Performance Characterization of String Libraries Implemented with AJIT 64-bit Extensions

Gauri Patrikar[1]

[1]Project Research Assistant, IITB

September 1, 2021

Abstract

We have developed string libraries using the SPARC V8 assembly with the newly added extensions to the AJIT processor. These instructions were designed to handle 64 bits at once and aid in enhancing the performance.

The libraries have been written in assembly instead of C as it is more chanllenging to develop a compiler.

The performance for implemented string functions is characterized and improvement in performance is noted for all. The libraries were compared with the existing uclibc C string libraries.

1 Introduction

1.1 64 bit extensions

The AJIT processor is based on the 32 bit SPARC V8 ISA. New assembly instructions that perform 64 bit operations have been added to enhance the performance Some examples of the 64 bit extensions that were used were-

- zbytedpos - gives an 8 bit output corresponding to a 64 bit input. It checks if any byte in the input is zero and sets the corresponding output bit to 1, else bit is set to 0.

- addd - performs 64 bit addition, result is also 64 bit.
- subdcc - performs a 64 bit subtraction while setting condition codes.
- ord - performs a 64 bit or, result is also 64 bit.
- srld - performs a maximum of 64 bit logical right shift.

These instructions are performed in a single clock cycle and help reduce number of operations, hence make string manipulation much faster.
Strings terminate with a null byte. Hence, finding the zeroth byte is important. Using the zbytedpos instruction, we can find the null byte in a 64 bit number in one instruction itself. In case of other instructions also, operations are performed two at once, for eg., increment two memory locations at once. This decreases the number of instructions required.

1.2 Sparc ABI

The programs have been written in assembly, simply because generating a compiler was a more complex task.

These libraries can be called using a C program. These programs are based on the SPARC ABI protocol. It is a calling convention used when an assembly program needs to be called using a C program. It gives the protocol for argument passing, writing the prologue and epilogue of the assembly functions and so on. This ensures that the process of shifting between the C and the assembly programs is seamless.

1.3 Restrictions

One restriction that holds in all these functions is that the addresses of the two strings need to be doubleword aligned. This is checked before starting each program.

In case of the strcat function, the termination of destination string should also be doubleword aligned.

If they are not aligned the code will exit without doing anything.

1.4 Functions and comparisons

The following functions are being implemented here -

1. strcpy
2. strncpy
3. strcmp
4. strcasecmp
5. strcat

. We are comparing these new libraries with the uclibc string library. Both of them are compiled to the SPARC V8. An improvement of about 40% to 90% was observed in the various functions

1.5 Accessing Libraries

To access these libraries, add include "astring.h", which contains the definitions for all of the functions. Further details for each function are given below.
The functions can be called using c program, as you would call any other string library, adding 'a64' to the name of the function, eg. strcmp funciton can be called by a64strcmp. All the inputs are exactly the same as the existing libraries, as is given in detail below.

Each function is described in the following sections. Its inputs and expected outputs are described below.

2 a64strcpy

The function copies the string pointed to by src, including the terminating null byte, to the buffer pointed to by dest. You can call it using -

```
char a64strcpy(char *dest, char *src);
```

The strings may not overlap, and the destination string dest must be large enough to receive the copy.

It returns the pointer to the destination string.

2.1 Program Flow

Following is a brief description of the how the program works. We use two instructions from the new extensions- zbytedpos, srld and addd.

1. Check memory alignment, has to be 64 bit aligned.
2. If aligned, operate in 64 bit mode, else return.
3. If aligned, load doubleword from memory.
4. Check for null byte using the new zbytedpos instruction. Compare with zero to see if null byte exists.
5. If no null byte, increment both memory locations using the new addd instruction, and perform a double word store.
6. Repeat this process till null byte found.
7. Check which byte is null, by shifting each byte and comparing .
8. if it is not null, increment memory and store byte,
9. continue till null byte found and return.

3 a64strncpy

First n bytes of src is copied to dest. You can call it using -

```
astrncpy (char *dest, char*src, size_t n)
```

Even if no null byte found in these n bytes and if length of dest is smaller than src, dest will not be not null terminated.

If there a null byte among the first N bytes, the rest of the length of dest is written with null bytes.

It returns a pointer to the destination string.

3.1 Program Flow

Following is a brief description of the how the program works. We use two instructions from the new extensions- zbytedpos,srld and addd.

1. Check memory alignment, has to be 64 bit aligned.
2. If aligned, operate in 64 bit mode, else return.
3. If aligned, load doubleword from memory.
4. Decrement counter and check if n is less than zero.
5. If greater than zero, check for null byte using the new zbytedpos instruction. Compare with zero to see if null byte exists.
6. If no null byte, increment both memory locations using the new addd instruction, and perform a double word store.
7. Repeat this until either n is less than or equal to zero or null byte is found.
8. If n is less than zero, icremenr it and start loading and string bytewise till n is zero.
9. end program at n = 0.

