Some Performance Experiments for Simple Data Structures

Francisco J. Ballesteros
Enrique Soriano
RoSAC−2011−1

Laboratorio de Sistemas — Universidad Rey Juan Carlos
April 5 2011, Madrid, Spain.
`http://lsub.org`

ABSTRACT

Actual program performance is non−intuitive. Stroustrup, the author of C++, included measurements in a talk for ordered insertion in C++ vectors and lists. We measured the same issue using C in Plan 9, Mac OS X, and Linux, and C++ in Mac OS X. This document describes the results.

1. Which is faster, a list or a vector?

In a recent talk by Stroustrup in Madrid, it was pointed out that it is not intuitive if a list or a vector is a better data structure for ordered insertion for a given number of elements. In principle, according to bibliography, the list should win for container sizes greater than two (or a close number). However, a slide presented results from an experiment, which we reproduce in figure 1.

Perhaps surprisingly, in the experiment shown in the talk, the vector remains a better data structure for this purpose until 40.000 elements have been inserted (or a close number). This was for small element sizes.

However, things are even better (worse?). We reproduced the experiment in Plan 9 from Bell Labs (in C), Mac OS X (in C and C++) and Linux (in C). And we obtained some contradictory results from the resulting measures!

It seems that effects like hardware caches, operating system memory management, standard library implementations for the language used, etc. can in fact dominate what happens in the end to the performance of the program.

That is, in practice, results from complexity theory seems to be totally neglected. In our opinion, what happens is that software is so complex, and there are so many layers of software underneath the application code, that it is not even clear which data structures are better; at least for the simple case we describe here.

This work supported in part by Spanish CAM S2009/TIC−1692

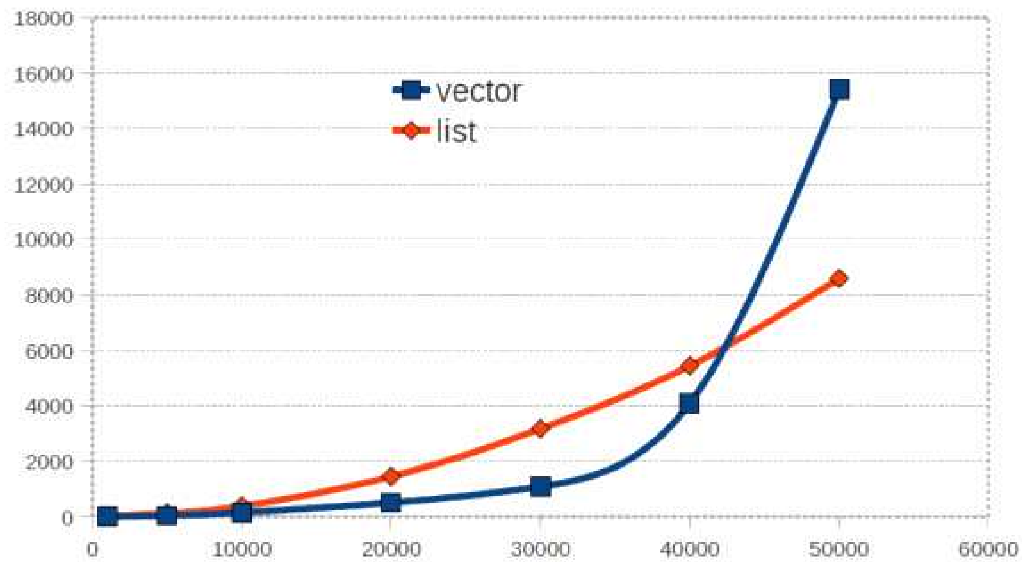

Figure 1 Ordered insertion times for C++ vector and list (Stroustrup, Madrid 2011 talk).

2. Reproducing the experiment

2.1. C program on Plan 9

We tried to reproduce the experiment, using Plan 9 from Bell Labs as the operating system running on a quadruple processor AMD64 Phenom 2.2GHz with 4GiB of memory installed. The machine has 64 bits per word, but the operating system and compilers installed keep it running in 32 bits (which is considered a machine word in what follows).

A C program was used to perform insertions on one of three different data structures. The program is reproduced in appendix A. The three data structures are: A regular array of elements (`Arry`); A linked list of elements (`List`); and an array of pointers to elements (`Ptrs`). See figure 2.

Each element is represented as an integer plus some optional space, so that we could reproduce the experiment for different element sizes (all of them multiples of the machine word).

```
struct El
{
	int n;		/* element value */
	int dummy[];
};
```

The array grows as elements are added, growing for a customizable number of new elements each time.

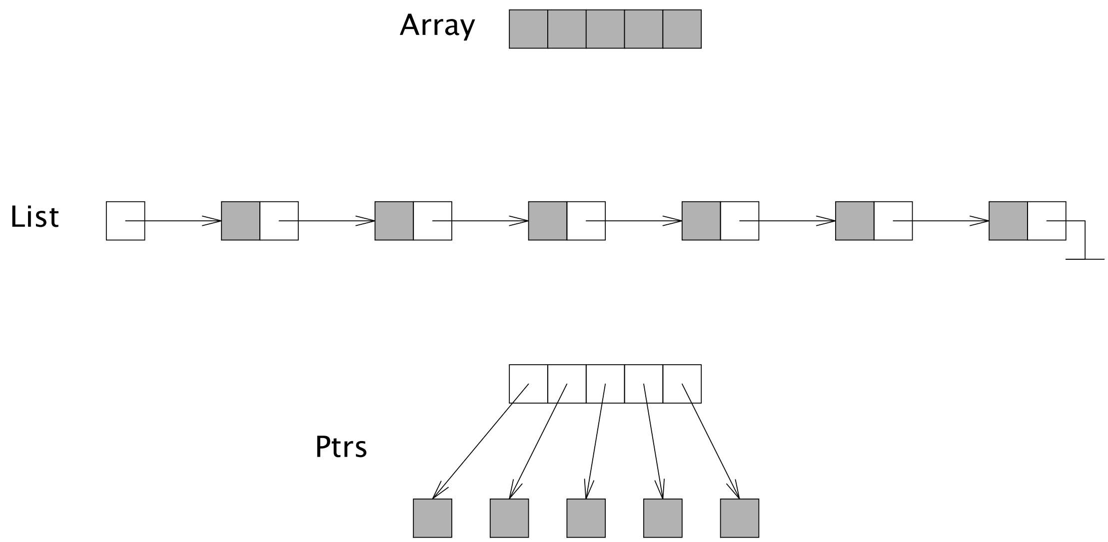

Figure 2 Data structures used for the experiment: array (`Arry`), list (`Node*`), and array of pointers (`Ptrs`). Filled boxes are elements.

```
struct Arry
{
     int nels;       /* number of elements used */
     int naels;      /* number of elements allocated */
     El *els;        /* array of elements */
};
```

Each linked list node is as expected:

```
struct Node
{
     Node *next;     /* element in list */
     int n;          /* element value */
     int dummy[];
};
```

The array of pointers is similar to the array above, but refers to external elements instead of containing them:

```
struct Ptrs
{
     int nels;       /* number of elements used */
     int naels;      /* number of elements allocated */
     El **els;       /* array of elements */
};
```

2.2. C++ program on Mac OS X

We used Mac OS X running on a 2.4GHz Core 2 Duo T7700 with 2GiB of memory installed. The C++ compiler was g++ version 4.2.1, and the libraries were libstdc++ 7.9.0 and libSystem 125.2.1.

The data structures used to compare lists and arrays are the STL implementations for `list<int>` and `vector<int>`. The source code for the program is included in appendix B.

2.3. C program on Mac OS X

We used the same Mac OS X machine than in the C++ set-up. The compiler was gcc version 4.2.1, and the standard library was libSystem 125.2.1. The C program is a port of the used in the Plan 9 experiment, using the same data structures.

2.4. C program on Linux

For the experiments on Linux, we used another machine, an Intel Pentium 4 CPU 2.40GHz with 512 MiB of RAM. The compiler was gcc version 4.4.1, and the standard C library was glibc 2.10.1. The C program is a port of the one used in the Plan 9 experiment, using the same data structures.

2.5. Experimental set up

Each experiment consisted on measuring the insertion of a given number of elements into one of the three data structures, with a fixed element size (and fixed increment size for C arrays). The elements inserted where integers (plus some optional space if required) taken in ascending order, in descending order, or in (pseudo-)randomized order. In the last case, the sequence of randomized integers was the same for all experiments, to make it fair.

Because these data structures are not isolated from the rest of the application when used in practice, 64 bytes of dynamic memory are allocated between each insertion in all the experiments. This memory is never released.

Measures of time are taken using *nsec*(2) in Plan 9 and *clock*(3) in C++ and C on Mac OS X and Linux. They include the insertion in the data structure and the allocation of memory for elements (allocation of elements in the case of the linked list and the array of pointers, and reallocation for the arrays). They do not include loops, extra allocations used by the program, and (pseudo-)random numbers generation.

3. Effect of increment in arrays

In the C implementation, the value for the increment in growing arrays may be important. This section tries to measure that effect. We inserted 10000 elements in randomized order into the array, for an element size of 4 bytes (1 integer in our experiment): once growing the array 1 element at a time, then growing it 16 elements at a time, and finally growing it 128 elements at a time. Figure 3 shows the time taken for the three experiments in nanoseconds, for Plan 9, Mac OS X, and Linux respectively. The relevant portion of code is as shown in this excerpt from appendix A:

```
	if((a->naels%incr) == 0){
		a->naels += incr;
		a->els = realloc(a->els, a->naels*elsz);
		if(a->els == nil)
			return -1;
	}
```

Figure 4 shows the results of the same experiment, using an element size of 64 bytes instead of 4 bytes, again, for Plan 9, Mac OS X, and Linux respectively.

For 4-byte elements, the graphs do not show significant difference regarding time (due to the scale). However, there can be seen important differences if the growing delta for the array is 128. For example, the Plan 9 program, for arrays up to 400 elements, is 83% faster with *incr=1* than with *incr=128* (mean of 25 independent executions incrementing the array size by 16 elements each time, from 16 to 400 elements).

For the same array sizes, the Mac OS X C program is 115% faster with *incr=1* than with *incr=128*. This is definitely non-intuitive! It seems that it is better to grow the array each time than it is to grow it from time to time. This is quite surprising, since `realloc` is called in each insertion when *incr=1.* Intuitively, one could expect to observe that the program performs better (at least, equally) with large increments. This is not the case.

Figure 4 shows the times for 64-byte elements. With *incr=128*, the Plan 9 program runs quickly out of memory (flat-dotted line in the graph above of figure 4). For increments of 1 and 16, the program can run for a longer number of elements. In Plan 9, *incr=1* performs worse than with *incr=16*, but it is still reasonable for 64-byte elements. In Mac OS X, *incr=1* results better, but comparable. In Linux (running on an older machine), results are comparable (but again, larger values of *incr* do not lead to better results).

For the next experiments, we use an increment of 1, growing the array each time.

3.1. Memory usage

Although figures are not shown here, the amount of memory used in these experiments over Plan 9 is quite different depending on the increment for array growth. In particular, with *incr=1* (growing the array one element at a time) the program consumes a lot less memory in the resulting process image (at the end of the experiment) than it consumes growing the array 128 elements at a time. An increment of 16 causes 14 times more memory to be consumed with respect to the increment of 1. An increment of 128 causes 141 more memory consumption. Also, using 64-byte elements and an increment of 128 makes the program run out of (virtual) memory in our Plan 9 system. Thus, the effect in memory footprint is not to be underestimated.

In what follows we consider only execution time, and not memory consumption.

4. Forward insertion experiment

Figure 5 shows the effect of forward insertion of 4-byte elements (1, 2, 3, etc.) in the data structures, using C in Plan 9. Inserting 4-byte elements in `Arry` takes much less time than inserting on the other two data structures in the long run. For few elements (see the bottom graph) using `Ptrs` is worse than using `List`. However, for a number of elements between 1000 and 2000 elements `Ptrs` becomes better than `List`.

Figure 6 shows the times for 64-byte elements using the Plan 9 C program. For 64-byte elements things change. Instead of being faster, `Arry` becomes slower, and `Ptrs` is not affected as much as the other two data structures. Also, there is a huge jump in execution time after inserting in the array about 3000 elements, which did not happen with 4-byte elements (probably would happen with a higher number of elements, not measured). Also, only for 64-byte elements, `List` is better than `Arry` for less than 700 elements in the data structure (aprox). No crossing point has been found for 4-byte elements: one is either better or worse than the other.

Inserting 4-byte elements using C++ in Mac OS X leads to the results shown in figure 7. Compare with figure 5 (the same experiment using C in Plan 9). Results are the opposite!

Times for inserting 4-byte and 64-byte elements using C in Mac OS X are shown in figures 8 and 9 respectivelly. Times for Linux are depicted in figures 10 and 11. For all these experiments, `Arry` results better than `Ptrs` and `List`, in this order.

Results for C and C++ are the opposite. Moreover, results of 64-byte elements differ from the C program over Plan 9 and the C program over Linux and Mac OS X.

So, which data structure should we use?

5. Backward insertion

Figure 12 shows the effect of backward insertion of 4-byte elements (descending order) in the data structures using C in Plan 9. This time the list wins on the long run, as expected (in backward insertion, elements are always inserted in the head of the list). The same experiment, using 64-byte elements, leads to results shown in figure 13. Results are the equivalent, only that the vector gets worse due to the increase in element size.

