

Functional Unparsing Performance in Objective Caml

T. Kurt Bond

tkb@tkb.mpl.com

ABSTRACT

An extended version of the continuation-passing-style string formatters from "Functional Unparsing" by Olivier Danvy is implemented in Objective Caml and performance is compared with the `sprintf` and `printf` functions of Objective Caml and C.

1. Background

In[DANVY98] Danvy shows how an extensible, type-safe string formatting function equivalent to C's `sprintf` can be conveniently and efficiently programmed using continuation-passing-style (CPS) in ML.

`Cpsio` and `Cpsio_exp` are modules that implement Functional Unparsing string formatting in Objective Caml[OCAML] and extend it with direct output. They implement the equivalents of the `%d`, `%g`, and `%s` control directives of `sprintf` and allow specification of field widths equivalent to the `sprintf` `%"` and `%20` control directives, which respectively encode the width of a field in either the control string or in the argument list. Here is an list of `sprintf` control directives and their `Cpsio` pattern directive equivalents:

<code>printf</code> format	<code>%d</code>	<code>%20d</code>	<code>%"d</code>	<code>%f</code>	<code>%20f</code>	<code>%"f</code>	<code>%s</code>	<code>%20s</code>	<code>%"s</code>
<code>Cpsio</code> equivalent	<code>int</code>	<code>wint 20</code>	<code>intw</code>	<code>flt</code>	<code>wflt 20</code>	<code>fltw</code>	<code>str</code>	<code>wstr 20</code>	<code>strw</code>

Figure 1 is an example in `sprintf` style and Figure 2 is an example in `Cpsio` style.

2. Tests Explained

Modified versions of the examples in Figure 1 and Figure 2 are used to benchmark the modules. Because the `format` function in the `Cpsio` module returns a string, the basic test is to create a string containing the formatted output and output it with a lower-level output function: `print_string` in Objective Caml or `fputs` in C. This basic test exists in two forms: one that formats a multi-line output in one function call and outputs it with one function call, and one that formats the same output a line at a time and outputs it a line at a time. The test programs for C `printf`, Objective Caml `printf`, and `Cpsio_exp` modify the basic test to use `printf` (or its equivalent construct in `Cpsio_exp`) to format and output directly.

The test programs are each run once for each type of test and during that run the test is executed 100000 times.

Processing Type	Tests							
	1	2	3	4	5	6	7	8
one format expression	•		•		•		•	
many format expressions		•		•		•		•
format to string	•	•			•	•		
format to output			•	•			•	•
cache format function					•	•	•	•

```

let test1 () =
  print_string
  (sprintf "%25s %d\n%25s %g\n%25s %s\n\n%25s ]%*d[\n%25s ]%*d[\n\
%25s ]%*g[\n%25s ]%*g[\n%25s ]%*s[\n%25s ]%*s[\n"
  "int:" 10 "float:" 1.234 "string:" "foo"
  "int with width:" 20 24 "int with -width:" (-20) 42
  "float with width:" 20 1.234 "float with -width:" (-20) 567.8
  "string with width:" 20 "Hello"
  "string with -width:" (-20) "Goodbye")

```

Figure 1. printf-style Example

```

let test1 () =
  print_string
  (format
   (wlit "int:"           25 $ lit " " $ int           $ nl $
    wlit "float:"        25 $ lit " " $ flt           $ nl $
    wlit "string:"       25 $ lit " " $ str           $ nml 2 $
    wlit "int with width:" 25 $ lit " ]" $ intw $ lit "[" $ nl $
    wlit "int with -width:" 25 $ lit " ]" $ intw $ lit "[" $ nl $
    wlit "float with width:" 25 $ lit " ]" $ fltw $ lit "[" $ nl $
    wlit "float with -width:" 25 $ lit " ]" $ fltw $ lit "[" $ nl $
    wlit "string with width:" 25 $ lit " ]" $ strw $ lit "[" $ nl $
    wlit "string with -width:" 25 $ lit " ]" $ strw $ lit "[" $ nl )
   ()
   10 1.234 "foo"           (* int, float, string *)
   24 20 42 (-20)          (* int with width, -width spec *)
   1.234 20 567.8 (-20)    (* float with width, -width spec *)
   "Hello" 20              (* string with width spec *)
   "Goodbye" (-20)        (* string with -width spec *)
  )

```

Figure 2. Functional Unparsing-style Example

3. Results

The labels at the beginning of each bar indicate what language and formatting functions are being used for format and output and whether the Objective Caml versions are byte-code or native-code.

- cio** C version using `sprintf` and `printf`.
- printfio** Byte-code version using Objective Caml `sprintf` and `printf`.
- printfio.opt** Native-code version using Objective Caml `sprintf` and `printf`.
- cpsio** Byte-code version using printing combinators equivalent to `sprintf`.
- cpsio.opt** Native-code version using printing combinators equivalent to `sprintf`.
- cpsio_exp** Byte-code version using printing combinators equivalent to `sprintf` and `printf`.
- cpsio_exp.opt** Native-code version using printing combinators equivalent to `sprintf` and `printf`.

The number at the beginning of each bar indicates what type of formatting and output is being done.

- 1 Output using one large `sprintf` or equivalent for entire multi-line output and `fputs` or `print_string` to print the result.
- 2 Output using one `sprintf` or equivalent and `fputs` or `print_string` for each line of output.
- 3 Output using one large `printf` or equivalent for entire multi-line output.
- 4 Output using one `printf` or equivalent for each line of output.

