recursion removal, which transforms recursive programs into iterative ones, has therefore great importance, but its implementation in compilers is rare except tail recursion removal.

Memory management is necessary for program execution. In imperative languages memory operations like `malloc` and `free` are used. They complicate the large development, because programmers have to know whole lifetime of memory areas, otherwise needed data is overwritten or unnecessary data remains unnoticed forever, which results in quite hard-to-remedy bugs. Under environments with garbage collection (GC for abbreviation) operations for acquiring memory area like `cons` are needed for programmers, because garbage collector as a subprocess collects memory areas which are not referred to any more. Thus GC solves problems of memory management and makes programming easy, as recursion removal is; thus new programming languages like Java start to adopt this technique. As is pointed out frequently, however, programs using garbage collection run slowly.

This report, based on [10], explains through analysing program behaviors and experiments that recursion removal brings great effects on execution time, and additionally time for garbage collection. Especially we believe the latter point, garbage collection, casts new viewpoint

*JSPS Research Fellow, kaz@futamura.info.waseda.ac.jp
†futamura@futamura.info.waseda.ac.jp

to *compile-time garbage collection.*

This reports consists of the following sections. Section 2 shows recursion removal methods and expected refinements, which will be verified by experiments with gcc in the following Section 3. Here we observe effects of recursion removal are greatly influenced by cache. Section 4 explains improvements toward programs under garbage collection environments and show results at Common Lisp. Recursion removal exhibits good improvement both on execution and GC, but in some cases overheads outweigh it. Section 5 summarizes related works, and finally Section 6 concludes.

2 Recursion removal

As [6] explained recursion has an advantage to make program development easy, but iterative programs run faster on von Neumann-style computers than recursive ones. This fact drives researches of recursion removal from [3], but its implementation in compilers are quite rare except for tail recursion, which can be easily translated into iteration.

2.1 Effects expected from recursion removal

By recursion removal we can improve these three points.

costs of recursive calls Recursive programs need recursive calls in their execution. This requires calling stacks and consumes much time for pushing and popping them. Iterative programs eliminate recursive calls, and gain faster execution.

locality of space Elimination of recursive calls has much larger meaning for execution time in current computer environments. Today's processors take advantage of *cache memory.* Cache is a quite fast but small memory area closely located at the processor. Due to its size, programs have to be configured to utilize it efficiently. Recursive programs and iterative ones have great difference here, as the contrast is exemplified in Figure 1. Recursion fails to meet this, because recursion requires stacks and this worsens locality. Iterative programs, in the contrary, only need some fixed size of memory and refer to the area continually, and locality of reference is much higher than recursion.

possibility of inlining One more advantage of iterative programs are possibility to be inlined. Un-inlined subroutines are called separately and take time. Recursive routines are not possible to be inlined, while compilers can inline iterative ones. Recursion removal eliminate this calling penalties.

We have also to be aware of overheads introduced by recursion removal. In some translations, recursion removal newly introduces extra calculation or data structure, and they sometimes spoil or worsen execution time.

2.2 Methods

We need to fix the aim of our recursion removal. The definition of recursion removal we give here is: to translate recursive programs into iterative ones without increasing computational complexity and using stacks. This definition gives two restrictions. Considering this transformation aims at speed up of original programs, the former, complexity, is quite reasonable. The latter, stack, is also acceptable, considering the essence of executing recursion is stacks and that is what we want to eliminate.

Here we give an overview of recursion removal presented in [7]. This method, with basis on *accumulating parameters*, aims at linear recursion, in which at most one recursive call appears in its definition. The top left program of Figure 2 is the general form of right linear recursive programs, which has a recursive call on its right side. Transformation of this target program differs from its auxiliary function a. They utilized the following two properties of auxiliary functions.

