Vmalloc: A General and Efficient Memory Allocator

KIEM-PHONG VO

AT&T Laboratories 180 Park Avenue, Florham Park, NJ 07932, U.S.A.
(kpv@research.att.com)

SUMMARY

On C/Unix systems, the *malloc* interface is standard for dynamic memory allocation. Despite its popularity, *malloc*'s shortcomings frequently cause programmers to code around it. The new library *Vmalloc* generalizes *malloc* to give programmers more control over memory allocation. *Vmalloc* introduces the idea of organizing memory into separate *regions*, each with a *discipline* to get raw memory and a *method* to manage allocation. Applications can write their own disciplines to manipulate arbitrary type of memory or just to better organize memory in a region by creating new regions out of its memory. The provided set of allocation methods include general purpose allocation, fast special cases and aids for memory debugging or profiling. A compatible *malloc* interface enables current applications to select allocation methods using environment variables so they can tune for performance or perform other tasks such as profiling memory usage, generating traces of allocation calls or debugging memory errors. A performance study comparing *Vmalloc* and currently popular *malloc* implementations shows that *Vmalloc* is competitive to the best of these allocators. Applications can gain further performance improvement by using the right mixture of regions with different *Vmalloc* methods.

KEYWORDS: Memory allocation, debugging, profiling, region, method, discipline

1 Introduction

Dynamic memory allocation is an integral part of programming. Programs in C and C++ (via constructors and destructors) routinely allocate memory using the familiar ANSI-C standard interface *malloc* established around 1979 by Doug McIlroy. *Malloc* manipulates *heap memory* using the functions `malloc(s)` to allocate a block of size `s`, `free(b)` to free a previously allocated block `b`, and `realloc(b,s)` to resize a block `b` to size `s`. No optimal solution to dynamic memory allocation exists [1, 2, 3] so, over the years, many *malloc* implementations were proposed with different trade-offs in time and space efficiency. A study by David Korn and Phong Vo [4] in 1985 presented and compared 11 *malloc* versions. Only a few of these survived the test of time. The first widely used *malloc* was written by McIlroy and became part of many Bell Labs Research and System V versions of the UNIX system. This *malloc* is based on a first-fit strategy and can be significantly slow in large memories. C. Kingsley wrote a *malloc* based on a power-of-two buddy system around 1980. This version is distributed with versions of the BSD UNIX system. It is fast but wastes significant space. In 1983, C.J. Stephenson [5] proposed a better-fit strategy based on the Cartesian tree data structure [6] . This strategy was implemented by C. Aoki and C. Adams and is now part of SUN OS. Also around 1983, Vo implemented a *malloc* based on the best-fit strategy and bottom-up splay tree data structure [7] . This version is now distributed with UNIX System V Release 4.

The *malloc* interface is simple and elegant. However, its simplicity imposes certain constraints on implementation and usability. For example, since only the address of usable data is returned in an allocation request, the size of the block must be kept elsewhere in the event that the block is freed or resized. Such overhead can be significant for programs that allocate small blocks or seldom free. Many programs allocate a few different block types where each type has a fixed size. Here, *malloc* may be inefficient in space because of the mentioned overhead and also inefficient in time because of the implied search for free space. To improve performance, many applications invent special interfaces that use *malloc* to allocate large chunks then manage that space themselves. The CustoMalloc package [8] improves on this approach by analyzing allocation traces from program execution to synthesize special *malloc* interfaces for popular sizes. This approach still has to pay the space overhead and may be misled by the trace data. Another type of allocation problem is to manage memory other than heap memory. For example, on modern environments, it is increasingly desirable to use shared memory to speed up process communication and mapped memory for faster IO and data persistence. *Malloc* is simply not designed for such purposes. This is unfortunate as the significant time and effort put into developing good algorithms and heuristics for *malloc* must be reinvented or at least repackaged in each new situation.

The *Vmalloc* library generalizes the *malloc* interface to give programmers more control over memory allocation strategy. *Vmalloc* introduces the idea of allocating memory in separate *region*s each of which has a *discipline* to obtain raw memory and a *method* to manage that memory. Aside from memory acquisition, a region discipline may also include a function to handle exceptional allocation events. The library provides disciplines for standard ways of obtaining memory. Application can defines their own disciplines to manipulate any type of memory including shared or mapped memory. The method of a region specifies how memory resource is managed. The predefined set of methods includes general purpose allocation via a best-fit allocator, fast allocation of important special cases, and allocation with aids for memory debugging and profiling. Thus, using *Vmalloc*, an application can select the appropriate mixture of regions that most effectively deals with its memory needs. For current applications, *Vmalloc* also provides a compatible *malloc* interface that allows method selection by environment variables.

The rest of the paper is organized as follows. Section 2 describes the *Vmalloc* library. Section 3 presents a performance study that both compares the general allocation method of *Vmalloc* against currently popular *malloc* versions and shows the efficiency of the specialized *Vmalloc* methods. Section 4 gives examples of how to use the compatible *malloc* interface. Section 5 summarizes the results.

```
Vmalloc_t* vm = vmopen(Vmdcheap,Vmlast,0);
for(a while)
{ ...
  b = vmalloc(vm,block_size);
  b = vmresize(vm,b,new_size,1);
  vmfree(vm,b);
  ...
  vmclear(vm);
  vmcompact(vm);
}
vmclose(vm);
```

Figure 1: An example of region creation and space allocation

2 The *Vmalloc* library

Vmalloc enables applications to allocate arbitrary types of memory, and to pick allocation strategies that matches allocation requirements. This is done via three basic structures:

- *Region*: Each allocation request is handled in some region. The library provides a standard heap region `Vmheap`. This region performs general purpose memory allocation on *heap memory* which, on UNIX systems, is obtained by the `sbrk()` system call. Applications can create other regions as necessary. Region operations are discussed in Section 2.1.
- *Discipline*: Each region has a discipline to obtain raw memory and to handle exceptional events. The library provides two standard disciplines: `Vmdcsbrk` to obtain memory with the system call `sbrk()`, and `Vmdcheap` to obtain memory from the heap region. Applications can define new disciplines for other types of memory. Discipline usage is discussed in Section 2.2.
- *Method*: Each region selects a particular method of memory management. The available methods are: `Vmbest` for general purpose allocation based on an approximate best-fit strategy, `Vmpool` to allocate blocks of fixed sizes, `Vmlast` for allocation where only the last block can be freed or resized, `Vmdebug` for allocation with aids for memory error detection, and `Vmprofile` for allocation with aids for memory profiling. Methods are discussed in Section 2.3.