Using C++ for 4-byte elements, we obtain the results shown in figure 14. Figures 15 and 16 show the results of inserting 4-byte and 64-byte elements using C in Mac OS X. Results of inserting 4-byte and 64-byte elements using C in Linux are depicted in figures 17 and 18 respectivelly.

For collections up to 400 4-byte elements, `Arry` and `List` are comparable. For larger collections and larger elements, `List` wins, as expected.

6. Randomized insertion

We come to the experiment that motivated this work. This could be compared to the one shown by Stroustrup (but shouldn't).

We inserted 4-byte elements in randomized order into the data structures, using C on Plan 9. `Arry` is better in the long run (but note the memory effects described above). For fewer elements (i.e., about 1500 or less) `List` becomes better. On the other hand, `Ptrs` seems to compete well with the other two ones. See figure 19.

In Plan 9, using 64-byte elements instead, the results are those shown in figure 20. Instead of being faster, `Arry` becomes slower. The increase in element size makes the array take longer. For large collections, `Ptrs` is a good candidate in this case (better than the list in the long run).

Compare now figure 19 with results using C++, shown in figure 21. Surprisingly, our result is the opposite once more. Also, considering the number of elements, the result is also the opposite of the result shown by Stroustrup in his talk. Figure 26 shows our results for the scale used in the Stroustrup's graph.

Times for inserting 4-byte and 64-byte elements using C in Mac OS X results in the graphs depicted in figures 22 and 23. Like in Plan 9, for large collections, `Arry` wins for 4-byte elements, and `Ptrs` wins for 64-byte elements.

The results of inserting a large number of 4-byte and 64-byte elements using C in Linux are depicted in figures 25 and 26 respectivelly. For huge collections, the array wins in this set up.

7. Summary

There is too much complexity. The cache hierarchies in the hardware, the operating system used, the C library and the standard library for the language used; all of them conspire to introduce effects that may even invert the results that you could expect. Clearly, in the end, the results obtained may be justified by different physical (that is, practical) effects and theory would be in accordance with the experiments if we consider such effects. However, it seems that we should use the simplest data structures that simplify our programs, and do not pay attention to the data structures used before measuring our program in our particular compiler, system, and hardware platform.

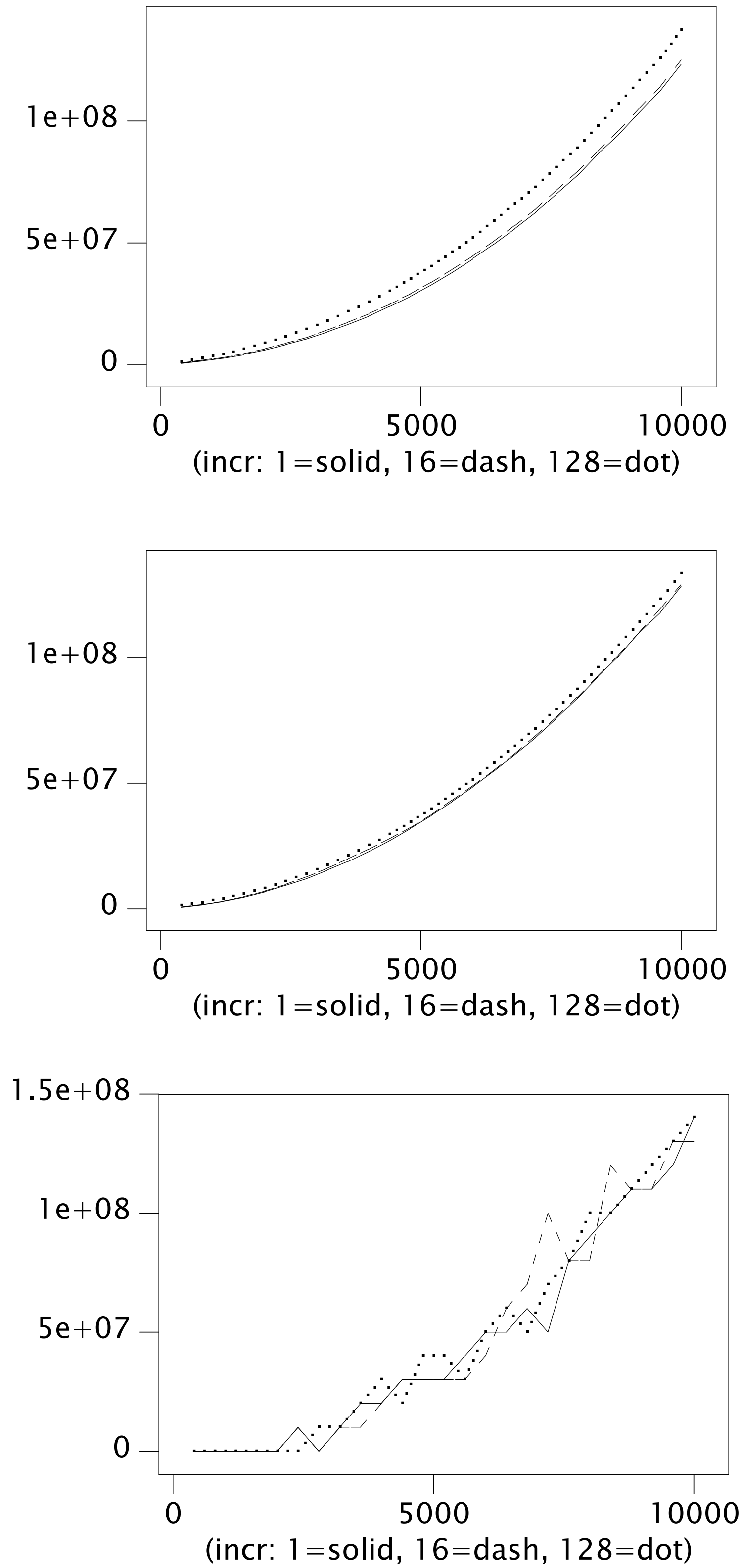

Figure 3 Time (ns) for the C program inserting 4-byte elements into `Arry` as a function of the number of elements for growing increments of 1, 16, and 128: Plan 9 (top), Mac OS X (middle), and Linux (bottom).

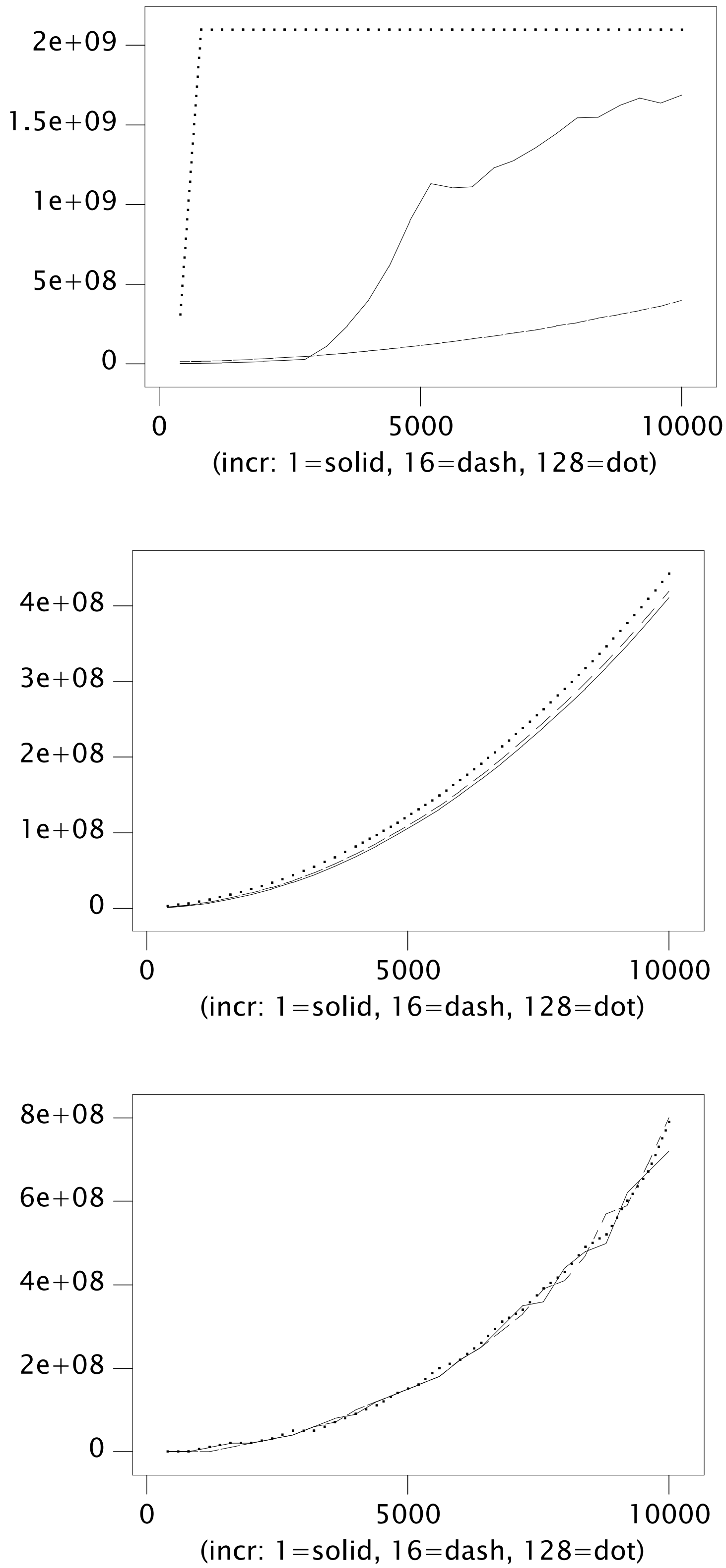

Figure 4 Time (ns) for the C program inserting 64-byte elements into `Arry` as a function of the number of elements for growing increments of 1, 16, and 128: Plan 9 (top), Mac OS X (middle), and Linux (bottom).

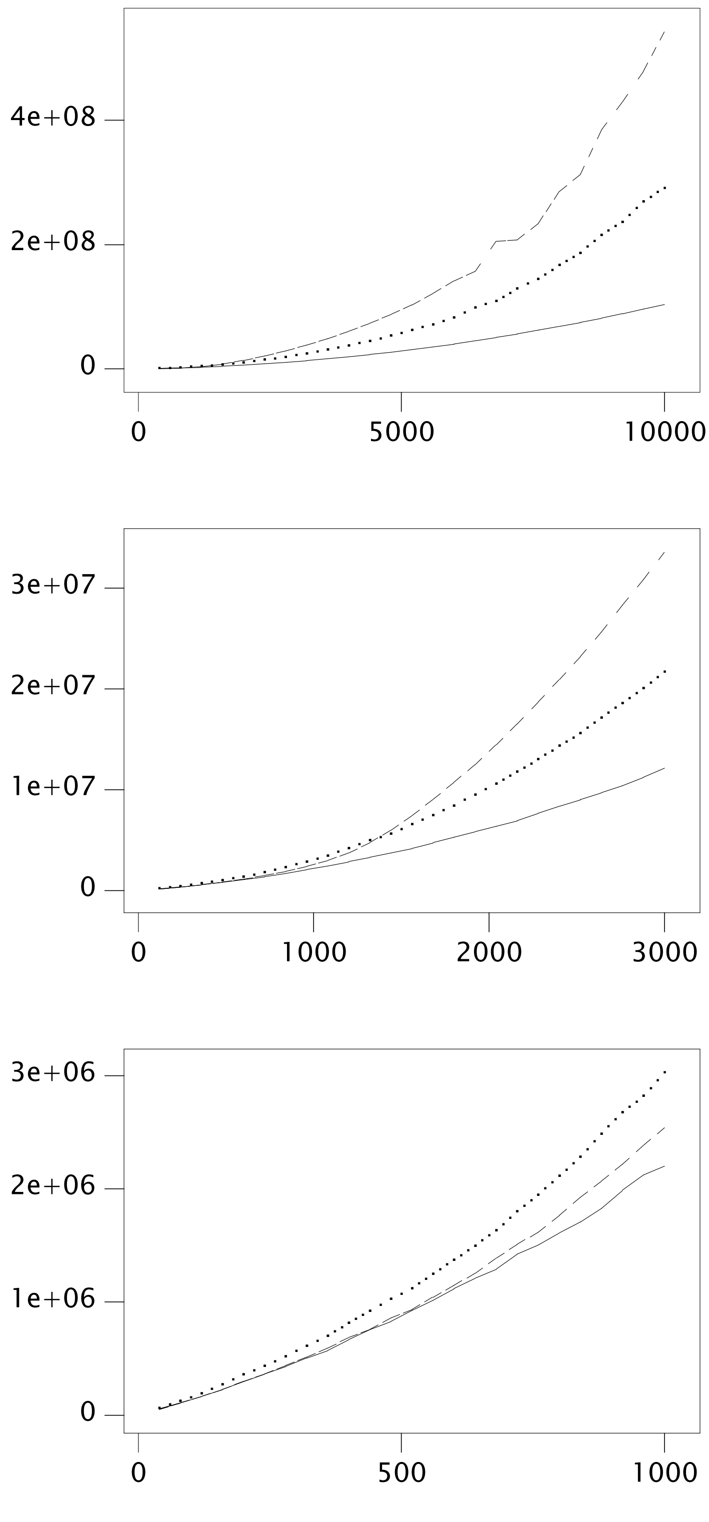

Figure 5 Time (ns) for inserting 4-byte elements in ascending order as a function of the number of elements using C in Plan 9; for `Arry` (solid line), `List` (dashed line), and `Ptrs` (dotted line).