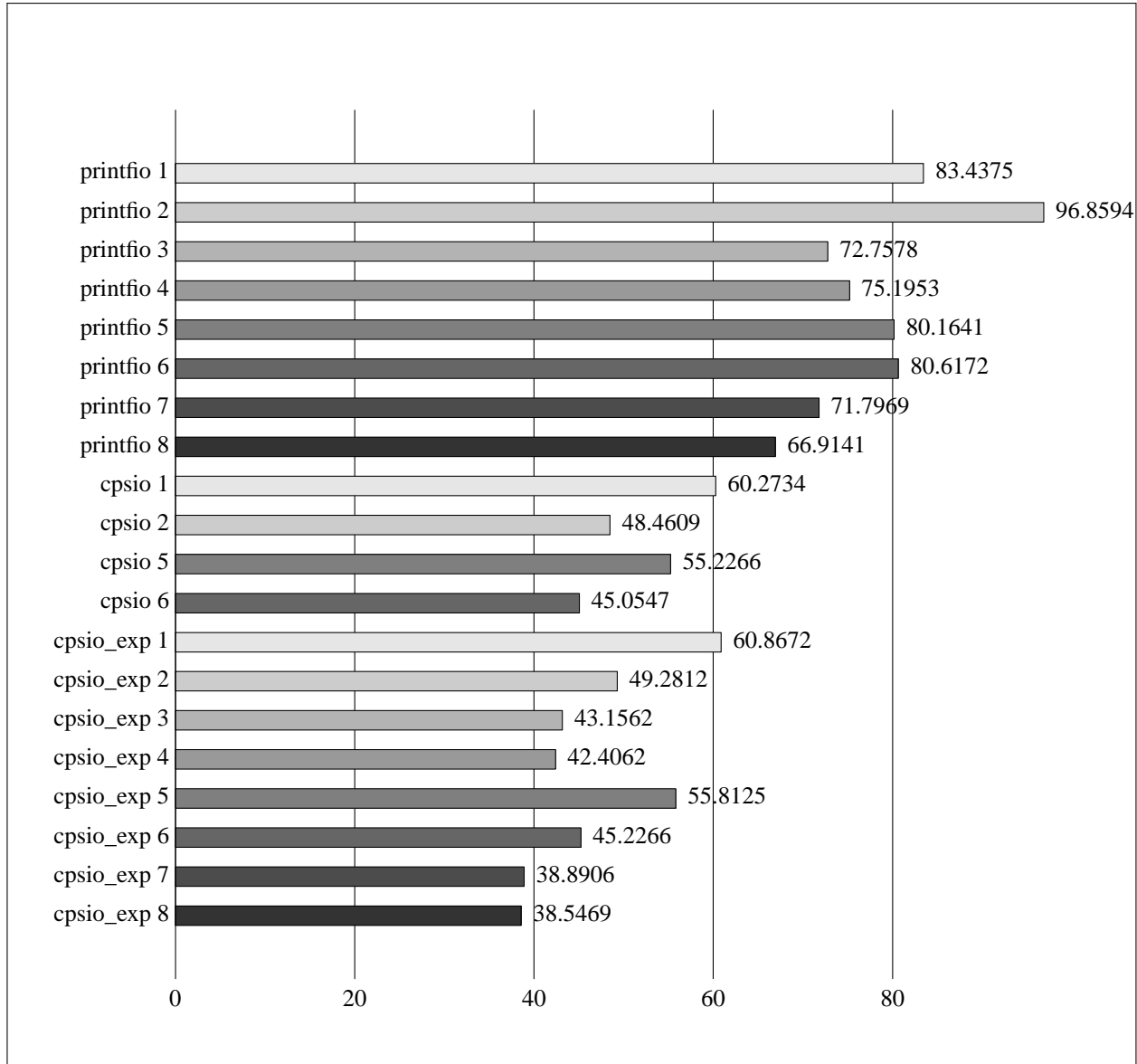


Figure 3. Byte-code Execution Time in Seconds

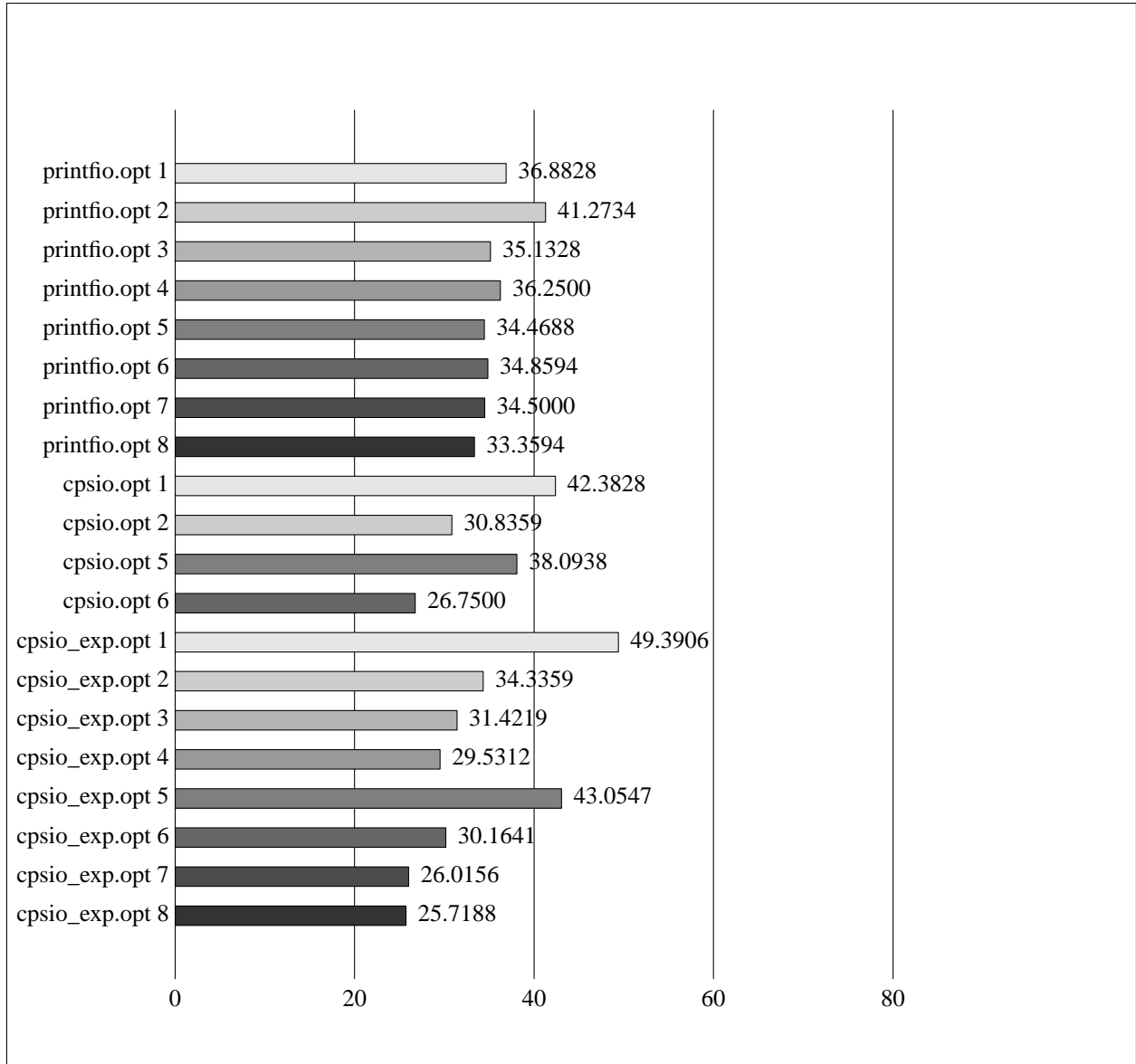


Figure 4. Native-code Execution Time in Seconds

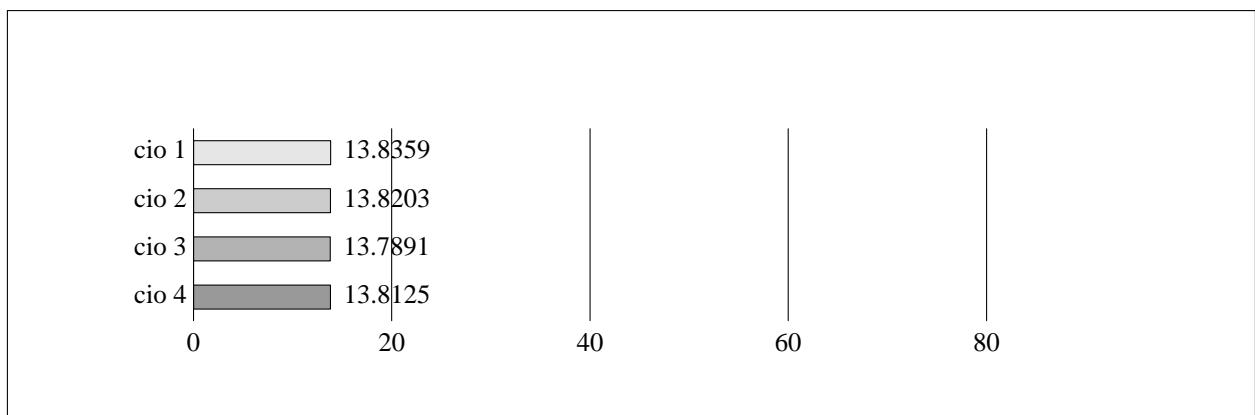


Figure 5. C Execution Time in Seconds