2.1 Region operations

Figure 1 shows an example code fragment. This code fragment runs a loop for a while. Each iteration of the loop constructs and manipulates some data structures. The structures are deleted at the end of an iteration. This is the typical mode of operation in an application such as a language interpreter. The main loop reads one or more language statements, constructs and interprets a structure representing these statements, then cleans up before continuing.

The call `vmopen(disc,meth,flags)` creates a region with discipline `disc`, method `meth`, and some control bits in `flags`. Thus, line 1 of Figure 1 creates a region `vm` with the discipline `Vmdcheap` (see Section 2.2) and the allocation method `Vmlast`. The discipline `Vmdcheap` obtains heap memory from the standardly provided region `Vmheap`. The `Vmlast` method allocates space but does not allow a block to be freed or resized unless it is the very last one allocated. Based on this constraint, `Vmlast` can avoid most space overhead and search time. The constraint is fine here since memory is only freed at the bottom of the loop (via `vmclear()`). The `flags` argument of `vmopen()` is composed from a few bits. The ones that apply here are `VM_TRACE` to generate a trace of allocation calls (see Section 4 for sample trace outputs), and `VM_TRUST` to turn off region locking and validation checks during allocation calls for faster speed. Since `flags` is `0` here, standard region handling will be used and `vm` will be locked during allocation.

Lines 4, 5, and 6 of Figure 1 show that allocations are performed with the calls:

- `vmalloc(vm,size)`: allocates from region `vm` a block of `size` bytes. The block is suitably aligned to store any C or C++ type.
- `vmfree(vm,b)`: makes the previously allocated space block `b` available for future allocations. Note that if `VM_TRUST` is off, both `vmfree()` and `vmresize()` will check to see if `b` was in fact allocated from `vm`.
- `vmresize(vm,b,size,type)`: resizes the block `b` to `size` bytes. `type` is composed from the bits `VM_RSMOVE`, `VM_RSCOPY`, and `VM_RSCLEAR`. `VM_RSMOVE` means that `b` cannot be resized in place, a new block of size `size` will be allocated. `VM_RSCOPY` is like `VM_RSMOVE` but it also copies data from the old block to the new block. `VM_RSCLEAR` means that any new space beyond the old size will be zero-filled.

The call `vmclear(vm)` on line 8 of Figure 1 clears the region `vm`, i.e., to reclaim all allocated space. `vmclear()` is useful to globally free all currently busy blocks in a region. It is particular helpful with the `Vmlast` method which only allows freeing of the last block. The call `vmcompact(vm)` on line 9 reduces unused space in the region via the discipline. In this example, this means to return such space to the `Vmheap` region. This is an important consideration if some iterations may consume large amounts of space. Finally, after the loop terminates, `vmclose(vm)` is called on line 11 to close the region `vm` and release all of its memory.

Other functions are available for obtaining information about regions and allocated blocks. For example, `vmstat(vm,statb)` returns in the buffer `statb` summary statistics on busy and free space, `vmaddr(vm,addr)` checks to see if `addr` is a part of some allocated block, and `vmsize(vm,b)` returns the true size of `b`.

2.2 Writing and using disciplines

The example in Figure 1 uses the system-provided discipline `Vmdcheap` to obtain memory from the heap region `Vmheap`. Applications can provide their own disciplines for special memory organization.

```
1. void* heapmem(Vmalloc_t* vm, void* addr,
2.               size_t csz, size_t nsz, Vmdisc_t* disc)
3. {   if(csz == 0)
4.          return vmalloc(Vmheap,nsz);
5.      else if(nsz == 0)
6.          return vmfree(Vmheap,addr) >= 0 ? addr : (void*)0;
7.      else return vmresize(Vmheap,addr,nsz,0);
8. }
```

Figure 2: A memory-obtaining discipline function

A discipline defines functions to get memory and to handle exceptional events. It is of type `Vmdisc_t` which has members:

```
Vmemory_f   memoryf;
Vmexcept_f  exceptf;
size_t      round;
```

The call `(*exceptf)(vm,type,obj,disc)` announces events. `vm` is the region originating the event and `disc` is the discipline. `type` and `obj` define the type of the event and the object that causes it. Events supported are: `VM_NOMEM` to indicate that region is out of memory, `VM_BADADDR` to indicate that a call to `vmfree()` or `vmresize()` was given an invalid address, and `VM_OPEN` and `VM_CLOSE` to indicate region opening and closing. The last two events are useful to initialize and finalize memory shared in multiple regions or mapped from persistent storage.

The call `(*memoryf)(vm,addr,csz,nsz,disc)` obtains or releases memory for region `vm`. The arguments `csz` and `nsz` define the current and new sizes of a memory segment. In any `memoryf` call, at most one of `csz` and `nsz` can be zero. If `csz` is zero, `memoryf` returns a new segment of memory of size `nsz`. Otherwise, `memoryf` tries to change the size of the segment pointed to by `addr` from `csz` to `nsz` without moving it. So, even when `nsz` is zero and the given segment is successfully freed, `memoryf` should return `addr` to indicate that the resize succeeds. Sizes in `memoryf` calls are always multiples of `round`. Depending on the region, `round` should be chosen to optimize some relevant system-dependent parameters such as page or disk block sizes. If `round` is zero, the library will pick a convenient size such as the page size.

Figure 2 shows the `heapmem()` function taken from the `Vmdcheap` discipline. This function obtains and frees space from the `Vmheap` region via *Vmalloc* allocation functions. On line 4, since at most one of `csz` and `nsz` can be zero, the value of `nsz` in the call `vmalloc(Vmheap,nsz)` must be non-zero. On line 6, if `addr` is successfully freed, `heapmem()` returns `addr` to indicate success to the calling function. On line 7, `vmresize()` is called to resize an existing block of memory without moving it (the last argument of the call is zero). This is important because elsewhere there may be pointers pointing into some part of the segment `addr`.

