http://dx.doi.org/10.5755/j01.eie.26.2.25871 ELEKTRONIKA IR ELEKTROTECHNIKA, ISSN 1392-1215, VOL. 26, NO. 2, 2020

Graphic Library Optimization for MIPS Architecture

Teodora Novkovic[1, *], Zeljko Lukac[1], Petar Jovanovic[2], Ivan Kastelan[1]
[1]*Department of Computing and Control Engineering, Faculty of Technical Sciences, University of Novi Sad,*
Trg Dositeja Obradovica 6, 21000 Novi Sad, Serbia
[2]*RT-RK, Institute for Computer Based Systems,*
Narodnog fronta 23a, 21000 Novi Sad, Serbia
teodora.novkovic@rt-rk.uns.ac.rs

***Abstract*—The aim of this paper and research was to analyse the efficiency of the compiler-generated code for the graphics library and to present results obtained by optimization for the MIPS (Million Instructions Per Second) architecture. *Libpng* is the official Portable Network Graphics reference library for use in applications that read, create, and manipulate PNG (Portable Network Graphics) raster image files. Given the data structure in the PNG files, as well as the capabilities of the MIPS instruction set, it was expected that significant improvements could be made. Graphic library *libpng* is optimized by using MIPS instruction set extension and tested on MIPS Malta 74K platform. Test results show, that by using MIPS optimization test, execution times are substantially improved. Our *libpng* optimization have achieved performance increase of 10 %–78 % depending on optimized routine.**

***Index Terms*—DSP; Embedded software; Image processing; MIPS; Optimization; PNG; SIMD.**

I. INTRODUCTION

The field of multimedia communications is evolving at a high speed and is conditioned by the introduction of new video and audio standards in new multimedia electronic devices. An important requirement for the end user is that multimedia electronic devices operate at a satisfactory speed in real-time. As new technologies require the increasing processing power of computer architectures, which is to some extent limited, it is necessary to significantly optimize software support on a given computer architecture to enable real-time operation.

Embedded system capabilities are a function of both the hardware platform and the software implementation strategy [1], [2]. When there are plenty of hardware resources with higher processing speeds and power like desktop computers, achieving real-time performance is not a major issue. Today, there has been an exponential growth in the use of mobile embedded systems like cell phones, tablets, video game console, etc. in the consumer electronics market, which have a relatively small power consumption and form factor. Multiple applications like video streaming, smart cameras, context-aware computing, and virtual reality are resident on these devices to meet consumer needs.

The aim of this paper and research is to analyse the efficiency of the compiler-generated code for the PNG graphics library and to present results obtained by optimization for the MIPS (Million Instructions Per Second) architecture for real-time operation.

Portable Network Graphics (PNG) is an undistorted and bitmapped image format that employs lossless data compression. PNG was created to improve upon and replace GIF (Graphics Interchange Format) as an image-file format not requiring a patent license [3].

The official graphic library that is used in applications for PNG image processing is *libpng*. This library has been designed to handle multiple sessions at one time, to be easily modifiable, to be portable to the vast majority of machines (16-, 32-, and 64-bit) available, and to be easy to use.

This graphic library supports almost all of PNGs features:

– indexed-color,
– grayscale images,
– truecolor images,
– optional alpha channel,
– sample depths range from 1 bit to 16 bits.

This graphic library is dependent on *zlib* library for data compression and decompression routines [4].

By analysing the assembler code obtained with the compiler, we found that it is possible to improve the performance by writing the assembler manually. *Libpng* has significant potential for SIMD (Single Instruction Multiple Data) acceleration due to the type of data and operations performed during image processing.

MIPS 74K core is a MIPS high-performance processor core and includes features that can support applications with significant signal processing requirements. These processors can be found in set-top boxes, VoIP SoCs (Voice over IP System on a Chip), networking equipment, digital TVs, DVD players and recorders, and in applications that use signal processing. MIPS 74K core operates at up to 1.11 GHz in a 65 nm process and supports MIPS DSP (Digital Signal Processing) oriented instruction set called DSP ASE (Application Specific Extension) Revision 2 that uses SIMD extensions to obtaining vectorised execution. [5].

Manuscript received 13 November, 2019; accepted 20 February, 2020.
This work has been partially funded by the Ministry of Education, Science and Technological Development of the Republic of Serbia under the grant (No. TR32014).

The purpose of DSP ASE instruction is to speed up algorithms in areas such as audio and video compression, image processing, communications, etc.

SIMD extensions to the embedded processors have come to support the increasing requirements by providing data parallelism. This extension improves the performance for multimedia applications, audio and video codecs, image processing, etc.

