Optimisation of Multimedia Applications for the Philips Wasabi Multiprocessor System

Demid Borodin*, Andrei Terechko**, Ben Juurlink*, and Paulus Stravers**

*Computer Engineering Laboratory
Faculty of EEMCS, TU Delft
Mekelweg 4, 2628 CD Delft
The Netherlands

**Philips Research Laboratory
Prof. Holstlaan 4
5656 AA Eindhoven
The Netherlands

E-mail: D.Borodin@EWI.TUDelft.NL
Phone: +31 (0)6 41861436, Fax: +31 (0)15 2784898

Abstract— **Libavcodec is an open source software library that contains many different audio/video codecs. In this work, it is optimised and parallelised for the Philips Wasabi chip multiprocessor, which is currently being developed at Philips. Wasabi contains several DSPs and one or more general-purpose processors. The TriMedia-style optimisations (in particular utilising the SIMD custom operations) improve the performance of the MPEG2 and MPEG4 decoders by approximately 41% and 30%, respectively. Parallelisation of the video encoder achieves a linear speedup for up to 6 CPUs. Thereafter, it slightly levels off. Additional work is required to make libavcodec more scalable so that it can exploit more processors efficiently. In addition, an interface between the TriMedia(s) executing libavcodec and the general-purpose processors running host applications that use libavcodec is proposed. This interface enables applications to efficiently use libavcodec running on the TriMedias, without having to port the applications themselves to the TriMedia.**

Keywords— **libavcodec; TriMedia; multiprocessor; optimisation; VLIW architecture**

I. Introduction

The demand for computational power increases continuously in the field of digital multimedia. The power of dedicated multimedia processors working alone has become insufficient. Currently Philips considers utilising parallel multiprocessor architectures for multimedia processing. The advantages of dedicated multimedia processors combined with parallelisation should open new possibilities for computationally intensive multimedia tasks. In addition, Philips considers using open source software quickly optimised for the multimedia processors.

Libavcodec is an open source software library within the FFmpeg project [3]. It contains many different audio/video coders/decoders (codecs) and is known to be a very fast MPEG4 codec [18], [7], [2]. It has been optimised for several multimedia extensions including Intel's MMX [9], AMD's 3DNow [5], and others. Wasabi is an instance of the CAKE architecture [15], [16], a chip multiprocessor targeted at media applications which is currently being developed at Philips Research. It consists of several TriMedia [10] processors and one or more general-purpose processors which communicate via a shared level-2 cache. The TriMedia is a processor with a very long instruction word (VLIW) architecture that supports many media operations.

The goals of this work are (1) to port libavcodec to the TriMedia, (2) to improve performance by applying architecture specific optimisations, (3) to parallelise libavcodec to exploit multiple processors, and (4) to provide an interface between the TriMedia processor(s) executing libavcodec and the general-purpose processor(s) running application(s) that use libavcodec.

This paper is organised as follows.

In Section II we describe the difficulties encountered when porting FFmpeg to the TriMedia. FFmpeg exploits many features of the most recent GNU C compiler (gcc), whereas the TriMedia compiler accepts standard ANSI C. Hence certain C constructs had to be replaced by others and certain library functions had to be implemented.

Section III describes how libavcodec was optimised for the TriMedia. Among the optimisations that have been applied are conventional optimisations such as algebraic simplifications, techniques to increase instruction-level par-

allelism such as branch elimination, and restricted pointers to inform the compiler that there is no aliasing. In addition, we have used the custom operations to process several short data types simultaneously in a SIMD fashion. To measure performance a cycle-accurate simulator of the TriMedia has been employed. For benchmarking several high-resolution MPEG2 and MPEG4 video sequences have been decoded. The optimisations improve the performance of the MPEG2 and MPEG4 decoders by approximately 41% and 30%, respectively.

Section IV describes the parallelisation of the video encoder which involved changing the interface from POSIX pthreads to the TriMedia multi-threading API TM OSAL. An almost linear speedup for up to 6 processors is achieved. Additional work is required to make libavcodec more scalable so that it can exploit more processors efficiently.

Section V proposes an interface between libavcodec running on the TriMedia(s) and applications running on GPP(s). This interface enables applications to efficiently use libavcodec running on the TriMedias, without having to port the applications themselves to the TriMedia.

Finally, Section VI draws conclusions of this work and Section VII provides suggestions for future work.