The example in Figure 2 does not use the region argument `vm` and the discipline argument `disc`. However, it is easy to imagine cases where these will be useful. For example, a discipline based on file

```
typedef struct _vmdcmmap_s
{   Vmdisc_t  disc;      /* Vmalloc discipline */
    int       fd;        /* file descriptor    */
    ...                  /* whatever else      */
} Vmdcmmap_t;
```

Figure 3: An application-extended discipline structure

memory mapping will need a place to store a file descriptor for the file being manipulated. A way to do this is to extend the *Vmalloc* discipline structure as in Figure 3. If `mmapdisc` and `disc` are of types `Vmdcmmap_t*` and `Vmdisc_t*`, C and C++ casting rules allow the constructs `(Vmdisc_t*)mmapdisc` and `(Vmdcmmap_t*)disc` to work.

2.3 Allocation methods

Each region must select a method for memory allocation. The predefined methods are: `Vmbest`, `Vmpool`, `Vmlast`, `Vmdebug`, and `Vmprofile`.

2.3.1 General purpose allocation with `Vmbest`

`Vmbest` is a general purpose allocator. The basic allocation strategy is best-fit, i.e., an allocation request is satisfied from a smallest free area that fits the required size. Free areas are kept in a top-down splay tree [7] for fast search. Even though there is an example of quadratic fragmentation for best-fit [2] , experience and theoretical evidence in bin-packing problems [4, 5, 9, 10] indicate that best-fit behaves nicely in general. `Vmbest` has a few additional heuristics to improve speed and reduce fragmentation:

- Free blocks are cached until a future allocation. This strategy benefits programs that free everything before exiting by avoiding expensive coalescing. It also benefits programs that continually free and allocate the same block types by giving the allocator the opportunity to avoid some searches if requested sizes fit recently freed blocks.
- Frequent and small allocation sizes are handled in an adaptive manner to speed up allocation. This strategy benefits programs that allocate small data types.
- When a block is resized to grow and has to be moved because it cannot be extended in place, a small amount is added to the size before searching for new space. This strategy reduces data movement and fragmentation because such a block is often resized again.
- The "wilderness preservation heuristic" [4] is observed. This means that the free area with highest address, or the wilderness, is only allocated as a last resource. This strategy prevents unnecessary growth of the arena.

`Vmbest` is analogous to the ANSI-C *malloc* interface. Section 3.1 presents a performance study comparing `Vmbest` and currently popular *malloc* implementations.

2.3.2 Special purpose allocation with `Vmpool` and `Vmlast`

`Vmpool` and `Vmlast` are special purpose allocators. `Vmpool` allocates blocks of a single size determined by the first call to `vmalloc()` after `vmopen()` or `vmclear()`. `Vmlast` allocates without freeing or resizing except on the last allocated block. Both `Vmpool` and `Vmlast` are faster and more space-efficient than `Vmbest` because the allocators do not have to maintain information on a per-block basis. Section 3.2 shows examples of performance gain using these methods.

2.3.3 Memory debugging with `Vmdebug`

`Vmdebug` is a general purpose allocator equipped with aids to detect common memory violations such as memory overwrites or freeing and resizing unallocated data. Thus, `Vmdebug` performs a subset of functions provided in Purify [11] . One advantage of `Vmdebug` over Purify is that it does not change the executable code which may hide certain elusive bugs. In addition, since it is just another allocation method, `Vmdebug` can be put to use any time during program execution by simply creating a new region. Two region flags affect the behavior of `Vmdebug`: `VM_DBABORT` aborts the program upon a detected error and `VM_DBCHECK` checks the region integrity on each allocation request. Since checking region integrity can be expensive, applications may leave `VM_DBCHECK` off and occasionally call `vmdbcheck(vm)` instead. The call `vmdbwatch(vm,addr)` can be used to watch when an address `addr` is met in an allocation function. Section 4 gives examples of how `Vmdebug` is used in the *malloc* compatibility interface.

2.3.4 Memory profiling with `Vmprofile`

`Vmprofile` is a general purpose allocator that also collects data on space allocation. The profiler summarizes space allocated and freed at each applicable program text line. The call `vmprofile(vm,fd)` outputs profile data in region `vm` to file descriptor `fd`. The special value `NULL` for `vm` causes output of profile data for on all regions instrumented with `Vmprofile`. Section 4 gives an example of profiling output.

3 Performance

This section presents a study comparing the general purpose allocator `Vmbest` against currently popular *malloc* implementations and gives a few examples of performance benefits of using `Vmlast` and `Vmpool` in appropriate contexts.

3.1 Comparing `Vmbest` to popular *malloc* versions

Previous allocation performance studies [4, 3] often employed randomly generated data to exercise the allocators. Though such simulations can give useful insights into the implemented algorithms, it is hard to create data that truly model operations in real programs [12] . Zorn and Grunwald [13] propose a good methodology in which allocation performance is measured using actual applications. However, they measure allocation performance using direct program execution. This is fine for measuring space because only the allocators allocate space but measuring time can be tricky due to other work done in the programs. To be consistent in measuring both space and time, the approach taken here is:

- Construct a simulator to execute any sequence of: allocate, free, and resize. The simulator can exercise any allocator by linking with it.
- Link the programs of interest with *Vmalloc* to generate traces of allocation requests using the `VM_TRACE` option (Sections 2.1 and 4).
- Process the traces into the format required by the simulator. This format is designed to reduce computation to a minimum of invoking allocation functions and accumulating resource usages.
- Execute the properly instrumented simulator with given data to measure resource consumption using a particular allocator. Space obtained in each allocation is cleared to ensure that the respective memory pages are indeed allocated by the operating system and improve the accuracy of system time.
- Each allocation is set to be at a minimum of 2 words so that blocks can be maintained efficiently in a doubly linked list as they are allocated or freed. This enables comparison of the Boehm-Weiser conservative garbage collector [14] and other allocators on the same footing.

The allocators that will be compared against `Vmbest` are:

- *V*: by Phong Vo, distributed with System V Release 4. This *malloc* is based on a best-fit strategy using a bottom-up splay tree for free blocks.
- *S*: by Chris Aoki and C. Adams, distributed with SUN OS. This *malloc* is Stephenson's better-fit allocator [5] .
- *P*: by Chris Kingsley, modified and distributed with the Perl language interpreter. This *malloc* uses a power-of-two buddy system.
- *X*: by Doug McIlroy, used in the 10th Edition Bell Labs Research UNIX system. This *malloc* is based on a first-fit strategy with a roving pointer. Small blocks are cached on freeing to speed up subsequent allocations.
- *H*: by Mike Haertel, distributed with the GNU C library, dated Mar 1 1994. This allocator segregates blocks of same size in same pages.
- *L*: by Doug Lea, distributed with the GNU C++ library, version 2.5.3b.