The paper is organized as follows. MIPS architecture overview is presented in Section II. The overview of MIPS32 instruction set is presented in Section III. *Libpng* optimization is described in Section IV. Finally, implementation results are given in Section V, and conclusions in Section VI. After these sections, we listed all references that we used in this paper. To the best of our knowledge, there are no works on the *libpng* topic and there are no works or results of *libpng* optimization. In the absence of literature and papers related to the *libpng* graphics library, we used references related to embedded systems, PNG optimizations, MIPS specifications, and manuals for DSPs and instruction sets.

II. MIPS ARCHITECTURE

The MIPS32 74K core is the first member of family of synthesizable 32-bit RISC (Reduced Instruction Set Computing) CPU (Central Processing Unit) cores and offers the highest performance yet from a synthesizable core. All 74K family cores implement the MIPS32 Release 2 instruction set architecture and supports the DSP ASE Revision 2 instruction set extensions. This platform has the following options:

– I and D-Caches: 4-way set associative: 0 Kbytes, 8 Kbytes, 16 Kbytes, 32 Kbytes or 64 Kbytes in size;

– L2 (secondary) cache: 128 Kbytes–1 Mbyte in size;

– Fast multiplier: 1-per-clock repeat rate for 32×32 multiply and multiply/accumulate;

– DSP ASE Revision 2: this instruction set extension adds many new computational instructions with a fixed-point math unit crafted to speed up popular signal-processing algorithms. Some of these functions do two math operations at once on two 16-bit values held in one 32-bit register;

– Floating Point Unit (FPU): if fitted, this is a 64-bit unit, which most often runs at half or two-thirds the clock rate of the integer unit;

– The “*CorExtend*” instruction set extension: defines a hardware interface, which makes it relatively straightforward to add logic to implement new computational instructions in your CPU using predefined instruction encodings.

The MIPS 74K core contains a load/store unit and separate data path, and can execute up to two instructions in parallel or up to four instructions - two floating point and two integers. These units share thirty-two 32-bit GPR and four 64-bit accumulators. The data path contains a 32-bit ALU (Arithmetic Logic) and MDU (Multiply/Divide) unit [6].

Figure 1 shows a block diagram of the MIPS 74K core.

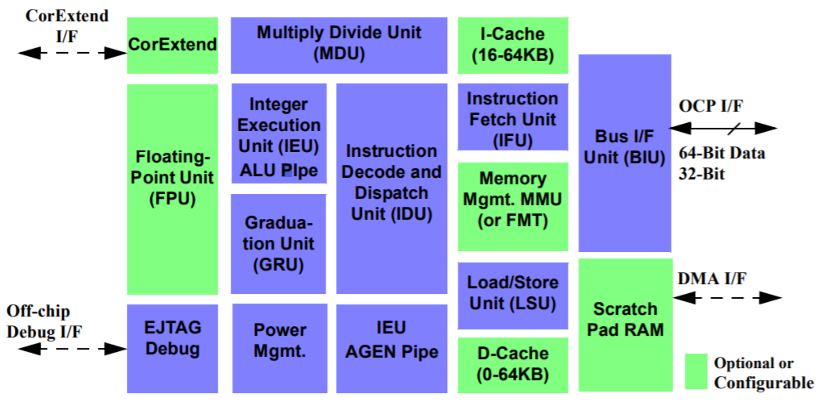

Fig. 1. MIPS32 74K Processor Core Block Diagram.

III. OVERVIEW OF MIPS32 INSTRUCTION SET

The MIPS 74K core supports the baseline MIPS32 instruction set and two instruction set extensions:

– DSP ASE Revision 2;

– MIPS 16e for 16-bit compressed instruction set.

In this paper, we will focus on DSP ASE Revision 2. This instruction set to the MIPS32 architecture was introduced by MIPS Technologies to optimize the performance of signal processing and multimedia applications running on MIPS core processors. The DSP ASE is a set of new instructions and new architectural state with computational support for fractional data types, SIMD, saturation, and other operations commonly used in DSP applications [7].

The MIPS DSP ASE Revision 2 provides support for a number of powerful data processing operations. There are instructions for fractional arithmetic (Q15/Q31) and for saturating arithmetic. Additionally, for smaller data sizes, SIMD operations are supported allowing 2 × 16 bit or 4 × 8 bit operations to occur simultaneously. Many of the DSP ASE instructions work in SIMD mode and perform certain operations simultaneously on several data packed into 32-bit vectors [5]. Instructions operating on vectors can be recognized because the name includes .ph (paired-half, usually signed, and often fractional) or .qb (quad-byte, always unsigned, only occasionally fractional) [8]. These instructions automatically perform additional operations such as rounding and saturation, which without them requires a significant number of processor cycles for implementation.