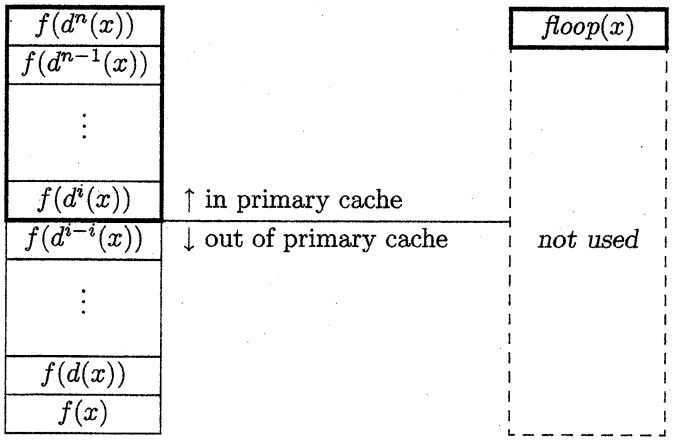


Figure 1: Recursive and iterative calls and their cache behavior

$f(x) =$ **if** $p(x)$
then $b(x)$
else $a(c(x), f(d(x)))$

floop$(x) =$ **if** $p(x)$
then $b(x)$
else do
$v := c(x);\ u := x$
while $\neg p(d(u))$ **do**
$u := d(u)$
$v := h(v, u)$
od
return $a(v, b(d(u)))$
od

floop'$(x) =$ **if** $p(x)$
then $b(x)$
else do
$v := c(x);\ w := v;\ u := x$
while $\neg p(d(u))$ **do**
$u := d(u)$
$w := a'(w, c(u))$
od
$a(w, b(d(u)))$
return v
od

Figure 2: Right linear recursive program f (*left top*) and its iterative form *floop* using a cumulative function (*left bottom*); another iterative form *floop'* using left-pseudo-associativity (*right*)

(1) cumulative functions We can recursion remove by finding a cumulative function $h(v,u)$ of the auxiliary function a. h has to satisfy under the condition $\neg p(u)$

$$a(v, f(u)) = a(h(v,u), f(d(u)))$$

for any expression v. Note that h cannot have recursive calls to f or any free variables.

h can be easily found when a is associative.

$$\begin{aligned} a(v, f(u)) &= a(v, a(c(u), f(d(u)))) \\ &= a(a(v, c(u)), f(d(u))) \end{aligned}$$

implies $h(v,u) = a(v, c(u))$. Consider $g(x)$:

$$\begin{aligned} g(x) = \textbf{if } & p(x) \\ \textbf{then } & b(x) \\ \textbf{else } & c_1(x) + c_2(x) \times g(d(x)) \end{aligned}$$

The auxiliary function of $g(x)$ doesn't have associativity, but its program is recursion-removable using two cumulative functions h_1 and h_2 into the following form:

$$\begin{aligned} &gloop(x) = \\ &\quad \textbf{if } p(x) \textbf{ then } b(x) \\ &\quad \textbf{else do} \\ &\qquad v_1 := c_1(x);\ v_2 := c_2(x);\ u := x \\ &\qquad \textbf{while } \neg p(d(u)) \\ &\qquad\quad \textbf{do } u := d(u) \\ &\qquad\qquad v_1 := h_1(v_1, v_2, u) \\ &\qquad\qquad v_2 := h_2(v_1, v_2, u) \textbf{ od} \\ &\qquad \textbf{return } v_1 + v_2 \times b(d(u)) \\ &\qquad \textbf{od} \end{aligned}$$

with

$$\begin{aligned} h_1(v_1, v_2, u) &= v_1 + v_2 \times c_1(u), \\ h_2(v_1, v_2, u) &= v_2 \times c_2(u). \end{aligned}$$

(2) left-pseudo-associativity While *append* is associative, the core operation `cons` does not have associativity. This hinders recursion removal by cumulative functions. Here we focus on `rplacd` or `rplaca`, which replace `cdr` and `car` part of the cons cell pointed by the first argument with the second, and returns the pointer to the cons cell. These functions has pseudo-associativity and we utilize this property for transformation.

