MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 1462 March, 1994

Garbage Collection is Fast, But a Stack is Faster

James S. Miller (`JMiller@zurich.ai.mit.edu`)
and
Guillermo J. Rozas (`gjr@zurich.ai.mit.edu`)

This publication can be retrieved by anonymous ftp to publications.ai.mit.edu.
The pathname for this publication is: ai-publications/1994/AIM-1462.ps.Z

Abstract

Prompted by claims that garbage collection can outperform stack allocation when sufficient physical memory is available, we present a careful analysis and set of cross-architecture measurements comparing these two approaches for the implementation of continuation (procedure call) frames. When the frames are allocated on a heap they require additional space, increase the amount of data transferred between memory and registers, and, on current architectures, require more instructions. We find that stack allocation of continuation frames outperforms heap allocation in some cases by almost a factor of three. Thus, stacks remain an important implementation technique for procedure calls, even in the presence of an efficient, compacting garbage collector and large amounts of memory.

KEYWORDS: compilers, garbage collection, storage management, performance evaluation.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-89-J-3202.

1 Introduction

In a well-known letter, Appel[1] argues that optimizing compilers for languages that support garbage collection need not attempt to allocate space using a stack if sufficient physical memory is available. Appel's letter reintroduces the copying collector algorithms[2, 3] which, while popular in implementations, have been largely ignored in the literature. These collectors have the property that the cost of garbage collection is proportional to the amount of memory in use (rather than the amount of garbage), and hence is asymptotically zero as the ratio of available memory to memory in use increases.

We restate Appel's claim as follows. Consider a program that requires the allocation and release of n structures as it runs. In a stack-based implementation, the cost of memory management is

$$\text{cost}_s(n) = \text{creation}_s(n) + \text{destruction}_s(n),$$

while in a heap-based implementation the comparable formula is

$$\text{cost}_h(n) = \text{creation}_h(n) + \text{gc}(\text{live}(n)),$$

where $\text{live}(n)$ is the number of structures in active use when a garbage collection occurs. Appel argues that when $\frac{PhysMem}{\text{live}(n)} > 7$, where $PhysMem$ is the amount of physical memory[1], the cost of stack allocation exceeds the cost of heap allocation. Note that creation_s, destruction_s, creation_h, and gc are all linear in their argument.

While we agree with Appel's argument in the fully general case, he fails to consider the details of continuation frames, i.e., records allocated by compiled procedures allowing them to resume after calling another procedure. It is these frames that compilers for languages descended from Algol allocate on the stack, and they do so because they are allocated and deallocated[2] in a stack-like manner, and must be referenced (to retrieve a return address) immediately prior to being released. When a stack is used to allocate these frames, this additional knowledge can be used to efficiently reclaim the storage for the frame by simply popping it off of the stack. By contrast, heap allocation of these frames requires explicitly maintaining a linked list of continuation frames—and this explicit manipulation requires additional storage, memory traffic, and (typically) instructions.

In particular, restricting our attention to continuation frames, we argue that

$$\text{creation}_h(n) > \text{creation}_s(n) + \text{destruction}_s(n),$$

and thus *even if garbage collection never occurs or costs nothing*, the cost of heap allocation exceeds the cost of stack allocation. Our argument depends critically on the *implementation* of the stack, not merely on the abstract stack data type: real systems implement real stacks using a contiguous block of memory with a single pointer into that block.

[1]Technically, $PhysMem$ is the size of a semi-space for a two space garbage collector.

[2]In languages with a `call-with-current-continuation`[4, 7] operator or backtracking operations[8, 5] continuation frames do not form a single list, but rather a tree. While stacks can (and are) used to allocate continuation frames in these languages, the performance tradeoffs are not as straightforward as those presented here. This is an active area of research.

2 Analysis

We now make our argument concrete by presenting actual instruction sequences for both types of implementations. We have done our best to give the benefit of the doubt to the garbage collector by ignoring issues such as the headers or type markers required by most algorithms to determine the size or composition of the garbage-collected heap. We also assume that both heap and stack overflow will be detected by hardware traps, requiring no in-line instructions to test for these conditions.

We have chosen a linear recursive algorithm, the computation of $n!$; the source code is shown in Figure 1. Optimized assembly code for a typical RISC architecture (Digital's Alpha) is shown in Figure 2. This code clearly demonstrates the differences in linkage convention required by the choice of stack vs. heap implementation. The essential point to notice is the increased number of memory references required by the heap allocation technique, arising from the need to maintain the singly-linked frame structure as compared with the simple address-based system used with stack allocation. Although the detailed instruction counts will vary, this overhead will be present on any general purpose computer system. For the particular machine we have chosen, the details are as follows (the item numbers are keyed to Figure 2):

1. The heap allocation version must save the address of the current continuation frame in the new frame before the recursive call.
2. The heap allocation version must copy the heap pointer into the current frame pointer register because these two registers operate independently: in general, neither can be directly computed from the contents of the other.
3. The heap allocation version must restore the frame pointer when the recursive call completes.
4. The stack allocation version must deallocate the frame by modifying the stack pointer when the recursive call completes.

Simply counting instructions gives a rough estimate of the performance of the two versions of the code. The heap allocation version requires $13n + 2$ instructions to compute $n!$, including $3n$ stores to and $3n$ loads from memory. The stack allocation version requires $11n + 2$ instructions with $2n$ stores and $2n$ loads. The stack version also requires one fewer register.

In terms of our earlier analysis, we can assign the following costs (in units of instructions executed):

$$\begin{array}{lcll} \text{creation}_s(n) & = & 3n & \text{(of which 2 are memory stores)} \\ \text{destruction}_s(n) & = & n & \text{(no memory references)} \\ \text{creation}_h(n) & = & 5n & \text{(of which 3 are memory stores)} \end{array}$$

```
int fact(int n)                    (define (fact n)
{ if (n==0)                          (if (= n 0)
    return 1;                            1
  else return n*fact(n-1);               (* n (fact (- n 1)))))
}

int fib(int n)                     (define (fib n)
{ if (n<2)                           (if (< n 2)
    return 1;                            1
  else return fib(n-1) + fib(n-2);       (+ (fib (- n 1))
}                                           (fib (- n 2)))))
```

Figure 1: Factorial and Fibonacci Functions: C and Scheme

Stack	Heap	Comment
`FACT:`		`; Rarg1 contains` N
`beq  Rarg1, DONE`	`beq  Rarg1, DONE`	`; Goto DONE if` $N = 0$
`stl  Rarg1, -4(Rsp)`	`stl  Rarg1, -4(Rhp)`	`; Save` N `for recursive call`
`stl  Rret, -8(Rsp)`	`stl  Rret, -8(Rhp)`	`; Save return address`
◇1◇	`stl  Rframe, -12(Rhp)`	`; Save previous frame`
`subl Rsp, #8, Rsp`	`subl Rhp, #12, Rhp`	`; Allocate contination frame`
◇2◇	`mov  Rhp, Rframe`	`; Point frame to new space`
`subl Rarg1, #1, Rarg1`	`subl Rarg1, #1, Rarg1`	`;` $N \leftarrow N - 1$
`bsr  Rret, FACT`	`bsr  Rret, FACT`	`; Recurse, ret. addr. in Rret`
`AFTERFACT:`		`; Upon return from recursion`
`ldl  Rt1, 4(Rsp)`	`ldl  Rt1, 8(Rframe)`	`; Restore old` N
`ldl  Rret, 0(Rsp)`	`ldl  Rret, 4(Rframe)`	`; Restore return address`
◇3◇	`ldl  Rframe, 0(Rframe)`	`; Restore continuation frame`
`addl Rsp, #8, Rsp`	◇4◇	`Deallocate stack frame`
`mull Rt1, Rval, Rval`	`mull Rt1, Rval, Rval`	`;` $result \leftarrow result \times N$
`jmp  Rzero, (Rret)`	`jmp  Rzero, (Rret)`	`; Return`
`DONE:`		
`lda  Rval, 1(Rzero)`	`lda  Rval, 1(Rzero)`	`; Base case:` $result \leftarrow 1$
`jmp  Rzero, (Rret)`	`jmp  Rzero, (Rret)`	`; Return`

Register convention:

- Arguments passed in registers `Rarg1` ... `Rarg`n. Return value is in `Rval`.
- Temporary registers `Rt1` ... `Rt`m.
- `Rzero` contains 0. Writes are ignored.
- `Rret` contains the return address.
- `Rsp` contains the stack pointer. *Stack version only.*
- `Rframe` points to the current continuation frame. *Heap version only.*
- `Rhp` points to the next available location for heap allocation. *Heap version only.*
- Upon exit the *only* registers with defined values are: `Rval`, `Rsp`, `Rframe` and `Rhp`.

The text is keyed to the numbers surrounded by the ◇ symbol. The Alpha assembler automatically reorganizes code to improve pipeline performance; these optimizations are not shown.

Figure 2: Alpha Assembly Code for `factorial`

In addition, heap allocation requires one additional load instruction per call to restore the previous frame pointer. Overall, the heap version requires 6 instructions per call as compared to 4 per call for the stack version.

3 Measurement

We find the conceptual analysis above to be interesting but not fully compelling. The performance of real machines is considerably more complicated than mere instruction counting; real machines have finite memory, limited caches, small translation look-aside buffers, potentially high load and store latencies, multiple instruction issue and other performance-affecting features. We have undertaken a series of measurements to see how well we are able to predict performance on real systems. Figure 3 shows the results of these measurements. We describe here the details of the measurements taken on the Alpha processor; details of other systems, experimental methodology, and the raw data are in the Appendix.

The measurements in Figure 3 were taken using a Digital Equipment Corporation 3000/400 system (133MHz processor) with 128MBytes of primary memory. Because of the relatively long latency of the integer multiply (`mull`) instruction on this machine, we ran the code in Figure 2 both as shown and with the `mull` instruction changed to integer addition (`addl`). The numbers shown in this table were generated by running the assembly language code shown in Figure 2, with different values of n, a total of 20 times each using the OSF/1 operating system V1.3 (Rev. 111) and a driver loop written in `C`. Because of inevitable operating system overhead the numbers are not precisely replicable, and severely outlying numbers were removed (in no case did we drop more than 4 values, and for $n \geq 10^4$ never more than 2). The remaining numbers are averaged. The timings are based on the `gettimeofday` system call. The `C` driver allocates the memory to be used by the assembly language code; the amount of memory allocated is precisely the amount required by the heap version of the program, $3n$ words of storage. To increase the precision for $n \leq 10^4$, these values were actually computed by running the program $\frac{200{,}000}{n}$ times between timing calls; this process was then repeated 20 times. This helps factor out the overhead of the system call, making all of the values shown in Figure 3 easily reproducible.

Using the instruction counts from the previous section, we predict that performance of the heap allocation algorithm should be 18% worse than the stack allocation version (13 instructions vs. 11 instructions). The actual measurements indicate that the penalty is less than this for programs with a limited number of continuation frames (fewer than 10^4). For larger numbers of frames the cost of heap allocation rises considerably to about 25%. We believe that this jump in cost arises from exceeding the size of the off-chip memory cache: at this point the extra memory traffic required by the heap allocation version becomes more expensive relative to the (constant) computation within the loop.