Another feature of the ASE is the inclusion of additional HI/LO accumulator registers to improve the parallelization of independent accumulation routines. All 32-bit operand arithmetic DSP instructions (except multiply) are executed in the ALU pipe, while the 64-bit operand arithmetic and multiply class DSP instructions are executed in the MDU pipe.

New instructions in DSP ASE Revision 2 help in minimizing the data packing, unpacking, shuffling, etc. That is required for SIMD arithmetic. The MIPS32 DSP ASE instructions can be classified depending on the implemented function [3]:

– *Arithmetic Class* - perform basic arithmetic operations, such as addition and subtraction (ADDQ.PH/QB and SUB.PH/QB), as well as some more specialized operations like data packing/unpacking, absolute value (ABSQ), horizontal sum of the source register data elements (RADDU), circular buffers (MODSUB), etc.;

– *Shift Class* - implement logical and arithmetic shift of the individual SIMD data elements that are packed into 32-bit registers (SHLL/R.QB/PH). The shift amount can be variable (specified by a register) or fixed. In addition, there is saturation option in the left shift instruction (SHLL_S.PH/QB);

– *Multiply Class* - include instructions for integer and fractional element-wise multiplication (MULEU_S.PH and MULQ_RS.PH), dot product accumulate and subtract (DPAU, DPSU, DPAQ_S, DPSQ_S), and compute the real and imaginary parts of a complex multiply (MULSAQ_S.W.PH and DPAQ_S.W.PH) accumulating the results to a specified accumulator;

– *Bit Manipulation Class* - include instruction that reverses the order of the 16 right-most bits in the source register (BITREV), as well as the data replication instructions REPL that copy a given scalar value to all the SIMD elements of the destination register. There is also the INSV instruction that support variable position and size values;

– *Compare and Pick Class* - include instruction that performs SIMD (element-wise) comparison of the data in the source registers (CMP.LT.PH), as well as instruction that selects elements from two registers based on the result of the comparison (PICK.QB and PICK.PH);

– *Accumulator and DSPControl Register Access Class* - contains the EXTR instruction that extract values from the accumulator after shifting it to the right, as well as the EXTP instruction extracts a number of bits from a specified position. Instructions WRDSP and RDDSP transfer data between the general-purpose registers and the DSPControl Register;

– *Indexed Load Class* - add a new addressing mode for loading integer and fixed-point data in the form of base + index. The three variants, LBUX, LHX, and LWX, load unsigned bytes, half-words, and 32-bit words, respectively.

IV. LIBPNG OPTIMIZATION

One of the problems of embedded system design is how to make software efficiently run on the used processor. Generally, software optimization has to be done for optimal system performance by matching the program code with the processor [9], [10]. During the process of code optimization, we have to take the architecture of the target processor, pipeline, compiler features, and features of instruction set into account for effective execution on target processor.

First phase in our work was to find the bottlenecks of the PNG graphic library. This involves analysing of the existing code, data structure, profiling analysis, and executing all existing tests before optimizing the code. We used following steps to do graphic library optimization.

A. Software Analysis

Firstly, we did software module partition and performance estimation in C. This step can help to understand the program structure and to find optimization goals in higher-level language. In this step, we examine the underlying data types that the program operates. How many bits are there in a single data element? Can multiple data elements be packed into a 32-bit register for SIMD data types?

B. Profiling

Performance analysis, better known as profiling, is an examination of program behavior using information gathered during program execution itself. This analysis identifies parts of the program that are suitable for optimization both of execution speed and memory usage. It is necessary to know, which parts of the software solution are most often executed, because there is no point to do optimization of routines that are rarely executed. Performance analysis shows the number of processor cycles required for each function, thus calculating its occupancy.

We used *Oprofile* tool for profiling analysis and to find functions that take the most time across the whole system. *OProfile* is an open source project that includes a statistical profiler for Linux systems capable of profiling all running code at low overhead. Profile data can be produced on the function-level or instruction-level detail. Source trees annotated with profile information can be created. That enables collection of various low-level data and association with particular sections of code [11].

Profiling was done for original source code and for all test cases developed by the authors of the *libpng* graphics library. Some of results are presented in Table I, where a list of routines suitable for the optimization is given. The number of cycles and the percentage of function share in test case for original source code of the graphic library follow the function name. Table I shows functions relevant only to *libpng* graphic library.

Profiling results showed that there were three computationally intensive tasks when reading PNG files: decoding row filters, expanding interlacing, and combining interlaced or transparent row data with previous row data.

In addition, we analysed all of routines to determine if optimization could be done for each one. The results show that it is not possible to optimize all of *libpng* routines, so we opted for partial optimization of problematic routines.

We have selected routines that we will optimize based on the operations they perform and the types of data. Each routine, where it was possible to vectorize operations and pack more data into a single 32-bit register, was chosen to be optimized.