Definition of left-pseudo-associativity is given as:

> Let $a(x,y)$ a function which has side-effects and returns the pointer to x. a is left-pseudo-associative if a', which is called as a *realization function* of a, exists and satisfies the following two equations:
>
> $$\begin{aligned} a(x,y) &= \mathrm{prog}(a'(x,y), x) \\ a(x, a(y,z)) &= \mathrm{prog}(a'(a'(x,y), z), x) \end{aligned}$$
>
> where $\mathtt{prog}(u,v)$ evaluates u and v, and returns the value of v.

Finding a' makes recursion removal possible from a right linear recursive program $f(x)$ to *floop'*(x) in Figure 2. Now we can recursion remove programs with $\mathtt{cons}(a,b)$ by regarding it as $\mathtt{rplacd}(\mathtt{list}(a), b)$. In case $a(x,y) = \mathtt{rplacd}(x,y)$, taking $\mathtt{prog}(a(x,y), y)$ as $a'(x,y)$ satisfies the equations presented previously.

3 Experiments in gcc

First tests of recursion removal were done under gcc on Sun Ultra 5 running SunOS 5.6, which uses UltraSparcIIi, a recent cpu with sufficient primary cache, and on Macintosh Color Classic II running NetBSD 1.3, where 68030, an older and slow cpu with few cache, resides. Details of environments appears at Table 1. Subject programs are numeric functions c and g, which calculate $\lceil \frac{x}{2} \rceil$ for non-negative integer x and $\frac{x+1}{2}$ for integer $x \geq 2$ respectively:

$$\begin{aligned} c(x) = \textbf{if } & x = 0 \\ \textbf{then } & 0 \\ \textbf{else } & x - c(x-1) \end{aligned}$$

$$\begin{aligned} g(x) = \textbf{if } & x = 2 \\ \textbf{then } & \tfrac{3}{2} \\ \textbf{else } & 1 + \tfrac{x-1}{x} \cdot g(x-1) \end{aligned}$$

Figure 3 plots results of g's 1000 calculations for various arguments. This figure shows

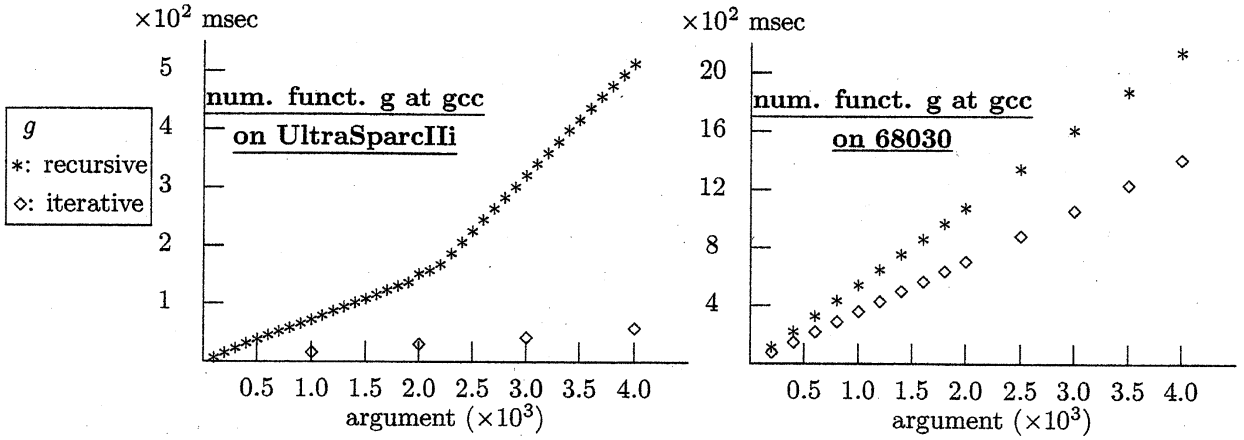


Figure 3: Results of numeric function g at gcc: on UltraSparcIIi (*left*) and on 68030 (*right*)