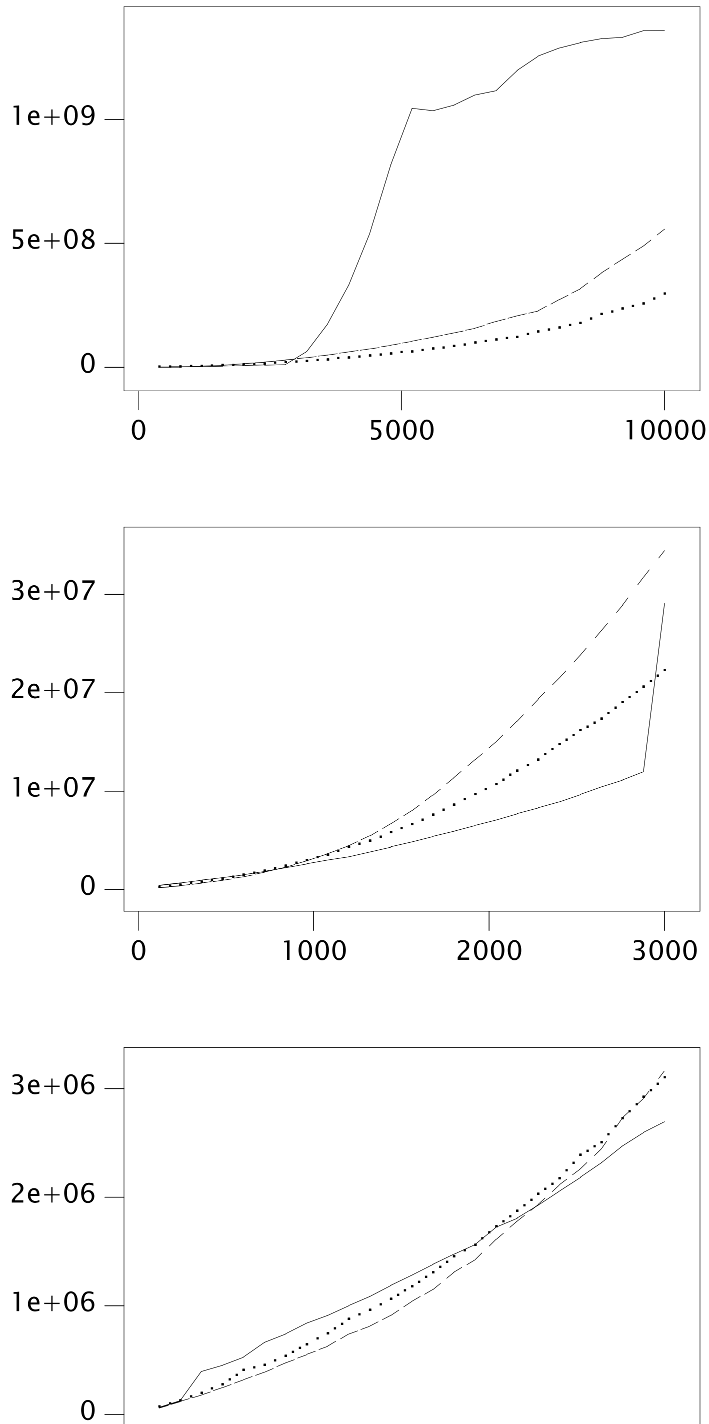

Figure 6 Time (ns) for inserting 64-byte elements in ascending order as a function of the number of elements using C in Plan 9; for `Arry` (solid line), `List` (dashed line), and `Ptrs` (dotted line).

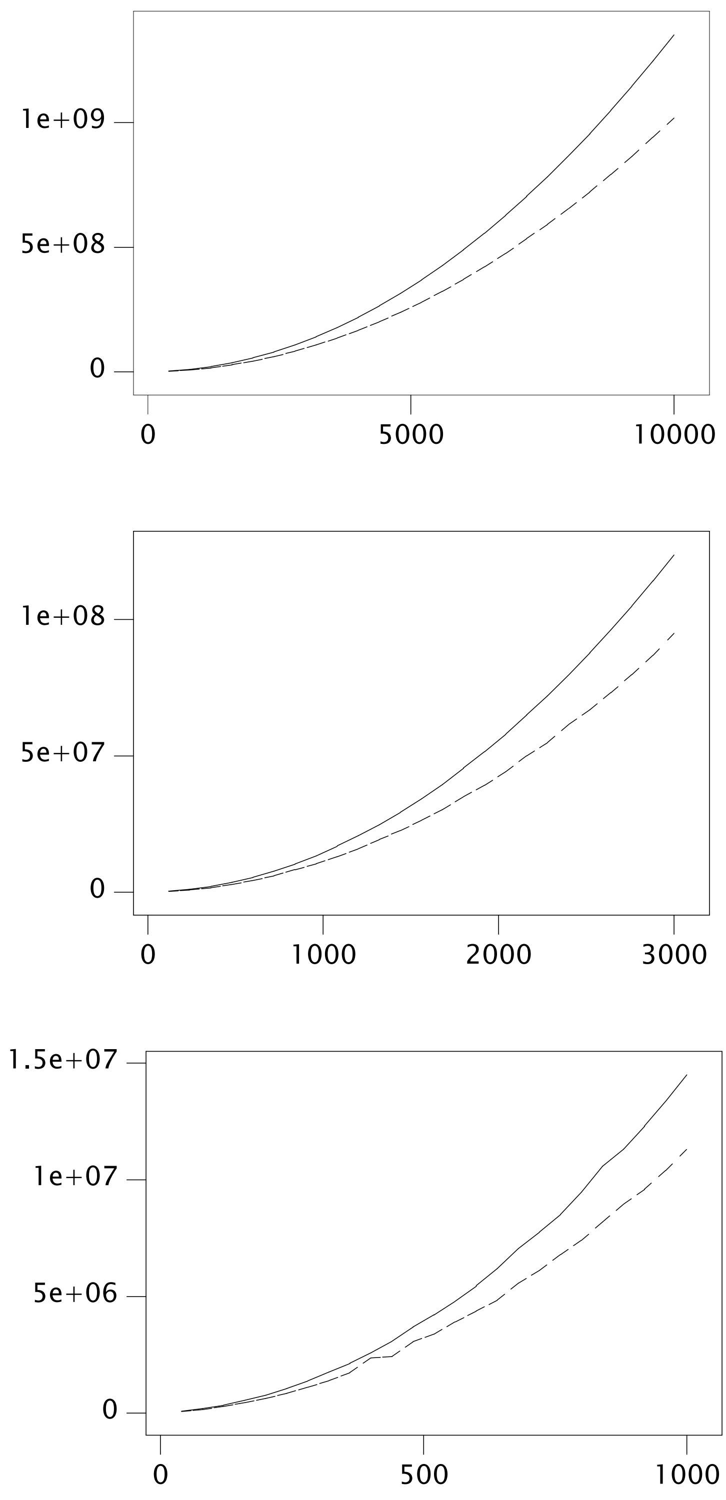

Figure 7 Time (ns) for inserting 4-byte elements in ascending order as a function of the number of elements; for C++ STL vector (solid lines) and list (dashed lines), running on Mac OS X.

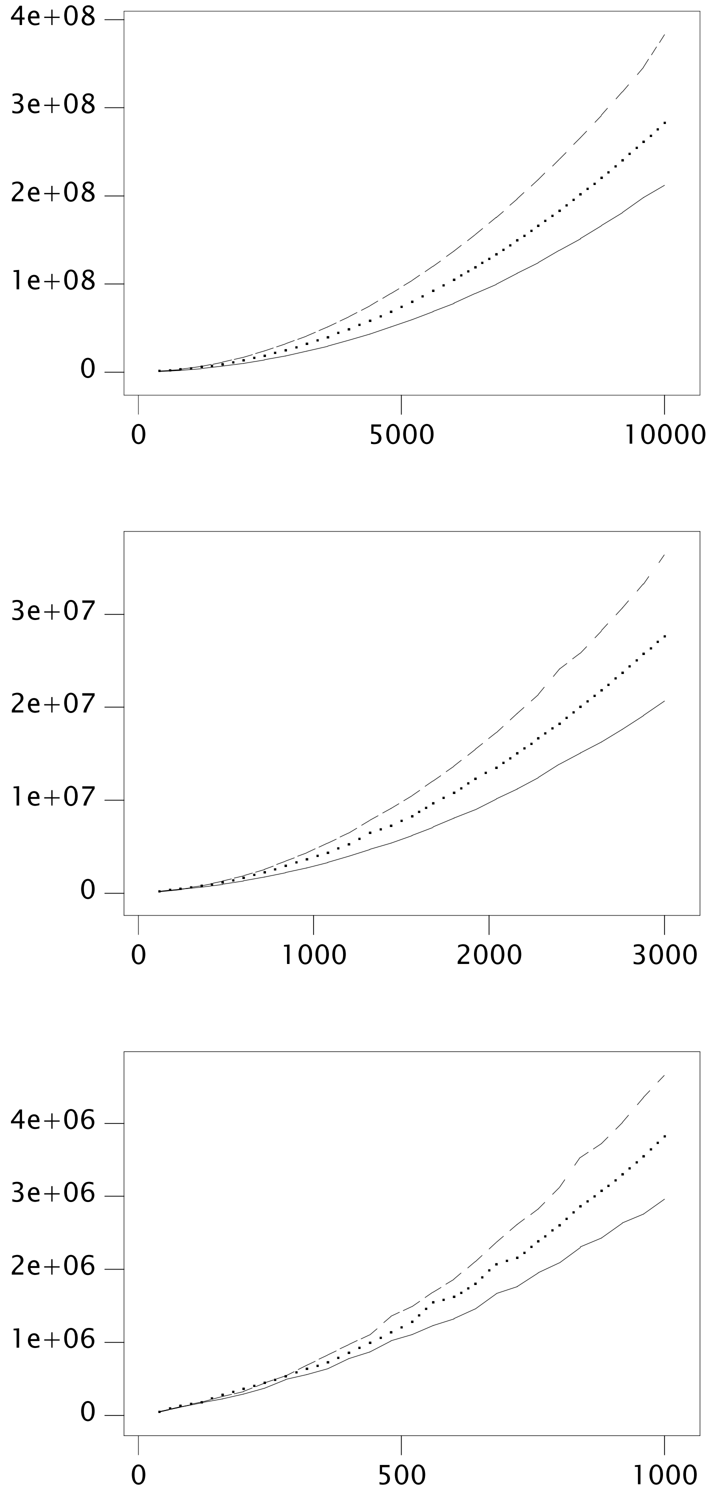

Figure 8 Time (ns) for inserting 4-byte elements in ascending order as a function of the number of elements using C in Mac OS X; for `Arry` (solid line), `List` (dashed line), and `Ptrs` (dotted line).

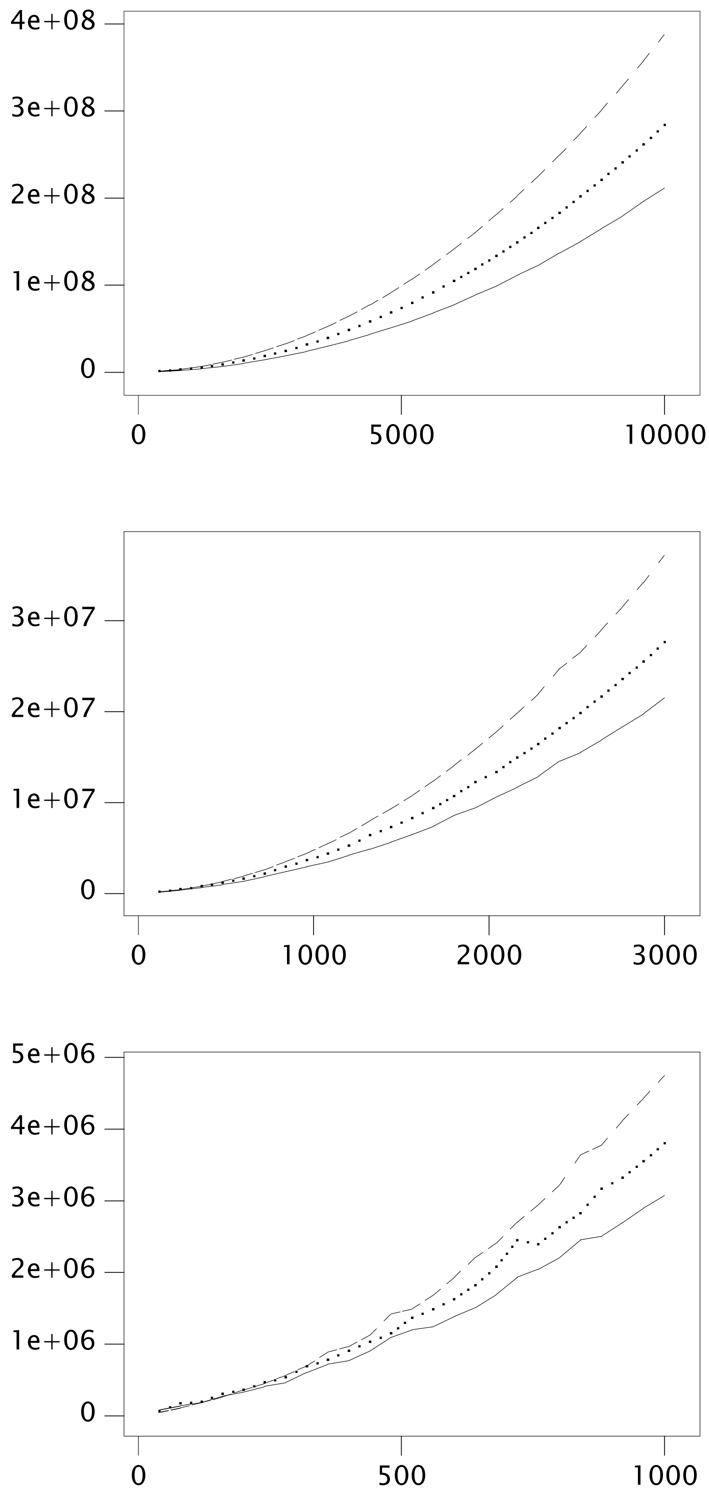

Figure 9 Time (ns) for inserting 64-byte elements in ascending order as a function of the number of elements using C in Mac OS X; for `Arry` (solid line), `List` (dashed line), and `Ptrs` (dotted line).