TABLE I. PROFILING RESULTS FOR ORIGINAL SOURCE CODE - LIST OF ROUTINES THAT COULD BE OPTIMIZED WITH NUMBER OF PROCESSOR CYCLES AND CPU OCCUPANCY.

***Libpng* graphic library profiling**		
Routine	**Samples**	**%**
png_write_find_filter	179509	85.39
png_read_filter_row_paeth_multibyte_pixel	150527	74.01
png_do_read_interlace	32169	25.07
png_combine_row	28563	19.82
png_do_chop	2369	9.53
png_read_filter_row_sub	2203	9.11
png_read_filter_row_avg	1756	6.97
png_read_filter_row_up	1637	6.09
png_do_scale_16_to_8	1223	5.66
png_do_gamma	657	2.84

C. Compiler Analysis

A new compiler that enables code translation for that architecture accompanies the new architecture on the market. Initially, first compiler versions were not perfect in the sense that it failed to produce the most optimal code, so assistance by programmers and hand-coding assembler writing are required in some parts of the routine. Over time, the compiler evolves and improves, and with each successive version, the code that translates is better and more optimized.

In this work, we used GCC (GNU Compiler Collection) compiler 4.5.1. [12], on Linux operating system to translate the *libpng* graphic library and run on MIPS architecture. This compiler and GNU Assembler provide options *–mdsp* and *–mdspr2* that enable the availability of Revision 1 or Revision 2 MIPS DSP ASE instructions, respectively, for code generation.

There are few methods [13] of utilizing the MIPS DSP ASE that are supported by the GCC GNU Assembler (GAS) [14] for MIPS cores:

– Hand-coding in assembly language;
– Hand-coding in inline assembly;
– ASM-macros;
– Intrinsics;
– Fixed-point data types and operators in C;
– Auto-vectorization.

Hand-coding in assembly language is the most time-consuming method, but can produce code with the highest performance.

Firstly, we tried to use auto-vectorization. It is necessary to activate automatic vectorization using a specific compilation flags (-mdspr2 -O3 -EL). The advantage of auto-vectorization is that the compiler can recognize scalar variables in order to utilize SIMD instructions automatically. Unfortunately, this method failed to get good results for all routines, so we decided for handwriting inline assembly.

D. Inline Assembly

Using the inline assembly can reduce the number of instructions required to be executed by the processor. Also, it is important because of its ability to operate and make its output visible on C/C++ variables. With inline assembly, it is possible to do partial optimization for speed-critical sections of code.

Before writing the inline assembler, it is necessary to analyse all original routines that will be optimized. It is generally necessary to make modifications to the program structure, data structure, as well as to do loop unrolling [9] in order to prepare the original code for vectorization and optimizations. For example, we had to carry out in parallel on operands packed into a single 32-bit general-purpose register for SIMD type operations that could be 2x16-bit or 4x8-bit operations.

E. Assembly Tips & Rules

In order to do better optimizations of routines, a few specific and important tips and rules need to be followed:

Instruction's latencies. It is necessary to know the basic latencies of instructions as executed by the 74K core, i.e. how many cycles later can be issued as a dependent instruction. For these purposes, there are four classes of instructions [9]:

– Simple ALU instructions - run in 1 cycle. This includes bitwise logical instructions, *mov* (an alias for *addu* with *$0*), shifts up to 8 positions down or up, test-and-set instructions, and sign-extend instructions;
– Simple DSP ASE operations – 2 cycle latency. The same as most regular MIPS32 arithmetic without multiply and saturation;
– DSP instructions, which feature saturation or rounding - three-cycle latency;
– Special DSP multiply - 6–9 cycle latency.

Out-of-order execution. The 74K core supports out-of-order instruction execution, which can automatically reorder instructions to more efficiently pair up instructions for parallel execution and help hide instruction and cache latencies [10].

Loads and load-to-use delay - MIPS CPUs cannot deliver loaded data to the immediately following instruction without a delay. Typically, data is delivered one clock later, so we can try to put some useful and non-dependent instruction between the load and its first use.

Branch delay slot instruction. The MIPS architecture defines that the instruction following a branch (the "branch delay slot" instruction) is always executed. Also, there are special forms of branches called "branch likely", which execute the branch delay slot instruction only when the branch is taken.

All of these tips and rules were used to reduce program and data space and to rearrange data space by manually modification of the assembly code.

F. Optimization

Our goal was to reduce the number of memory accesses and to do as many operations as possible in parallel to get the best results. A reduced number of accesses to memory is derived by taking 32-bit value from memory instead of taking 8-bit or 16-bit values, which are then computed within a 32-bit value. Compared to the original software solution, they avoid unnecessary memory access up to eight times within a single loop pass.