II. Porting FFmpeg to the TriMedia

According to [11], the C programming language accepted by the TriMedia compiler *tmcc* is based on the following standards:

- American National Standard for Programming Languages–C, ANS X3.159–1989
- ISO/IEC 9899:1990
- Technical Corrigendum 1 (1994) to ISO/IEC 9899:1990
- IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE 754-1985

"ANS X3.159–1989" is the basis for the later ISO standard "ISO/IEC 9899:1990," which includes the prior standard entirely, with only minor editorial changes. Further in the text, the ISO C is meant by "ANSI C" or "Standard C".

There are also extensions to ANSI C in the TriMedia compiler, such as the concept of restricted pointers, as proposed by the Numerical C extensions group in the proposal X3J11/95-049, WG 14/N448 [6].

A. The C Language Changes Applied to FFmpeg

FFmpeg is written for the most recent GNU C compilers. It utilises many features that are not supported by the TriMedia compiler. Several examples are given below.

Structure assignments: given a structure definition such as

```
typedef struct {
    char  ch;
    int   i;
    float f;
} struct_type;
```

the structure assignment

```
struct_type struct_name;
struct_name =
    (struct_type) { 'a', 50, 0.5 };
```

must be replaced by

```
struct_type struct_name;
struct_name.ch = 'a';
struct_name.i  = 50;
struct_name.f  = 0.5;
```

Structure initialisation: for the same structure type as in the previous example, the initialisation

```
struct_type struct_name =
    { .f=0.5, .i=50 };
```

should be replaced by

```
struct_type struct_name =
    { 0, 50, 0.5 };
```

Declaration of arrays with non-constant dimensions: a declaration such as

```
int int_array[w];
```

where w is not a constant, should be replaced by a function call allocating memory dynamically:

```
int *int_array =
    (int*) malloc( w * sizeof(int) );
```

Finally, the attributes of the GNU C compiler are not supported by the TriMedia compiler. They should be carefully eliminated. For example, the attribute

```
__attribute__(unused)
```

can be simply removed since it is only used to let the compiler know that no warnings should be given if variables are unused. Normally the compiler would give a warning about an unused variable.

B. Unsupported Library Functions and Type Declarations

There are some C library type declarations used in the FFmpeg project that are not supported by the TriMedia Compilation System (TCS). To solve this problem, FFmpeg was extended with some include files, sometimes with minor changes, taken from Linux standard include directories.

Besides that, several functions used in FFmpeg are not present in TCS. FFmpeg was extended with the implementation of these functions. Due to the project time limitation, not all these functions were implemented completely.

Instead, simplified versions were implemented, based on some assumptions. For example, the functions *snprintf* and *vsnprintf* simply call the standard functions *sprintf* and *vsprintf* respectively, which are available in the TCS standard library. The argument controlling the maximum length of the produced string is ignored in both cases. This assumption turned out to be safe for FFmpeg.

C. Operating System Specific Features

There are some operating system (primarily Linux) specific features used in FFmpeg. The audio/video grabbing implementation uses them, for example. These features were disregarded, since they are out of scope of this work.

D. Other Porting Issues

When porting FFmpeg to Philips TriMedia, the following aspects had to be also taken into account:

- The compiler had to be switched to Little Endian byte ordering since the default is Big Endian and FFmpeg assumes Little Endian.
- The TriMedia hardware does not support double-precision floating-point (64-bit) operations. As FFmpeg/libavcodec needs it, the compiler is switched to double-precision floating-point simulation mode using the *-fp64* compiler command line option. Unfortunately, this has a negative effect on performance since the simulation is done in software.
- The first versions of the TriMedia do not support unaligned load and store operations. As they are used in libavcodec, the environment had to be changed to the TriMedia 2270. That is, TCS 4.5 was used instead of TCS 4.2 which did not support the TriMedia 2270 architecture. The compiler command-line option *-target tm2270Minimal* is used to set the target architecture. The employed simulator is *tm2270sim* instead of *tmsim*.

After the porting is finished, FFmpeg runs on the *tm2270sim* simulator. The default memory size assumed by the simulator is 8 MB. This is not sufficient for the experiments, so the simulator was explicitly instructed to assume 128 MB of memory space.

III. Optimisation of Libavcodec for the TriMedia

A. Introduction

The measure to determine a program's performance used in the work is the number of clock cycles required to perform a task. The goal is to minimise the execution time of a program, i.e. to minimise the number of clock cycles as much as possible.