Table 1: System details

machine	processor	memory	(cache) inst./ data/second.	OS	language
Sun Ultra Enterprise 2	dual UltraSparcI (200MHz)	256MB	16KB/16KB/ 1MB	SunOS5.5.1	ACL4.3.1 GCL2.2
Sun Ultra 5	UltraSparcIIi(270MHz)	64MB	16KB/16KB/ 256KB	SunOS5.6	gcc2.8.1
PowerMac9600	PowerPC604 (233MHz)	160MB	32KB/32KB/ 512KB	MacOS8.1	MCL4.2
ColorClassic II	68030 (33MHz)	20MB	256B/ 256B/ N.A.	NetBSD1.3	gcc 2.7.2.2

1. UltraSparcIIi shows much greater improvement (at most 9.1 times faster in the figure) than 68030 (regularly 1.5 times faster);

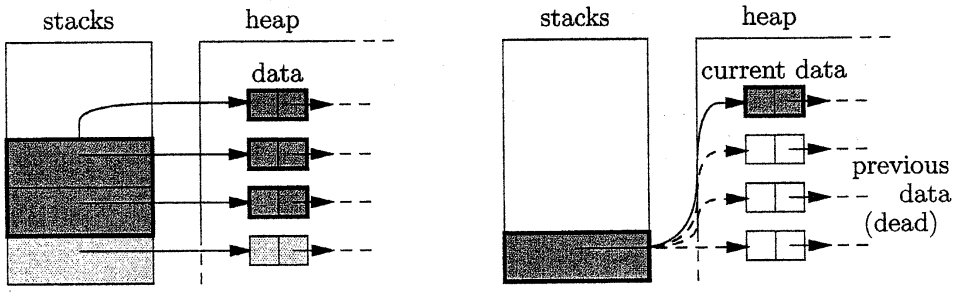
2. a break appears around 2,000 in UltraSparcIIi's recursive results.

The second is what Subsection 2.2 explained: cache miss appears over 2,000 for recursion in UltraSparcIIi and never for iteration. A tiny primary cache and no secondary cache don't work for either recursion or iteration in 68030. Today's processors have adequate cache for minimizing speed gap between memory and processors, and this example shows recursion removal suits the current computing environments greatly.

Experiments in other environments, that is UltraSparc, Pentium and PowerPC, showed scattering in improvements; calculating c with an argument 6,000 becomes 1.3 times faster in one environment and 16.9 times faster in another for example. What is in common is, however, the more arguments become, the more effects of recursion removal is in recent processors having cache of adequate size.

4 Experiments in Common Lisp

Next language we are going to investigate is Common Lisp. Its language processor has a build-in garbage collector and this is one point of our research. Environments for experiments are Allegro Common Lisp (ACL) and Gnu Common Lisp (GCL) running on Sun Ultra Enterprise 2, and Macintosh Common Lisp (MCL) on Power-Macintosh 9600.



grey: memory space to be checked by GC process
dark grey: memory space which is in cache when GC process is invoked

Figure 4: Behaviors of stacks and heap in case temporary data is created: recursion (*left*) and iteration (*right*)

4.1 Effects expected for garbage collection

This subsection gives analysis of the relation between recursion removal and garbage collection.

What we have to notice that GC process is influenced not only by heap cells but other memory area like stacks. GC process, which collects dead cells, i.e. cells without pointers pointing to them, for new allocation, uses *roots* for checking cells. This root consists of global variables and stacks. If this stack grows, GC process takes more time to check roots for finding pointers to cells. Stack is also one element of memory, thus there is memory and cache competition between stack and cells.

Considering these issues, following four improvements on garbage collection are expected by recursion removal. Note that recursion removal only aims at changing stack behavior, not at decreasing total quantity of cells or data creation.

(1) decreasing traversal cost In programming by recursion each stack elements points to (some part of) argument, data made by the stack and upper stack for next data, as seen in Figure 4. For each time GC process traverses all stack elements pushed and all pointers the stack elements have. Recursion removal reduces this cost of traversal by minimizing stacks to some fixed size and reducing pointers to ones needed for current execution. A considerable effect can be expected because root scanning can take up to 70% of the total GC cost [4]. Additionally, iterative programs generated by recursion removal decrease the number of pointers, pointing only to middle of argument lists, top and middle of generating lists, and one of the current temporary result.