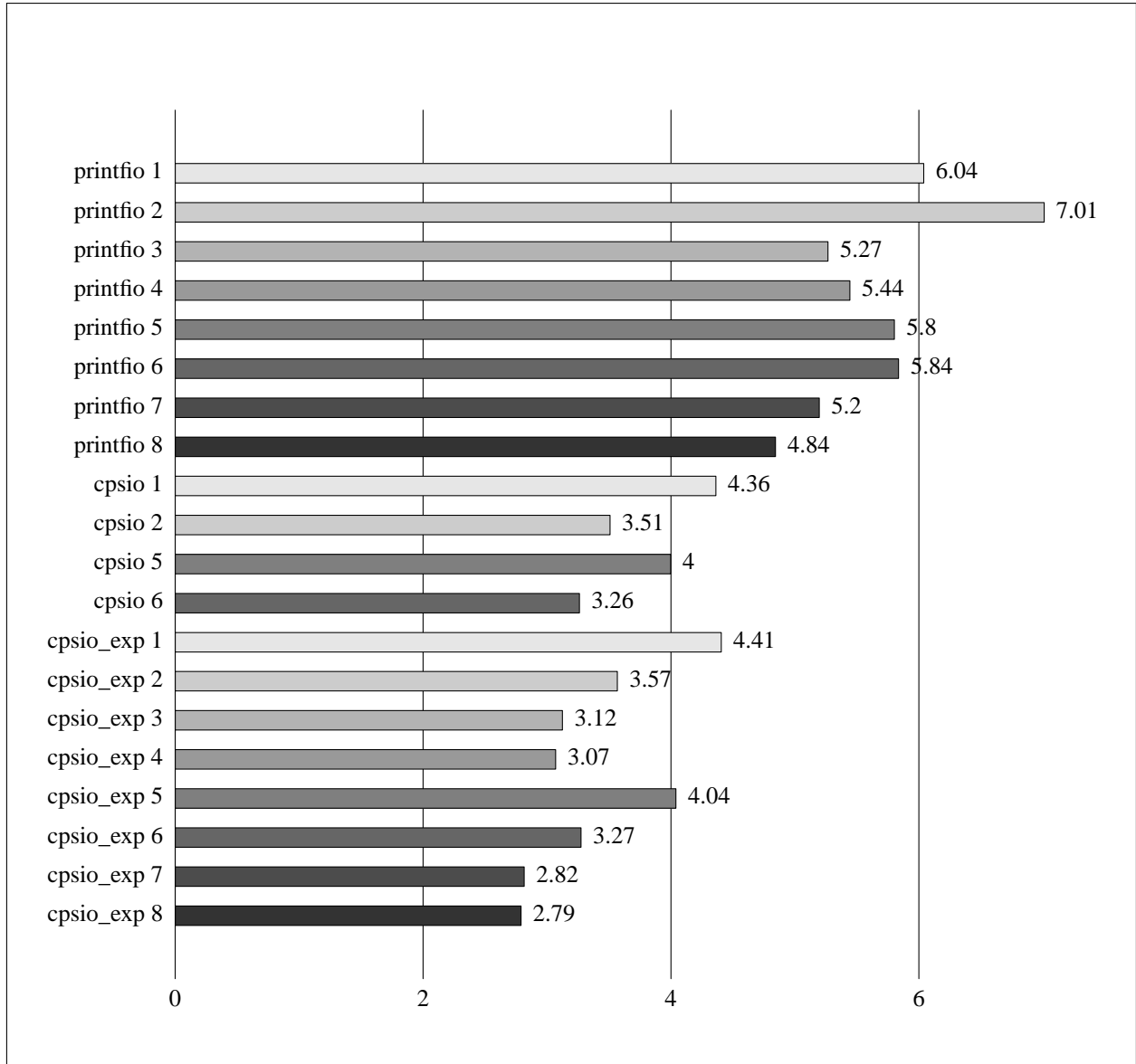


Figure 6. Ratio of Byte-code to Average C Execution Times

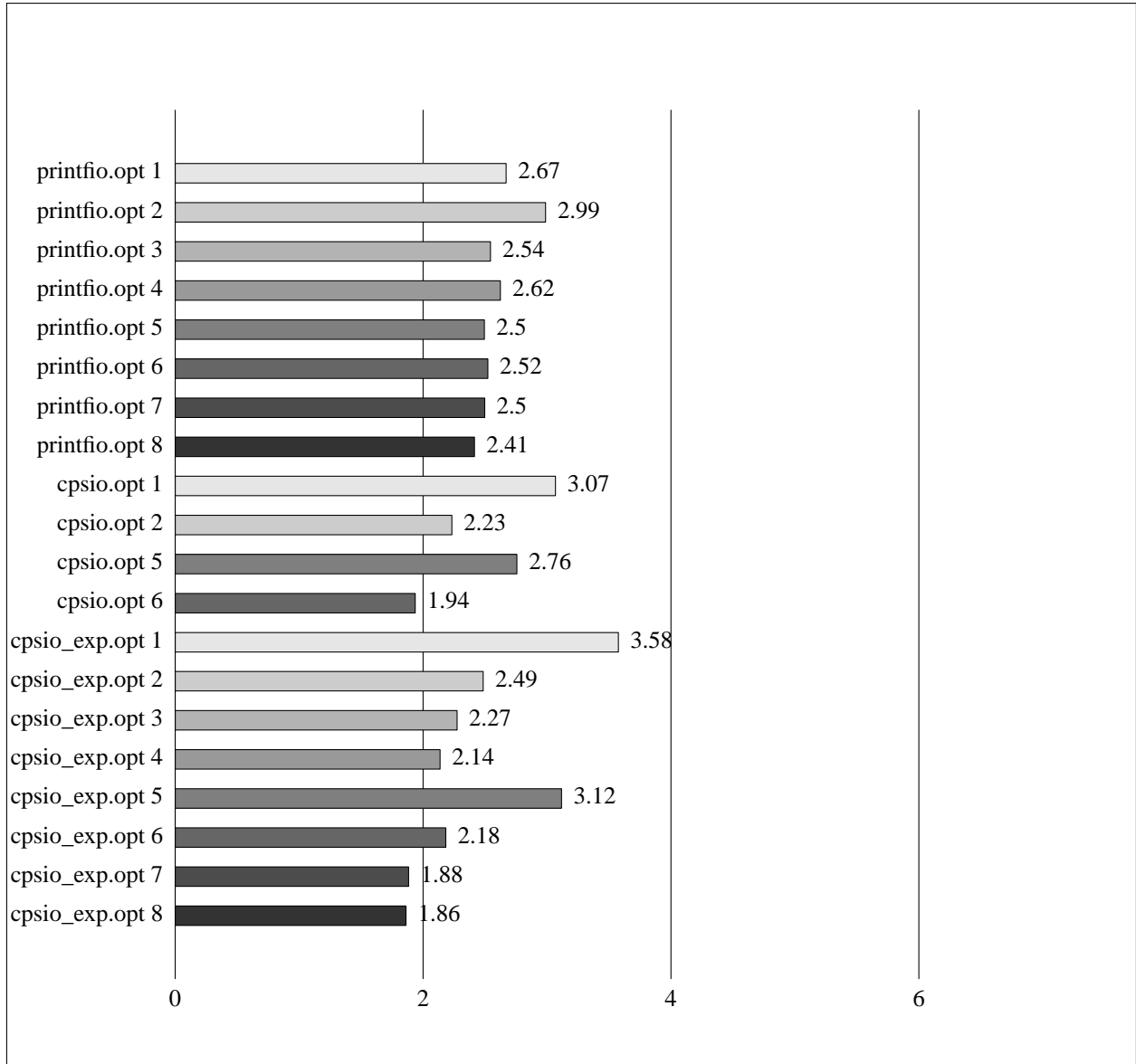


Figure 7. Ratio of Native-code to Average C Execution Times

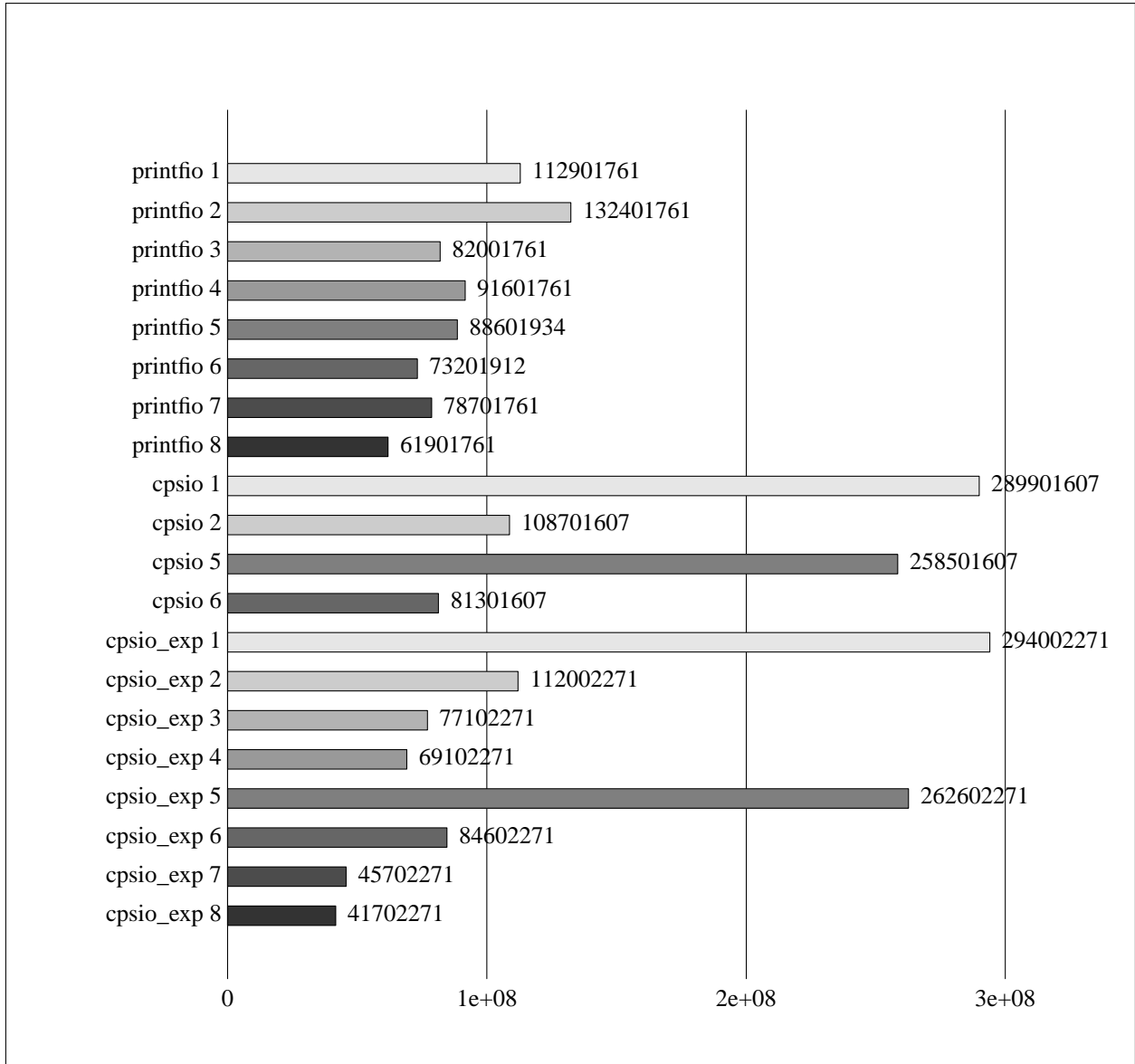


Figure 8. Byte-code Minor Words Allocated