We observed one additional phenomenon. When the number of frames increases even further (to 10^7), the programs require more virtual address space than there is physical memory on the machines. The effect on performance is dramatic (over an order of magnitude on the Alpha) and it occurs first on the heap allocation version since these require 50% more memory for the same number of frames. We were unable to run the benchmarks with enough frames to force both the stack and heap versions to enter this paging mode, so we cannot continue the performance comparison into this regime.

The data in Figure 3 represents a scenario in which, for both the stack and heap allocator, there is no reuse of continuation frames and the memory used is maximally compact. This is a simple *linear recursive* process, and represents the case in which stack and heap performance are as close to identical as possible (barring, of course, *iterative processes* which create no continuation frames).

One of the advantages of stacks relative to heaps, however, is their ability to immediately reuse storage. The amount of stack space in use is exactly the amount used by the live continuation frames. On a heap, the live continuation frames are interspersed with inactive frames and the space is compacted when a garbage-collection occurs. Since the stack maintains locality, it performs better in the presence of memory hierarchies—at least between garbage collections. Notice that, in the best case, a garbage collection can only improve the locality to match that attained by the stack.

To explore the impact of this loss of locality, we examine a different algorithm: the doubly recursive `fibonacci` function shown in Figure 1. This algorithm requires space (i.e. live continuation frames) linear in n. Using the stack allocation scheme, the actual memory use is linear. With a heap allocation strategy, however, the amount of memory in use is exponential in n in the absence of garbage collection. We again hand-coded `fib` into two assembly language programs, one using stack allocation and the other heap allocation. In order to avoid unduly penalizing the heap allocated version, we wrote the code to use the same continuation frame for computing both $\mathrm{fib}(n-1)$ and $\mathrm{fib}(n-2)$[3]. This alters the constant factor in the maximum size of the heap without changing its order of growth. The results are shown in Figure 4. In this case, we see a performance penalty for using heap allocation that rises steadily with n, to ultimately double the running time of the program.

A compacting garbage collector will improve the running time for the heap allocated version when n is large. For small values of n, however, it is unreasonable to presume that a garbage collection would occur. Furthermore, the cost of testing, invoking, and running the garbage collector is likely to outweigh any advantage it might have. In either case, of course, the performance cannot exceed what was seen in Figure 3, since those measurements are conceptually what would be measured if a garbage collection occured after the completion of every recursive procedure call.

As one final check on our earlier analysis, we repeated these experiments on three other computer architectures to verify that the results arise, as we claim, from prop-

[3]This optimization would be incorrect if the language supports `call-with-current-continuation` or backtracking.

	Fact						Sum					
		Alpha		PA	68K	486		Alpha		PA	68K	486
n	Stack	Heap	Ratio	Ratio	Ratio	Ratio	Stack	Heap	Ratio	Ratio	Ratio	Ratio
10^1	0.22	0.23	1.03	1.20	1.05	1.19	0.14	0.14	1.05	1.32	1.16	1.30
10^2	0.20	0.21	1.04	1.24	1.06	1.20	0.12	0.12	1.06	1.41	1.20	1.41
10^3	0.20	0.21	1.04	1.24	1.15	1.35	0.12	0.15	1.26	1.41	1.47	1.79
10^4	0.20	0.21	1.04	1.24	1.14	1.29	0.14	0.16	1.10	1.42	1.40	1.69
10^5	0.24	0.31	1.30	1.36	1.15	1.31	0.18	0.26	1.41	1.54	1.39	1.52
10^6	0.30	0.37	1.26	1.29	1.13	1.28	0.25	0.33	1.30	1.47	1.39	1.50

All times in microseconds. Ratio is $\frac{Heap}{Stack}$.

Figure 3: Measured performance

		Alpha		PA	68K	486
n	Stack	Heap	Ratio	Ratio	Ratio	Ratio
5	0.10	0.10	1.08	1.15	1.03	1.08
10	1.09	1.19	1.10	1.19	1.03	1.13
15	12.09	16.86	1.40	1.20	1.21	1.66
20	134.13	212.19	1.58	1.56	1.22	2.39
25	1487.24	3030.59	2.04	1.55	1.23	2.98
30	16481.76	33591.33	2.04	1.56	1.23	2.97

All times are in microseconds. Ratio is $\frac{Heap}{Stack}$.

Figure 4: Measured performance on `fibonacci`

erties inherent in the use of heap allocation. We recoded the procedures for the Motorola 68040, Intel 486, and Hewlett-Packard Precision Architecture processors. All of these processors have special purpose instructions to accelerate stack-like operations, and we use these in the stack implementation and wherever possible in the heap implementation. Unlike the Alpha, where an explicit instruction must be used to bump the stack or heap pointer, these machines can perform that operation as a side-effect of the data motion instructions used to store and restore data from the continuation frames. Thus, $\text{destruction}_s(n) = 0$. The results are also shown in Figures 3 and 4.

4 Conclusion

Compilers have traditionally used a stack to store continuation frames, even when the language they implement requires a garbage-collected heap. This tradition has been recently challenged, based on the observation that the cost of garbage collection can be minimized by a careful choice of algorithm and sufficiently large memory. Our investigation, across four architectures and a number of illustrative programs, shows that the traditional strategy outperforms the use of the heap for storing continuation frames. While the numbers vary in detail, in no case does a heap perform better than a stack; and we have measured performance degradation of over a factor of two when a large number of procedure calls must be executed.

The difficulty with heap allocated continuation frames comes from two factors:

1. The size of the continuation frame must be larger if allocated on the heap in order to accomodate a pointer to the previous frame. When frames are on the stack the previous frame pointer can be calculated using address arithmetic on the current frame pointer.
2. Because continuation frames form a singly linked structure when allocated on the heap, the maintenance of the link information requires instructions. In addition, since these instructions reference memory, they are relatively expensive on current machines.

The observation that garbage collection comes for free under certain assumptions is correct. Unfortunately, in the important case of continuation frames, the cost of heap allocation *even without* the added cost of garbage collection exceeds the cost of stack allocation and release.

References

[1] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, 1987.

[2] H. Baker and C. Hewitt. The incremental garbage collection of processes. AI Memo 454, Mass. Inst. of Technology, Artificial Intelligence Laboratory, December 1977.

[3] R. R. Fenichel and J. C. Yochelson. A lisp garbage-collector for virtual-memory computer systems. *Comm. ACM*, 12(11):611–612, 1969.

[4] IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[5] Roberty Kowalski. Predicate logic as a programming language. Technical Report 70, University of Edinburgh, 1973.

[6] James S. Miller and Guillermo Juan Rozas. Garbage collection is fast, but a stack is faster. AI Memo 1462, Mass. Inst. of Technology, Artificial Intelligence Laboratory, 1993.

[7] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[8] Gerald Jay Sussman, Terry Winograd, and Eugene Charniak. MicroPlanner reference manual. AI Memo 203a, Mass. Inst. of Technology, Artificial Intelligence Laboratory, 1971.

A Measured Code

This section contains the actual assembly language code measured on the various machines.

A.1 Alpha

This code is written to be expanded by the `cpp` macro expansion facility.

```
/* -*- Midas -*- */

#include <regdef.h>

#define rarg1  a0
#define rsp    a1
#define rret   ra
#define rt1    t1
#define rval   v0
#define rzero  zero
#define rhp    a1
#define rframe t0

        .align  4
        .globl  stack_fact
        .ent    stack_fact 2
stack_fact:
        beq     rarg1,stack_fact_done
        stl     rarg1,-4(rsp)
        stl     rret,-8(rsp)
        subl    rsp,8,rsp
        subl    rarg1,1,rarg1
        bsr     rret,stack_fact

stack_fact_after:
        ldl     rt1,4(rsp)
        ldl     rret,0(rsp)
        addl    rsp,8,rsp
        mull    rt1,rval,rval
        jmp     rzero,(rret)

stack_fact_done:
        lda     rval,1(rzero)
        jmp     rzero,(rret)
        .end    stack_fact

        .align  4
        .globl  heap_fact
        .ent    heap_fact,2
heap_fact:
        beq     rarg1,heap_fact_done
        stl     rarg1,-4(rhp)
        stl     rret,-8(rhp)
        stl     rframe,-12(rhp)
        subl    rhp,12,rhp
        mov     rhp,rframe
        subl    rarg1,1,rarg1
        bsr     rret,heap_fact

heap_fact_after:
        ldl     rt1,8(rframe)
        ldl     rret,4(rframe)
        ldl     rframe,0(rframe)
        mull    rt1,rval,rval
        jmp     rzero,(rret)

heap_fact_done:
        lda     rval,1(rzero)
        jmp     rzero,(rret)
        .end    heap_fact

        .align  4
        .globl  stack_sum
        .ent    stack_sum 2
stack_sum:
        beq     rarg1,stack_sum_done
        stl     rarg1,-4(rsp)
        stl     rret,-8(rsp)
        subl    rsp,8,rsp
        subl    rarg1,1,rarg1
        bsr     rret,stack_sum

stack_sum_after:
        ldl     rt1,4(rsp)
        ldl     rret,0(rsp)
        addl    rsp,8,rsp
        addl    rt1,rval,rval
        jmp     rzero,(rret)

stack_sum_done:
        lda     rval,1(rzero)
        jmp     rzero,(rret)
        .end    stack_sum

        .align  4
        .globl  heap_sum
        .ent    heap_sum 2
heap_sum:
        beq     rarg1,heap_sum_done
        stl     rarg1,-4(rhp)
        stl     rret,-8(rhp)
```

```
        stl     rframe,-12(rhp)
        subl    rhp,12,rhp
        mov     rhp,rframe
        subl    rarg1,1,rarg1
        bsr     rret,heap_sum

heap_sum_after:
        ldl     rt1,8(rframe)
        ldl     rret,4(rframe)
        ldl     rframe,0(rframe)
        addl    rt1,rval,rval
        jmp     rzero,(rret)

heap_sum_done:
        lda     rval,1(rzero)
        jmp     rzero,(rret)
        .end    heap_sum

        .align  4
        .globl  stack_fib
        .ent    stack_fib 2
stack_fib:
        cmplt   rarg1,2,rt1
        bne     rt1,stack_fib_done
        stl     rarg1,-4(rsp)
        stl     rret,-8(rsp)
        subl    rsp,8,rsp
        subl    rarg1,1,rarg1
        bsr     rret,stack_fib

stack_fib_after_1:
        ldl     rarg1,4(rsp)
        stl     rval,4(rsp)
        subl    rarg1,2,rarg1
        bsr     rret,stack_fib

stack_fib_after_2:
        ldl     4(rsp),rt1
        ldl     0(rsp),rret
        addl    rsp,8,rsp
        addl    rt1,rval,rval
        jmp     rzero,(rret)

stack_fib_done:
        mov     rarg1,rval
        jmp     rzero,(rret)
        .end    stack_fib

        .align  4
        .globl  heap_fib
        .ent    heap_fib 2
heap_fib:
        cmplt   rarg1,2,rt1
        bne     rt1,heap_fib_done
        stl     rarg1,-4(rhp)
        stl     rret,-8(rhp)
        stl     rframe,-12(rhp)
        subl    rhp,12,rhp
        mov     rhp,rframe
        subl    rarg1,1,rarg1
        bsr     rret,heap_fib

heap_fib_after_1:
        ldl     rarg1,8(rframe)
        stl     rval,8(rframe)
        subl    rarg1,2,rarg1
        bsr     rzero,(rret)

heap_fib_after_2:
        ldl     rt1,8(rframe)
        ldl     rret,4(rframe)
        ldl     rframe,0(rframe)
        addl    rt1,rval,rval
        jmp     rzero,(rret)

heap_fib_done:
        mov     rarg1,rval
        jmp     rzero,(rret)
        .end    heap_fib
```