Dataset	Allocate	Free	Resize	MaxAllocate	MaxBusy
gawk	723,470	722,922	150,888	47,684K	38K
db.2X	880,688	879,648	0	10,953K	20K
db.ioQ	66,626	11,912	0	1,777K	1,411K
mt.ioQ	69,387	10,677	0	1,867K	1,575K
C++parser	44,730	5,381	0	1,024K	848K
graph	111,782	14,882	0	1,706K	1,590K
S	102,146	83,124	56	800,369K	5,887K
ciao	163,044	145,113	3,246	912,507K	6,839K
fragment	10,001	0	10,000	1,563,203K	547K

Table 1: Summary of datasets in the simulation study

- *B*: by Hans Boehm and Mark Weiser, a conservative garbage collector, version 4.5. Here, `GC_malloc_uncollectable()` and `GC_free()` are used so that garbage collection is bypassed and only allocation performance is measured. Allocated space is not cleared because that is already done by the allocator.
- *C*: the same Boehm-Weiser garbage collector. Here, `GC_malloc()` is used and objects are freed by removing them from the linked list discussed above. Thus, in this case, the garbage collection performance is measured.

The datasets studied here contain allocation traces from a diverse set of applications including parsers, database queries, data analyses and interactive graphics. Each dataset either performs a large number of allocations or allocates a large amount of data, or both. Table 1 summarizes information about the datasets. The first three numerical columns display total numbers of different types of operations. The fourth column shows the total of space requested via `malloc()` or `realloc()`. The last column shows the maximum busy space at any time. Below are brief descriptions of the programs and their input data:

- *gawk*: The GNU *awk* program with input data as described in [13] .
- *db.ioQ, db.2X*: Two processes in the Daytona database system [15] for compiling and executing a large query.
- *C++parser*: A C++ language parser parsing a large program.
- *mt.ioQ*: The program that prepares an allocation trace from *Vmalloc* for the allocation simulator, here using the *db.ioQ* trace as input. This program allocates memory via three separate regions (Section 3.2).
- *graph*: A graph processor parsing a large directed graph specification and building an in-core representation of the graph.
- *S*: The *S* statistical analysis system [16] processing its regression test suite.

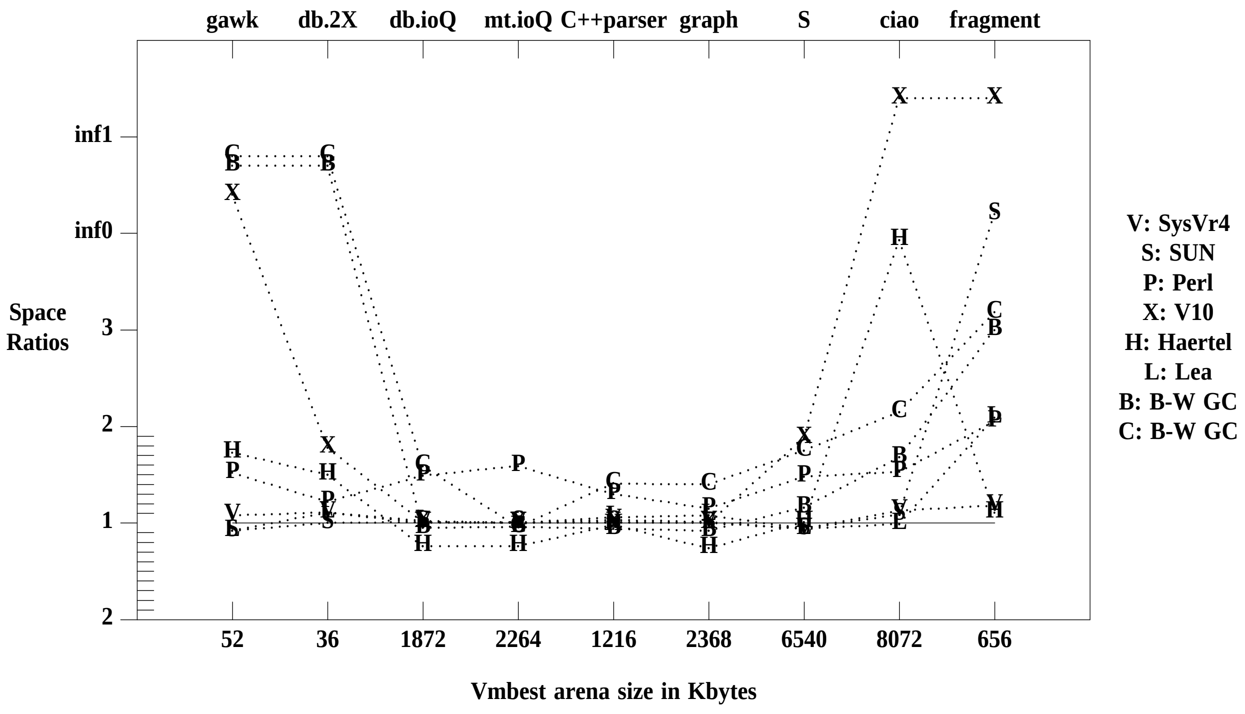
```
size = 32;
big = malloc(size);
for(n = 0; n < 10000; ++n)
{    small[n] = malloc(24);
     big = realloc(big,size += 32);
}
```

Figure 4: An allocation pattern causing fragmentation in some allocators

- *ciao*: A program and data visualization system [17] .
- *fragment*: A program running the allocation pattern in Figure 4. This pattern occurred in an early version of the IFS language compiler [18, 19] .