Taking into account the VLIW architecture of the target platform, the ideal case is when the processor is saturated. That is, only the processor's computing resources (the number and configuration of the available functional units) limit the performance of an application. The TriMedia VLIW processor allows to issue 5 operations in one instruction (clock cycle), and ideally the amount of instruction level parallelism (ILP) should be sufficient to execute 5 useful operations every clock cycle.

Unfortunately in an unoptimised application there are many factors that affect the ILP negatively. The dependencies of the operations on each other force the compiler to schedule a lot of NOPs (No OPerations) instead of useful operations. A particular problem for a TriMedia program is branches. The compiled code is organised in so-called decision trees, and if there is not enough plain code between branches, a large part of the decision trees can consist of NOPs only, introducing many "wasted" instructions.

B. Which Parts to Optimise?

Since libavcodec is a very large project, it is impossible to optimise it completely in a limited time period. Some parts to optimise have to be chosen.

The TriMedia simulator provides statistics about the time consumption of the functions in FFmpeg. The TriMedia profiler was used to obtain the most expensive functions of libavcodec in terms of execution time. These are the most computationally-intensive functions from the codecs, for example the functions implementing the IDCT (Inverse Discrete Cosine Transform) algorithm.

The experiments were performed with input media files encoded by different codecs. The most computationally-intensive functions common to the largest number of the most important (such as MPEG2 and MPEG4) codecs were chosen for optimisation. For example, the function `idctSparseColPut` is one of the most time-consuming functions in the decoding process of MPEG2, MPEG4 and MJpeg codecs. In MPEG2 decoding this function takes 9.14% of the total number of execution cycles. Several other functions implementing the IDCT algorithm also consume a significant portion of the total execution time.

C. Optimisation Methods

The TriMedia compiler provides a large variety of automatic optimisations that can be applied. Unfortunately, the compiler's ability is restricted, and certain optimisations have to be applied manually. The compiler can schedule plain code without branches and dependencies most efficiently. One of the main goals of manual code optimisation is, therefore, to eliminate branches and dependen-

cies as much as possible. Below an example is given of how branches can be eliminated in several functions of the IDCT algorithm, and a technique to avoid false dependencies (restricted pointers) is described.

The TriMedia issues 5 operations every clock cycle. These operations can be elementary, RISC-like, as well as custom operations. The TriMedia custom operations are similar to MMX operations. They process several small data type packed in a single 32-bit register in parallel. An example of such a custom operation performing several arithmetic operations is *ifir16* which is a sum of products of signed 16-bit halfwords (see Figure 1):

$$\textit{ifir16(a,b)} = a_{high} \times b_{high} + a_{low} \times b_{low}$$

where a_{high} (b_{high}) and a_{low} (b_{low}) are high and low 16-bit signed half-words of the 32-bit operands a and b. The latency of the *ifir16* operation is three clock cycles, the same as that of the *imul* operation (32-bit integer multiplication).

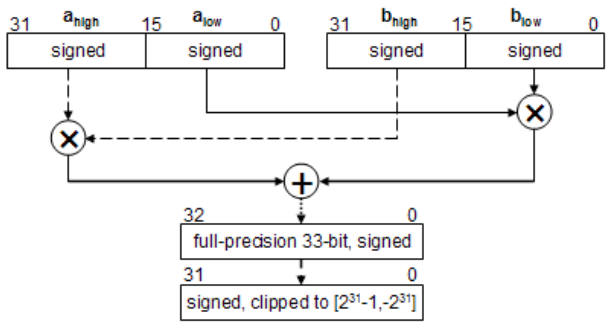

Fig. 1
ifir16 TRIMEDIA CUSTOM OPERATION

The custom operations are specialised for multimedia algorithms and are implemented in hardware, providing a large performance improvement. The current TriMedia compiler is not able to compile to custom operations, so a programmer has to insert them manually.

Below is an example of optimisations based on custom operations applied in libavcodec. It is taken from the function `idctSparseColPut`, one of the most time-consuming functions in DVD (MPEG2), DivX (MPEG4), and MJpeg decoding:

```
a0 += W2 * col[8*2];
if (col[8*4])
    a0 += W4 * col[8*4];
if (col[8*6])
    a0 += W6 * col[8*6];
```

This code has been replaced by:

```
temp = IFIR16( PACK16LSB(W2,W4),
               PACK16LSB(col[8*2],
               col[8*4]) );
a0  += IFIR16( PACK16LSB(temp,W6),
               PACK16LSB(1,col[8*6]) );
```

The *ifir16* and *pack16lsb* TriMedia custom operations are used here. The *pack16lsb* operation packs the two least significant half-words of its input operands into its destination (32-bit) operand.

```
IFIR16( PACK16LSB(W2,W4),
        PACK16LSB(col[8*2],col[8*4]) )
```

is equivalent to

```
W2 * col[8*2] + W4 * col[8*4]
```

Thus, instead of 2 integer multiplications (which take $2 \times 3 = 6$ clock cycles) and one addition (1 clock cycle) everything is done by 2 packing operations ($2 \times 1 = 2$ clock cycles) and one *ifir16* operation (3 clock cycles).