A.2 HP PA

This code is written to be expanded by the `m4` macro expansion facility. Some lines have been split and indented for presentation purposes.

```
changecom(`;');;; -* Midas -*-

define(rarg1,`26')
define(rsp,`25')
define(rret,`2')
define(rt1,`24')
define(rval,`28')
define(rzero,`0')
define(rhp,`25')
define(rframe,`23')

        .SPACE  $TEXT$
        .SUBSPA $CODE$,QUAD=0,ALIGN=8,
                ACCESS=44,CODE_ONLY
stack_sum
        .PROC
        .CALLINFO CALLER,FRAME=0
        .ENTRY
        COMB,=,N rzero,rarg1,stack_sum_done
        STWM    rarg1,-4(0,rsp)
        STWM    rret,-4(0,rsp)
        BL      stack_sum,rret
        ADDI    -1,rarg1,rarg1

stack_sum_after
        LDWM    4(0,rsp),rret
        LDWM    4(0,rsp),rt1
        BV      0(rret)
        ADD     rt1,rval,rval

stack_sum_done
        BV      0(rret)
        .EXIT
        LDI     1,rval
        .PROCEND

        .SPACE  $TEXT$
        .SUBSPA $CODE$,QUAD=0,ALIGN=8,
```

```
                ACCESS=44,CODE_ONLY
heap_sum
        .PROC
        .CALLINFO CALLER,FRAME=0
        .ENTRY
        COMB,=,N rzero,rarg1,heap_sum_done
        STWM    rarg1,-4(0,rhp)
        STWM    rret,-4(0,rhp)
        STWM    rframe,-4(0,rhp)
        COPY    rhp,rframe
        BL      heap_sum,rret
        ADDI    -1,rarg1,rarg1

heap_sum_after
        LDW     8(0,rframe),rt1
        LDW     4(0,rframe),rret
        LDW     0(0,rframe),rframe
        BV      0(rret)
        ADD     rt1,rval,rval

heap_sum_done
        BV      0(rret)
        .EXIT
        LDI     1,rval
        .PROCEND


        .SPACE  $TEXT$
        .SUBSPA $CODE$,QUAD=0,ALIGN=8,
                ACCESS=44,CODE_ONLY
stack_fact
        .PROC
        .CALLINFO CALLER,FRAME=0
        .ENTRY
        COMB,=,N rzero,rarg1,stack_fact_done
        STWM    rarg1,-4(0,rsp)
        STWM    rret,-4(0,rsp)
        BL      stack_fact,rret
        ADDI    -1,rarg1,rarg1

stack_fact_after
        STW     rval,-4(0,rsp)
        LDWM    4(0,rsp),rret
        FLDWS,MA 4(0,rsp),%fr4L
        FLDWS   -12(0,rsp),%fr4R
        XMPYU   %fr4L,%fr4R,%fr5
        FSTWS   %fr5R,-4(0,rsp)
        BV      0(rret)
        LDW     -4(0,rsp),rval

stack_fact_done
        BV      0(rret)
        .EXIT
        LDI     1,rval
        .PROCEND

        .SPACE  $TEXT$
        .SUBSPA $CODE$,QUAD=0,ALIGN=8,
                ACCESS=44,CODE_ONLY
heap_fact
        .PROC
        .CALLINFO CALLER,FRAME=0
        .ENTRY
        COMB,=,N rzero,rarg1,heap_fact_done
        STWM    rarg1,-4(0,rhp)
        STWM    rret,-4(0,rhp)
        STWM    rframe,-4(0,rhp)
        COPY    rhp,rframe
        BL      heap_fact,rret
        ADDI    -1,rarg1,rarg1

heap_fact_after
        STW     rval,-4(0,rhp)
        FLDWS   8(0,rframe),%fr4L
        LDW     4(0,rframe),rret
        LDW     0(0,rframe),rframe
        FLDWS   -4(0,rhp),%fr4R
        XMPYU   %fr4L,%fr4R,%fr5
        FSTWS   %fr5R,-4(0,rsp)
        BV      0(rret)
        LDW     -4(0,rsp),rval

heap_fact_done
        BV      0(rret)
        .EXIT
        LDI     1,rval
        .PROCEND

        .SPACE  $TEXT$
        .SUBSPA $CODE$,QUAD=0,ALIGN=8,
                ACCESS=44,CODE_ONLY
stack_fib
        .PROC
        .CALLINFO CALLER,FRAME=0
        .ENTRY
        COMIB,>,N 2,rarg1,stack_fib_done
        STWM    rarg1,-4(0,rsp)
        STWM    rret,-4(0,rsp)
        BL      stack_fib,rret
        ADDI    -1,rarg1,rarg1

stack_fib_after_1
        LDW     4(0,rsp),rarg1
        STW     rval,4(0,rsp)
        BL      stack_fib,rret
        ADDI    -2,rarg1,rarg1

stack_fib_after_2
        LDWM    4(0,rsp),rret
        LDWM    4(0,rsp),rt1
        BV      0(rret)
        ADD     rt1,rval,rval

stack_fib_done
        BV      0(rret)
        .EXIT
        COPY    rarg1,rval
        .PROCEND

        .SPACE  $TEXT$
        .SUBSPA $CODE$,QUAD=0,ALIGN=8,
                ACCESS=44,CODE_ONLY
```

```
heap_fib
        .PROC
        .CALLINFO CALLER,FRAME=0
        .ENTRY
        COMIB,>,N 2,rarg1,heap_fib_done
        STWM    rarg1,-4(0,rhp)
        STWM    rret,-4(0,rhp)
        STWM    rframe,-4(0,rhp)
        COPY    rhp,rframe
        BL      heap_fib,rret
        ADDI    -1,rarg1,rarg1

heap_fib_after_1
        LDW     8(0,rframe),rarg1
        STW     rval,8(0,rframe)
        BL      heap_fib,rret
        ADDI    -2,rarg1,rarg1

heap_fib_after_2
        LDW     4(0,rframe),rret
        LDW     8(0,rframe),rt1
        LDW     0(0,rframe),rframe
        BV      0(rret)
        ADD     rt1,rval,rval

heap_fib_done
        BV      0(rret)
        .EXIT
        COPY    rarg1,rval
        .PROCEND

        .SPACE  $TEXT$
        .SUBSPA $CODE$
        .EXPORT stack_fact,PRIV_LEV=3,
                ARGW0=GR,ARGW1=GR,RTNVAL=GR
        .EXPORT heap_fact,PRIV_LEV=3,
                ARGW0=GR,ARGW1=GR,RTNVAL=GR
        .EXPORT stack_sum,PRIV_LEV=3,
                ARGW0=GR,ARGW1=GR,RTNVAL=GR
        .EXPORT heap_sum,PRIV_LEV=3,
                ARGW0=GR,ARGW1=GR,RTNVAL=GR
        .EXPORT stack_fib,PRIV_LEV=3,
                ARGW0=GR,ARGW1=GR,RTNVAL=GR
        .EXPORT heap_fib,PRIV_LEV=3,
                ARGW0=GR,ARGW1=GR,RTNVAL=GR
        .END
```

A.3 MC68K

This code is written to be expanded by the `m4` macro expansion facility.

```
### -*-Midas-*-

define(rarg1,`%d1')
define(rsp,`%sp')
define(rval,`%d0')
define(rhp,`%sp')
define(rframe,`%a1')
define(ratemp,`%a0')

        global  _stack_fact
_stack_fact:
        pea     (%a5)           # save a5
        lea     (%sp),%a5
        mov.l   8(%a5),rarg1    # n
        mov.l   12(%a5),%sp     # memory
        bsr     i_stack_fact
        mov.l   %a5,%sp
        mov.l   (%sp)+,%a5
        rts

        global  _heap_fact
_heap_fact:
        pea     (%a5)           # save a5
        lea     (%sp),%a5
        mov.l   8(%a5),rarg1    # n
        mov.l   12(%a5),%sp     # memory
        bsr     i_heap_fact
        mov.l   %a5,%sp
        mov.l   (%sp)+,%a5
        rts

        global  _stack_sum
_stack_sum:
        pea     (%a5)           # save a5
        lea     (%sp),%a5
        mov.l   8(%a5),rarg1    # n
        mov.l   12(%a5),%sp     # memory
        bsr     i_stack_sum
        mov.l   %a5,%sp
        mov.l   (%sp)+,%a5
        rts

        global  _heap_sum
_heap_sum:
        pea     (%a5)           # save a5
        lea     (%sp),%a5
        mov.l   8(%a5),rarg1    # n
        mov.l   12(%a5),%sp     # memory
        bsr     i_heap_sum
        mov.l   %a5,%sp
        mov.l   (%sp)+,%a5
        rts

        global  _stack_fib
_stack_fib:
        pea     (%a5)           # save a5
        lea     (%sp),%a5
        mov.l   8(%a5),rarg1    # n
        mov.l   12(%a5),%sp     # memory
        bsr     i_stack_fib
        mov.l   %a5,%sp
        mov.l   (%sp)+,%a5
        rts

        global  _heap_fib
_heap_fib:
        pea     (%a5)           # save a5
        lea     (%sp),%a5
        mov.l   8(%a5),rarg1    # n
        mov.l   12(%a5),%sp     # memory
        bsr     i_heap_fib
        mov.l   %a5,%sp
```

```
        mov.l   (%sp)+,%a5
        rts

i_stack_fact:
        tst.l   rarg1
        beq     i_stack_fact_done
        mov.l   rarg1,-(rsp)
        subq.l  &1,rarg1
        bsr     i_stack_fact

i_stack_fact_after:
        muls.l  (rsp)+,rval
        rts

i_stack_fact_done:
        movq    &1,rval
        rts

i_heap_fact:
        tst.l   rarg1
        beq     i_heap_fact_done
        link    rframe,&-4
        mov.l   rarg1,-4(rframe)
        subq.l  &1,rarg1
        bsr     i_heap_fact

i_heap_fact_after:
        mov.l   4(rframe),ratemp
        muls.l  -4(rframe),rval
        mov.l   (rframe),rframe
        jmp     (ratemp)

i_heap_fact_done:
        movq    &1,rval
        rts

i_stack_sum:
        tst.l   rarg1
        beq     i_stack_sum_done
        mov.l   rarg1,-(rsp)
        subq.l  &1,rarg1
        bsr     i_stack_sum

i_stack_sum_after:
        add.l   (rsp)+,rval
        rts

i_stack_sum_done:
        movq    &1,rval
        rts

i_heap_sum:
        tst.l   rarg1
        beq     i_heap_sum_done
        link    rframe,&-4
        mov.l   rarg1,-4(rframe)
        subq.l  &1,rarg1
        bsr     i_heap_sum

i_heap_sum_after:
        mov.l   4(rframe),ratemp
        add.l   -4(rframe),rval
        mov.l   (rframe),rframe
        jmp     (ratemp)

i_heap_sum_done:
        movq    &1,rval
        rts

i_stack_fib:
        cmp.l   rarg1,&2
        blt     i_stack_fib_done
        mov.l   rarg1,-(rsp)
        subq.l  &1,rarg1
        bsr     i_stack_fib

i_stack_fib_after_1:
        mov.l   (rsp),rarg1
        mov.l   rval,(rsp)
        subq.l  &2,rarg1
        bsr     i_stack_fib

i_stack_fib_after_2:
        add.l   (rsp)+,rval
        rts

i_stack_fib_done:
        mov.l   rarg1,rval
        rts

i_heap_fib:
        cmp.l   rarg1,&2
        blt     i_heap_fib_done
        link    rframe,&-4
        mov.l   rarg1,-4(rframe)
        subq.l  &1,rarg1
        bsr     i_heap_fib

i_heap_fib_after_1:
        mov.l   -4(rframe),rarg1
        mov.l   rval,-4(rframe)
        subq.l  &2,rarg1
        bsr     i_heap_fib

i_heap_fib_after_2:
        mov.l   4(rframe),ratemp
        add.l   -4(rframe),rval
        mov.l   (rframe),rframe
        jmp     (ratemp)

i_heap_fib_done:
        mov.l   rarg1,rval
        rts
```