Figure 5 shows the time and space performance comparisons between `Vmbest` and the described *malloc* versions. To give an idea of magnitude, bottoms of the graphs are labeled with the time measures and arena sizes for `Vmbest`. Measurements were done on a completely idle Sparc-5 running SUNOS4.1. Each time value is a sum of cpu and system times obtained by running the simulator 10 times and taking the average. Each data point on the graphs is constructed by dividing the time or space value of the respective allocator by that of `Vmbest` if the former is larger; otherwise, the reverse is done. Thus, the horizontal lines at 1 in both graphs represent `Vmbest`. An allocator is slower or faster (less or more space efficient) than `Vmbest` if the corresponding data point is above or below this line. A data point between *inf0* and *inf1* means that the respective allocator is able to service all allocation requests but its time or space value is at least 4 times that of `Vmbest`. A datapoint beyond *inf1* means that the respective allocator fails – typically because it runs out of memory. Below are a few observations about the data:

- `Vmbest` is competitive to the best allocators in time. However, note that the times used by `Vmbest` on Figure 5 are only small fractions of application running times. For example, *gawk* takes on average about 95 seconds to process the given input and only about 7 seconds of that time is for allocation. This means that the speed of an allocator is not an important factor as long as it remains reasonable. Much more important is space fragmentation which has implication on memory contention not just for the immediate process but also others running concurrently on the system.
- Allocators B, C, H, S, and X suffer significant fragmentation on a few datasets. In fact, X runs out of memory on *ciao* and *fragment*. P typically used 30% to 40% more space than the better allocators because it rounds any request size up to the next power of 2. L manages space well in general except on the dataset *fragment*. V and `Vmbest` do fine in all cases.
- Table 2 summarizes `realloc()` calls on *gawk*, *ciao* and *fragment*. The "move" columns show the number of times that resized blocks are moved to new locations. The "copy" columns show the total amounts of data copied in such cases. *gawk* has the most `realloc()` calls but mostly for small, short-lived blocks. *ciao* uses a few large, long-lived buffers that are resized



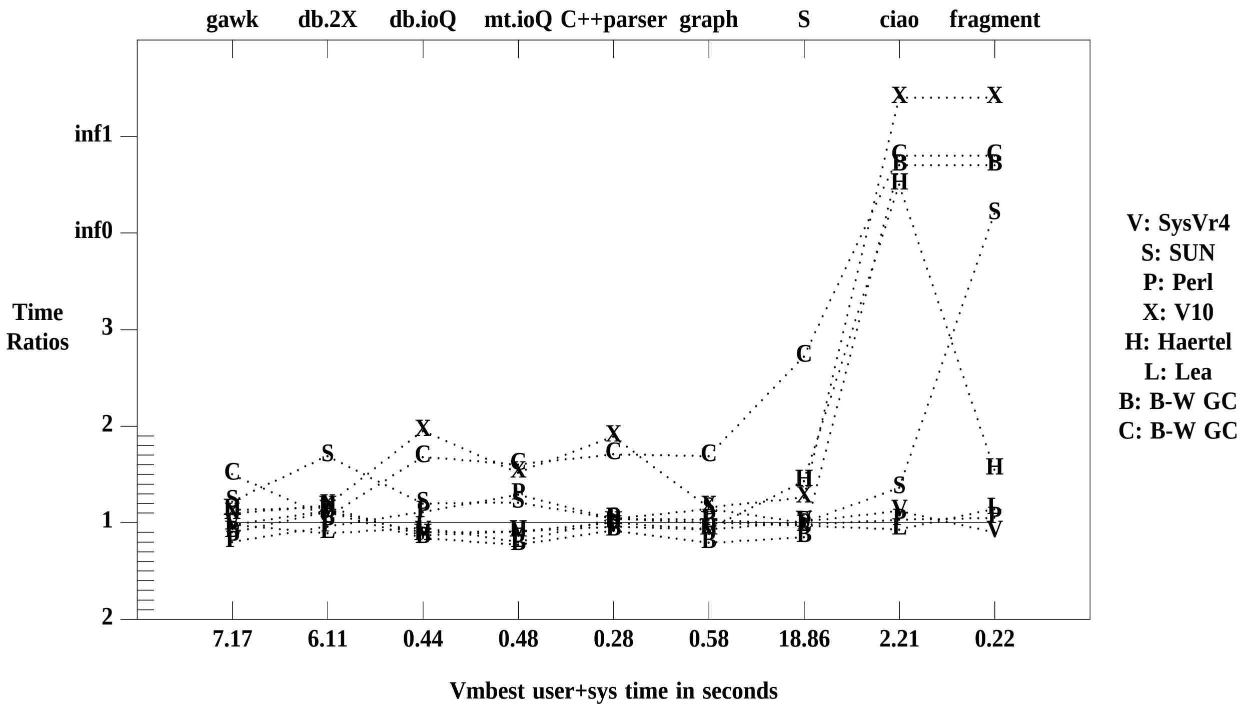


Figure 5: Time and space allocation performances normalized to `Vmbest`

	gawk(150,888)		*ciao*(3,246)		*fragment*(10,000)	
Allocator	move	copy	move	copy	move	copy
`Vmbest`	20,114	470K	223	11,546K	12	335K
V: SysVr4	150,869	3,536K	295	14,373K	8	442K
S: SUN	72,194	1,573K	298	14,267K	317	49,288K
P: Perl	150,884	3,536K	163	2,540K	13	512K
X: V10	17,165	326K				
H: Haertel	150,869	3,536K	404	81,543K	12	356K
L: Lea	105,479	2,357K	209	11,938K	16	842K
B: B-W GC	150,884	3,536K	488	91,692K	99	12,332K
C: B-W GC	150,867	3,536K	470	91,690K	96	12,330K

Table 2: `realloc()` statistics for the datasets *gawk* and *ciao*

many times. For *gawk* and *ciao*, the `Vmbest`'s resize strategy of adding a small amount to a moved block helps to reduce the total number of moves and the amount of copied data. P's policy of rounding a size up to its next power of 2 does even better on *ciao* because the buffers are large.

- Except for H, V, and `Vmbest`, the other allocators lose large amount of space on the dataset *fragment*. A likely reason is because they do not observe the wilderness preservation heuristic (Section 2.3.1). When the arena requires extension, all allocators extend it by a multiple of some fixed value (usually the page size). This often leaves some extra space after the respective allocation request is satisfied. Without wilderness preservation, this space is immediately available for allocation. On *fragment*, arena extension happens mostly when the `big` block in Figure 4 grows. Using the extra space for a `small` allocation prevents the `big` block from being able to grow in place and causes it to move in its next resizing. V and `Vmbest` preserve wilderness and avoid this problem. H manages to mostly avoid the problem due to its technique of allocating page at a time for groups of blocks of the same size. Since the extra space is unlikely to be page aligned, H is unlikely to use it for a `small` allocation.