4 a64strcmp

The function compares the two strings s1 and s2. You can call it using -

```
char astrcmp (char* s1, char *s2)
```

It returns the difference between the two strings(s1-s2).

4.1 Program Flow

Following is a brief description of the how the program works. We use three instructions from the new extensions- zbytedpos, subdcc and addd.

1. Check memory alignment, has to be 64 bit aligned.

2. If aligned, operate in 64 bit mode, else bytewise.
3. If aligned, load doubleword from memory for both strings.
4. Compare both doublewords for equivalence.
5. Check for null byte using the new zbytedpos instruction for both the strings Compare with zero to see if null byte exists.
6. If no null byte, increment both memory locations using the new addd instruction.
7. Repeat this process till either null byte is found or the two strings are different.
8. If it is equal, and null byte is found, we can say the strings are equal.
9. If unequal, go bytewise to check exact byte where it differs.
10. return when the different byte is found.

5 a64strcasecmp

The astrcasecmp function is similar to astrcmp, except it compares ignoring the case of the string. You can call it using -

```
astrcasecmp (char *s1, char *s2)
```

It compares the two strings s1 and s2 ignorng the case of the byte. For eg., the byte 'a' and 'A' should be considered equivalent. and It returns the difference between the two strings (s1-s2)

5.1 Program Flow

Following is a brief description of the how the program works. We use three instructions from the new extensions- zbytedpos, subdcc and addd.

1. Check memory alignment, has to be 64 bit aligned.
2. If aligned, operate in 64 bit mode, else return.

3. If aligned, load doubleword from memory for both strings.
4. Make them lovercase by making 6th bit of each byte as '1'.
5. Check if strings are equal.
6. Check for null byte using the new zbytedpos instruction for both the strings Compare with zero to see if null byte exists.
7. If no null byte, increment both memory locations. using the new addd instruction.
8. Repeat this process till either null byte is found, or the two strings are different.
9. If equal, but null byte, then the two strings are equal.
10. If unequal, check which byte is unequal.
11. return when different byte is found.

6 a64strcat

The function copies the string pointed to by src, including the terminating null byte, to the end of the dest string. You can call it using -

```
char astrcat(char *dest, char *src);
```

Here, the dest string must also terminate at 64 bit aligned memory boundary. This is to ensure that the source can perform doubleword stores to the dest string. The strings may not overlap, and the destination string dest must be large enough to receive the copy.

It returns the pointer to the destination string.

6.1 Program Flow

Following is a brief description of the how the program works. We use two instructions from the new extensions- zbytedpos,srld and addd.

1. Check memory alignment, has to be 64 bit aligned.

2. If aligned, operate in 64 bit mode, else return.
3. Increment memory till null of dest string is found. This is the memory that the src string will copy to.
4. If aligned, load doubleword from memory.
5. Check for null byte using the new zbytedpos instruction. Compare with zero to see if null byte exists.
6. If no null byte, increment both memory locations using the new addd instruction, and perform a double word store.
7. Repeat this process till null byte found.
8. Check which byte is null, by shifting each byte and comparing .
9. if it is not null, increment memory and store byte,
10. continue till null byte found and return.

7 Performance comparison

As we can see from the figure, improvement of minimum of 40% and a maximum of upto 90% is observed.

This reduce in instructions is mainly due to the fact that about only half the number of iterations were required in the main loops (where maximum time is spent by the function), as we could use 64 bit operations.

Performance comparison of libraries

	No of bytes of string	C STRING	AJIT STRING	% DEC
STRCMP	160(eq)	437	232	46.9
	164(uneq)	450	247	45.11
	320	797	392	50.8
	324	810	407	49.7
STRCPY	160	358	194	45.81
	164	365	209	42.7
	320	638	314	50.78
	324	645	329	48.9
STRNCPY	160	1343	234	82.57
	164	1375	266	80.6
	320	2623	394	84.9
	324	2655	426	83.9
STRCAT	960	4011	2441	39.14
	1920	7733	4603	40.4
STRCASECMP	160(eq)	2965	273	90.79
	164(uneq)	3013	278	90.77
	320	5845	473	91.9
	324	5893	478	91.8

8 Conclusion

An implementation of the string libraries using 64 bit instructions in assembly was discussed. The performance was compared to that of the existing uclibc C string libraries.
Performance improvement was observed in all cases. In case of strncpy and strcasecmp functions, a significant improvement of 80%-92% was observed.

We conclude that these libraries can be used for performance enhancement.