A.4 i486

This code is written to be expanded by the **m4** macro expansion facility.

```
### -*- Midas -*-

define(rarg1,`%ecx')
define(rsp,`%esp')
define(rval,`%eax')
```

```
define(rhp,`%esp')
define(rframe,`%ebp')
define(rt1,`%esi')

        .text

        .align 2
        .globl  _stack_fact
_stack_fact:
        pushl   %ebp            # save ebp
        pushl   %ebx            # save ebx
        push    %esi            # save esi
        movl    %esp,%ebx       # save esp in ebx
        movl    16(%ebx),rarg1# n
        movl    20(%ebx),%esp # memory
        call    i_stack_fact
        movl    %ebx,%esp
        popl    %esi
        popl    %ebx
        popl    %ebp
        ret

        .align 2
        .globl  _heap_fact
_heap_fact:
        pushl   %ebp            # save ebp
        pushl   %ebx            # save ebx
        push    %esi            # save esi
        movl    %esp,%ebx       # save esp in ebx
        movl    16(%ebx),rarg1# n
        movl    20(%ebx),%esp # memory
        call    i_heap_fact
        movl    %ebx,%esp
        popl    %esi
        popl    %ebx
        popl    %ebp
        ret

        .align 2
        .globl  _stack_sum
_stack_sum:
        pushl   %ebp            # save ebp
        pushl   %ebx            # save ebx
        push    %esi            # save esi
        movl    %esp,%ebx       # save esp in ebx
        movl    16(%ebx),rarg1# n
        movl    20(%ebx),%esp # memory
        call    i_stack_sum
        movl    %ebx,%esp
        popl    %esi
        popl    %ebx
        popl    %ebp
        ret

        .align 2
        .globl  _heap_sum
_heap_sum:
        pushl   %ebp            # save ebp
        pushl   %ebx            # save ebx
        push    %esi            # save esi
        movl    %esp,%ebx       # save esp in ebx
        movl    16(%ebx),rarg1# n
        movl    20(%ebx),%esp # memory
        call    i_heap_sum
        movl    %ebx,%esp
        popl    %esi
        popl    %ebx
        popl    %ebp
        ret

        .align 2
        .globl  _stack_fib
_stack_fib:
        pushl   %ebp            # save ebp
        pushl   %ebx            # save ebx
        push    %esi            # save esi
        movl    %esp,%ebx       # save esp in ebx
        movl    16(%ebx),rarg1# n
        movl    20(%ebx),%esp # memory
        call    i_stack_fib
        movl    %ebx,%esp
        popl    %esi
        popl    %ebx
        popl    %ebp
        ret

        .align 2
        .globl  _heap_fib
_heap_fib:
        pushl   %ebp            # save ebp
        pushl   %ebx            # save ebx
        push    %esi            # save esi
        movl    %esp,%ebx       # save esp in ebx
        movl    16(%ebx),rarg1# n
        movl    20(%ebx),%esp # memory
        call    i_heap_fib
        movl    %ebx,%esp
        popl    %esi
        popl    %ebx
        popl    %ebp
        ret

        .align 2
i_stack_fact:
        cmpl    $0,rarg1
        je      i_stack_fact_done
        pushl   rarg1
        subl    $1,rarg1
        call    i_stack_fact

i_stack_fact_after:
        popl    rt1
        imull   rt1             # implicit %eax = rval
        ret

i_stack_fact_done:
        movl    $1,rval
        ret

        .align 2
i_heap_fact:
        cmpl    $0,rarg1
```

```
        je      i_heap_fact_done
        pushl   rarg1
        pushl   rframe
        movl    rsp,rframe
        subl    $1,rarg1
        call    i_heap_fact

i_heap_fact_after:
        movl    8(rframe),rt1 # return address
        imull   4(rframe) # implicit %eax = rval
        movl    0(rframe),rframe
        jmp     *rt1

i_heap_fact_done:
        movl    $1,rval
        ret

        .align 2
i_stack_sum:
        cmpl    $0,rarg1
        je      i_stack_sum_done
        pushl   rarg1
        subl    $1,rarg1
        call    i_stack_sum

i_stack_sum_after:
        popl    rt1
        addl    rt1,rval
        ret

i_stack_sum_done:
        movl    $1,rval
        ret

        .align 2
i_heap_sum:
        cmpl    $0,rarg1
        je      i_heap_sum_done
        pushl   rarg1
        pushl   rframe
        movl    rsp,rframe
        subl    $1,rarg1
        call    i_heap_sum

i_heap_sum_after:
        movl    8(rframe),rt1 # return address
        addl    4(rframe),rval
        movl    0(rframe),rframe
        jmp     *rt1

i_heap_sum_done:
        movl    $1,rval
        ret

        .align 2
i_stack_fib:
        cmpl    $2,rarg1
        jl      i_stack_fib_done
        pushl   rarg1
        subl    $1,rarg1
        call    i_stack_fib

i_stack_fib_after_1:
        popl    rarg1
        pushl   rval
        subl    $2,rarg1
        call    i_stack_fib

i_stack_fib_after_2:
        popl    rt1
        addl    rt1,rval
        ret

i_stack_fib_done:
        movl    rarg1,rval
        ret

        .align 2
i_heap_fib:
        cmpl    $2,rarg1
        jl      i_heap_fib_done
        pushl   rarg1
        pushl   rframe
        movl    rsp,rframe
        subl    $1,rarg1
        call    i_heap_fib

i_heap_fib_after_1:
        movl    4(rframe),rarg1
        movl    rval,4(rframe)
        subl    $2,rarg1
        call    i_heap_fib

i_heap_fib_after_2:
        addl    4(rframe),rval
        movl    8(rframe),rt1
        movl    0(rframe),rframe
        jmp     *rt1

i_heap_fib_done:
        movl    rarg1,rval
        ret
```

B Driver Loop

Measurements of `fact` and `sum` were taken using the following driver program. The program was slightly modified to measure `fib` to account for the different space growth.

```
/* -*- C -*- */

#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>

extern int stack_fact (int, void *);
extern int stack_sum (int, void *);
extern int heap_fact (int, void *);
extern int heap_sum (int, void *);

#define SUBTRACT_TIMES(time2, time1)                                    \
  ((((time2.tv_sec) - (time1.tv_sec)) *                                 \
    1000000) +                                                          \
   ((time2.tv_usec) - (time1.tv_usec)))

#define CANONICALIZE_TIME(time)                 ((time <= 0) ? 1 : time)

#if 0
#define mem_alloc malloc
#define mem_free free
#else

void
mem_free (void * brk_val)
{
  extern int brk (void *);

  (void) brk (brk_val);
  return;
}

void *
mem_alloc (long nbytes)
{
  extern void * sbrk (ssize_t);
  void * base = (sbrk (nbytes));

  if (base == ((void *) -1))
    return ((void *) NULL);
  else
    return (base);
}

#endif
```

```
long
do_test (int (* fun) (int, void *), char * name, int n,
         int memory_size, int iterations, int iterations2, int hc)
{
  long * individual_times;
  char * memory_i, * memory;
  int result = -1, count, count2;
  struct timezone tzp;
  struct timeval time_0, time_1;
  long elapsed_time, total_time;

  individual_times = ((long *) (mem_alloc (iterations * (sizeof (long)))));
  if (individual_times == ((long *) NULL))
  {
    fprintf (stderr, "Unable to allocate %d bytes.\n",
             (iterations * (sizeof (long))));
    exit (1);
  }

  memory = ((char *) (mem_alloc (memory_size * (sizeof (int)))));
  if (memory == ((char *) NULL))
  {
    fprintf (stderr, "Unable to allocate %d bytes.\n",
             (memory_size * (sizeof (int))));
    exit (1);
  }

  memory_i = memory;
  memory += (memory_size * (sizeof (int)));

  if (hc == 1)
    result = ((* fun) (n, memory));

  total_time = 0;
  for (count = 0; count < iterations; count++)
  {
    gettimeofday (&time_0, &tzp);
    for (count2 = 0; count2 < iterations2; count2++)
      result = ((* fun) (n, memory));
    gettimeofday (&time_1, &tzp);
    elapsed_time = (SUBTRACT_TIMES (time_1, time_0));
    individual_times[count] = elapsed_time;
    total_time += elapsed_time;
  }

  mem_free (memory_i);

  printf
    ("%s (%d) returned %d was run %d:%d times and took %ld usec. total.\n",
     name, n, result, iterations, iterations2, total_time);

  for (count = 0; count < iterations; count++)
    printf ("\tIteration %d took %ld usec.\n", count, individual_times[count]);

  mem_free (individual_times);
  return (total_time);
}
```

```
void
main (int argc, char ** argv)
{
  int n, memory_size, iterations, iterations2, hc, result;
  long stack_fact_time, heap_fact_time, stack_sum_time, heap_sum_time;
  double fact_ratio, sum_ratio;
  long fact_diff, sum_diff;

  if ((argc < 2) || (argc > 6))
  {
    fprintf (stderr, "usage: %s n [iter iter2 hc memory-size ]\n", argv[0]);
    exit (1);
  }

  n = (atoi (argv[1]));

  if (argc > 2)
    iterations = (atoi (argv[2]));
  else
    iterations = 1;

  if (argc > 3)
    iterations2 = (atoi (argv[3]));
  else
    iterations2 = 1;

  if (argc > 4)
    hc = (atoi (argv[4]));
  else
    hc = 0;

  if (argc > 5)
    memory_size = (atoi (argv[5]));
  else
    memory_size = (n * 3);

  printf ("n = %d; memory_size = %d; iterations = %d; hc = %d\n",
          n, memory_size, iterations, hc);

  stack_fact_time = do_test (stack_fact, "stack_fact", n,
                             memory_size, iterations, iterations2, hc);
  heap_fact_time = do_test (heap_fact, "heap_fact", n,
                            memory_size, iterations, iterations2, hc);
  stack_sum_time = do_test (stack_sum, "stack_sum", n,
                            memory_size, iterations, iterations2, hc);
  heap_sum_time = do_test (heap_sum, "heap_sum", n,
                           memory_size, iterations, iterations2, hc);

  fact_diff = heap_fact_time - stack_fact_time;
  sum_diff = heap_sum_time - stack_sum_time;

  if (stack_fact_time != 0)
    fact_ratio = ((double) heap_fact_time) / ((double) stack_fact_time);
  else
    fact_ratio = -1.;

  if (stack_sum_time != 0)
    sum_ratio = ((double) heap_sum_time) / ((double) stack_sum_time);
  else
    sum_ratio = -1.;
```

```
  printf ("\n\tTotal\t\tPer Iteration\tPer n\t\tRatio\n");
  printf ("\tTime Delta\tTime Delta\tTime Delta\n\n");

  printf ("fact\t%ld usec.\t%ld usec.\t%3ld nsec.\t%f\n",
          fact_diff,
          ((fact_diff + (iterations / 2)) / iterations),
          (((fact_diff * 1000) + ((((long) iterations) * n) / 2))
           / (((long) iterations) * n)),
          fact_ratio);

  printf ("sum\t%ld usec.\t%ld usec.\t%3ld nsec.\t%f\n",
          sum_diff,
          ((sum_diff + (iterations / 2)) / iterations),
          (((sum_diff * 1000) + ((((long) iterations) * n) / 2))
           / (((long) iterations) * n)),
          sum_ratio);

  exit (0);
}
```