- On the dataset *gawk*, the final arena sizes for B and C are respectively 257K and 385K indicating that switching to garbage collection significantly increases space requirement. This is probably because free blocks are not collected fast enough. The peak difference is on *db.2X* where B uses 257K and C uses 773K. Turning on garbage collection also negatively affects time. On *gawk*, the allocation time increases from an average of 6.53s for B to 10.73s for C. The peak difference is on dataset *S* where B uses an average of 16.40s while C uses 51.33s. Garbage collection is a good technique that solves many programming problems. Applications with memory leakage sometimes look to it as a quick fix. The evidence here suggests that this is not always a good idea. Unless the leakage is severe, a program may be better off just ignoring it.

	C++parser		*graph*	
	`Vmlast`	`Vmbest`	`Vmlast`	`Vmbest`
Arena size	1,076K	1,216K	1,724K	2,368K
Cpu+Sys	0.22s	0.28s	0.50s	0.58s

Table 3: Performance comparison of `Vmlast` and `Vmbest`

3.2 Efficient allocation with `Vmpool` and `Vmlast`

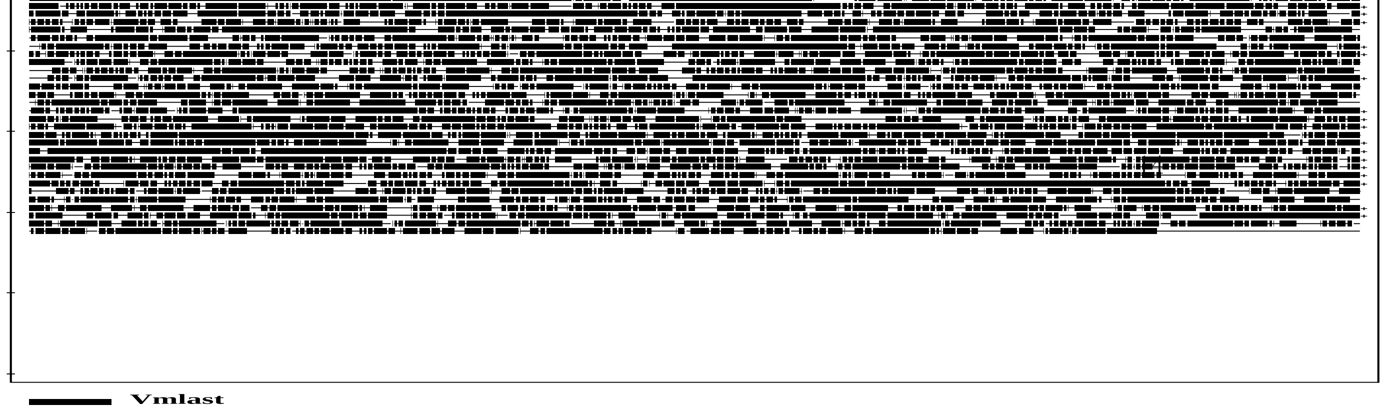
Programs can benefit from the special purpose allocators `Vmpool` and `Vmlast`. For example, it is a folklore that parsers and compilers often do well with an allocator that never frees. The datasets *C++parser* and *graph* are typical cases. Table 3 compares the space and time performances between `Vmbest` and `Vmlast` on these datasets. `Vmlast` uses less space than `Vmbest` even though it ignores most free calls. The reason for this becomes clear in Figure 6 which shows snapshots of the `Vmlast` and `Vmbest` allocation arenas for *C++parser* when the allocated space just exceeds 195K bytes. Each line in the pictures represents 8K bytes. Thick segments are allocated space and thin segments are free areas. The relatively large free areas in the `Vmlast` arena are lost because they are not reusable. The `Vmbest` arena is fairly well packed but it has numerous thin unused areas corresponding to block headers. The sum of such header space is much more than the lost space in `Vmlast`.

The program producing *mt.ioQ* allocates space via three regions, each with a different method, `Vmbest`, `Vmlast`, and `Vmpool`. All three regions use the standardly provided discipline `Vmdcsbrk` which is based on the `sbrk()` system call. At peak time, the allocated space is 1,575K bytes. With regions, this is satisfiable with an arena of size 1,812K and takes 0.35s allocation time. When allocations are forced to be done with a single `Vmbest` region, arena size increases to 2,264K and allocation time increases to 0.48s. Figure 7 shows the arenas when the allocated space just exceeds 270K bytes. The light gray areas in Figure 7 indicate that most allocations are done in the `Vmpool` region. Thus, the good performance for *mt.ioQ* is due to the header saving and fast allocation speed of `Vmpool`.

4 The compatible *malloc* interface

A *malloc* interface is provided to make *Vmalloc* functionality available to current applications. By default, this interface allocates via the heap region `Vmheap` which uses the `Vmbest` method and the `Vmdcsbrk` discipline. A program that creates regions with *Vmalloc* can redirect the *malloc* interface to any other region by setting the global variable `Vmregion` to the wanted region. This is a useful technique to redirect allocation before calling functions in reusable libraries that may allocate memory.

Applications can change the allocation method of *malloc* at start-up time by setting certain environment variables. For example, setting `VMETHOD=Vmlast` selects the `Vmlast` method. Note, however, that setting the method to `Vmpool` is usually not a good idea unless it is certain that the