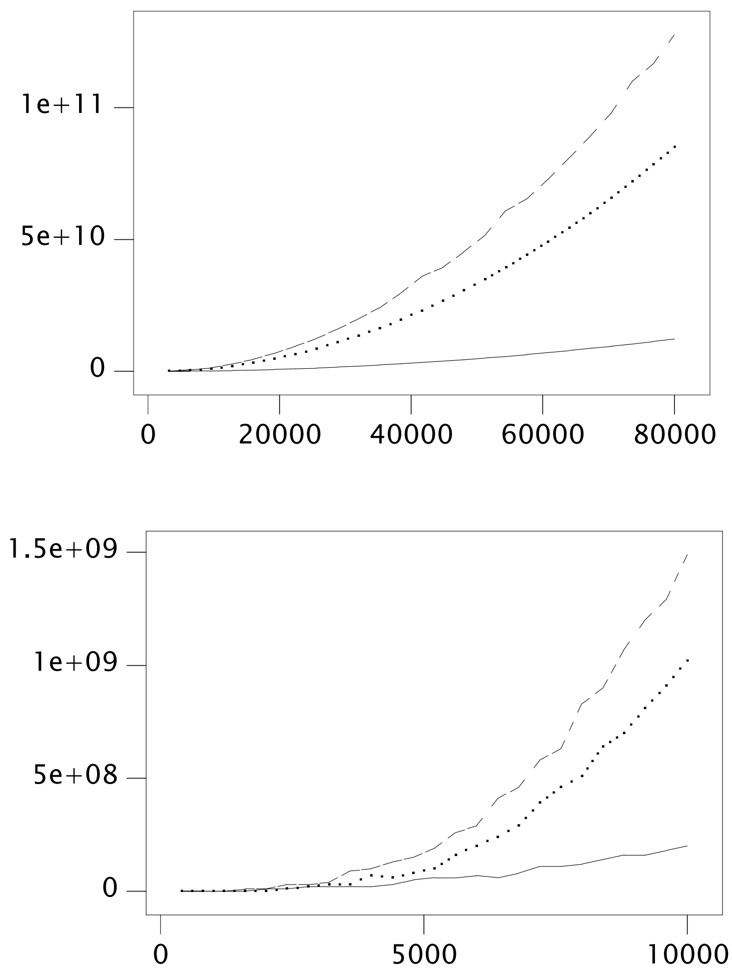

Figure 10 Time (ns) for inserting 4-byte elements in ascending order as a function of the number of elements using C in Linux; for `Arry` (solid line), `List` (dashed line), and `Ptrs` (dotted line).

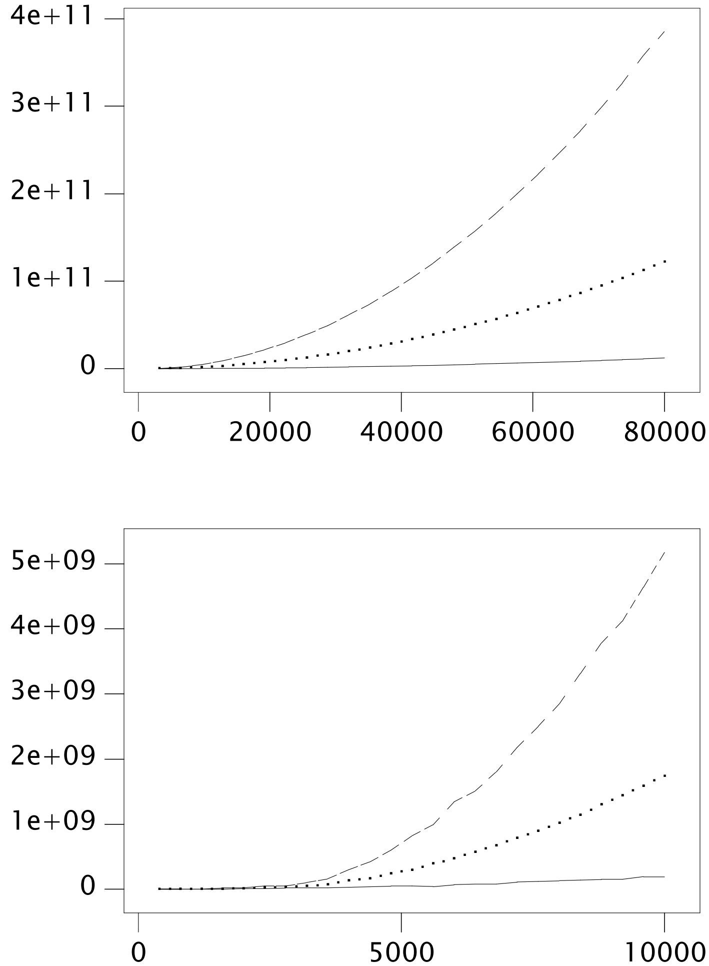

Figure 11 Time (ns) for inserting 64-byte elements in ascending order as a function of the number of elements using C in Linux; for `Arry` (solid line), `List` (dashed line), and `Ptrs` (dotted line).

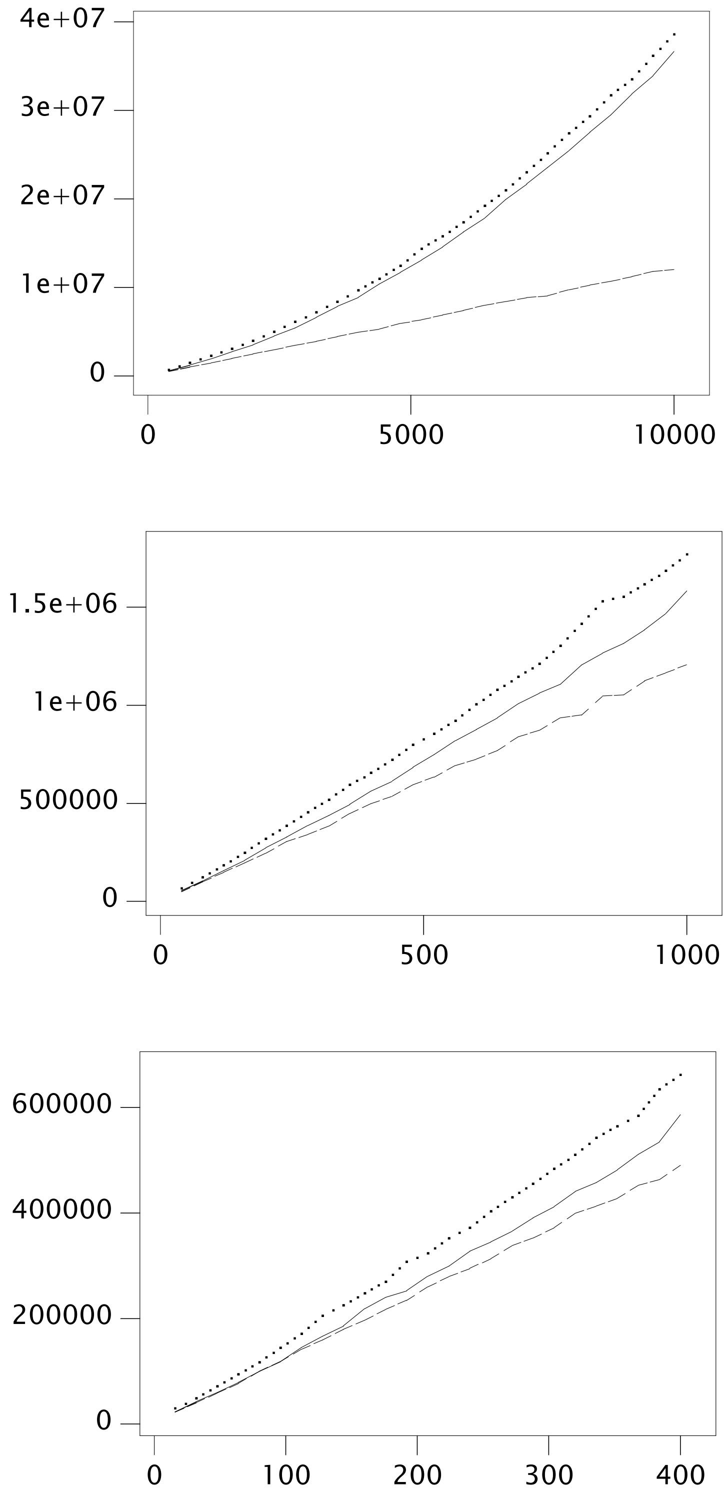

Figure 12 Time (ns) for inserting 4-byte elements in descending order as a function of the number of elements using C in Plan 9; for `Arry` (solid line), `List` (dashed line), and `Ptrs` (dotted line).

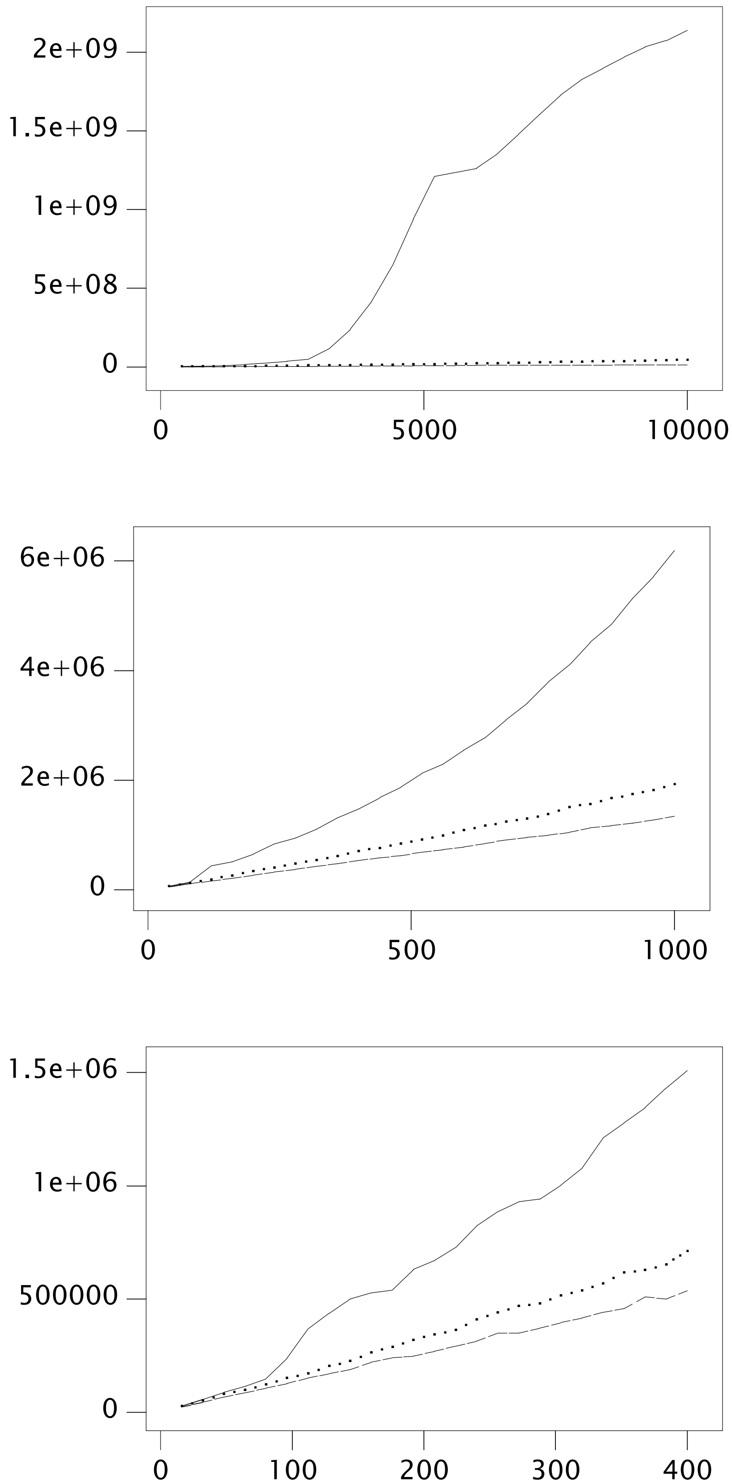

Figure 13 Time (ns) for inserting 64-byte elements in descending order as a function of the number of elements using C in Plan 9; for `Arry` (solid line), `List` (dashed line), and `Ptrs` (dotted line).