We analysed the original source code with an *Objdump* tool [14] that generates assembler code from the source C code. *Objdump* display information from object files. In this step, we identified where the delays occur in executing the instructions. These delays are most often caused by the interdependence between two consecutive instructions. Namely, if the execution time of an instruction is more than one cycle (for example, multiplication operations) and the result is used in the next instruction, the flow structure is stopped until the result of the first instruction becomes available. Such problems are solved by simply rearranging the order of execution of instructions, i.e. by inserting some independent instructions into the flow structure.

In addition, instructions with the greatest delay were placed at the beginning of the assembler block, and after these instructions, we set other instructions that could be executed in parallel.

The loading of coefficients and constants is done outside the loop avoiding additional memory access, which occurs in the original software solution.

The processing of most routines is based on computing with 8-bit data. Due to out of range, it is not possible to use DSP ASE instructions over 8-bit data sets. That is why 8-bit data are placed within each 16-bit data of one 32-bit registry. DSP ASE instructions were used over this data, which made parallelization possible. In particular, the PRECEU.PH.QBR instruction was used for these situations.

In digital signal processing, processing is generally performed on larger data sets cyclically. There are a number of iterations of the same operations, where only the input data is changed. In this kind of processing, it is possible to parallelize the processing of several iterations simultaneously. This method is called "loop unrolling", and thus avoids the delay caused by memory access. The unrolled loop assumes that the number of iterations of the original loop is multiple of the unroll factor. For example, if a loop is unrolled four times (4x), the number of iterations in the original loop has to be a multiple of four. This is usually not worrisome and the code can be written to work correctly for any number of iterations. Sometimes, it is easiest to just produce a few extra don't care values at the tail-end of the unrolled loop; just to ensure that the data buffers are large enough to hold all the output values from the unrolled loop [15].

In the following example, in Fig. 2 and Fig. 3, we present loop unrolling and some optimization details for routine *png_read_filter_row_paeth_multibyte_pixel*.

```
 1      int bpp = (row_info->pixel_depth + 7) >> 3;
 2      png_bytep rp_end = row + bpp;
 3      while (row < rp_end)
 4      {
 5         int a = *row + *prev_row++;
 6         *row++ = (png_byte)a;
 7      }
 8      rp_end += row_info->rowbytes - bpp;
 9
10      while (row < rp_end)
11      {
12         int a, b, c, pa, pb, pc, p;
13         c = *(prev_row - bpp);
14         a = *(row - bpp);
15         b = *prev_row++;
16         p = b - c;
17         pc = a - c;
18         pa = p < 0 ? -p : p;
19         pb = pc < 0 ? -pc : pc;
20         pc = (p + pc) < 0 ? -(p + pc) : p + pc;
21         if (pb < pa) pa = pb, a = b;
22         if (pc < pa) a = c;
23         c = b;
24         a += *row;
25         *row++ = (png_byte)a;
26      }
```

Fig. 2. Original C code for png_read_filter_row_paeth_multibyte_pixel routine.

Figure 2 shows original C code for png_read_filter_row_paeth_multibyte_pixel routine. There are two while loops in original code: first for reading each row data and second for data processing. Reading of each row data was done by taking 8-bit values from memory (Fig. 2 - Line 6). So, there are too many accesses to memory. We reduced number of accesses to memory by taking 32-bit value from memory (Fig. 3 - Line 9).

Original code uses *bpp* variable to calculate the end of row. This value is the number of bytes per pixel and could have only the value of? 3 or 4. We certainly do 4 pixel processing, so we avoided this calculation by using parallelization.

Also, the data processing in the second while loop worked with 8-bit values in original code. In our optimization, we used SIMD instructions to process 4 bytes in parallel, whenever it was possible. All MIPS DSP ASE instructions with .qb in names (Figure 3 - Lines 6, 14–17, etc.) use 4 bytes processing in parallel. Due to out of range for some processing, we had to place 8-bit data within each 16-bit data of one 32-bit registry. As we already mentioned, we used instruction PRECEU.PH.QBR for this purpose.

After this data packing, we had to use instructions that process 2 bytes in parallel. These instructions have .ph in names (Figure 3 - Lines 23–44, etc.). Also, whenever it was possible, we set other instructions after the instruction with greatest delay.