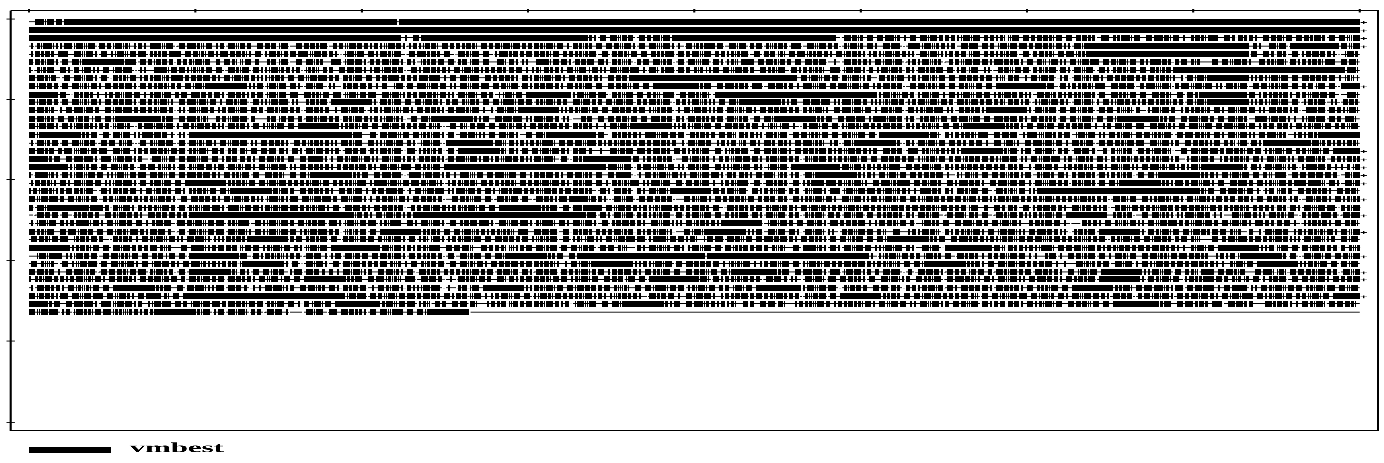


Figure 6: Arenas for *C++parser* with `Vmlast` and with `Vmbest`

Figure 7: Arenas for *mf.c* with different method regions and with only Vmbest

```
 1. #include   <vmalloc.h>
 2. foo(s)
 3. char* s;
 4. {     char* news = malloc(strlen(s));
 5.       strcpy(news,s);
 6.       free(s);
 7. }
 8. char* bar(s)
 9. char* s;
10. {     char* news = malloc(strlen(s)+1);
11.       strcpy(news,s);
12.       free(news);
13.       return news;
14. }
15. main()
16. {     char* s = "1234";
17.       foo(s);
18.       s = bar(s);
19.       free(s);
20. }
```

Figure 8: A buggy C program

program only allocates one type of data. Other variables provide finer controls depending on the method in use. It is best to show this with examples.

Consider the program in Figure 8 which contains a few common memory errors. Figure 9 shows that this program is first compiled and linked with *Vmalloc*. The flag `-DVMFL` enables recording of the file name and line number for each allocation call. This makes error messages more meaningful. The first execution results in a memory fault. Then, the `Vmdebug` method is selected by setting `VMDEBUG=1`. Three errors are detected. The first one is a memory corruption in a block allocated on line 4 of the code. This is because not enough space was allocated for the null byte at the end of a C string. The second error on line 6 concerns a block being freed but never allocated. This is because the function `foo()` was passed a literal string as its argument (line 17 of Figure 8). This error is what causes the memory fault! The third error concerns freeing a block already freed. To examine this, the aberrant block `0xd6e0` is added to `VMDEBUG` so that it would be watched for by the allocator. The resulting two `alert` messages show that the block is allocated on line 10 and subsequently freed on line 12. Thus, the `free()` call on line 19 is redundant and, under normal circumstances, would have corrupted the heap. Note that the memory fault is avoided with `Vmdebug` because the allocator prevents bad operations from proceeding too far.

Figure 10 shows that setting `VMTRACE=/dev/tty` causes whatever allocator in use to produce a trace of allocation calls. Such an allocation trace can be useful for an in-depth analysis of how a program behaves with respect to allocation. For a quick assessment of memory usage, the `Vmprofile`

```
$ cc -DVMFL -O t.c -lvmalloc -o t
$ t
Memory fault(core dump)

$ VMDEBUG=1 t
corrupted data:region=0xc648:block=0xd6b0:bad byte at=4:allocated at=t.c,4:
free error:region=0xc648:block=0xc180:unallocated block:detected at=t.c,6:
free error:region=0xc648:block=0xd6e0:already freed:detected at=t.c,19:

$ VMDEBUG=1,0xd6e0 t
corrupted data:region=0xc648:block=0xd6b0:bad byte at=4:allocated at=t.c,4:
free error:region=0xc648:block=0xc180:unallocated block:detected at=t.c,6:
alert:region=0xc648:block=0xd6e0:size=5:just allocated:detected at=t.c,10:
alert:region=0xc648:block=0xd6e0:size=5:being freed:detected at=t.c,12:
free error:region=0xc648:block=0xd6e0:already freed:detected at=t.c,19:
```

Figure 9: Examples of debugging with *Vmalloc*

```
$ VMTRACE=/dev/tty t
0x0:0xc648:4:0xc288:best:t.c,4:
0x0:0xc658:5:0xc288:best:t.c,10:
0xc658:0x0:8:0xc288:best:t.c,12:

$ VMPROFILE=/dev/tty t
ALLOCATION USAGE SUMMARY:n_alloc=3:n_free=1:s_alloc=21:s_free=5:
region=0xc648:n_alloc=3:n_free=1:s_alloc=21:s_free=5:max_busy=21:extent=4096:
file=<>:n_alloc=1:n_free=0:s_alloc=12:s_free=0:
         line=0:region=0xc648:n_alloc=1:n_free=0:s_alloc=12:s_free=0:
file=t.c:n_alloc=2:n_free=1:s_alloc=9:s_free=5:
         line=4:region=0xc648:n_alloc=1:n_free=0:s_alloc=4:s_free=0:
         line=10:region=0xc648:n_alloc=1:n_free=1:s_alloc=5:s_free=5:
```

Figure 10: Examples of tracing and profiling with *Vmalloc*

method can be used. Setting `VMPROFILE=/dev/tty` both selects `Vmprofile` and causes it to send profiling data to the terminal. The file name `<>` in Figure 10 summarizes allocations from libraries and code not instrumented with `vmalloc.h` and `-DVMFL`. A total of 9 bytes are allocated in `t.c` but only 5 are freed. The unfreed 4 bytes are allocated on line 4 in function `foo()` which returns without freeing them, a typical example of memory leakage.

5 Discussions

We have presented *Vmalloc*, a memory allocation library that allows a wide range of memory manipulation. Flexibility is accomplished by: (1) using regions to organize memory and group logically related allocation tasks, (2) obtaining memory via application-definable disciplines, and (3) customizing memory management with appropriate allocation methods. `Vmalloc` provides two standard disciplines, one to get memory using the system call `sbrk()` and the other to get memory from the standardly provided heap region using *Vmalloc* allocation calls. The latter discipline makes it simple to create special purpose regions out of heap memory. Examples were given to show the simplicity of writing and using disciplines and how to extend the discipline structure for application-specific requirements. The predefined set of allocation methods includes general purpose allocation, fast allocation of fixed size blocks or without freeing, memory debugging, and memory profiling.