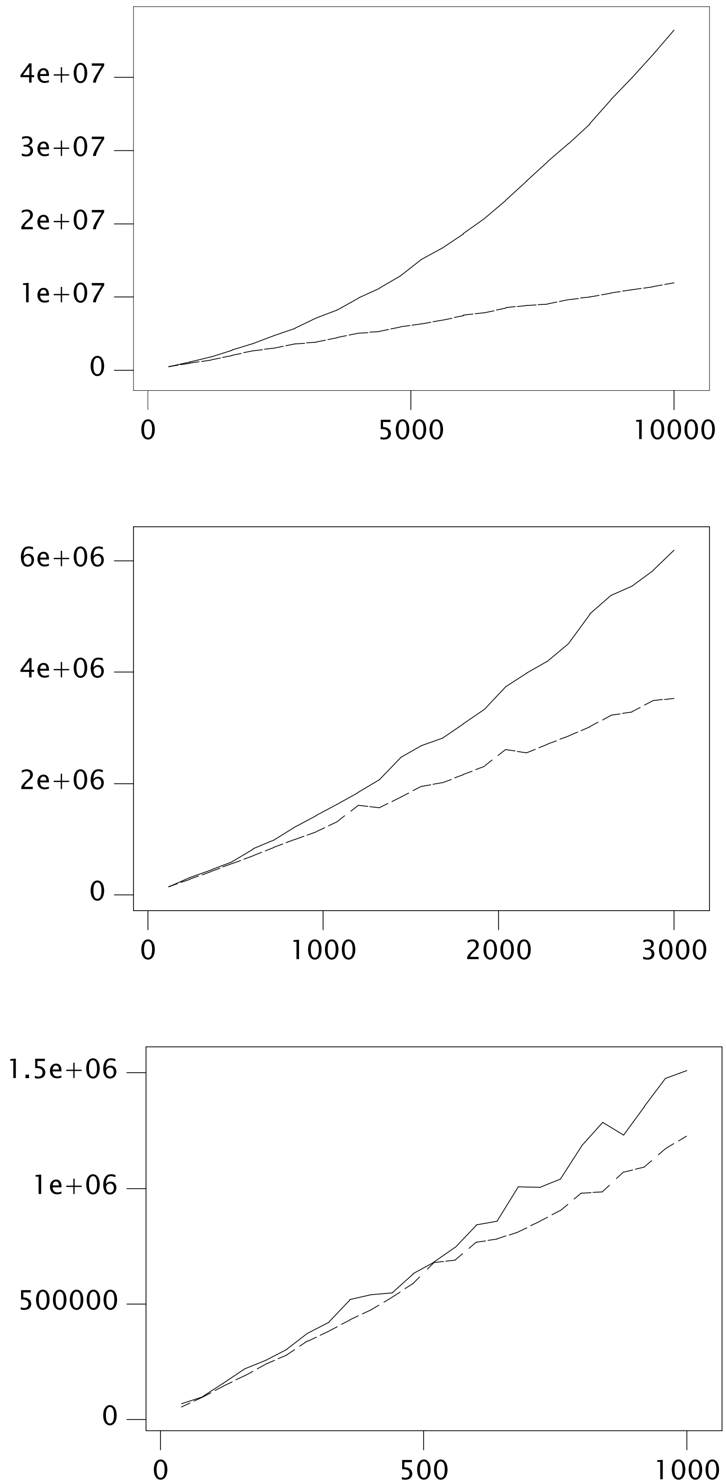

Figure 14 Time (ns) for inserting 4-byte elements in descending order as a function of the number of elements using C++ in Mac OS X; for C++ STL vector and list.

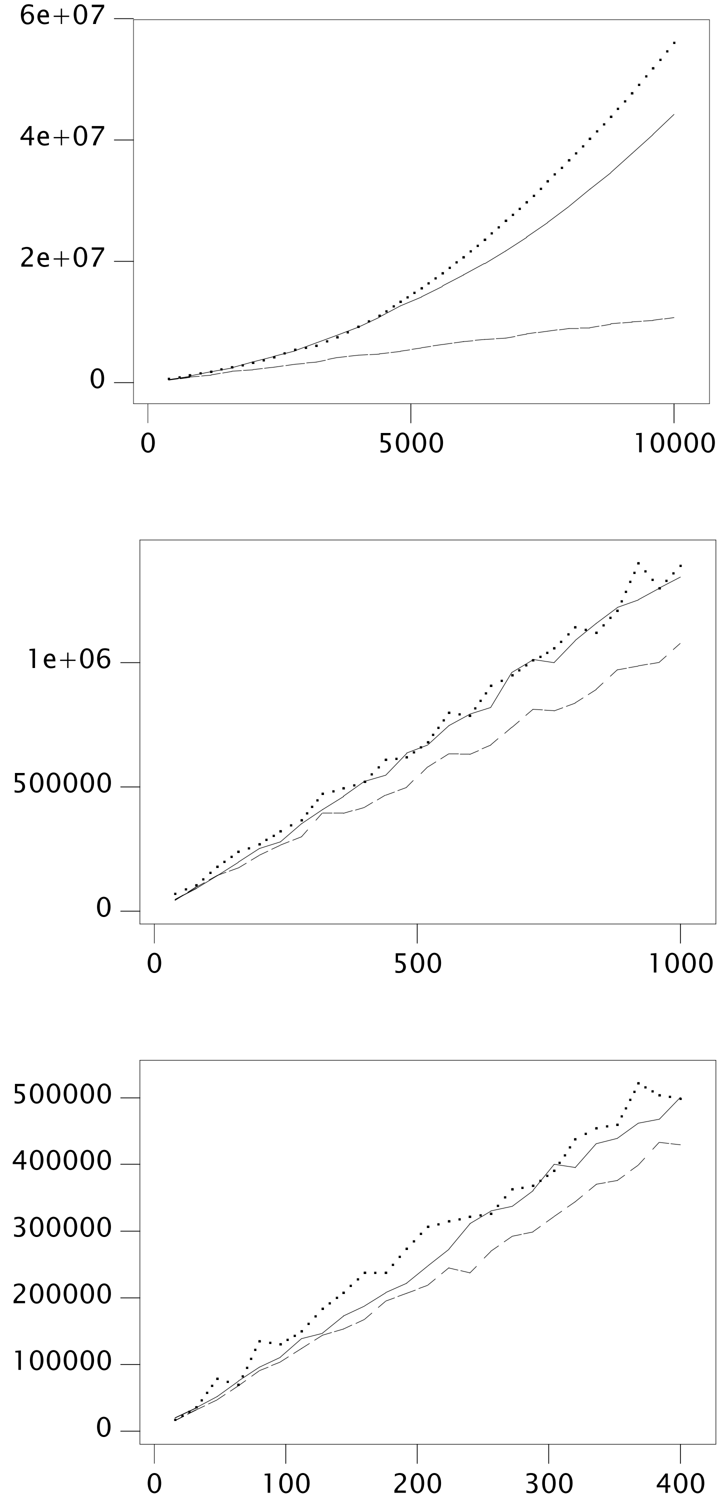

Figure 15 Time (ns) for inserting 4-byte elements in descending order as a function of the number of elements using C in Mac OS X; for `Arry` (solid line), `List` (dashed line), and `Ptrs` (dotted line).

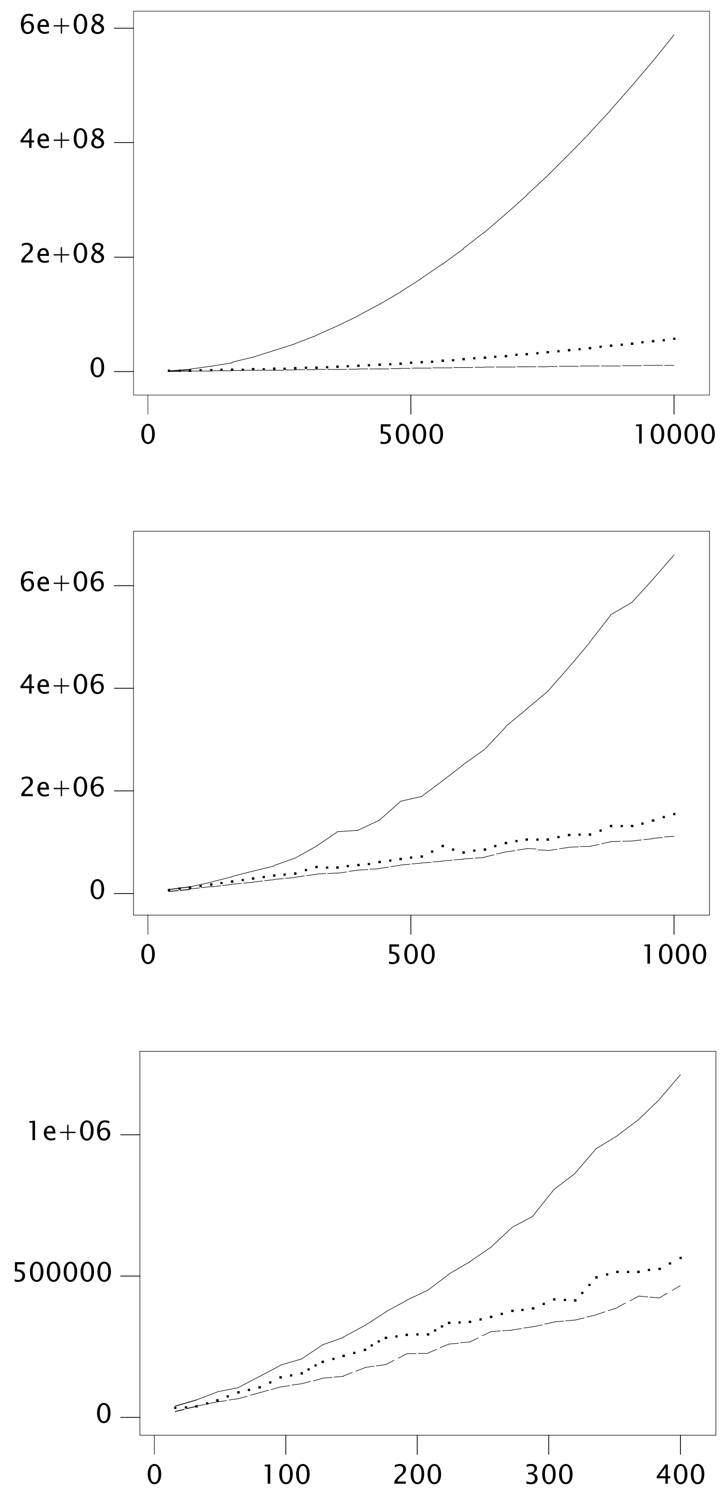

Figure 16 Time (ns) for inserting 64-byte elements in descending order as a function of the number of elements using C in Mac OS X; for `Arry` (solid line), `List` (dashed line), and `Ptrs` (dotted line).

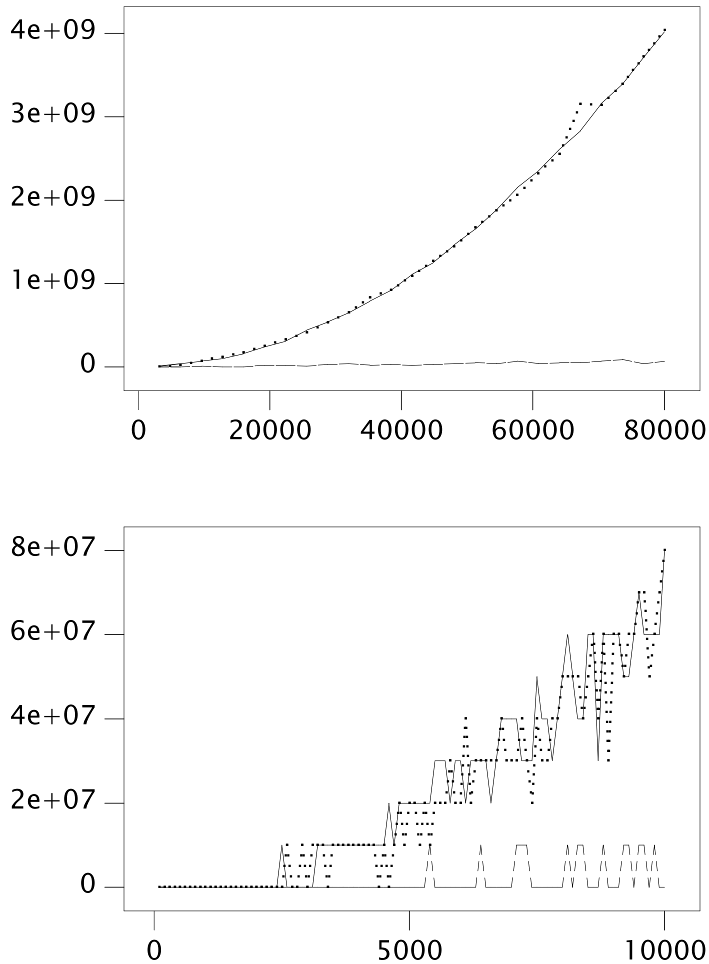

Figure 17 Time (ns) for inserting 4-byte elements in descending order as a function of the number of elements using C in Linux; for `Arry` (solid line), `List` (dashed line), and `Ptrs` (dotted line).