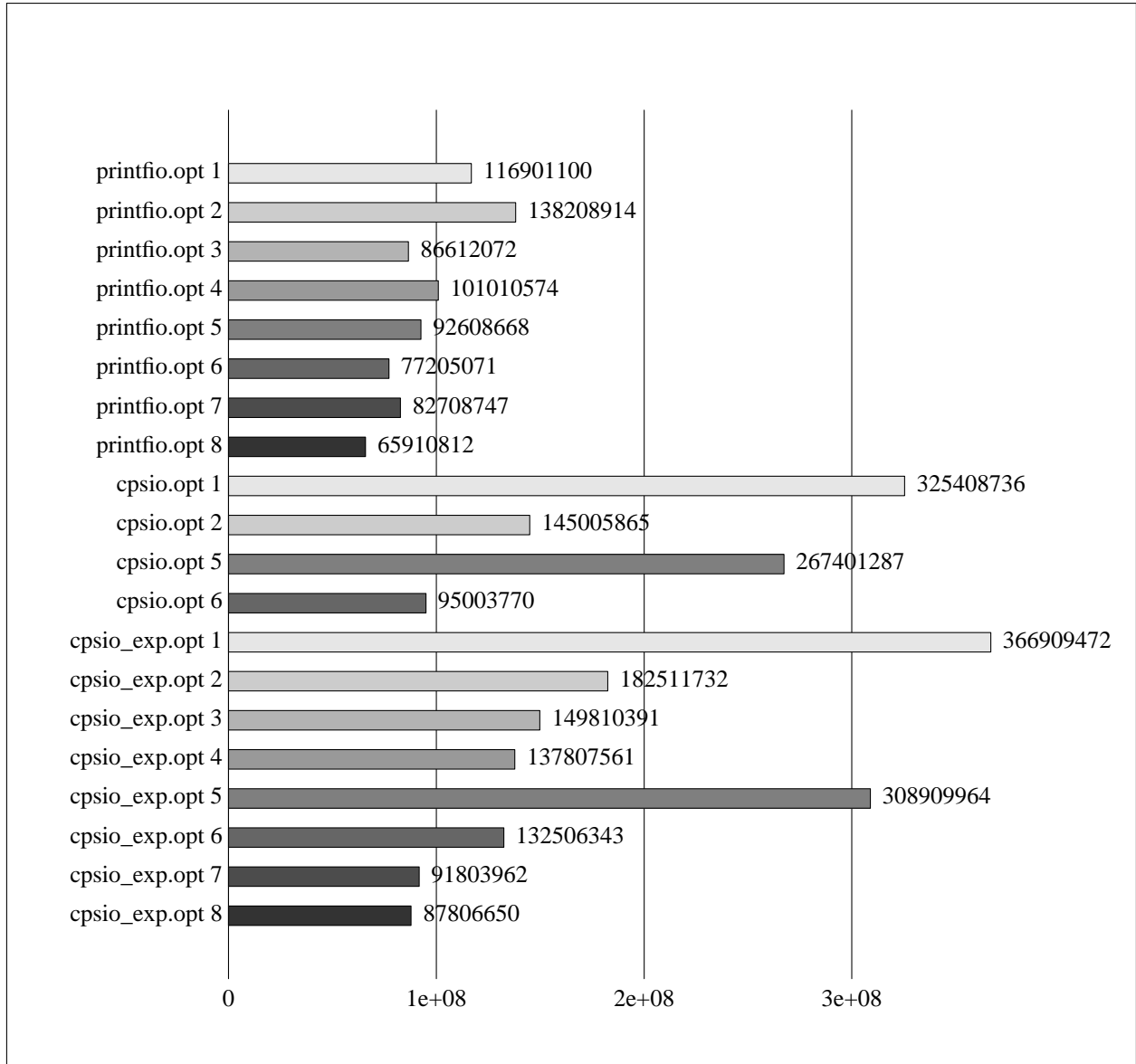


Figure 9. Native-code Minor Words Allocated

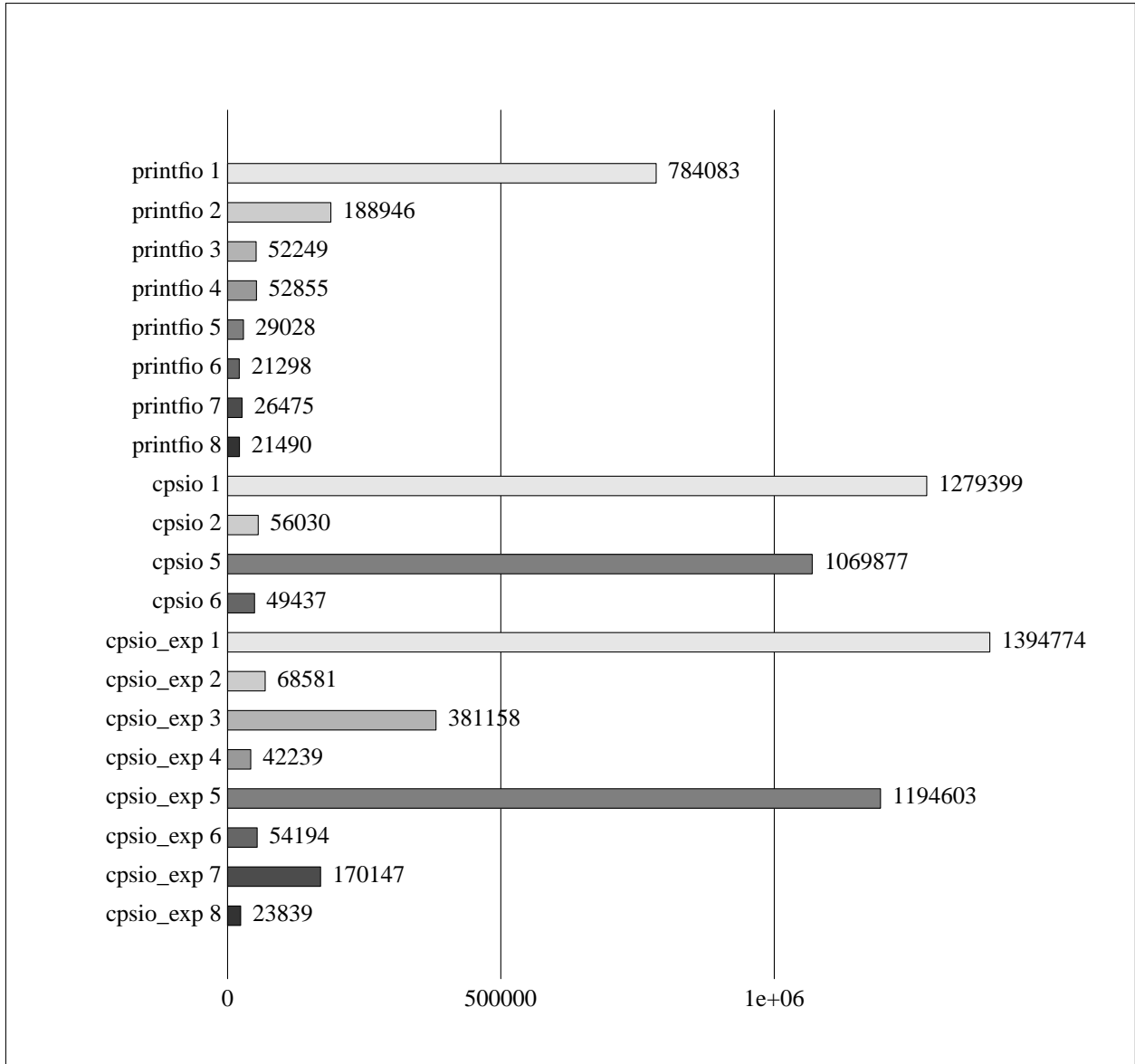


Figure 10. Byte-code Promoted Words

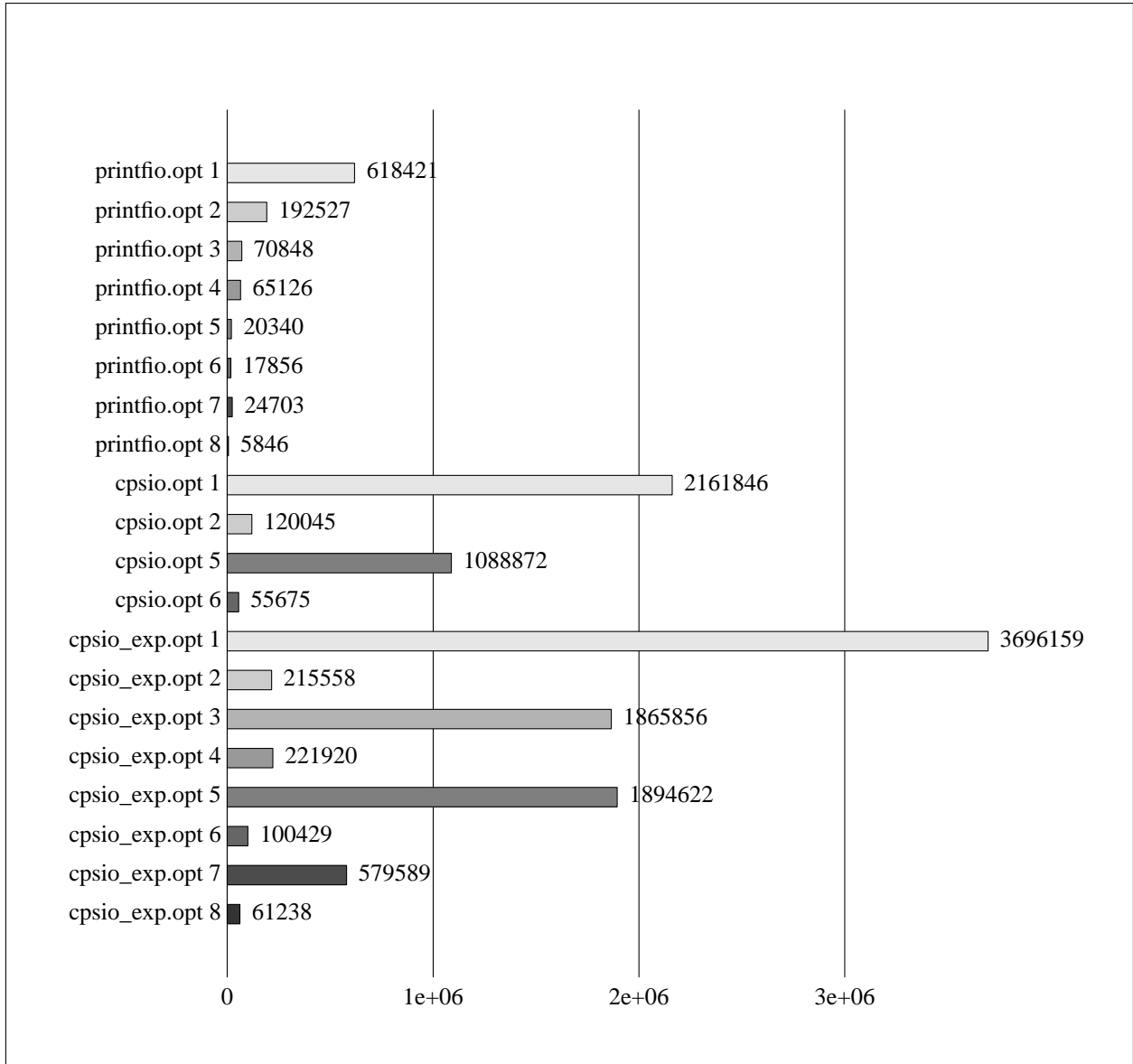


Figure 11. Native-code Promoted Words

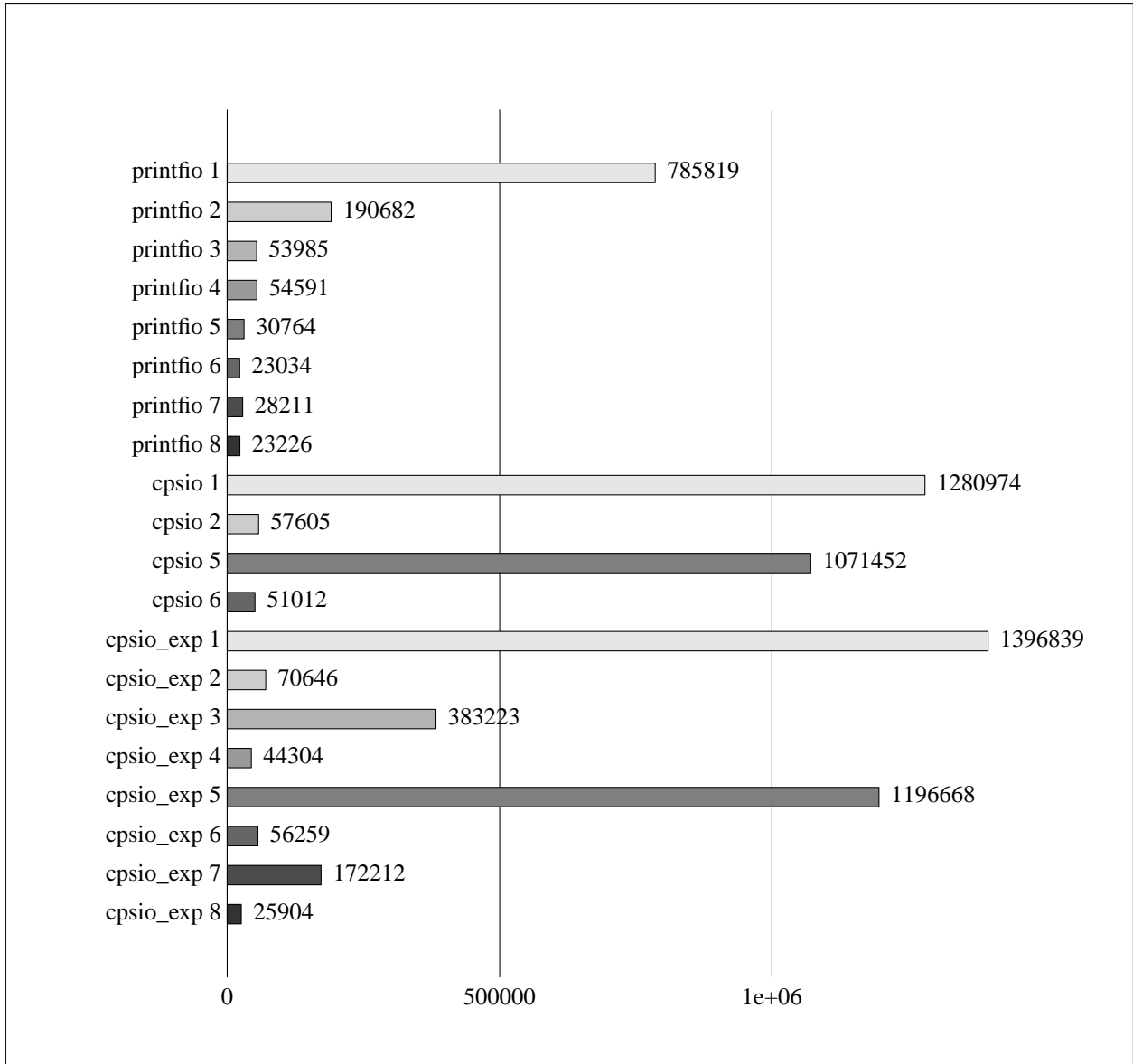