After optimization, the number of instructions in the original and optimized code was compared. Almost all routines are optimized to handle 4 pixels in one pass, so the number of instructions of optimized code compared to the original code is greatly reduced.

```
 1    unsigned int rowbytes = row_info->rowbytes;
 2
 3   __asm__ __volatile__ (
 4      "lw              $t0, 0(%[row])          \n"
 5      "lw              $t1, 0(%[prev_row])     \n"
 6      "addu.qb         $t2, $t0, $t1           \n"
 7      "addiu           $t1, %[rowbytes], -4    \n"
 8      "addu            $t0, %[row], $t1        \n" // t0 = end address
 9      "sw              $t2, 0(%[row])          \n"
10      "1:                                      \n"
11      "lw              $t1, 0(%[prev_row])     \n" // t1 = c
12      "lw              $t2, 4(%[prev_row])     \n" // t2 = b
13      "lw              $t3, 0(%[row])          \n" // t3 = a
14      "subu_s.qb       $t4, $t2, $t1           \n" // p = b - c
15      "subu_s.qb       $t5, $t1, $t2           \n" // t5 = c - b
16      "subu_s.qb       $t7, $t3, $t1           \n" // pc = a - c
17      "subu_s.qb       $t6, $t1, $t3           \n" // t6 = c - a
18      "or              $t4, $t4, $t5           \n" // t4 = abs(p) [pa]
19      "or              $t5, $t7, $t6           \n" // t5 = abs(pc) [pb]
20      "cmpu.lt.qb      $t5, $t4                \n" // if (pb < pa)
21      "pick.qb         $t4, $t5, $t4           \n" // t4 = pa
22      "pick.qb         $t8, $t2, $t3           \n" // a ? b : a
23      "preceu.ph.qbr   $t5, $t3                \n"
24      "preceu.ph.qbr   $t6, $t2                \n"
25      "preceu.ph.qbr   $t9, $t1                \n"
26      "addq.ph         $t6, $t5, $t6           \n" // t6 = lo(a+b)
27      "shll.ph         $t5, $t9, 1             \n" // t5 = lo(2*c)
28      "preceu.ph.qbr   $t7, $t4                \n"
29      "subq.ph         $t5, $t6, $t5           \n"
30      "preceu.ph.qbl   $t6, $t3                \n"
31      "absq_s.ph       $t5, $t5                \n" // t5 = abs(lo(pc))
32      "cmp.lt.ph       $t5, $t7                \n"
33      "preceu.ph.qbr   $t3, $t8                \n" // t3 = hi[a]
34      "preceu.ph.qbl   $t7, $t2                \n"
35      "pick.ph         $t5, $t9, $t3           \n" // hi[a] ? c : a
36      "preceu.ph.qbl   $t9, $t1                \n"
37      "addq.ph         $t7, $t6, $t7           \n" // t7 = hi(a+b)
38      "shll.ph         $t6, $t9, 1             \n" // t6 = hi(2*c)
39      "preceu.ph.qbl   $t3, $t4                \n"
40      "subq.ph         $t6, $t7, $t6           \n"
41      "addiu           %[prev_row], %[prev_row], 4  \n"
42      "absq_s.ph       $t6, $t6                \n" // t6 = abs(hi(pc))
43      "preceu.ph.qbl   $t7, $t8                \n" // t7 = lo[a]
44      "cmp.lt.ph       $t6, $t3                \n"
45      "lw              $t3, 4(%[row])          \n"
46      "pick.ph         $t6, $t9, $t7           \n"
47      "precr.qb.ph     $t8, $t6, $t5           \n" // t8 = full a
48      "addiu           %[row], %[row], 4       \n"
49      "addu.qb         $t8, $t8, $t3           \n"
50      "bne             $t0, %[row], 1b         \n"
51      " sw             $t8, 0(%[row])          \n"
52      "3:                                          \n"
53      : [row] "+r" (row), [prev_row] "+r" (prev_row)
54      : [rowbytes] "r" (rowbytes)
55      : "t0", "t1", "t2", "t3", "t4", "t5", "t6", "t7", "t8", "t9"
56    );
```

Fig. 3. MIPS DSP ASE optimization for *png_read_filter_row_paeth_multibyte_pixel* routine.

V. EXPERIMENTAL RESULTS

The results of the optimizations were obtained by measuring the performance of the graphics library using test programs developed by the authors of the *libpng* graphics library. The testing was performed on the MIPS 74K Malta development platform with an integrated Linux operating system.

Tests generally do conversions between PNG and other image formats. To run existing tests, it is advisable to use images required by the authors of the *libpng* graphics library, which can be found on the *libpng* official website. These existing tests check output's bit-exactness after images processing and have time execution measurement. Time measurement is done in terms of second (s).

Before using existing *libpng* tests, we created our standalone tests to verify bit-exactness and acceleration for all routines. These tests have shown that optimized routines give bit-exactness outputs for same inputs and acceleration of about from 35 % to 80 % depending on the optimized routine. It was expected that these standalone tests would perform better results than the existing tests, because these tests measure time and number of cycles for each one optimized routine separately. These tests use the same input data from the existing tests. Each optimized routine has a separate test, in which we first check bit-exactness of the output relative to the reference code, and then the processing time is measured in milliseconds (ms). By comparing results, we can see a significant improvement for each optimized routine as shown in Table II.