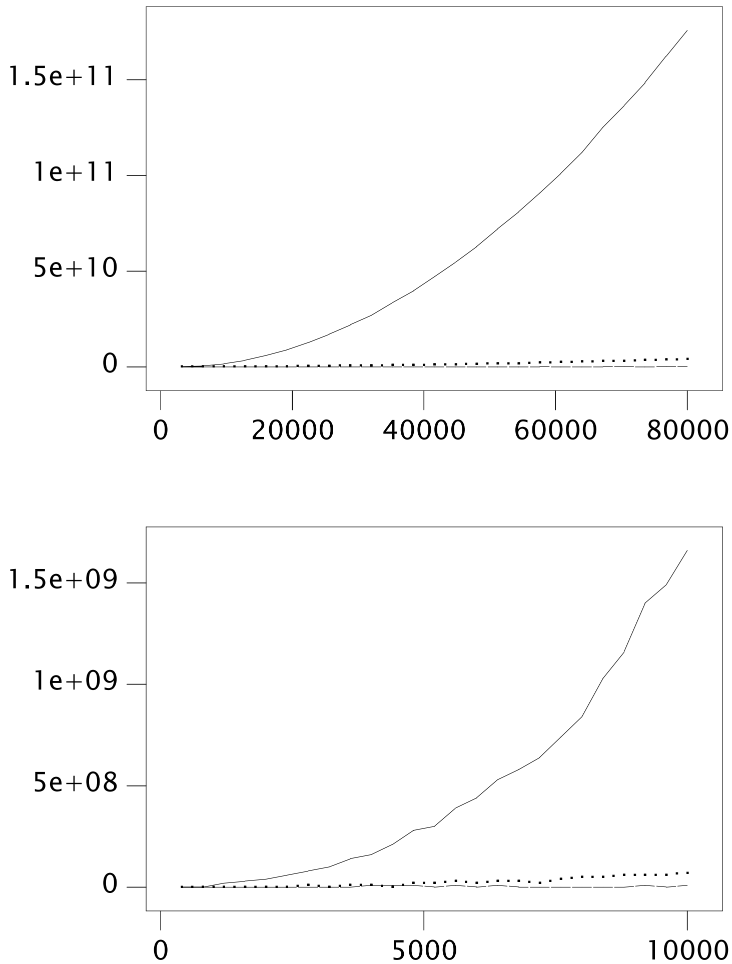

Figure 18 Time (ns) for inserting 64-byte elements in descending order as a function of the number of elements using C in Linux; for `Arry` (solid line), `List` (dashed line), and `Ptrs` (dotted line).

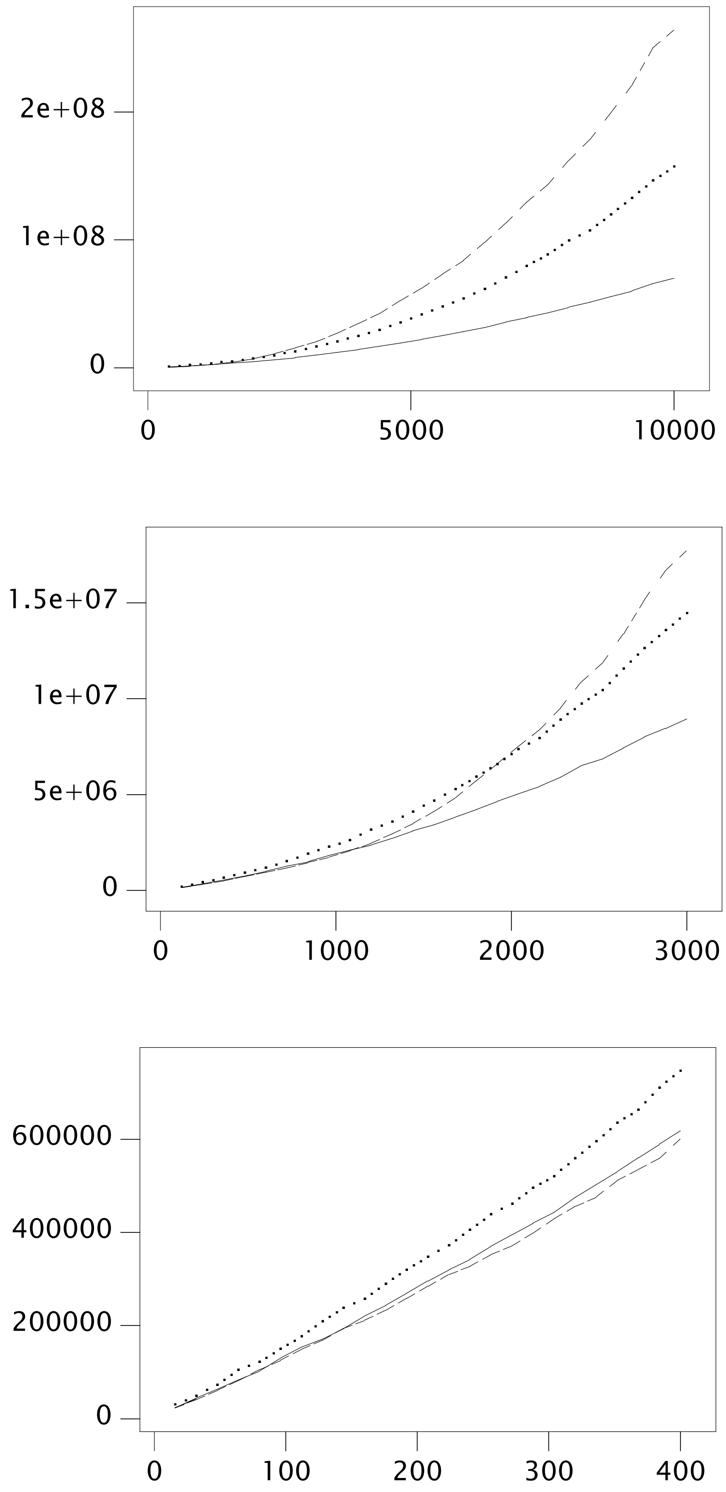

Figure 19 Time (ns) for inserting 4-byte elements in random order as a function of the number of elements using C in Plan 9; for `Arry` (solid line), `List` (dashed line), and `Ptrs` (dotted line).

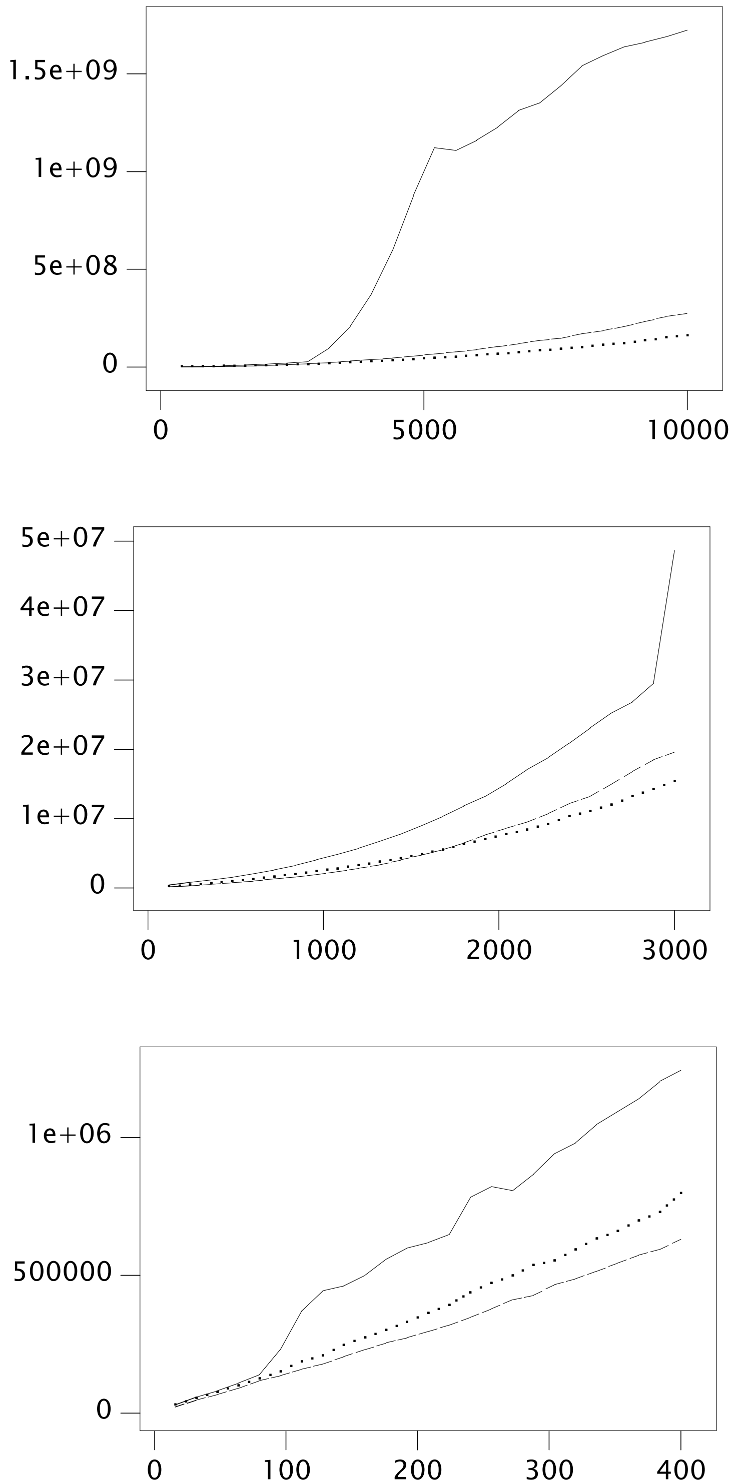

Figure 20 Time (ns) for inserting 64-byte elements in random order as a function of the number of elements using C in Plan 9; for `Arry` (solid line), `List` (dashed line), and `Ptrs` (dotted line).

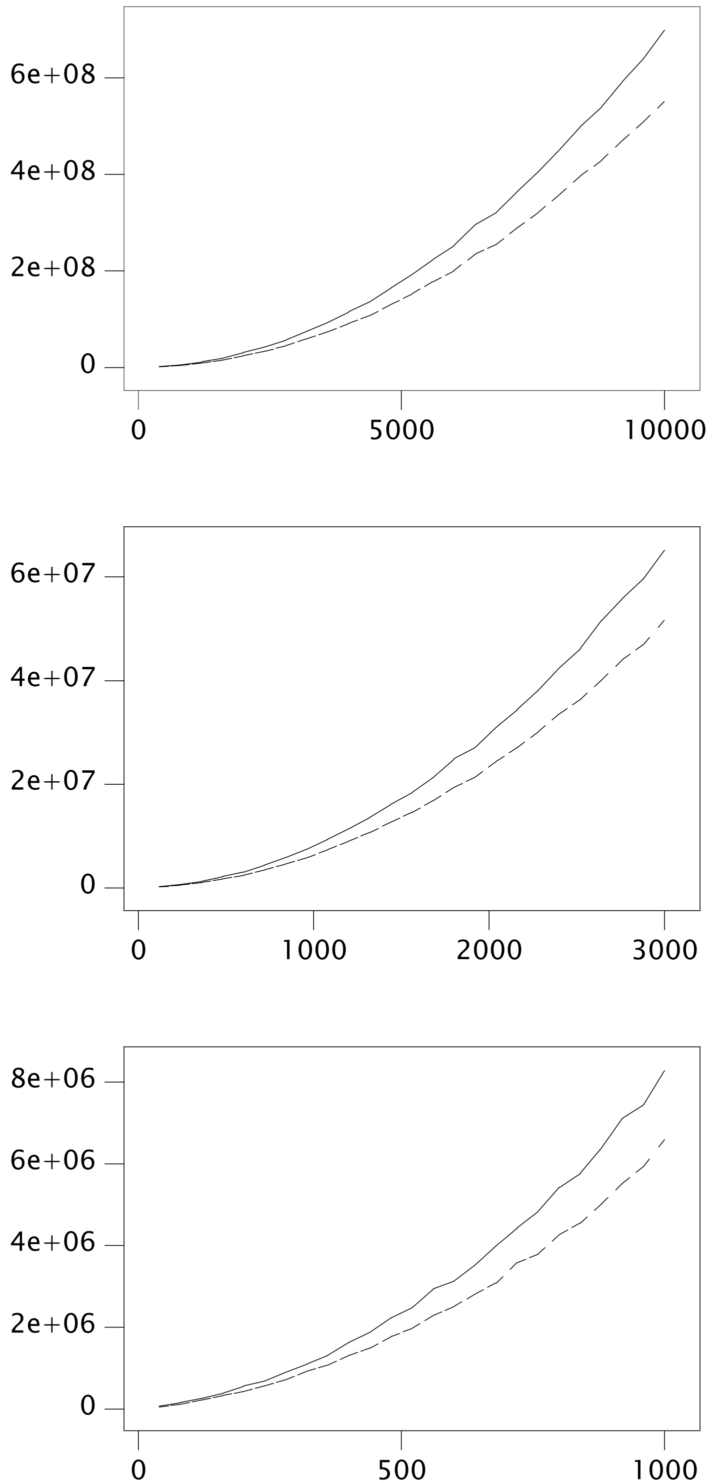

Figure 21 Time (ns) for inserting 4-byte elements in random order as a function of the number of elements using C++ in Mac OS X; for C++ STL vector and list.