Figure 12. Byte-code Major Words Allocated

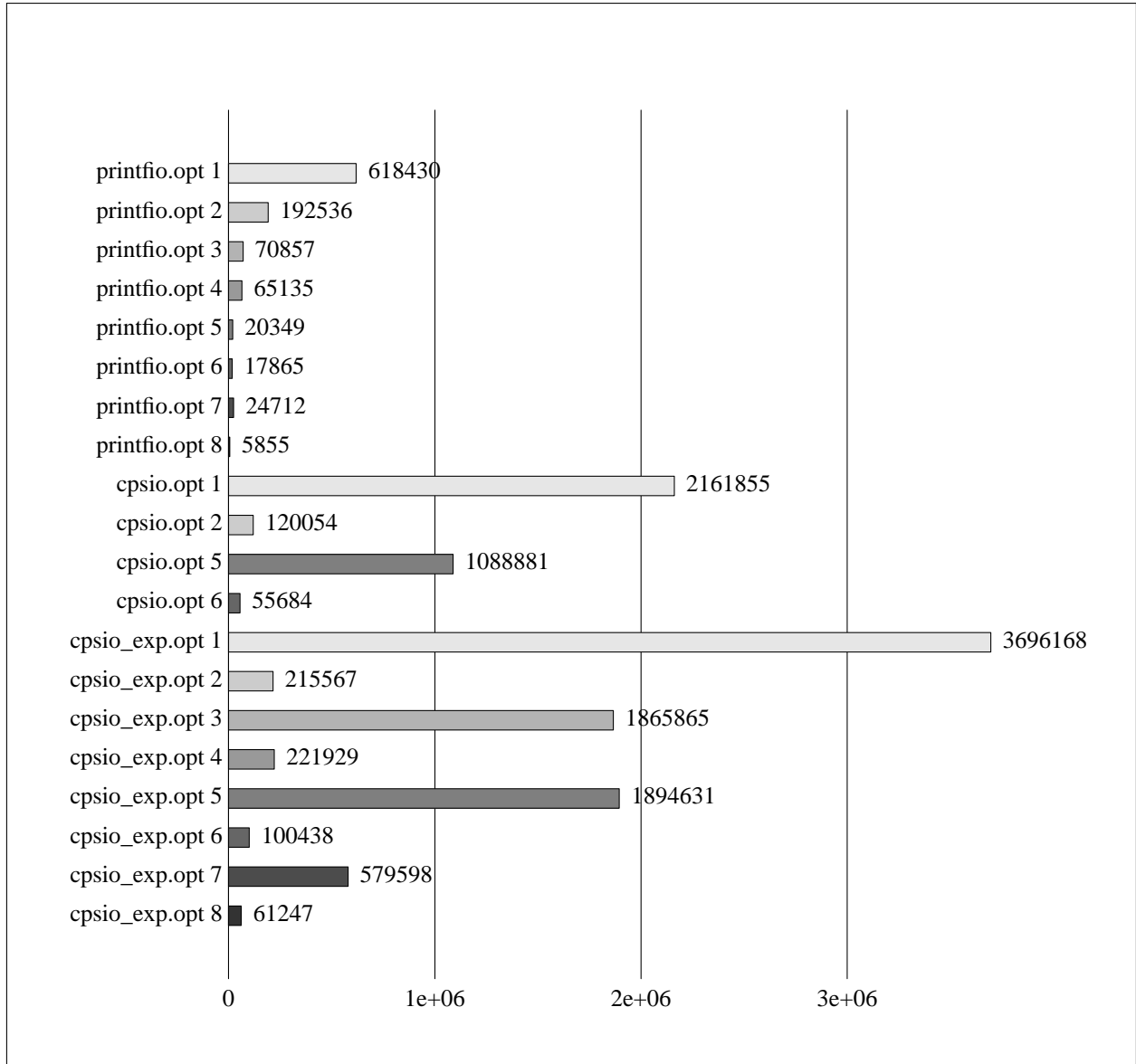


Figure 13. Native-code Major Words Allocated

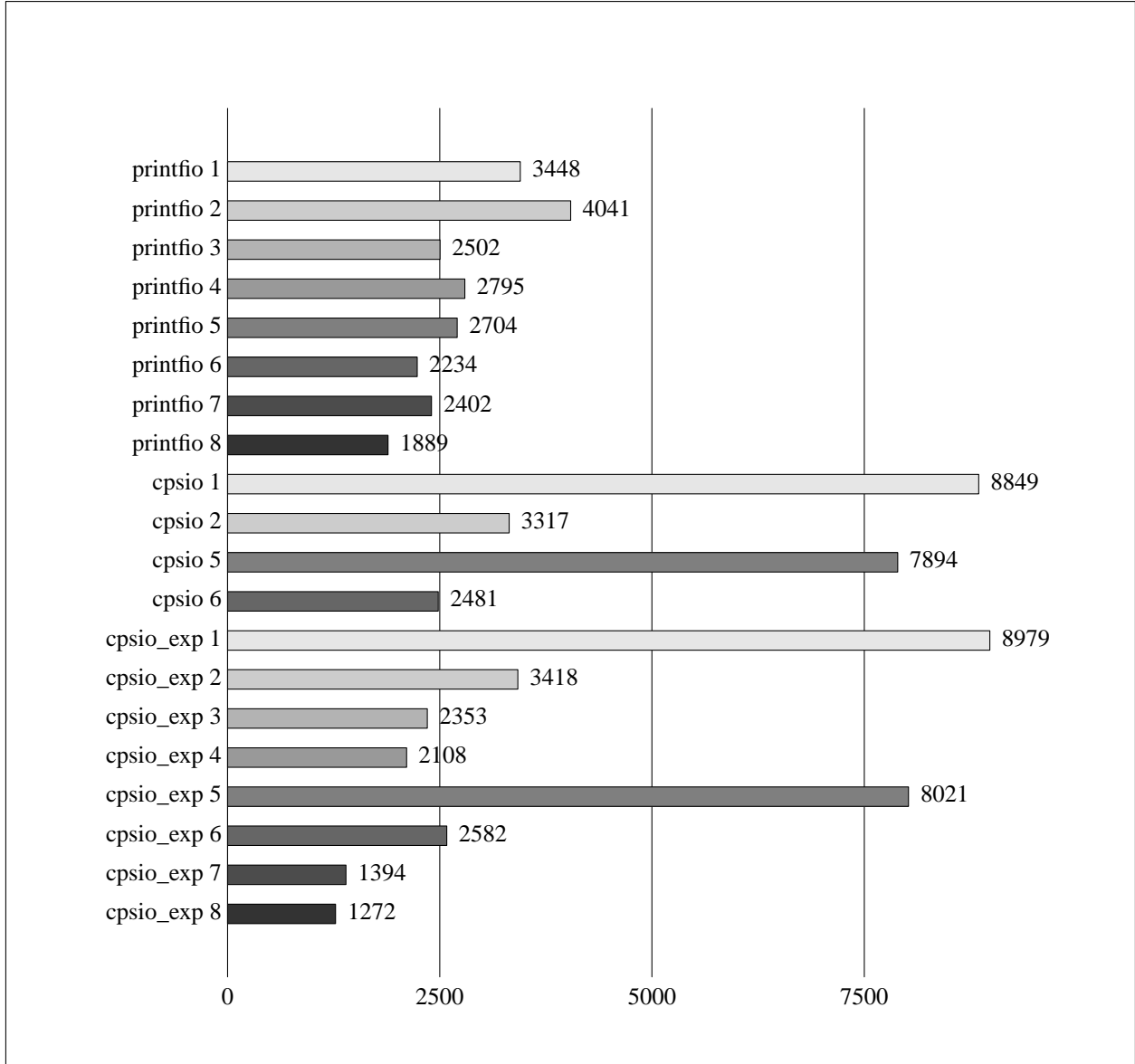


Figure 14. Byte-code Minor Collections

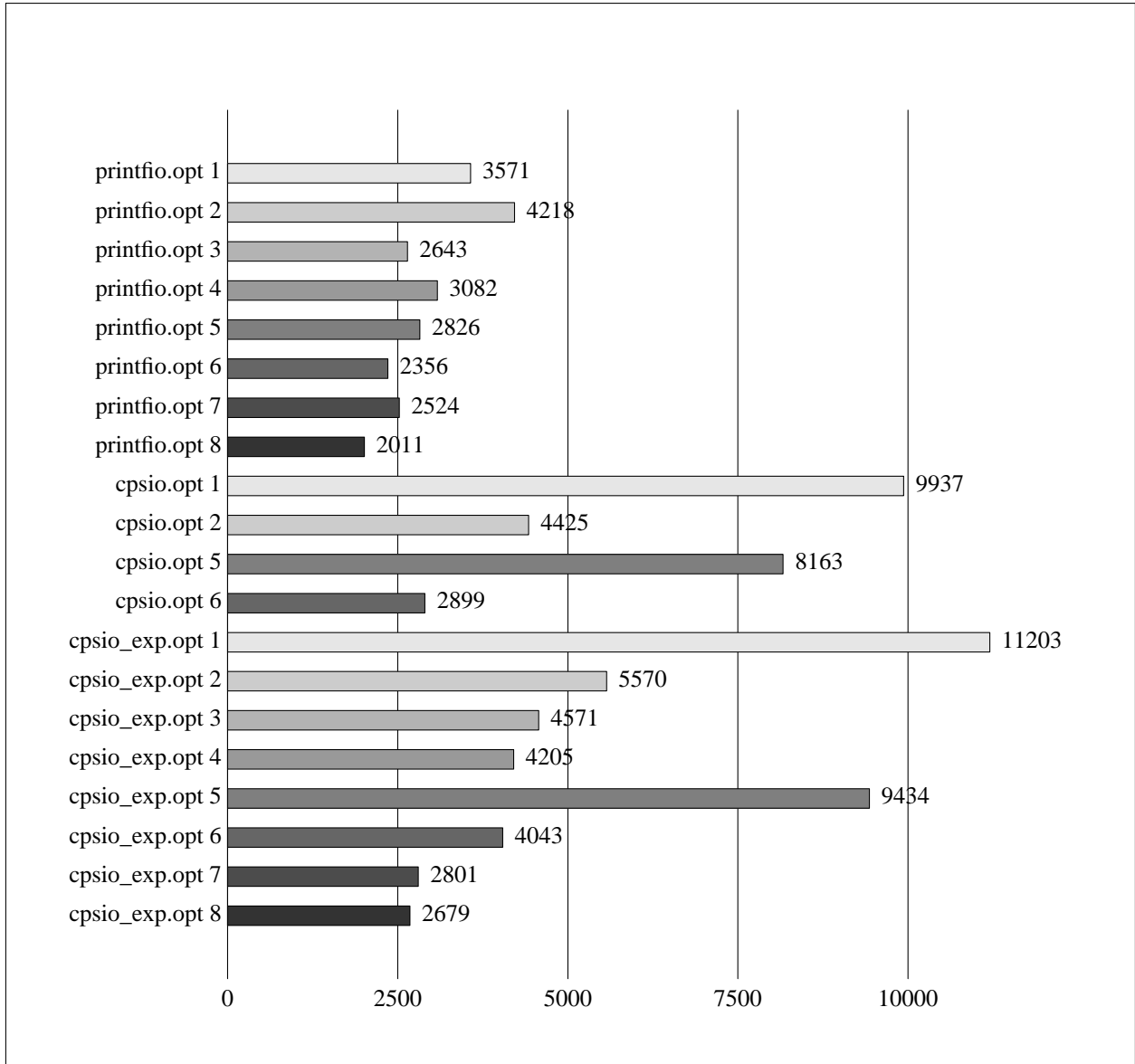