TABLE II. STANDALONE TEST RESULTS.

Standalone test for each routine			
Optimized routine	**ref [ms]**	**opt [ms]**	**gain [%]**
png_read_filter_row_sub	1604	645	59.79
png_read_filter_row_avg	2177	783	64.03
png_read_filter_row_up	1587	782	50.72
png_read_filter_row_paeth_multibyte_pixel	2135	893	58.17
png_do_gamma	1785	645	63.87
png_do_scale_16_to_8	1169	756	35.33
png_do_read_interlace	2258	786	65.19
png_combine_row	1763	961	45.49
png_do_chop	2581	511	80.20
png_write_find_filter	1884	1024	45.65

Names of optimized routines are given in the first column followed by the results for original source code (ref) and for optimized routines (opt) that were measured in milliseconds. The last column presents gain in percent.

After these standalone measurements, we have done profiling measurements with *Oprofile* tool for all *libpng* tests, with and without optimized routines. Table III shows results of reducing the cycle consumption of each function separately for both original and optimized routines. First column presents names of optimized routines followed by the test that is executed. After that, we presented results for original source code (ref) and for optimized routines (opt). Finally, the last column shows gain in percent. All optimized routines have achieved performance increase as we expected Table III).

Results in Table III show that the percentage of functions in the graphics library has changed significantly.

Routines *png_read_filter_row_paeth_multibyte_pixel* and *png_write_find_filter* had the highest CPU time and their acceleration increased by 25%–50 % and about 15 %, respectively, depending on the test case.

Also, routines *png_do_chop* (acceleration about 75 %) and *png_do_read_interlace* (acceleration about 65 %) have the greatest acceleration, probably because of the large number of constants, additions and shifts used in their original processing that we have been able to improve using all of the above techniques.

Last step was to measure execution time for all tests with and without optimized routines. These tests showed satisfactory results also Table IV).

Results in Table IV show the total gain of all graphics library optimizations. Results depend on the input images and the processing performed in the tests. The best results are shown by the *pngtopng* test (acceleration about 25 %), because this test uses almost all optimized routines.

TABLE III. PROFILING RESULTS.

Existing *libpng* tests - profiling results for all optimized routines				
Optimized routine	**Test**	**ref [%]**	**opt [%]**	**gain [%]**
png_read_filter_row_sub	testpng	2.31	1.21	47.62
	pngtopng	1.85	0.86	53.21
	rpng2-x	6.53	3.25	50.23
	pngm2pnm	9.11	5.02	44.90
png_read_filter_row_avg	testpng	1.94	0.82	57.73
	pngtopng	1.38	0.65	52.90
	rpng2-x	5.53	2.23	59.57
	pngm2pnm	6.96	2.37	65.95
png_read_filter_row_up	testpng	0.27	0.22	19.63
	pngtopng	0.23	0.13	42.17
	rpng2-x	0.86	0.24	72.44
	pngm2pnm	0.63	0.41	34.12
png_read_filter_row_ paeth_multibyte_pixel	pngtopng	12.33	5.80	52.96
	rpng2-x	74.01	56.68	23.42
	pngm2pnm	76.22	56.48	25.89
png_do_gamma	pngtopng	2.83	1.51	46.64
png_do_scale_16_to_8	pngtopng	5.66	3.06	45.64
png_do_read_interlace	testpng	5.47	1.50	72.63
	pngtopng	5.80	4.74	18.34
	rpng2-x	25.07	8.45	66.28
	pngm2pnm	28.05	10.90	61.15
png_combine_row	testpng	1.56	1.29	17.03
	pngtopng	1.67	1.50	10.36
	rpng2-x	19.82	16.49	16.79
	pngm2pnm	9.29	7.40	20.38
png_do_chop	rpng2-x	8.43	2.16	74.34
	pngm2pnm	9.53	2.08	78.16
png_write_find_filter	testpng	85.40	71.67	16.08
	pngtopng	95.33	82.45	13.51

TABLE IV. TIME MEASUREMENT - OVERALL.