(2) shortening lifetime Decreasing pointers changes lifetime of cells or data. Consider the case where temporary results are required to proceed calculation which is illustrated in Figure 4. In recursive case, every temporary result is pointed by each stack element, surviving long until its calculation terminates. Iterative programs, on the contrary, only need the current result, and once the next temporary result is gained the previous is discarded. This lifetime shortening also applies to intermediate data passed between two functions. Intermediate data are not pointed by global variables. When an element of the intermediate list is used, iterative programs replace pointer from the element to the next, implying there are no pointers to data already consumed by the outer iterative program.

This lifetime shortening has great meanings to garbage collection. It does not only reduce cells or data to be checked by GC process. The quantity of the free space after one garbage collection widens and the duration between GCs

becomes longer. Total cell allocation doesn't change by recursion removal, then this decrease of frequency implies decrease of GC invocation.

(3) memory competition In a fixed memory, growing stacks pressurize heap area and increases frequency of garbage collection. This implies recursion essentially increases the number of invoking GC process. Recursion-removed programs can have larger heap, and the smaller invocation of GC.

(4) cache competition As Section 3 pointed out, recursion removal restrains cache miss by minimizing stack size. This also applies to garbage collection. Recursive programs refer to stack elements, data or cells, and arguments, and GC process checks them all: they conflict on cache, and when GC process is invoked only recently referred items remains in cache, as Figure 4 shows, resulting in longer garbage collection. Smaller number of stack elements reduces cache miss and speed up GC process. This effect is massive for programs generating temporary results. Reachable data is the most recently created one, and it is hardly conceivable that this is swept out of cache when GC process is invoked.

4.2 Consing functions

First experiments are programs using conses, that is, *merge*, *halve* and mergesorts. The first two are independent functions which do not call other recursive functions, while the last is a compound function which is composed of plural functions.

Figure 5 shows results of 50 times computation to merge two ascending lists of length 30,000, and results to halve a list of length 60,000 50 times. *halve* takes one list and halve it, and returns a new cons cell which have pointers in `car` and `cdr` parts to halved two lists:

$$halve(\texttt{(3 1 2 5 4)}) = \texttt{((3 2 4) (1 5))}$$

The auxiliary function of *halve* is not a single `cons` but a more complicated one, but we can find its realization function utilizing the auxiliary function.

Generally, good improvements have achieved for both of total and GC time, at most 5.7 times faster on total and 19.8 times faster on GC. The bottom part of figures, plotting, shows sequences of each garbage collection time. Remarkable in iteration are:

- each of GC time are kept under 200 msec, on average 150 msec;
- frequency of GC invocation is reduced.

Change in frequency differs on its execution condition like heap size, and sometimes GC process is invoked more frequently in recursion removed functions than the original recursive ones. In any case, recursion removal reduces each GC time.

Results of mergesorts is shown in Figure 6. Mergesorts can take two forms: one is constructed in a *top-down* manner, tree-recursively halving and merging a list; the other is constructed in a *bottom-up* manner, recursively continuing to merge neighboring two lists. While all parts of bottom-up can be recursion-removed, the calling function of top-down is tree-recursive and all but this can be iterative. The figures shows results of 5 times sorting of one uniform random sequence of length 60,000 generated by Random Data Server [8].

While the top-down mergesort doesn't show refinement by recursion removal as good as independent functions due to its tree structure, the bottom-up one, in which all parts can be recursion removed, is improved a little more. Reduction of GC time is not so much as independent functions like *merge*. This is because (a) frequency doesn't change so much; (b) each GC time of the recursive are kept small. In this mergesorts, we have expected effects of lifetime shortening for intermediate data are created, but it is not noticeable in this results. It is true inlining is possible here, but it is not done under environments experimented with.