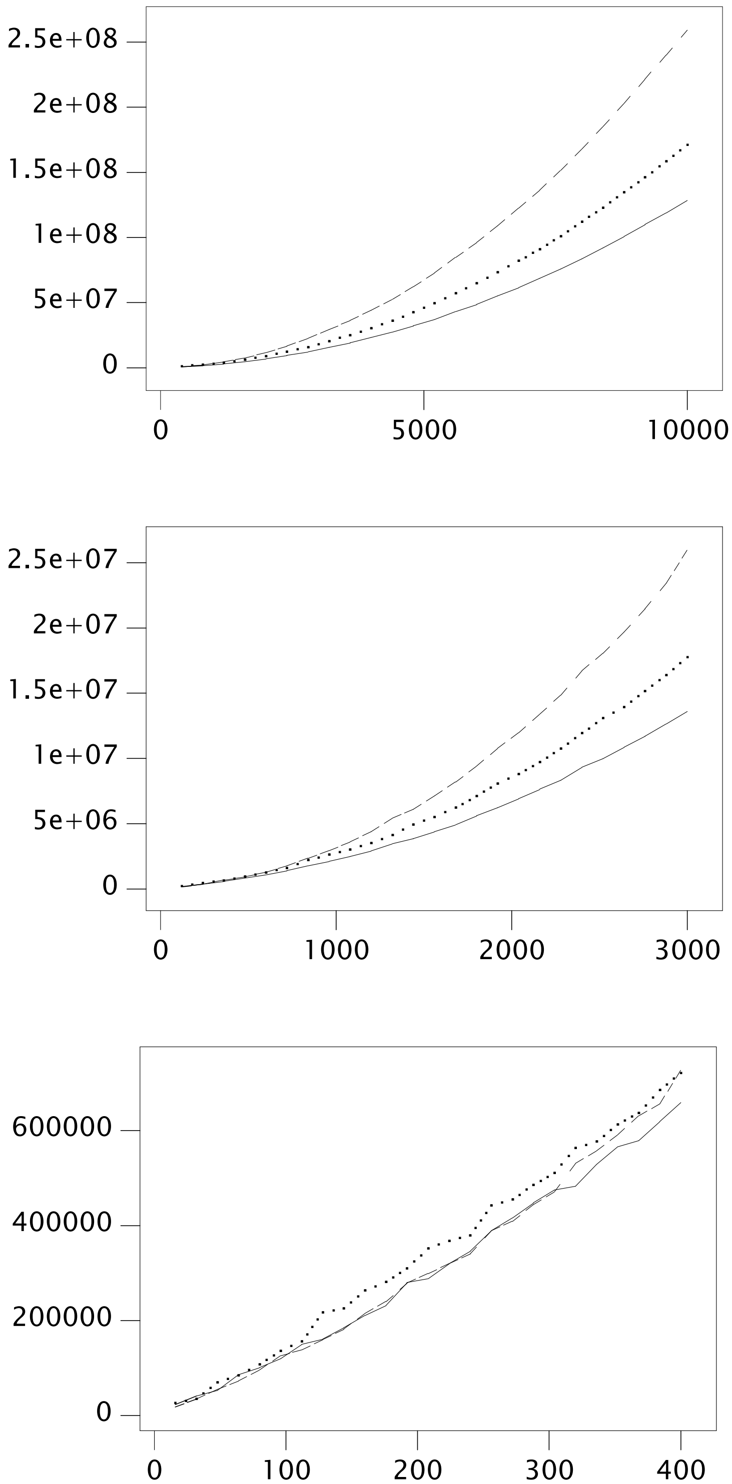

Figure 22 Time (ns) for inserting 4-byte elements in random order as a function of the number of elements using C in Mac OS X; for `Arry` (solid line), `List` (dashed line), and `Ptrs` (dotted line).

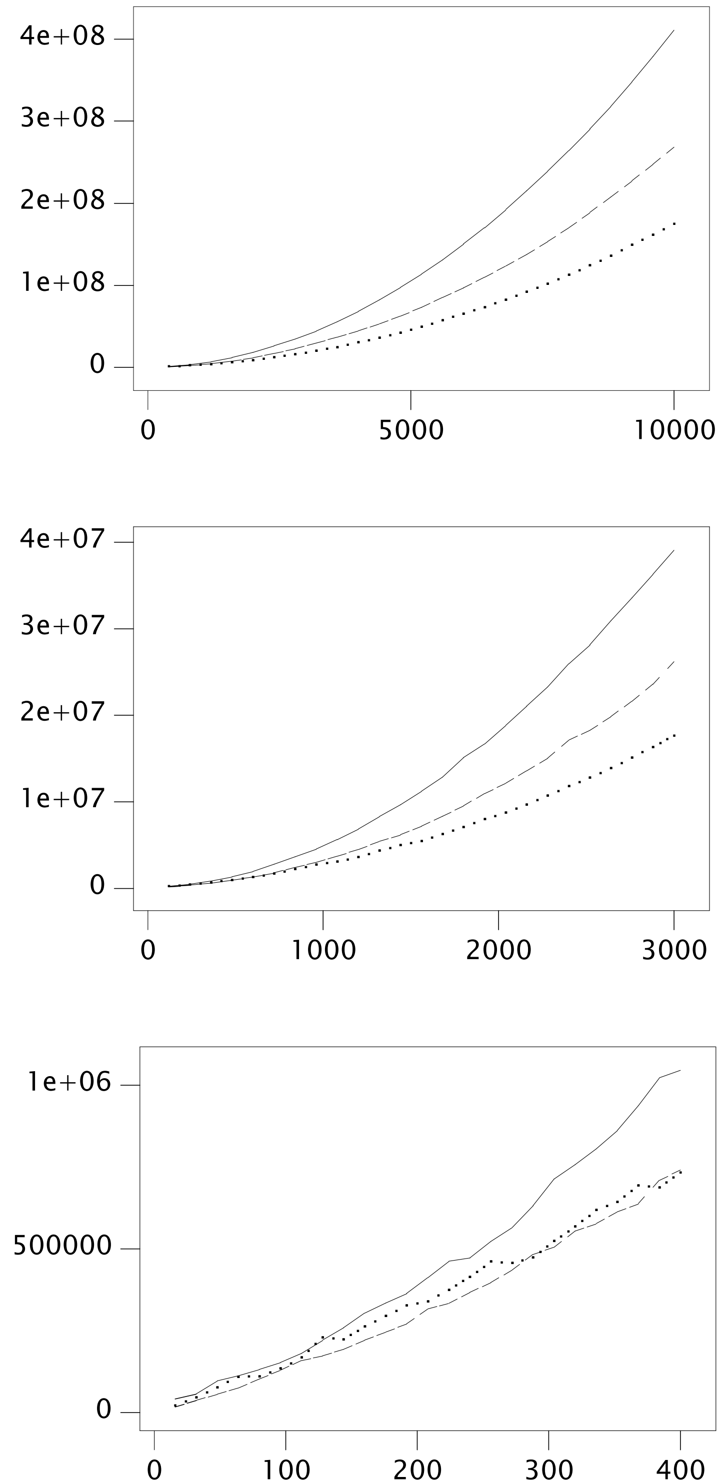

Figure 23 Time (ns) for inserting 64-byte elements in random order as a function of the number of elements using C in Mac OS X; for `Arry` (solid line), `List` (dashed line), and `Ptrs` (dotted line).

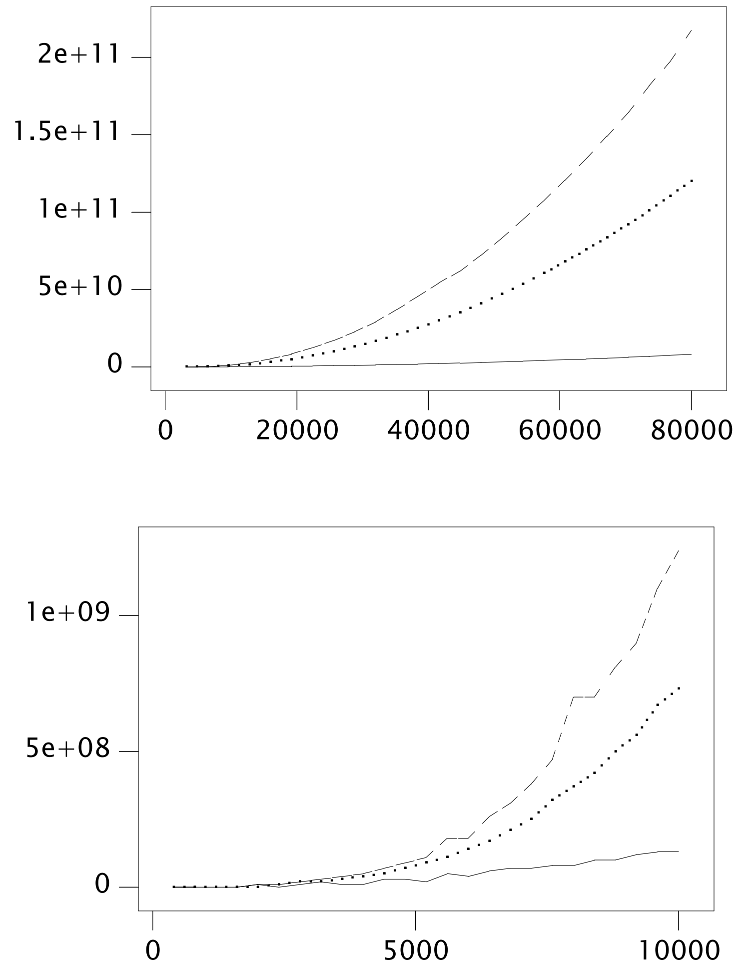

Figure 24 Time (ns) for inserting 4-byte elements in random order as a function of the number of elements using C in Linux; for `Arry` (solid line), `List` (dashed line), and `Ptrs` (dotted line).

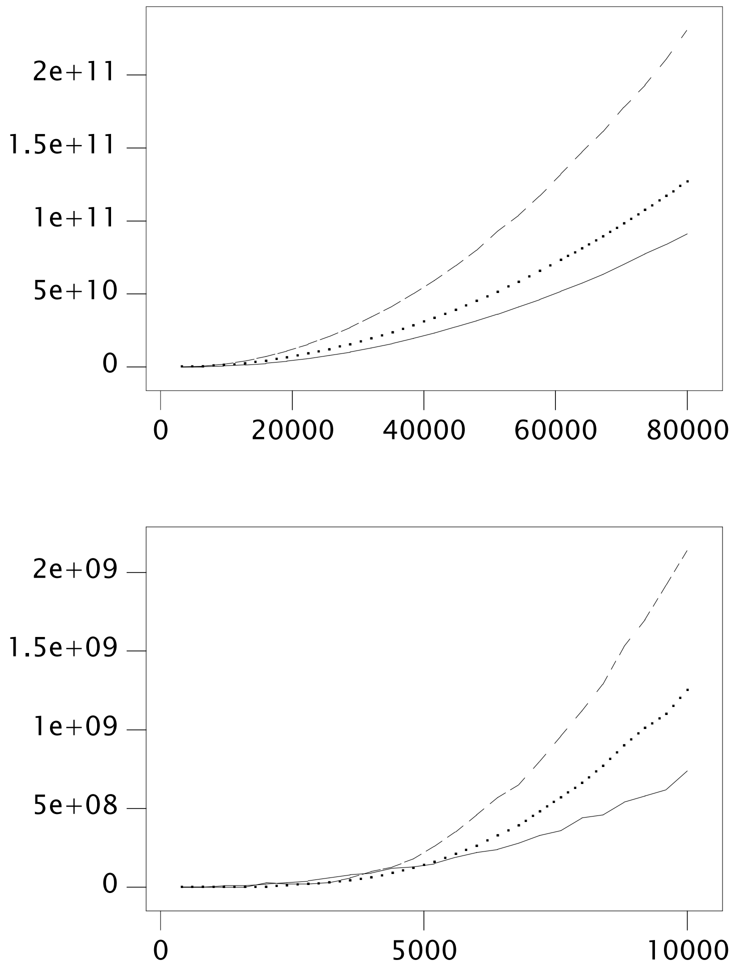

Figure 25 Time (ns) for inserting 64-byte elements in random order as a function of the number of elements using C in Linux; for `Arry` (solid line), `List` (dashed line), and `Ptrs` (dotted line).