C Raw Data and Data Reduction

The raw data was collected and editted into the following Scheme program to reduce the data.

```
(define-structure
  (data-value (conc-name data-value/)
              (constructor make-data-value (name n raw)))
  raw
  name
  n
  (length (length raw))
  (mean (mean raw))
  (gmean (gmean raw))
  (std-dev (std-dev raw))
  (cooked false))

(define (mean x)
  (/ (apply + x) (exact->inexact (length x))))

(define (gmean x)
  (expt (apply * x) (exact->inexact (/ 1 (length x)))))

(define (std-dev x)
  (define (square x) (* x x))
  (let ((ave (mean x)))
    (sqrt (/ (apply + (map (lambda (val) (square (- val ave)))
                           x))
             (- (length x) 1)))))

(define (make-scaled-data-value factor n name data)
  (make-data-value name n
                   (map (lambda (x) (/ x factor)) data)))

(define (cook! data-value procedure)
  (set-data-value/cooked! data-value (procedure data-value))
  'OK)

(define (toss-outliers dv)
  (list-transform-negative (data-value/raw dv)
    (let ((mean (data-value/mean dv))
          (std-dev*2 (* 2 (data-value/std-dev dv))))
      (lambda (value) (> (abs (- value mean)) std-dev*2) ))))
```

```
(define (round-to n-places)
  (let ((power (expt 10 n-places)))
    (lambda (x)
      (/ (round (* power x)) power))))

(define (cook-em! vals)
  (for-each
   (lambda (dv)
     (cook! dv toss-outliers)
     (write-line (list (data-value/name dv)
                       (- (data-value/length dv)
                          (length (data-value/cooked dv))))))
   vals))

(define (show-mean-and-std vals)
  (for-each (lambda (dv)
              (let ((mean (mean (data-value/cooked dv)))
                    (rounder (round-to 2))
                    (n (data-value/n dv)))
                (write-line
                 (list (data-value/name dv)
                       (rounder (/ mean n))
                       (rounder
                        (/ (std-dev (data-value/cooked dv))
                           (* n mean)))))))
            vals))

(define (pair-up l)
  (if (null? l)
      '()
      (cons (list (car l) (cadr l))
            (pair-up (cddr l)))))

(define (show-ratios vals)
  (let ((paired-values (pair-up vals)))
    (for-each
     (let ((rounder (round-to 3)))
       (lambda (dvs)
         (write-line (list (data-value/name (car dvs))
                           (rounder
                            (/ (mean (data-value/cooked (cadr dvs)))
                               (mean (data-value/cooked (car dvs)))))))))
     paired-values)))

(define (do-it architecture procedures vals)
  (write-line (list architecture procedures "COOK"))
  (newline)
  (cook-em! vals)
  (newline)
  (write-line (list architecture procedures
                    "Mean and Standard Deviation"))
  (newline)
  (show-mean-and-std vals)
  (newline)
  (write-line (list architecture procedures "Ratios"))
  (newline)
  (show-ratios vals))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
ALPHA (Win's capsicum, a 3000/400 with 133MHz CPU 128MBytes memory
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (alpha)
  (define stack_fact-3
    (make-scaled-data-value
     200 1000 'stack_fact-3
     '(40016 40016 40016 40016 40016 39040 41968 39040 40016 40016
             40016 40592 40016 39040 40016 40016 40016 39040 40992
             40016)))

  (define heap_fact-3
    (make-scaled-data-value
     200 1000 'heap_fact-3
     '(41968 41968 40992 41968 40992 40992 41968 40992 42944 40992
             41968 40992 41968 41568 41968 40992 41968 40992 41968
             40992)))

  (define stack_sum-3
    (make-scaled-data-value
     200 1000 'stack_sum-3
     '(23424 23424 22448 23424 23424 22448 23424 23424 23424 23424
             22448 22448 23424 23424 24400 23424 23424 23424 23424
             22448)))

  (define heap_sum-3
    (make-scaled-data-value
     200 1000 'heap_sum-3
     '(29280 29280 29280 29271 29264 28304 29280 30256 29280 29280
             29280 29280 29280 29280 29280 28304 29280 29280 29280
             28304)))

  (define stack_fact-2
    (make-scaled-data-value
     2000 100 'stack_fact-2
     '(41968 40016 40016 40016 40016 40016 40016 40992 40016 40016
             40016 40592 40016 40016 40984 39999 40016 40016 40016
             40992)))

  (define heap_fact-2
    (make-scaled-data-value
     2000 100 'heap_fact-2
     '(41968 41968 40992 40992 40992 41968 41968 40992 41968 41968
             41968 41968 40992 41968 41568 41968 41968 40992 41968
             40992)))

  (define stack_sum-2
    (make-scaled-data-value
     2000 100 'stack_sum-2
     '(23424 23424 23424 23424 23424 23424 23424 23424 23424 23424
             23424 22448 23424 23424 23424 23424 23424 23424 23424
             23424)))

  (define heap_sum-2
    (make-scaled-data-value
     2000 100 'heap_sum-2
     '(24400 25376 24400 25376 24400 25952 24400 25376 25376 24400
             25376 24400 25376 24400 24400 25376 24400 26352 24400
             25376)))

  (define stack_fact-1
    (make-scaled-data-value
```

```
   20000 10 'stack_fact-1
   '(44896 43920 43920 44896 42944 43920 43920 43920 43920 42944
           43920 43920 44896 43920 42944 44496 43920 43920 42944
           43920)))

(define heap_fact-1
  (make-scaled-data-value
   20000 10 'heap_fact-1
   '(45872 45872 44896 45872 45872 45872 44896 44896 44896 45872
           44896 44896 46848 44896 44896 45872 45472 45872 44886
           45857)))

(define stack_sum-1
  (make-scaled-data-value
   20000 10 'stack_sum-1
   '(27328 27328 28304 27328 26352 28304 27328 27328 27328 27328
           27328 27328 26352 27328 27328 27328 27328 27328 27328
           27328)))

(define heap_sum-1
  (make-scaled-data-value
   20000 10 'heap_sum-1
   '(29280 28304 29280 28880 28304 29280 29280 28304 28304 29280
           28304 29280 28304 29280 28304 30256 29280 28304 29280
           28304)))

(define stack_fact-4
  (make-scaled-data-value
   20 10000 'stack_fact-4
   '(40592 40992 40016 40016 40992 40992 40016 40016 40016 40016
           40016 40016 40016 40016 40016 40016 41968 40016
           40016)))

(define heap_fact-4
  (make-scaled-data-value
   20 10000 'heap_fact-4
   '(43920 40992 42944 41968 41568 41968 41968 42944 41968 40992
           41968 41968 41968 41968 40992 41968 41968 40992 41968
           42944)))

(define stack_sum-4
  (make-scaled-data-value
   20 10000 'stack_sum-4
   '(29280 28304 28304 29280 28880 28304 29280 29280 28292 28292
           30256 28304 29280 28304 28304 28304 28304 29280 29280
           29280)))

(define heap_sum-4
  (make-scaled-data-value
   20 10000 'heap_sum-4
   '(32208 31232 32208 31232 34160 31232 32208 31232 32208 31232
           32208 31232 32208 31232 32208 31808 32208 31232 32208
           31232)))

(define stack_fact-5
  (make-data-value
   'stack_fact-5 100000
   '(34160 24976 24400 23424 23424 23424 23424 23424 24400 23424 24400
           23424 23424 23424 23424 24400 23424 23424 23424 23424)))

(define heap_fact-5
```

```
  (make-data-value
   'heap_fact-5 100000
   '(37088 31232 30256 31232 30256 30256 31232 31232 30256 31232 31232
          30256 30256 30256 31232 30256 32208 31808 31232 30256)))

(define stack_sum-5
  (make-data-value
   'stack_sum-5 100000
   '(368928 18544 18544 18544 18544 18544 18544 18544 18544 17568 18544
           18544 18544 18544 18544 17568 18544 18544 18544 18544 18544)))

(define heap_sum-5
  (make-data-value
   'heap_sum-5 100000
   '(27328 26352 26352 25376 26352 25376 26352 25376 26352 26352 25376
          26352 25376 26352 25376 25376 26352 26352 25376 26352)))

(define stack_fact-6
  (make-data-value
   'stack_fact-6 1000000
   '(381616 298656 296704 297280 297680 295703 299232 296704 295728
           298256 296704 296704 296704 298256 297680 295728 298256
           295728 297680 299207)))

(define heap_fact-6
  (make-data-value
   'heap_fact-6 1000000
   '(421632 373808 375360 372832 373808 374384 374784 383168 381616
           373784 375360 373808 372832 375360 373808 375360 372832
           373808 374384 373808)))

(define stack_sum-6
  (make-data-value
   'stack_sum-6 1000000
   '(252759 254336 250832 253760 251808 253360 251808 252784 250832
           254336 251808 253760 250832 253360 253760 252784 252759
           254336 251808 253760)))

(define heap_sum-6
  (make-data-value
   'heap_sum-6 1000000
   '(325984 330464 326960 326960 328512 328912 326960 329488 327936
           326936 328512 327936 325984 329488 327936 325984 328512
           327936 325984 329488)))

(define stack_fact-7
  (make-data-value
   'stack_fact-7 10000000
   '(4360119 3112215 3029256 3031207 3051703 3032208 3034136 3031207
            3033159 3032784 3030232 3035111 3034135 3033184 3033160
            3031207 3035111 3032208 3038040 3035111)))

(define heap_fact-7
  (make-data-value
   'heap_fact-7 10000000
   '(112783613 106876358 93831961 88238184 92866107 112269274 106182857
              100982851 102376106 99392322 98944738 97354651 99212133
              100388395 103351687 86775972 87163420 87232715 88112667
              94177990)))

(define stack_sum-7
```

```
  (make-data-value
   'stack_sum-7 10000000
   '(3317776 2637302 2618358 2595936 2595334 2593958 2590480 2595910
             2594934 2590480 2592983 2593382 2597888 2593958 2592432
             2595910 2593382 2593984 2594934 2594358)))

(define heap_sum-7
  (make-data-value
   'heap_sum-7 10000000
   '(113252058 91017981 91719302 97720252 102820729 100691347 98321037
               103970100 98914474 103821193 96754398 89168300 103249753
               101138909 86165365 89804191 97936901 123155339 115575163
               110995482)))

(define stack_fact-66
  (make-data-value
   'stack_fact-66 6666666
   '(2636352 2020472 2020496 2019496 2020496 2019496 2021072 2021447
             2022448 2028280 2020496 2020472 2018544 2025352 2021472
             2020472 2018544 2021448 2021472 2021448)))

(define heap_fact-66
  (make-data-value
   'heap_fact-66 6666666
   '(2862360 2535824 2533448 2540704 2534424 2533848 2536400 2533847
             2536376 2532496 2531896 2535424 2537752 2535400 2535824
             2537352 2538728 2533472 2537752 2535400)))