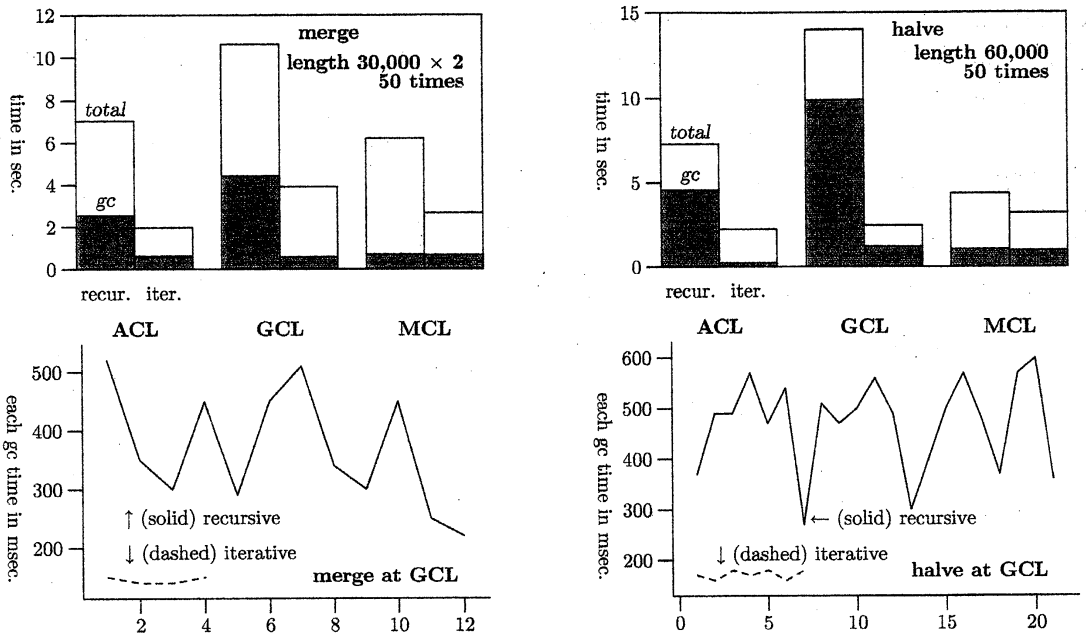


Figure 5: Results of independent functions at Common Lisp: *merge* (*left*) and *halve* (*right*)

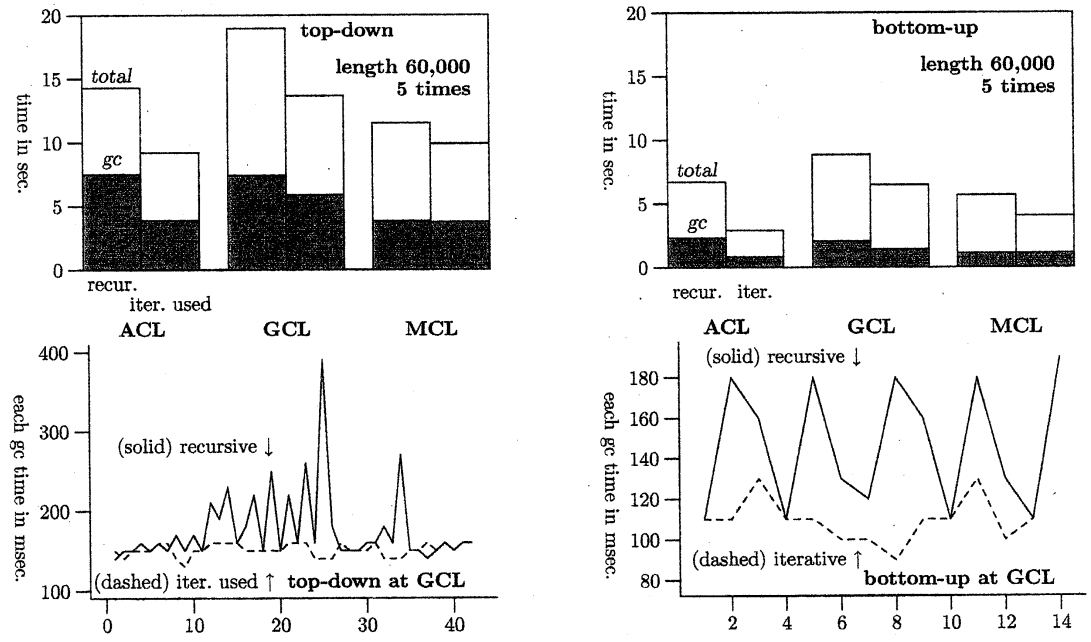


Figure 6: Results of *mergesorts* at Common Lisp: top-down (*left*) and bottom-up (*right*)