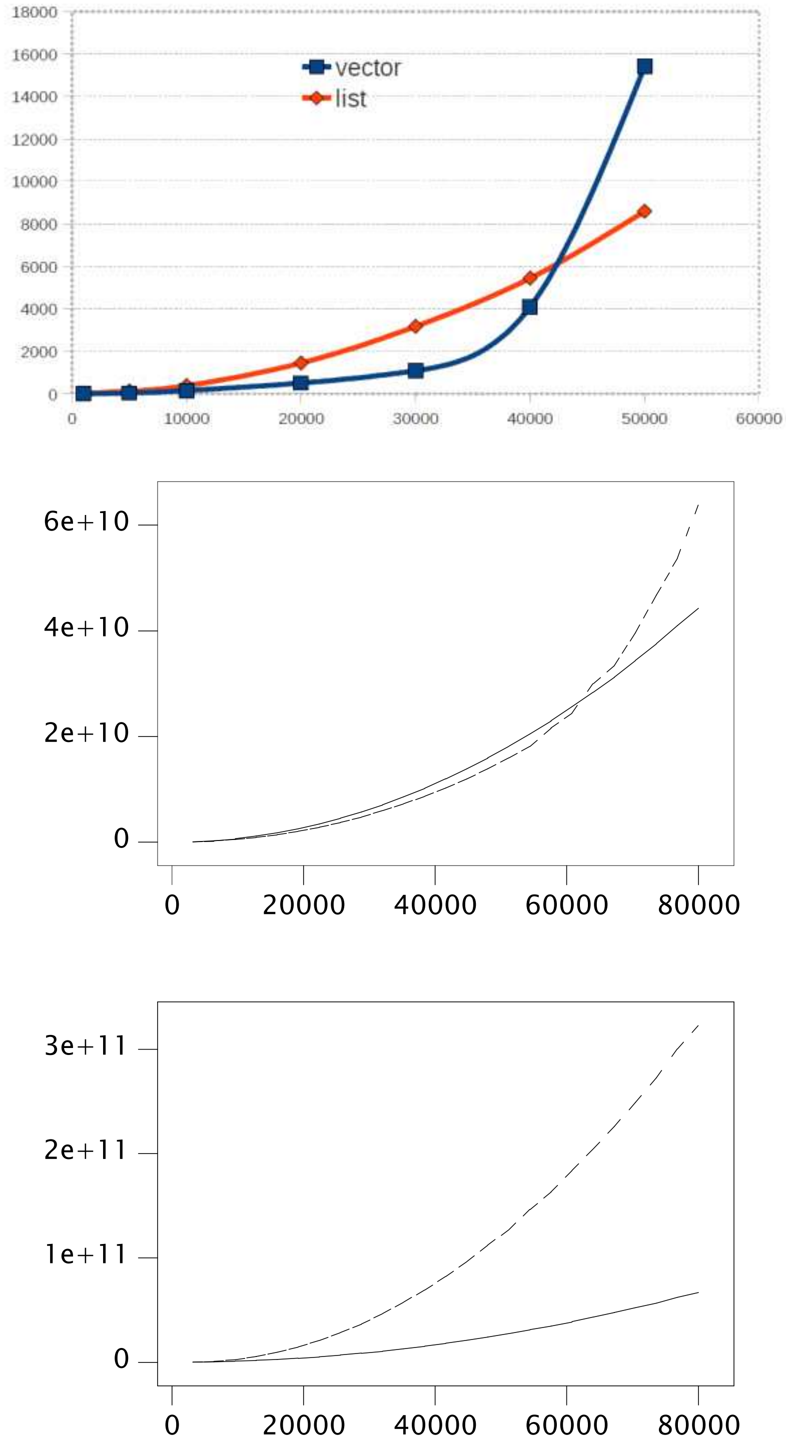

Figure 26 Top: Stroustrup's results. Middle: Time (ns) for inserting 4-byte elements in random order as a function of the number of elements using C++ in Mac OS X; for C++ STL vector (solid) and list (dashed). Bottom: The same experiment performed in Linux (older machine).

Appendix A: Plan 9 C source code

listarry.c

```
#include <u.h>
#include <libc.h>

/*
 *   Measure insertion into ordered sequences
 */

enum
{
     Incr = 16,
     Num = 1000,
     I2LN = 16,

     Fwd = 0,
     Bck = 1,
     Rnd = -1,

     Tarry = 0,
     Tlist,
     Tptrs,
};

typedef struct Arry Arry;
typedef struct Ptrs Ptrs;
typedef struct Node Node;
typedef struct El El;

struct El
{
     int n;          /* element value */
     int dummy[];
};

struct Arry
{
     int nels; /* number of elements used */
     int naels;     /* number of elements allocated */
     El *els;  /* array of elements */
};

struct Node
{
     Node *next;    /* element in list */
     int n;          /* element value */
     int dummy[];
};

struct Ptrs
{
     int nels; /* number of elements used */
     int naels;     /* number of elements allocated */
     El **els; /* array of elements */
};
```

```
#pragma      varargck    type  "A"   Arry*
#pragma      varargck    type  "L"   Node**
#pragma      varargck    type  "P"   Ptrs*

static int incr = Incr;       /* in array realloc */
static int elsz = sizeof(int);     /* number of bytes in element */
static int otherallocs;       /* do mallocs to pollute space */

static void
usage(void)
{
     fprint(2, "usage: %s [-malpfbrvw] [-e nwords] [-i incr] [-n num]\n", argv0);
     exits("usage");
}

static void*
anew(void)
{
     return mallocz(sizeof(Arry), 1);
}

static void*
lnew(void)
{
     return mallocz(sizeof(Node**), 1);
}

static void*
pnew(void)
{
     return mallocz(sizeof(Ptrs), 1);
}

static int
ains(void *x, int el)
{
     int i;
     Arry *a = x;

     if((a->naels%incr) == 0){
          a->naels += incr;
          a->els = realloc(a->els, a->naels*elsz);
          if(a->els == nil)
               return -1;
     }
     for(i = 0; i < a->nels && a->els[i].n < el; i++)
          ;
     if(i < a->nels)
          memmove(&a->els[i+1], &a->els[i], elsz*(a->nels-i));
     a->els[i].n = el;
     a->nels++;
     return 0;
}

static int
lins(void *x, int el)
{
     Node **l, *n;
```

```
	l = x;
	for(; (n = *l) != nil && n->n < el; l = &n->next)
		;
	n = malloc(sizeof(Node*)+elsz);
	if(n == nil)
		return -1;
	n->n = el;
	n->next = *l;
	*l = n;
	return 0;
}

static int
pins(void *x, int el)
{
	int i;
	Ptrs *a = x;

	if((a->naels%incr) == 0){
		a->naels += incr;
		a->els = realloc(a->els, a->naels*sizeof(El*));
		if(a->els == nil)
			return -1;
	}
	for(i = 0; i < a->nels && a->els[i]->n < el; i++)
		;
	if(i < a->nels)
		memmove(&a->els[i+1], &a->els[i], sizeof(El*)*(a->nels-i));
	a->els[i] = malloc(elsz);
	if(a->els[i] == nil)
		return -1;
	a->els[i]->n = el;
	a->nels++;
	return 0;
}

static int
Afmt(Fmt *f)
{
	Arry *a;
	int i;

	a = va_arg(f->args, Arry*);
	if(a == nil)
		return fmtprint(f, "<nilarry>");
	fmtprint(f, "[");
	for(i = 0; i < a->nels; i++){
		if(i > 0){
			fmtprint(f, ", ");
			if(i < a->nels-1 && (i%I2LN) == 0)
				fmtprint(f, "\n");
		}
		fmtprint(f, "%d", a->els[i].n);
	}
	return fmtprint(f, "]");
}
```

```
static int
Lfmt(Fmt *f)
{
	Node **x, *l;
	int i;

	x = va_arg(f->args, Node**);
	l = *x;
	fmtprint(f, "(");
	for(i = 0; l != nil; l = l->next){
		if(i++ > 0){
			fmtprint(f, ", ");
			if((i%I2LN) == 0)
				fmtprint(f, "\n");
		}
		fmtprint(f, "%d", l->n);
	}
	return fmtprint(f, ")");
}

static int
Pfmt(Fmt *f)
{
	Ptrs *a;
	int i;

	a = va_arg(f->args, Ptrs*);
	fmtprint(f, "[");
	for(i = 0; i < a->nels; i++){
		if(i > 0){
			fmtprint(f, ", ");
			if(i < a->nels-1 && (i%I2LN) == 0)
				fmtprint(f, "\n");
		}
		fmtprint(f, "%d", a->els[i]->n);
	}
	return fmtprint(f, "]");
}

static vlong
test(int (*ins)(void*, int), void *a, int n, int dir)
{
	vlong t0, t1, tot;
	int i, r;
	char *s;
```

```
	tot = 0LL;
	for(i = 0; i < n; i++){
		r = nrand(n);
		t0 = nsec();
		switch(dir){
		case Fwd:
			r = ins(a, i);
			break;
		case Bck:
			r = ins(a, n-i);
			break;
		default:
			r = ins(a, r);
		}
		if(r < 0){
			print("no memory");
			exits(nil);
		}
		t1 = nsec();
		tot += (t1 - t0);
		if(otherallocs){
			s = malloc(64);
			USED(s);
		}
	}
	return tot;
}

static void
afree(void *x)
{
	Arry *a;

	a = x;
	free(a->els);
	free(a);
}

static void
lfree(void *x)
{
	Node **a, *l, *n;

	a = x;
	l = *a;
	while(l != nil){
		n = l;
		l = l->next;
		free(n);
	}
	free(a);
}

static void
pfree(void *x)
{
	Ptrs *a;
	int i;
```

```
	a = x;
	for(i = 0; i < a->nels; i++)
		free(a->els[i]);
	free(a->els);
	free(a);
}

static void
adump(void *x)
{
	print("%A\n", x);
}

static void
ldump(void *x)
{
	print("%L\n", x);
}

static void
pdump(void *x)
{
	print("%P\n", x);
}


void
main(int argc, char*argv[])
{
	void *x;
	int dir, adt, n, vflag, wflag;
	vlong ∂;
	struct{
		void*(*new)(void);
		int(*ins)(void*,int);
		void(*free)(void*);
		void(*dump)(void*);
	} fns[] = {
		[Tarry]	{anew, ains, afree, adump},
		[Tlist]	{lnew, lins, lfree, ldump},
		[Tptrs]	{pnew, pins, pfree, pdump},
	};
```

```
	dir = Fwd;
	adt = Tarry;
	n = Num;
	∂ = 0;
	vflag = wflag = 0;
	ARGBEGIN{
	case 'a':
		break;
	case 'l':
		adt = Tlist;
		break;
	case 'p':
		adt = Tptrs;
		break;
	case 'f':
		break;
	case 'b':
		dir = Bck;
		break;
	case 'r':
		dir = Rnd;
		break;
	case 'i':
		incr = atoi(EARGF(usage()));
		if(incr < 1)
			sysfatal("incr < 1");
		break;
	case 'n':
		n = atoi(EARGF(usage()));
		if(n < 1)
			sysfatal("n < 1");
		break;
	case 'e':
		elsz = atoi(EARGF(usage()));
		if(elsz < sizeof(int))
			sysfatal("elsz < 1");
		if(elsz%sizeof(int))
			sysfatal("elsz is not a sizeof(int) multiple");
		break;
	case 'm':
		otherallocs = 1;
		break;
	case 'v':
		vflag = 1;
		break;
	case 'w':
		wflag = 1;
		break;
	default:
		usage();
	}ARGEND;
	if(argc != 0)
		usage();
	fmtinstall('A', Afmt);
	fmtinstall('L', Lfmt);
	fmtinstall('P', Pfmt);
	srand(13);
	x = fns[adt].new();
	∂ = test(fns[adt].ins, x, n, dir);
```

```
	if(vflag)
		fns[adt].dump(x);
	print("%lld\n", ∂);
	if(wflag){
		print("waiting...\n");
		sleep(3600 * 1000);
	}
if(0)		fns[adt].free(x);
	exits(nil);
}
```

Appendix B: C++ Source code

listarry.cpp

```
#include <iostream>
#include <vector>
#include <list>

enum
{
     Num = 1000,

     Fwd = 0,
     Bck = 1,
     Rnd = -1,

     Tarry = 0,
     Tlist,
};


using namespace std;

const long long CLK2MS = CLOCKS_PER_SEC / 1000000;
const int N = 500;

/*
 * Definition of ARGBEGIN, ARGEND, ... omitted.
 */

void usage()
{
          cerr << "usage: " << argv0 << " [-alfbr] [-n num]" << endl;
          exit(1);
}

int
main(int argc, char *argv[])
{
     vector<int> arry;
     vector<int>::iterator vi;
     list<int> lst;
     list<int>::iterator li;
     int i, r;
     clock_t t0, t1;
     long long tot;
     void *s;
     int adt, dir, n, otherallocs, vflag;
```

```
	dir = Fwd;
	adt = Tarry;
	n = Num;
	otherallocs = vflag = 0;
	ARGBEGIN{
	case 'a':
		break;
	case 'l':
		adt = Tlist;
		break;
	case 'f':
		break;
	case 'b':
		dir = Bck;
		break;
	case 'r':
		dir = Rnd;
		break;

	case 'n':
		n = atoi(EARGF(usage()));
		if(n < 1){
			cerr << "n < 1" << endl;
			return 1;
		}
		break;
	case 'm':
		otherallocs = 1;
		break;
	case 'v':
		vflag = 1;
		break;
	}ARGEND;
	if(argc != 0)
		usage();

	tot = 0;
	srand(13);
	for(i = 0; i < n; i++){
		switch(dir){
		case Fwd:
			r = i;
			break;
		case Bck:
			r = n-i;
			break;
		default:
			r = rand() % n;
		}
```

```
            t0 = clock();
            /* No dispatching here */
            if(adt == Tarry){
                for(vi = arry.begin(); vi != arry.end(); vi++)
                    if(*vi >= r){
                        arry.insert(vi, r);
                        break;
                    }
                if(vi == arry.end())
                    arry.push_back(r);

            }else{
                for(li = lst.begin(); li != lst.end(); li++)
                    if(*li >= r){
                        lst.insert(li, r);
                        break;
                    }
                if(li == lst.end())
                    lst.push_back(r);
            }
            t1 = clock();
            tot += t1 - t0;
            if(otherallocs)
                s = malloc(64);
        }

        if(vflag){
            cout << "{";
            if(adt == Tarry)
                for(vi = arry.begin(); vi != arry.end(); vi++)
                    cout << " " << *vi;
            else
                for(li = lst.begin(); li != lst.end(); li++)
                    cout << " " << *li;
            cout << "}" << endl;
        }
        cout << tot * CLK2MS  << "000" << endl;
        return 0;
}
```