(define stack_sum-66
  (make-data-value
   'stack_sum-66 6666666
   '(1736480 1726144 1727672 1727696 1728647 1724192 1730600 1725744
             1726144 1730600 1727696 1726696 1728096 1726696 1727696
             1727696 1727096 1726720 1728648 1729072)))

(define heap_sum-66
  (make-data-value
   'heap_sum-66 6666666
   '(2232264 2230336 2229336 2230336 2230888 2236167 2228384 2235192
             2231888 2233240 2228384 2229336 2231888 2228360 2230312
             2230336 2231288 2228960 2236168 2227408)))

(define all-values
  (list stack_fact-1
        heap_fact-1
        stack_sum-1
        heap_sum-1
        stack_fact-2
        heap_fact-2
        stack_sum-2
        heap_sum-2
        stack_fact-3
        heap_fact-3
        stack_sum-3
        heap_sum-3
        stack_fact-4
        heap_fact-4
        stack_sum-4
        heap_sum-4
        stack_fact-5
        heap_fact-5
```

```
        stack_sum-5
        heap_sum-5
        stack_fact-6
        heap_fact-6
        stack_sum-6
        heap_sum-6
        stack_fact-66
        heap_fact-66
        stack_sum-66
        heap_sum-66
        stack_fact-7
        heap_fact-7
        stack_sum-7
        heap_sum-7))

(do-it 'alpha 'fact-and-sum all-values)

(define stack_fib-5
  (make-scaled-data-value
   20 10000 'stack_fib-5
   '(25376 18544 20496 18544 19520 18544 18544 19520 19520 18544
           19520 18544 19520 18544 18544 19520 18544 18544 19520
           18544)))

(define heap_fib-5
  (make-scaled-data-value
   20 10000 'heap_fib-5
   '(20496 20496 20496 20496 21472 21472 20496 20496 20496 20496
           20496 20496 20496 20496 20496 21072 20496 20496 20496
           20496)))

(define stack_fib-10
  (make-scaled-data-value
   20 5000 'stack_fib-10
   '(109312 108336 108336 108336 109312 108336 108912 109307 109291
            108336 108336 108336 108336 110288 108336 108912 108336
            108336 109312 108336)))

(define heap_fib-10
  (make-scaled-data-value
   20 5000 'heap_fib-10
   '(120048 120048 119072 118672 119072 119072 119072 119072 118096
            119072 119072 119648 118096 118096 120048 118096 119072
            118096 120048 118096)))

(define stack_fib-15
  (make-scaled-data-value
   20 1000 'stack_fib-15
   '(242624 242048 241072 242048 241648 242999 242048 241072 241648
            242048 241072 242048 241648 241072 241072 241072 242624
            241072 242048 240096)))

(define heap_fib-15
  (make-scaled-data-value
   20 1000 'heap_fib-15
   '(338246 337696 337696 336320 339648 336720 336320 338672 337696
            337296 336720 337696 337271 336720 337696 337296 337696
            336720 337296 336720)))

(define stack_fib-20
  (make-scaled-data-value
```

```
     20 100 'stack_fib-20
     '(269376 268400 268976 267399 268400 268400 268976 268400 267424
              268400 268000 268400 266448 268976 267424 268400 267424
              269952 268374 267424)))

  (define heap_fib-20
    (make-scaled-data-value
     20 100 'heap_fib-20
     '(430016 424560 425136 423584 422608 424160 425536 424160 425510
              424160 423584 423584 425136 424560 424160 424560 424560
              424135 423584 426112)))

  (define stack_fib-25
    (make-scaled-data-value
     20 10 'stack_fib-25
     '(299232 296704 297680 295728 299206 297680 295728 298256 297680
              302560 298256 295728 297680 296704 298256 297680 296704
              298256 296679 297680)))

  (define heap_fib-25
    (make-scaled-data-value
     20 10 'heap_fib-25
     '(619360 606672 605120 606672 604144 606646 606672 606096 606672
              606096 606672 605696 606070 605696 606096 607648 606672
              606096 606647 604144)))

  (define stack_fib-30
    (make-scaled-data-value
     20 1 'stack_fib-30
     '(331440 329888 328912 329488 328912 328912 330449 329878 327936
              331440 329888 327936 330464 329888 328912 330464 334768
              329888 329488 328886)))

  (define heap_fib-30
    (make-scaled-data-value
     20 1 'heap_fib-30
     '(833104 672064 671488 672064 672064 670486 673040 673040 669536
              672064 673040 672438 671088 673040 670512 672064 671088
              671463 671088 673040)))

  (define all-fibs
    (list stack_fib-5
          heap_fib-5
          stack_fib-10
          heap_fib-10
          stack_fib-15
          heap_fib-15
          stack_fib-20
          heap_fib-20
          stack_fib-25
          heap_fib-25
          stack_fib-30
          heap_fib-30))

  (do-it 'alpha 'fib all-fibs))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
HPPA (montreaux) a 720 (50MHz CPU) with 48MBytes memory
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (hppa)
```

```
(define stack_fact-10
  (make-scaled-data-value
   20 10000 'stack_fact-10
   '(49569 49409 49535 49399 49481 49561 49581 52589 49481 50152
           50166 49425 49457 49410 49519 49506 49474 49401 49510
           49410)))

(define heap_fact-10
  (make-scaled-data-value
   20 10000 'heap_fact-10
   '(59525 59544 59441 59494 59553 59599 62616 60768 59714 59826
           59503 59545 59634 59504 59442 59539 59499 59441 59548
           59435)))

(define stack_sum-10
  (make-scaled-data-value
   20 10000 'stack_sum-10
   '(31589 31641 31528 31638 31593 34555 31596 31532 32048 31821
           31899 31552 31593 31530 31529 31678 31538 31642 31529
           31606)))

(define heap_sum-10
  (make-scaled-data-value
   20 10000 'heap_sum-10
   '(41746 41680 41566 41571 41621 41572 41678 41600 41627 41565
           41670 41668 41634 44587 42082 41706 42278 41586 41632
           41564)))

(define stack_fact-100
  (make-scaled-data-value
   20 1000 'stack_fact-100
   '(42952 42913 43067 42921 43057 42957 42975 43075 42927 43130
           46154 43030 43529 43189 43397 42972 42929 42952 43027
           43031)))

(define heap_fact-100
  (make-scaled-data-value
   20 1000 'heap_fact-100
   '(53062 52938 53056 54627 53314 52928 53099 52993 53053 53066
           53003 56121 53614 53202 53330 53031 52943 53048 53164
           53007)))

(define stack_sum-100
  (make-scaled-data-value
   20 1000 'stack_sum-100
   '(24886 24843 24995 24851 24890 24846 24943 24845 24882 24852
           24953 24881 24847 24892 24908 24883 24948 24884 24946
           24950)))

(define heap_sum-100
  (make-scaled-data-value
   20 1000 'heap_sum-100
   '(37909 34956 36261 34900 35192 35482 34932 34944 34926 34897
           35044 34996 35003 34886 34925 34994 34932 34896 34985
           34889)))

(define stack_fact-1000
  (make-scaled-data-value
   20 100 'stack_fact-1000
   '(42964 45569 42329 42848 42904 42731 42248 42306 42257 42404
           42410 42316 42247 42252 42401 42246 42320 42246 42402
```

```
          42253)))

(define heap_fact-1000
  (make-scaled-data-value
   20 100 'heap_fact-1000
   '(52796 52552 52407 52355 55512 52813 52562 52692 52394 52291
          52417 52407 52406 52296 52406 52296 52387 52303 52409
          52290)))

(define stack_sum-1000
  (make-scaled-data-value
   20 100 'stack_sum-1000
   '(24839 24173 24331 24173 24340 24240 24175 27402 24273 24182
          24689 24351 24303 24806 24194 24220 24238 24209 24175
          24186)))

(define heap_sum-1000
  (make-scaled-data-value
   20 100 'heap_sum-1000
   '(34942 34421 34252 34293 34261 34387 34386 34213 34216 34312
          34222 34378 34215 34262 34271 34216 34364 34330 34317
          37271)))

(define stack_fact-10000
  (make-scaled-data-value
   20 10 'stack_fact-10000
   '(45617 42460 42482 42331 42311 42439 42380 42265 42373 42262
          42372 42274 42256 42368 42465 42362 42346 45778 42266
          44122)))

(define heap_fact-10000
  (make-scaled-data-value
   20 10 'heap_fact-10000
   '(57767 52413 52354 52471 52619 52417 52352 52469 52398 52409
          52348 52466 52397 52421 52541 52458 55906 52433 53326
          53016)))

(define stack_sum-10000
  (make-scaled-data-value
   20 10 'stack_sum-10000
   '(27428 24241 24203 24148 24215 24284 24150 24411 24146 24280
          24150 24199 24258 24230 24328 24149 24208 24149 24259
          24159)))

(define heap_sum-10000
  (make-scaled-data-value
   20 10 'heap_sum-10000
   '(38907 34418 34225 34228 34348 34425 34287 34330 34361 37823
          34361 34856 34581 34237 34722 34284 34281 34231 34341
          34290)))

(define stack_fact-100000
  (make-scaled-data-value
   20 1 'stack_fact-100000
   '(84380 60212 60037 60254 60047 60375 60278 62585 61941 61101
          60595 60129 60028 60230 60334 60123 60214 60069 60147
          60222)))

(define heap_fact-100000
  (make-scaled-data-value
   20 1 'heap_fact-100000
```

```
   '(114320 81605 85013 82143 82333 81331 81393 81477 81204 81349
            81245 81410 81299 81466 81456 86300 82160 81432 81360
            81395)))

(define stack_sum-100000
  (make-scaled-data-value
   20 1 'stack_sum-100000
   '(64734 41225 41043 41023 41195 41023 41203 41052 41010 41121
           41244 41244 41219 44590 41058 42125 41438 41570 41126
           41043)))

(define heap_sum-100000
  (make-scaled-data-value
   20 1 'heap_sum-100000
   '(96698 63261 63134 63225 63260 63124 63240 63245 63296 63292
           66739 64351 63581 63824 63264 63339 63299 63291 63294
           63121)))

(define stack_fact-1000000
  (make-scaled-data-value
   20 1 'stack_fact-1000000
   '(884472 682896 678273 684765 682821 681800 679688 683254 681371
            679456 684403 683033 679008 682984 682438 678696 683195
            683675 678067 682971)))

(define heap_fact-1000000
  (make-scaled-data-value
   20 1 'heap_fact-1000000
   '(1175697 880327 883871 881463 880607 880493 879693 876615 880127
             881461 880885 880310 880393 880855 881035 879436 877151
             880613 880532 880437)))

(define stack_sum-1000000
  (make-scaled-data-value
   20 1 'stack_sum-1000000
   '(678238 474478 478021 473358 478229 473407 478162 473188 477570
            473658 478118 474061 476697 475307 473017 478130 473232
            478303 473194 478488)))

(define heap_sum-1000000
  (make-scaled-data-value
   20 1 'heap_sum-1000000
   '(994281 699718 694885 700204 699559 696462 699757 700716 698372
            697086 699815 699204 695100 699852 700438 695025 700193