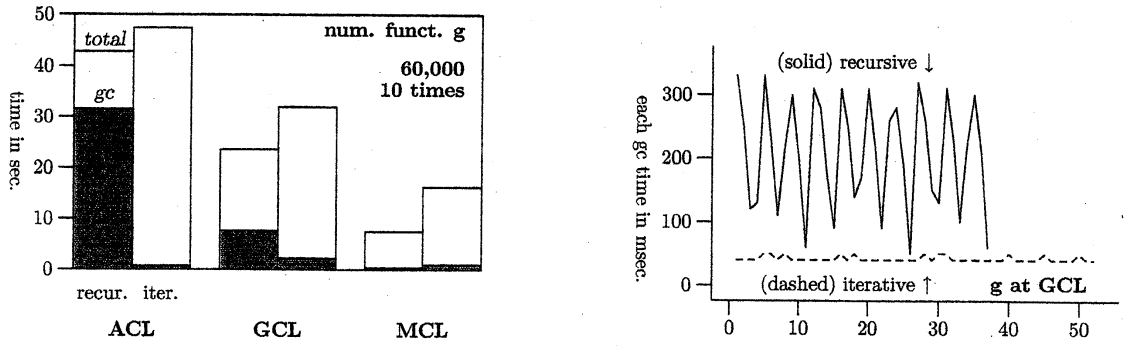


Figure 7: Results of numeric functions at Common Lisp: g

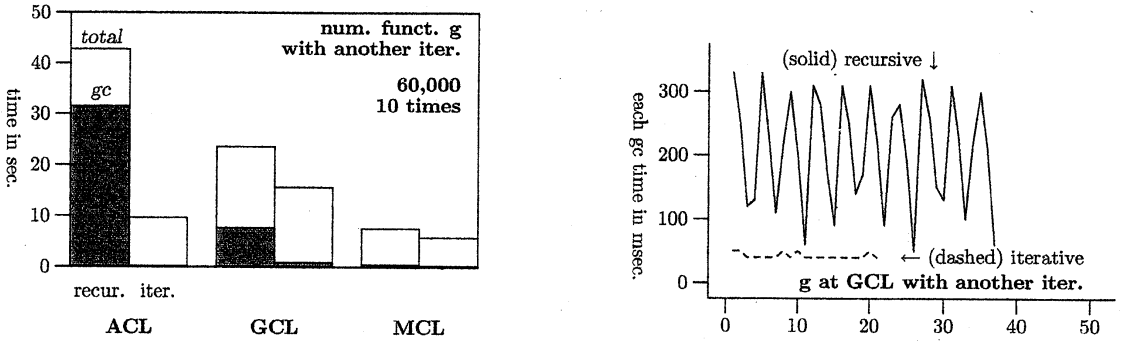


Figure 8: Results of numeric functions at Common Lisp: another g

4.3 Numeric functions

Now results of numeric functions, which creates not cons cells but temporarily calculated data in heap, will be shown. Here applies the analysis of temporary data in Subsection 4.1. Executed functions are c of integer type and g of double float type which appear in Subsection 3. Figure 7 shows results of calculating g with an argument 60,000 10 times, which shows typical characteristics.

Disappointingly almost all of total time worsened. One reason is these recursion-removed functions requires two cumulative functions and introduce more variables (see *gloop* in Subsection 2.2). One more reason is, as [11] explains, recursion removal changes computation from smaller numbers into computation from larger, complicated numbers. Interesting here is garbage collection. Indeed computation and generated data increase due to overheads, total and each GC time decreases in ACL and GCL.