Figure 15. Native-code Minor Collections

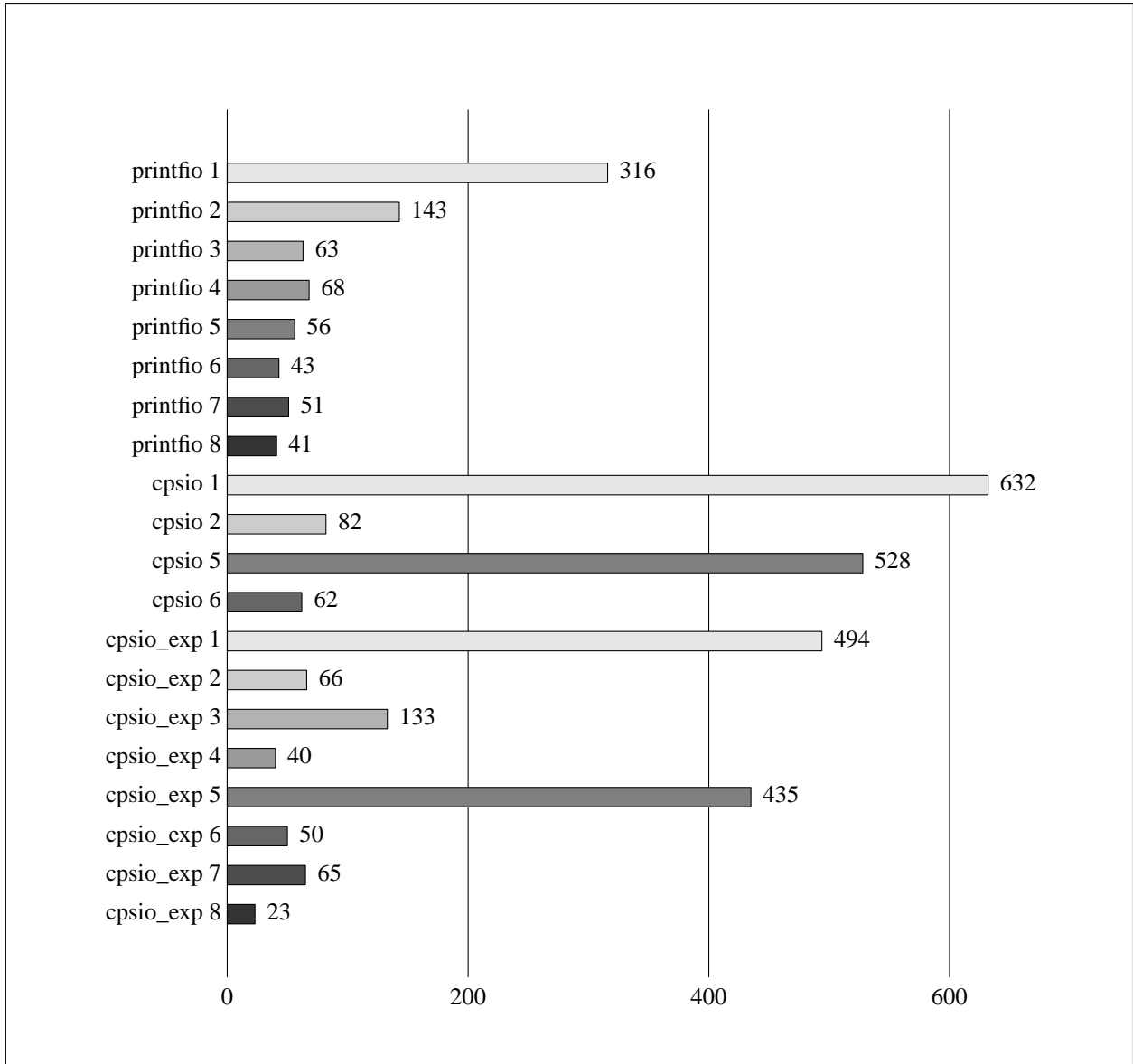


Figure 16. Byte-code Major Collections

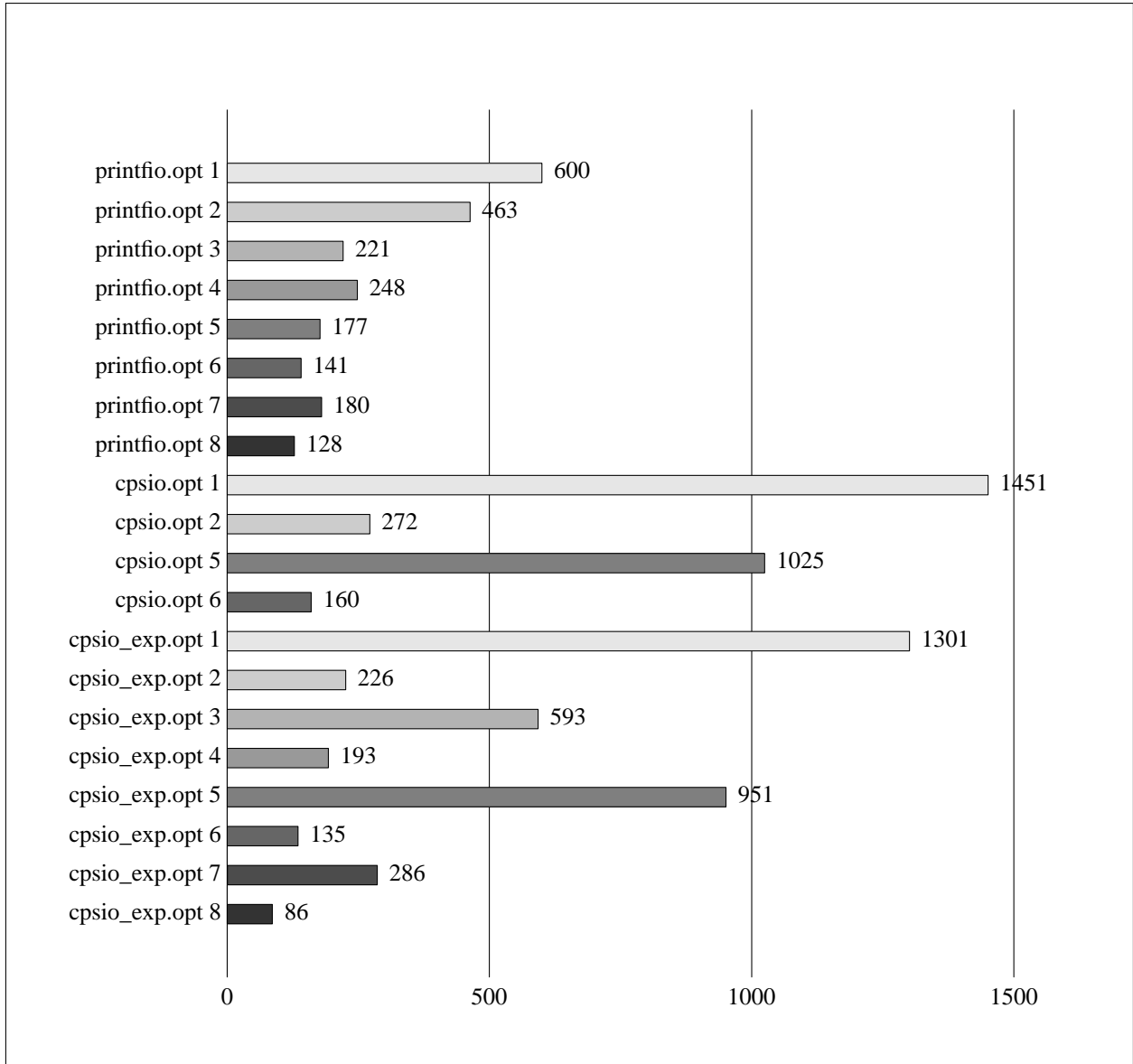


Figure 17. Native-code Major Collections

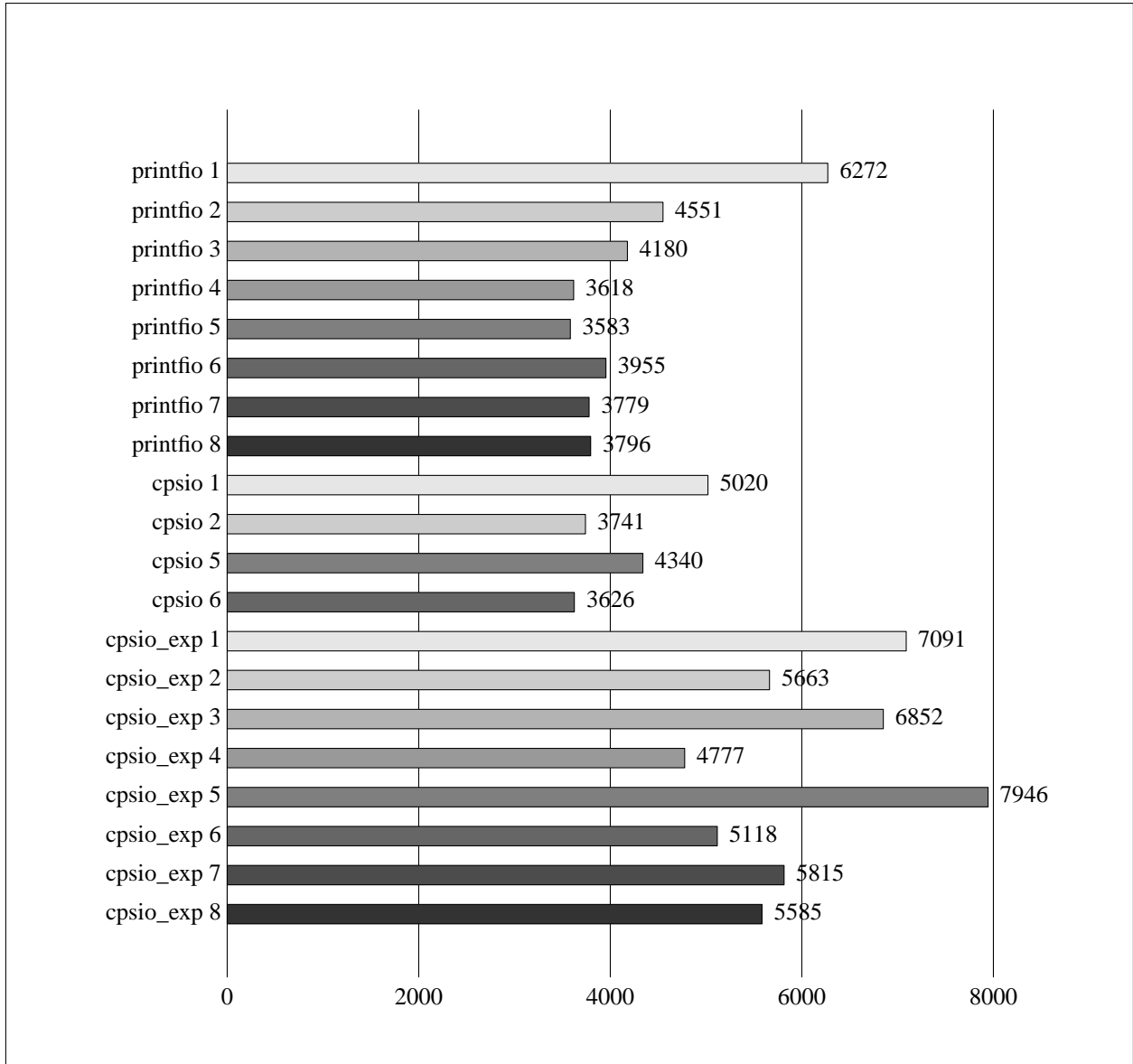


Figure 18. Byte-code Live Words