            699791 698312 696146)))

(define all-values
  (list stack_fact-10
        heap_fact-10
        stack_sum-10
        heap_sum-10
        stack_fact-100
        heap_fact-100
        stack_sum-100
        heap_sum-100
        stack_fact-1000
        heap_fact-1000
        stack_sum-1000
        heap_sum-10000
        stack_fact-10000
        heap_fact-10000
```

```
         stack_sum-10000
         heap_sum-10000
         stack_fact-100000
         heap_fact-100000
         stack_sum-100000
         heap_sum-100000
         stack_fact-1000000
         heap_fact-1000000
         stack_sum-1000000
         heap_sum-1000000))

  (do-it 'hppa 'fact-and-sum all-values)

  (define all-fibs
    (list stack_fib-5
         heap_fib-5
         stack_fib-10
         heap_fib-10
         stack_fib-15
         heap_fib-15
         stack_fib-20
         heap_fib-20
         stack_fib-25
         heap_fib-25
         stack_fib-30
         heap_fib-30))

  (do-it 'hppa 'fib all-fibs))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
68040 (Dumbo) HP9000/380 25MHz 68040 with 32MBytes of memory
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (mc68040)
  (define stack_fact-10
    (make-scaled-data-value
     20 10000 'stack_fact-10
     '(156284 156232 156663 156268 158444 156952 156208 155896 156635
              156176 158124 156372 156376 156224 156015 156680 157752
              156484 156708 156112)))

  (define heap_fact-10
    (make-scaled-data-value
     20 10000 'heap_fact-10
     '(164368 164476 164244 167841 164652 164256 164349 164408 164248
              166296 164749 164272 164228 164564 164308 166417 164628
              164276 163964 164769)))

  (define stack_sum-10
    (make-scaled-data-value
     20 10000 'stack_sum-10
     '(75195 75048 76748 75320 75216 75067 75416 75044 75204 74931
             74888 75228 75456 75087 74956 76764 75356 75216 74827
             75688)))

  (define heap_sum-10
    (make-scaled-data-value
     20 10000 'heap_sum-10
     '(87241 87320 87072 87417 87116 87428 87377 90208 87557 87240
             87740 87269 87244 87028 87177 87304 87817 87308 87084
             89313)))
```

```
(define stack_fact-100
  (make-scaled-data-value
   20 1000 'stack_fact-100
   '(143987 144347 145532 149919 148220 143891 143611 143835 144219
            146120 143807 144295 143871 143676 144279 143855 145771
            143811 144400 143795)))

(define heap_fact-100
  (make-scaled-data-value
   20 1000 'heap_fact-100
   '(152204 152275 152088 155288 152139 152236 151971 151968 152528
            153419 152356 152504 151927 151740 152363 152012 154000
            151819 152568 152080)))

(define stack_sum-100
  (make-scaled-data-value
   20 1000 'stack_sum-100
   '(62187 62447 62111 62711 62431 62175 62223 64444 62187 62431
           62115 62923 62259 62147 62564 62231 62147 62415 62147
           62815)))

(define heap_sum-100
  (make-scaled-data-value
   20 1000 'heap_sum-100
   '(74752 74392 76023 74824 74592 74624 74715 74464 74628 74400
           74259 74656 74812 74664 74251 74432 77944 74584 74271
           75124)))

(define stack_fact-1000
  (make-scaled-data-value
   20 100 'stack_fact-1000
   '(160220 157396 158063 157516 157128 158224 157164 159412 157980
            157472 157648 157143 158220 158740 157092 157708 156864
            157504 157544 156787)))

(define heap_fact-1000
  (make-scaled-data-value
   20 100 'heap_fact-1000
   '(182045 181089 180425 180341 181517 183709 180993 180601 180061
            181037 180453 183094 180793 180497 180345 181121 182457
            181145 180597 180017)))

(define stack_sum-1000
  (make-scaled-data-value
   20 100 'stack_sum-1000
   '(74040 72743 73276 72396 72211 74624 72692 72127 73192 72952
           72143 72600 72311 72560 72348 72695 73248 72372 73847
           72796)))

(define heap_sum-1000
  (make-scaled-data-value
   20 100 'heap_sum-1000
   '(108257 107638 106653 107002 106537 106749 107546 107773 110382
            107017 107405 106950 106801 106593 107322 107541 106789
            108958 106933 107222)))

(define stack_fact-10000
  (make-scaled-data-value
   20 10 'stack_fact-10000
   '(174272 167557 166312 165992 165541 166104 165744 167400 166581
```

```
            165672 165372 166245 165692 167492 166509 165648 165268
            166345 165696 169240)))

(define heap_fact-10000
  (make-scaled-data-value
   20 10 'heap_fact-10000
   '(202750 189641 189786 190237 191622 190329 189770 189685 190242
            191209 190062 190161 189810 190057 189790 191753 190286
            189573 189818 190177)))

(define stack_sum-10000
  (make-scaled-data-value
   20 10 'stack_sum-10000
   '(90264 80356 80252 80800 80116 80044 80288 80028 80304 80156
           80396 80300 80068 83704 80280 80316 80436 79988 80332
           80044)))

(define heap_sum-10000
  (make-scaled-data-value
   20 10 'heap_sum-10000
   '(124462 112414 111989 113794 112037 112274 112446 112033 111922
            112158 112221 112158 113653 112442 121238 112322 112101
            111950 112110 112433)))

(define stack_fact-100000
  (make-scaled-data-value
   20 1 'stack_fact-100000
   '(245737 167148 167108 167801 169048 167509 167348 167168 167329
            167520 170569 167328 167312 167321 167156 167665 169280
            167328 167437 167172)))

(define heap_fact-100000
  (make-scaled-data-value
   20 1 'heap_fact-100000
   '(309095 193706 192418 191617 191790 192053 193618 192406 191613
            191802 191757 193610 192034 192093 191826 192306 191605
            195222 192266 191613)))

(define stack_sum-100000
  (make-scaled-data-value
   20 1 'stack_sum-100000
   '(158860 81980 81684 81412 83540 81660 81857 81828 81484 81688
            81408 81672 81396 81888 81684 81516 83552 81797 81376
            82092)))

(define heap_sum-100000
  (make-scaled-data-value
   20 1 'heap_sum-100000
   '(230884 113762 113781 113762 115298 113653 113766 114546 113750
            113373 113822 113978 113629 115358 114042 113597 113982
            113790 113502 113577)))

(define stack_fact-1000000
  (make-scaled-data-value
   20 1 'stack_fact-1000000
   '(2458527 1677699 1677848 1678524 1679520 1678032 1678116 1678164
             1679016 1689992 1924317 1927164 1679316 1676996 1679616
             1678420 1676320 1680208 1677939 1677076)))

(define heap_fact-1000000
  (make-scaled-data-value
```

```
   20 1 'heap_fact-1000000
   '(3106003 1922328 1925732 1924765 1926494 1932211 1924156 1925548
            1924244 1925491 1924508 1924120 1925428 1923820 1925391
            1922480 1924712 1926304 1924523 1924240)))

(define stack_sum-1000000
  (make-scaled-data-value
   20 1 'stack_sum-1000000
   '(1595600 822307 822923 817247 818823 820847 818963 818655 816868
            819223 818723 820583 818639 819243 817911 818683 819103
            820771 818631 817695)))

(define heap_sum-1000000
  (make-scaled-data-value
   20 1 'heap_sum-1000000
   '(2315616 1141371 1140223 1142171 1140271 1143399 1140483 1140207
            1140143 1140223 1141635 1140123 1149818 1140287 1141607
            1140035 1140587 1140243 1140187 1143387)))

(define all-values
  (list stack_fact-10
        heap_fact-10
        stack_sum-10
        heap_sum-10
        stack_fact-100
        heap_fact-100
        stack_sum-100
        heap_sum-100
        stack_fact-1000
        heap_fact-1000
        stack_sum-1000
        heap_sum-1000
        stack_fact-10000
        heap_fact-10000
        stack_sum-10000
        heap_sum-10000
        stack_fact-100000
        heap_fact-100000
        stack_sum-100000
        heap_sum-100000
        stack_fact-1000000
        heap_fact-1000000
        stack_sum-1000000
        heap_sum-1000000))

(do-it '68k 'fact-and-sum all-values)

(define stack_fib-5
  (make-scaled-data-value
   20 100000 'stack_fib-5
   '(1024737 1027557 1024577 1025036 1024721 1024853 1025837 1024916
            1025393 1024529 1024501 1026696 1024709 1024485 1024517
            1025068 1026005 1024485 1024493 1025128)))

(define heap_fib-5
  (make-scaled-data-value
   20 100000 'heap_fib-5
   '(1053088 1055527 1053355 1056787 1057971 1053260 1054507 1053175
            1052848 1056211 1054711 1053131 1053068 1052947 1054119
            1054352 1052979 1053211 1052828 1054387)))
```

```
(define stack_fib-10
  (make-scaled-data-value
   20 10000 'stack_fib-10
   '(1123336 1123272 1122891 1123076 1124212 1123100 1122884 1124288
             1125495 1123676 1126212 1122804 1128359 1125700 1123308
             1124292 1123407 1124580 1122632 1123052)))

(define heap_fib-10
  (make-scaled-data-value
   20 10000 'heap_fib-10
   '(1159258 1159418 1160354 1161882 1159342 1158914 1159938 1159490
             1158946 1164642 1162961 1165394 1404730 1410237 1162310
             1159214 1163898 1160314 1166174 1159070)))

(define stack_fib-15
  (make-scaled-data-value
   20 1000 'stack_fib-15
   '(1244638 1243549 1245274 1245386 1243906 1243634 1247297 1245846
             1245278 1244010 1245097 1244978 1244026 1244118 1244805
             1244170 1245362 1243910 1244694 1244113)))

(define heap_fib-15
  (make-scaled-data-value
   20 1000 'heap_fib-15
   '(1508953 1508708 1505045 1503305 1503849 1502637 1503712 1503517
             1503465 1503605 1502053 1504381 1501921 1503060 1514057
             1505832 1501453 1503893 1502845 1503297)))

(define stack_fib-20
  (make-scaled-data-value
   20 100 'stack_fib-20
   '(1380735 1390298 1378423 1380863 1378688 1380774 1380583 1378856
             1379251 1382518 1378320 1380923 1380486 1378816 1380651
             1379243 1380819 1381298 1378244 1383178)))

(define heap_fib-20
  (make-scaled-data-value
   20 100 'heap_fib-20
   '(1699867 1691544 1694135 1686116 1687459 1692980 1685187 1687768
             1692635 1687656 1685596 1692975 1687684 1685724 1690051
             1687232 1686343 1689156 1691019 1685908)))

(define stack_fib-25
  (make-scaled-data-value
   20 10 'stack_fib-25
   '(1559962 1532467 1528684 1532547 1530984 1529327 1532843 1529436
             1530931 1529136 1532583 1529376 1531687 1530396 1531691
             1528908 1532335 1529144 1534743 1535791)))

(define heap_fib-25
  (make-scaled-data-value