In this method, reduction of GC frequency is hard to notice due to increase of computation. To confirm this benefit, we have checked results of another recursion removal. This transformation follows the evaluation strategy of 'left-most inner-most' (call-by-value). We have to be aware of the following points in case of this transformation:

- the inverse of descent functions $d^{-1}(x)$ is needed,
- conditions $p(x)$ have to be '=';

otherwise auxiliary stacks or lists are required to store the chain of $x, d(x), d^2(x), \ldots$. Our numeric examples, g for instance, fits them and easily translated.

Figure 8 shows the result of g. This time good performance of execution was achieved as well as garbage collection. Following the same

evaluation strategy extra computation or variables are not needed any more. We also notice that the frequency of garbage collection has decreased. There is no change in data creation, so we can conclude that this decrease is due to lifetime shortening introduced by recursion removal.

5 Related Works

As is mentioned in Section 1, recursion removal has energetically studied. Most notable is Burstall and Darlington [3]. This transformation basically utilized template matching to achieve recursion removal. Similar transformation as our cumulative functions is [1] for example. Cohen [5] focused on redundant recursive calls. One of recent works is Harrison and Khoshnevisan [9]. They utilized FP [2], which investigated function-level reasoning, for achieving recursion removal. These methods introduced so far transform basically numeric functions into iterative forms, and don't mention general consing functions. Another recent work by Liu and Stoller [11] introduced quite similar technique as our pseudo-associativity, and showed also similar results. Yet they don't think of benefits brought by recursion removal or benefits of cache.

On effects of garbage collection, Cheng, Harper and Lee [4] mentioned that in the worst case scanning costs of roots take up to 70% of garbage collection time. Their observation fits this report and we can say recursion removal can be the candidate to eliminate that time.

6 Conclusion and Future Works

This report showed the effects of recursion removal on total execution and garbage collection, with data by gcc and Common Lisp. Due to good cache utilization, functions have high possibility to become faster and this has effects to combined functions. In some kinds of programs, however, recursion removal worsens execution by introducing overheads. Garbage collection also benefits from recursion removal especially by the effects of reduction of traversal cost and cache competition.

There are two directions to tackle: power of transformation and automation. As Subsection 4.3 showed, recursion removal by cumulative functions didn't succeed in improving execution. We have to figure out how to circumvent introduction of overheads and achieve successful transformation, like emulation of evaluation strategies. Current methods explained here don't fit for automation yet due to dependency on heuristics. We are now trying to automate these methods, especially one using pseudo-associativity, by analyzing consing structures and making independent from heuristics.

References

[1] J. Arsac and Y. Kodratoff. Some techniques for recursion removal from recursive functions. *ACM Transactions on Programming Languages and Systems*, 4(2):295–322, Apr. 1982.

[2] J. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, Aug. 1978.

[3] R. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, Jan. 1977.

[4] P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI)*, pages 162–173, 1998.

[5] N. Cohen. Eliminating redundant recursive calls. *ACM Transactions on Programming Languages and Systems*, 5(3):265–299, Jan. 1983.

[6] L. Colussi. Recursion as an effective step in program development. *ACM Transactions on Programming Languages and Systems*, 6(1):55–67, Jan. 1984.

[7] Y. Futamura and H. Ootani. Recursion removal rules for linear recursive programs and their effectiveness. *Computer Software (Japanese)*, 15(3):38–49, May 1998.

[8] Y. Futamura, H. Ootani, K. Aoki, and N. Futamura. Fast generation of random permutations with simple indexes. *Computer Software (Japanese)*, 14(6):56–73, Nov. 1997.

[9] P. Harrison and H. Khoshnevisan. A new approach to recursion removal. *Theoretical Computer Science*, 93:91–113, Jan. 1992.

[10] K. Kakehi. Effects of stack and cache on garbage collection (Japanese). Master's thesis, Department of Information and Computer Science, Graduate School of Science and Engineering, Waseda University, Feb. 1999.

[11] Y. Liu and S. Stoller. From recursion to iteration: what are the optimizations? Technical report, Computer Science Department, Indiana University, July 1999.