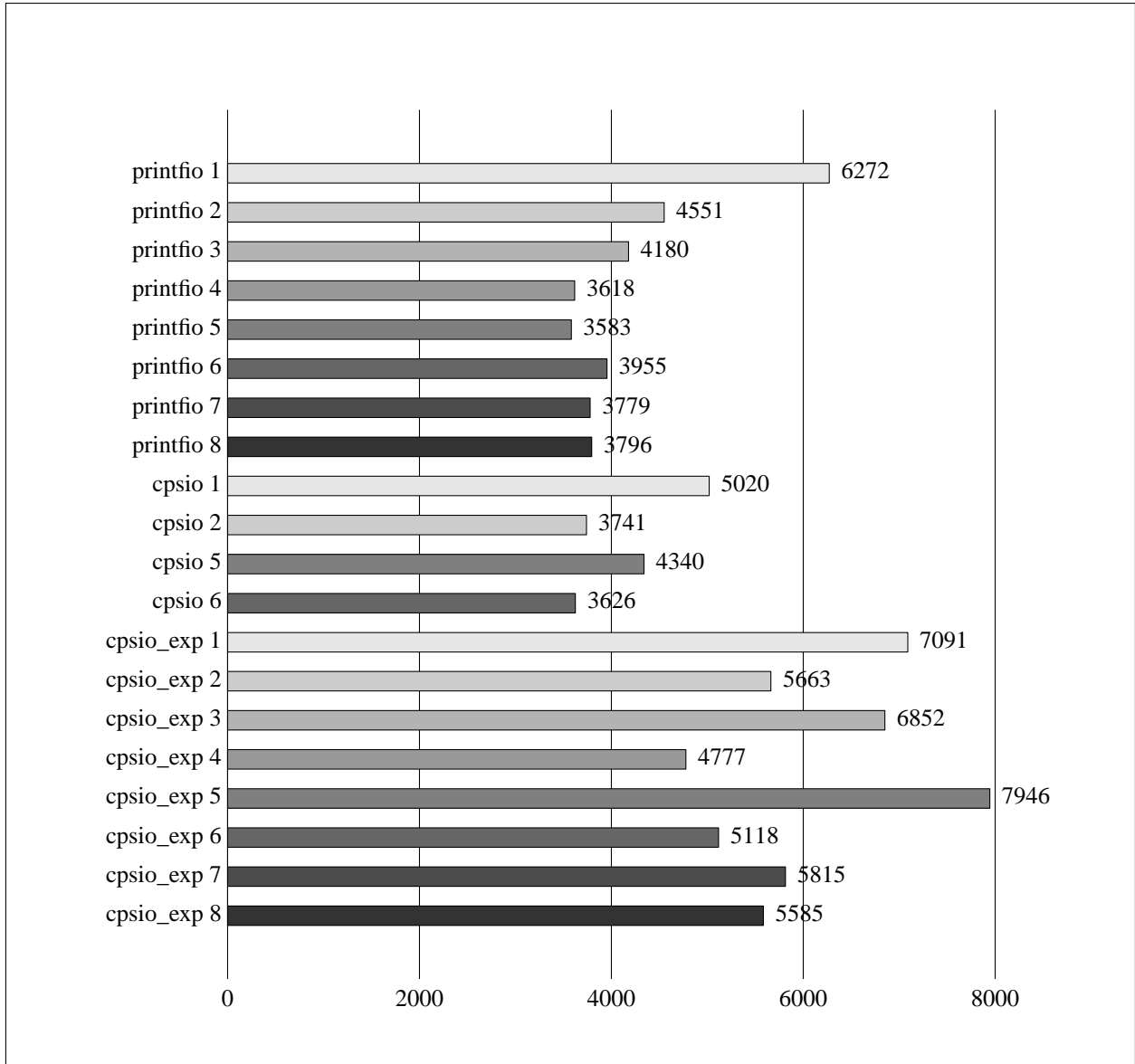


Figure 19. Native-code Live Words

4. References

- [DANVY98] Olivier Danvy, “Functional Unparsing,” BRICS RS-98-5, BRICS, Department of Computer Science, University of Aarhus (May 1998). <URL:<http://www.brics.dk/RS/98/12/>>
- [OCAML] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon, “The Objective Caml system, release 3.04, Documentation and user’s manual,” Institut National de Recherche en Informatique et en Automatique (December 10, 2001). <URL:<http://caml.inria.fr/ocaml/htmlman/index.html>>