A compatible *malloc* interface is provided. This package is instrumented with *Vmalloc* methods. Examples were given to show how a *malloc*-based application using *Vmalloc* can perform efficiently under normal circumstances using a good allocator and can also do memory debugging and profiling by selecting specialized allocators using environment variables. In particular, this means that production code can perform preliminary debugging at customers' sites, a valuable tool considering that memory bugs are often elusive and sensitive to the execution environments.

A performance study based on allocation traces from real applications compared the general purpose allocation method `Vmbest` of *Vmalloc* to other popular *malloc* implementations. The results show that `Vmbest` performs competitively to the best of these *malloc* versions in both space and time. In fact, only `Vmbest` and the System V Release 4 *malloc* manage to do well in all cases including those that cause a few of the *malloc* versions to completely or partially fail. The time efficiency of `Vmbest` is good considering that because of *Vmalloc*'s architecture, its calls go through more indirections and do more preliminary work than their *malloc*'s counterparts.

Careful organization of memory into regions and use of the special allocation methods `Vmpool` and `Vmlast` can save significant amounts of space and time. Examples were given to show efficiency improvement when the general purpose method `Vmbest` was replaced with appropriate special purpose methods. An example application showed that `Vmpool` is an effective allocator of fixed size blocks such as C and C++ structures. In fact, for C++, `Vmpool` would be a good base for implementing constructors and destructors of heavily used classes.

Beyond performance tuning, regions are useful for other reasons. In applications needing asynchronous processing such as threads or signal handling, *malloc* typically falls short due to reentrant problems. Multi-threaded programs based on *malloc* also have no convenient ways to clean up when a thread is finished. In general, *Vmalloc* calls are safe because they lock regions during operation and check for bad arguments in operations such as freeing or resizing. In a multi-threaded program, different regions can be used to support different threads. Then, safety issues concerning reentrance are lessened, and cleaning up on thread exit is just a region close.

Modern applications are often assembled using code from multiple sources. In such a setting, memory errors are hard to isolate and estimating the memory consumption in each package is at best an art. This is in part because it is hard to find out the exact origin of each allocation call with *malloc*. Packages such as Purify and the memory debugging and profiling methods of *Vmalloc* can alleviate some of the problems. A better solution would be to require different packages to allocate from different regions. Then, memory would be better organized, there is less chance of code in one package overstepping space allocated in another, and it is easy to gather space usage estimates on a per package basis.

We end with a discussion on the use of methods and disciplines as abstractions of memory management and memory acquisition functions. The *Vmalloc* method provides a single abstraction that hides different techniques to manage memory resources. Allocation calls on a region are identical, independent from whatever method specified at region opening. A benefit for applications is that code tuning and instrumentation with different methods can be done easily. An example of this is provided by *Vmalloc* itself in the compatible *malloc* interface which allows method selection by environment variables. This architecture also opens up the possibility of future method addition without too much upheaval in the interface.

A discipline packages data and functions for resource acquisition and handling of certain exceptional events. Despite its importance, resource acquisition is often glossed over by library implementors. An obvious example in many of the *malloc* implementations considered here is the hard-coding of the system calls `brk()` or `sbrk()` as means to obtain memory. This makes these packages unusable beyond heap memory. By allowing the creation of regions with application-definable disciplines, *Vmalloc* opens up a new level of flexibility in managing different memory types or just organizing the same memory but for different purposes. The most important part is that any extension of the library for exotic memory management can be easily created by application writers themselves without touching library code. In fact, some *Vmalloc* users have written disciplines for shared and mapped memory.

Acknowledgement

I would like to thank Doug McIlroy and David Korn for their valuable advices on interface design.

References

[1] D.E. Knuth. *The Art of Computer Programming, Volume 1.* Addison-Wesley, 1968.

[2] J.M. Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. *The Computer Journal*, 20:242–244, 1975.

[3] J.E. Shore. On the External Storage Fragmentation Produced by First-Fit and Best-Fit Allocation Strategies. *CACM*, 18:433–440, 1975.

[4] D.G. Korn and K.-P. Vo. In Search of a Better Malloc. In *Proc. of the Summer '85 Usenix Conference*, pages 489–506. USENIX, 1985.

[5] C.J. Stephenson. Fast fits: new methods for dynamic storage allocation. In *Proc. Ninth ACM Symp. on Operating System Principles*, pages 30–32. Bretton Woods, NH, 1983.

[6] J. Vuillemin. A unifying look at data structures. *CACM*, 23:229–239, 1980.

[7] D. Sleator and R.E. Tarjan. Self-Adjusting Binary Search Trees. *JACM*, 32:652–686, 1985.

[8] D. Grunwald and B. Zorn. CustoMalloc: Efficient Synthesized Memory Allocators. *Software - Practice And Experience*, 23(8):851–869, 1993.

[9] D.S. Johnson, A. Demers, J.D. Ullman, M.R. Garey, and R.L. Graham. Worst case performance bounds for simple one-dimensional packing algorithms. *SIAM J. Computing*, 3:299–325, 1974.

[10] P.W. Shor. The average case analysis of some on-line algorithms for bin-packing. *Combinatorica*, 6:179–200, 1986.

[11] R. Hastings and R. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter 1992 Usenix Conference*, pages 125–136. USENIX Association, 1992.

[12] B. Zorn and D. Grunwald. Evaluating Models of Memory Allocation. *ACM Transaction on Modeling and Computer Simulation*, 4(1):107–131, 1994.

[13] B. Zorn and D. Grunwald. Empirical measurements of six allocation-intensive C programs. *SIGPLAN Notices*, 27(12):71–80, 1992.

[14] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software - Practice and Experience*, pages 807–820, 1988.

[15] R. Greer. Daytona and the Fourth-Generation Language Cymbal. *Available from author at rxga@research.att.com*, 1995.

[16] R.A. Becker, J.M. Chambers, and A.R. Wilks. *The New S Language.* Chapman and Hall, 1988.

[17] Y.-F. Chen, G.S. Fowler, E. Koutsofios, and R.S. Wallach. Ciao: A Graphical Navigator for Software and Document Repositories. In *International Conference on Software Maintenance*, 1995.

[18] K.-P. Vo. IFS: A Tool to Build Integrated, Interactive Software. *AT&T Bell Labs Tech. J.*, 64(9):2097–2117, 1985.

[19] G.S. Fowler, J.J. Snyder, and K.-P. Vo. End-User Systems, Reusability and High Level Design. In *Proc. of the Usenix VHLL Conference*, pages 101–118. USENIX, 1994.