   20 10 'heap_fib-25
   '(2030934 1873911 1876946 1876814 1876702 1878202 1875646 1877951
             1875798 1874022 1877490 1875558 1875731 1877686 1879546
             1880130 1875902 1875198 1877294 2124722)))

(define stack_fib-30
  (make-scaled-data-value
   20 1 'stack_fib-30
   '(1945566 1696688 1695427 1698487 1697119 1797206 1762760 1697631
             1696743 1697959 1700435 1697207 1701555 1696688 1697671
```

```
              1697943 1695255 1699391 1698571 1696207)))

  (define heap_fib-30
    (make-scaled-data-value
     20 1 'heap_fib-30
     '(4185419 2134617 2112158 2096864 2095913 2090442 2088152 2092970
              2087008 2090906 2087104 2091826 2163824 2089356 2087926
              2086840 2087840 2087394 2089720 2088790)))

  (define all-fibs
    (list stack_fib-5
          heap_fib-5
          stack_fib-10
          heap_fib-10
          stack_fib-15
          heap_fib-15
          stack_fib-20
          heap_fib-20
          stack_fib-25
          heap_fib-25
          stack_fib-30
          heap_fib-30))

    (do-it '68k 'fib all-fibs))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
i486 (Patek) i486/DX2 66 MHz with 32MBytes of memory
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (i486)
  (define stack_fact-10
    (make-scaled-data-value
     20 10000 'stack_fact-10
     '(56510 55447 55436 55707 55794 55420 55566 55687 55687 55603
            55665 55740 55528 55514 55690 55737 55438 55614 56473
            55918)))

  (define heap_fact-10
    (make-scaled-data-value
     20 10000 'heap_fact-10
     '(65548 67039 65906 65216 67037 65924 65251 67113 65965 65118
            67030 65958 65060 66832 65834 65712 67220 66572 65713
            67212)))

  (define stack_sum-10
    (make-scaled-data-value
     20 10000 'stack_sum-10
     '(36672 37795 37870 37849 37794 37832 37868 37800 37842 37796
            37798 37908 37784 37795 37865 37891 37905 37781 37794
            37857)))

  (define heap_sum-10
    (make-scaled-data-value
     20 10000 'heap_sum-10
     '(48791 48495 48148 49189 50376 49467 48222 49057 50376 49589
            48146 48941 50376 49717 48146 48816 50302 49320 49217
            49562)))

  (define stack_fact-100
    (make-scaled-data-value
     20 1000 'stack_fact-100
```

```
   '(52391 51719 51817 51694 51749 51710 52189 51693 51753 51958
           51807 51694 51749 51693 52514 51711 51748 51692 51789
           51767)))

(define heap_fact-100
  (make-scaled-data-value
   20 1000 'heap_fact-100
   '(62626 62273 62156 62233 62260 62157 62222 62351 62274 62216
           64087 62292 62215 62158 62258 62231 62240 62260 62156
           62218)))

(define stack_sum-100
  (make-scaled-data-value
   20 1000 'stack_sum-100
   '(29693 29446 29302 29318 30017 29403 29318 29507 29413 29319
           29315 29316 29372 29317 29315 29411 29317 29316 29370
           29303)))

(define heap_sum-100
  (make-scaled-data-value
   20 1000 'heap_sum-100
   '(41840 41542 41478 41409 41487 41406 41490 41375 41434 41376
           41374 41526 41404 41441 41549 41425 41475 41405 41467
           41375)))

(define stack_fact-1000
  (make-scaled-data-value
   20 100 'stack_fact-1000
   '(58325 57221 56859 56999 56923 57156 56855 57414 57123 56848
           57055 56905 57109 56862 57029 57147 56987 57020 56918
           57085)))

(define heap_fact-1000
  (make-scaled-data-value
   20 100 'heap_fact-1000
   '(78285 76830 76846 77203 76883 76850 76736 76897 76782 76952
           76860 76773 76961 76860 76827 76871 77156 76866 76842
           76731)))

(define stack_sum-1000
  (make-scaled-data-value
   20 100 'stack_sum-1000
   '(31258 29720 29690 29824 29671 29683 29795 29876 29879 29726
           29840 29700 29693 29935 29751 29699 29705 29829 29707
           29682)))

(define heap_sum-1000
  (make-scaled-data-value
   20 100 'heap_sum-1000
   '(54940 53216 53332 53697 53229 53334 53276 53396 53226 53391
           53228 53399 53369 53343 53219 53462 53233 53338 53461
           53224)))

(define stack_fact-10000
  (make-scaled-data-value
   20 10 'stack_fact-10000
   '(81504 72305 72431 72199 72207 72338 72215 72010 72303 72251
           72910 72014 72182 72368 72028 72215 72298 72405 72034
           72324)))

(define heap_fact-10000
```

```
  (make-scaled-data-value
   20 10 'heap_fact-10000
   '(105531 92729 92545 93131 92553 92691 92607 92663 92734 92520
            92599 92555 92713 92959 92722 92557 92696 92609 92663
            96741)))

(define stack_sum-10000
  (make-scaled-data-value
   20 10 'stack_sum-10000
   '(62329 44575 44428 42185 46505 40249 39725 39720 39907 40182
           40040 39727 39723 39912 39720 40038 39724 39725 39909
           39725)))

(define heap_sum-10000
  (make-scaled-data-value
   20 10 'heap_sum-10000
   '(82083 68851 68793 68825 68855 69051 68640 68880 69550 68646
           68767 69030 68626 69269 69024 68890 68876 69043 68776
           68671)))

(define stack_fact-100000
  (make-scaled-data-value
   20 1 'stack_fact-100000
   '(166633 119617 119842 119656 119575 119440 119552 120214 119463
            119598 119437 119781 119771 119461 119584 119900 119547
            119756 119501 119829)))

(define heap_fact-100000
  (make-scaled-data-value
   20 1 'heap_fact-100000
   '(224122 156409 156944 156121 156400 156412 156453 158840 156434
            157912 156481 156415 156290 157183 156238 156957 156289
            156313 156615 156247)))

(define stack_sum-100000
  (make-scaled-data-value
   20 1 'stack_sum-100000
   '(130537 82599 83403 82890 82800 82955 82785 82626 83137 82798
            82952 82825 82937 82627 83290 82917 82807 82920 82826
            82629)))

(define heap_sum-100000
  (make-scaled-data-value
   20 1 'heap_sum-100000
   '(194803 126288 126136 126367 126018 126214 126036 126280 126094
            126155 126136 126404 126236 126034 126013 126257 126273
            125991 126144 126428)))

(define stack_fact-1000000
  (make-scaled-data-value
   20 1 'stack_fact-1000000
   '(1728015 1274891 1275088 1274826 1275315 1278018 1275163 1274913
             1275243 1275026 1275035 1275315 1275244 1301347 1275042
             1275296 1274860 1275067 1277540 1275370)))

(define heap_fact-1000000
  (make-scaled-data-value
   20 1 'heap_fact-1000000
   '(2305451 1629725 1629062 1629459 1629548 1629699 1632918 1630708
             1629356 1629262 1629042 1629732 1633909 1698553 1629944
             1629529 1634125 1629238 1629886 1629139)))
```

```
(define stack_sum-1000000
  (make-scaled-data-value
   20 1 'stack_sum-1000000
   '(1310212 856439 857043 856919 856684 856936 856911 857154 858389
             857707 856806 857074 856601 856935 857218 856408 856906
             856990 856683 856875)))

(define heap_sum-1000000
  (make-scaled-data-value
   20 1 'heap_sum-1000000
   '(1974375 1286119 1285269 1286179 1288749 1285647 1286244 1285736
             1285937 1285473 1286104 1285905 1285663 1286286 1289451
             1286958 1285348 1286303 1285918 1285525)))

(define all-values
  (list stack_fact-10
        heap_fact-10
        stack_sum-10
        heap_sum-10
        stack_fact-100
        heap_fact-100
        stack_sum-100
        heap_sum-100
        stack_fact-1000
        heap_fact-1000
        stack_sum-1000
        heap_sum-1000
        stack_fact-10000
        heap_fact-10000
        stack_sum-10000
        heap_sum-10000
        stack_fact-100000
        heap_fact-100000
        stack_sum-100000
        heap_sum-100000
        stack_fact-1000000
        heap_fact-1000000
        stack_sum-1000000
        heap_sum-1000000))

(do-it 'i486 'fact-and-sum all-values)

(define stack_fib-5
  (make-scaled-data-value
   20 1 'stack_fib-5
   '(308 33 25 23 24 24 23 24 23 24 24 23 23 24 23 24 24 23 25 24)))

(define heap_fib-5
  (make-scaled-data-value
   20 1 'heap_fib-5
   '(334 26 25 24 25 24 24 25 24 25 24 24 24 24 24 24 57 25 24 25)))

(define stack_fib-10
  (make-scaled-data-value
   20 1 'stack_fib-10
   '(352 74 65 65 65 65 65 65 66 65 66 65 66 65 67 65 66 65 66 65)))

(define heap_fib-10
  (make-scaled-data-value
   20 1 'heap_fib-10
```

```
   '(458 76 75 74 74 75 75 75 75 74 74 74 75 75 75 74 74 74 74 75)))

(define stack_fib-15
  (make-scaled-data-value
   20 1 'stack_fib-15
   '(821 537 537 526 526 526 560 527 526 526 526 526 527 527 527 527
         527 526 527 526)))

(define heap_fib-15
  (make-scaled-data-value
   20 1 'heap_fib-15
   '(2591 899 875 875 876 875 875 875 875 916 877 875 875 875 875
          875 874 875 876 875)))

(define stack_fib-20
  (make-scaled-data-value
   20 1 'stack_fib-20
   '(5923 5677 5645 5636 5659 5636 5652 5636 5652 5636 5652 5653
          5636 5652 5636 5652 5636 5737 5655 5636)))

(define heap_fib-20
  (make-scaled-data-value
   20 1 'heap_fib-20
   '(25840 13441 13412 13524 13399 13630 13372 13376 13415 13374
           13437 13353 13551 13440 14789 13430 13418 13372 13515
           13604)))

(define stack_fib-25
  (make-scaled-data-value
   20 1 'stack_fib-25
   '(62877 62561 62412 62752 62503 62391 62446 62423 62468 62449
           62389 62538 62470 62392 62481 62392 62462 62484 62391
           62582)))

(define heap_fib-25
  (make-scaled-data-value
   20 1 'heap_fib-25
   '(285983 186278 186455 186051 186587 186157 186276 186460 186214
            186273 186531 186224 186453 186216 186227 186679 186243
            187504 186439 186080)))

(define stack_fib-30
  (make-scaled-data-value
   20 1 'stack_fib-30
   '(693010 702383 702233 702075 702256 701592 692409 694180 693032
            692424 692229 692413 692321 692269 692361 692339 692181
            692384 692363 692357)))

(define heap_fib-30
  (make-scaled-data-value
   20 1 'heap_fib-30
   '(3265745 2065392 2064986 2069689 2065605 2064913 2066052 2065216
             2064950 2067606 2066191 2065429 2065168 2065149 2079006
             2065178 2065183 2068184 2065347 2065230)))

(define all-fibs
  (list stack_fib-5
        heap_fib-5
        stack_fib-10
        heap_fib-10
        stack_fib-15
```

```
      heap_fib-15
      stack_fib-20
      heap_fib-20
      stack_fib-25
      heap_fib-25
      stack_fib-30
      heap_fib-30))

(do-it 'i486 'fib all-fibs))
```