Existing *libpng* tests - overall gain				
Image	**Test**	**ref[s]**	**opt[s]**	**gain [%]**
BumbleBee_HedKase.png	testpng	x	x	x
	pngtopng	15.12	11.2	25.93
	pngm2pnm	14.52	13.61	6.28
iTunes.png	testpng	x	x	x
	pngtopng	2.16	1.95	9.72
	pngm2pnm	0.97	0.84	13.22
png4.png	testpng	6.78	5.41	20.28
	pngtopng	1.55	1.21	21.94
	pngm2pnm	6.88	6.87	0.15
lena_16g_lin.png	testpng	0.41	0.38	8.74
	pngtopng	0.27	0.26	4.07
	pngm2pnm	0.14	0.12	14.18
pnglogo-grr.png	testpng	8.26	7.61	7.88
	pngtopng	4.05	3.8	6.17
	pngm2pnm	0.88	0.71	18.68

The overall gain is lower than the gain of individual functions and profiling results due to compression and decompression routines from the *zlib* library that consume most of the CPU time and CPU cycles in the existing tests.

VI. CONCLUSIONS

In this paper, we have proposed embedded software code optimization of the graphic PNG library *libpng* on MIPS32 platform.

The *libpng* library optimizations for the MIPS architecture described in this paper show satisfactory results. Further enhancement of the execution speed of the PNG image processing algorithm on the MIPS platform can be achieved by introducing support for the MSA (MIPS SIMD Architecture) extension of the MIPS instruction set. MIPS MSA implements 128-bit wide vector registers that significantly increases the possibility of parallelization. In the meantime, we worked on optimization for the *zlib* library, but did not get satisfactory results for the MIPS DSP ASE instruction set. It is possible that the MSA instruction set would produce much better optimization results for the *zlib* library, and therefore the *libpng* graphic library that relies directly on *zlib*.

CONFLICTS OF INTEREST

The authors declare that they have no conflicts of interest.

REFERENCES

[1] W. Wolf, "A decade of hardware/software co-design", *IEEE Computer*, vol. 36, no.4, pp. 38–43, Apr. 2003. DOI: 10.1109/MC.2003.1193227.

[2] G. Berry, "Synchronous methodology for designing hardware, software and mixed embedded systems", in *Proc. of 17th International Conference* on *VLSI Design*, Mumbai, India, Jan. 2004, pp. 24–25. DOI: 10.1109/ICVD.2004.1260897.

[3] H. Mao, Z. Hu, L. Zhu, and H. Qin, "PNG file decoding optimization based embedded system", in *Proc. of 2012 8th IEEE International Conference on Wireless Communications, Networking and Mobile Computing*, Shanghai, China, Sept. 2012. DOI: 10.1109/WiCOM.2012.6478619.

[4] G. Randers-Pehrson, "A description on how to use and modify libpng", Oct. 2002. [Online]. Available: http://www.libpng.org/pub/png/libpng-1.0.3-manual.html

[5] *Effective DSP Programming using MIPS® DSP Application Specific Extensions*, Document no. MD00475, Jun. 2008. [Online]. Available: https://s3-eu-west-1.amazonaws.com/downloads-mips/mips-documentation/login-required/eEffective_programming_of_the_24ke_and_the_34k_core_families_for_dsp_code.pdf

[6] D. Sweetman, *See MIPS run Linux*, 2nd ed. Morgan Kaufmann publishers, 2007.

[7] *Efficient DSP ASE Programming in C: Tips and Tricks*, Document no. MD00485, Jun. 2011. [Online]. Available: https://s3-eu-west-1.amazonaws.com/downloads-mips/mips-documentation/login-required/efficient_dsp_ase_programming_in_c-tips_and_tricks.pdf

[8] *MIPS® Architecture for Programmers Volume IV-e: MIPS® DSP Module for MIPS32™ Architecture*, Document no. MD00374, Dec. 2014. [Online]. Available: https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00374-2B-MIPS32DSP-AFP-03.01.pdf

[9] *Programming the MIPS32® 74K™ Core*, Revision 02.13, June 04,2010, MIPS Technologies, Inc. 955 East Arques Avenue, Sunnyvale, CA 94085-4521.

[10] *An Independent Analysis of the MIPS Technologies MIPS32 74K Licensable Processor Core*, Berkeley Design Technology, Inc,, 2007.

[11] J. Levon, "OProfile Manual", Victoria University of Manchester, 2004. [Online]. Available: https://oprofile.sourceforge.io/doc/

[12] *GCC 4.5.1. Manual*, 2008 Free Software Foundation, Inc.

[13] *Five Methods of Utilizing the MIPS DSP ASE*, Document no. MD00783, Jun. 2011. [Online]. Available: https://s3-eu-west-1.amazonaws.com/downloads-mips/mips-documentation/login-required/five_methods_of_utilizing_the_mips_dsp_ase.pdf

[14] Objdump - Linux Manual Page. [Online]. Available: http://man7.org/linux/man-pages/man1/objdump.1.html

[15] J. C. Huang and T. Leng, "Generalized loop-unrolling: A method for program speedup", in *Proc. of 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology*, ASSET'99, Richardson, TX, USA, 1999. DOI: 10.1109/ASSET.1999.756775.