In the original code given above, the expensive multiplication operation is avoided if one operand is zero. This technique assumes that the multiplication operation is much more expensive than a comparison and/or conditional branch. For the TriMedia, this technique is not applicable, because the branches can result in a very bad schedule causing a lot of wasted clock cycles. Instead, the branches were eliminated completely without affecting the semantics, by adding something multiplied by zero.

This optimisation was applied to several functions in the IDCT algorithm, for several variables. As a result, the execution time of the function calling them decreased by 17%.

Restricted pointers help to avoid false dependencies, allowing the compiler to make assumptions that are not safe in the general case. Consider the following example:

```
int func( char *a, char *b, char *c )
{
    c[0] = a[0] * b[0];
    c[1] = a[0] * b[1];
}
```

In the general case the pointers can refer to the same or overlapping memory locations. For example, c can be equal to a. If so, the second assignment statement can be data dependent on the first one, since $c[0]$ equals $a[0]$. It means that the operations of the two statements cannot be executed in parallel, scheduled in the same VLIW instruction. To make sure that the pointers are not aliased to each other, the compiler must perform an inter-procedural analysis of the whole application. However, if the programmer is sure that the pointers refer to distinct arrays and do not alias, he can inform the compiler by declaring the pointers as restricted, by using the keyword *restrict* after the type specification.

File ID	*Before optimisations*		*After optimisations*		Performance improvement (%)
	Cycles ($\times 10^3$)	Frequency required (MHz)	Cycles ($\times 10^3$)	Frequency required (MHz)	
01	5046	504.6	3077	307.75	**39**
02	6419	641.92	3752	375.15	**42**
03	6347	634.73	3676	367.57	**42**
04	6784	678.45	4007	400.74	**41**
05	5587	279.37	3912	195.62	**30**
06	2992	299.16	2079	207.87	**31**
07	3192	319.25	2220	222.01	**30**
08	3498	349.79	2372	237.22	**32**
09	2347	234.67	1680	168.02	**28**
10	3518	351.77	2425	242.54	**31**
				Average:	**34.6**

TABLE I
PERFORMANCE BEFORE AND AFTER OPTIMISATION

Except the TriMedia-specific optimisations described above, algebraic simplifications, loop unrolling and function inlining have been applied to libavcodec. Function inlining gives considerable advantages: the call and return operations are eliminated, the code becomes less fragmented (plain) giving more possibilities for the scheduler to do a good job, and it reduces the number of decision trees since each function invocation adds a decision tree. However function inlining and loop unrolling should be used carefully when optimising for the TriMedia. They increase the number of instructions, which may affect the instruction cache, and increase the number of memory stall cycles.

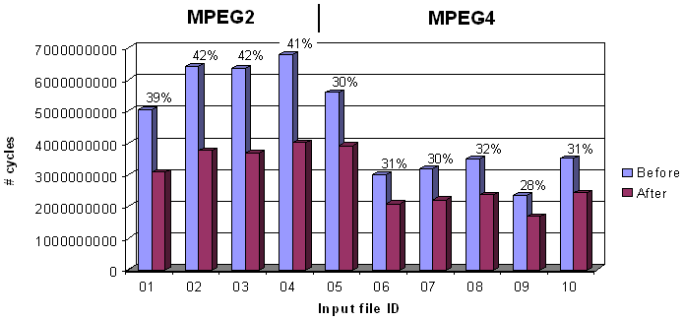

Fig. 2
PERFORMANCE BEFORE AND AFTER OPTIMISATION

D. Experimental Results

The performance is measured in terms of the number of clock cycles needed on the TriMedia to perform a task. We focus on the task of decoding video and audio streams in a file. Several video files with different properties are used. First a video file is processed using the "original" libavcodec, that is libavcodec as it was right after porting was finished, but before applying manual optimisations. Then the same video file is processed on libavcodec as it is after the optimisations. Only manual optimisations are considered. The difference in speed is called the *performance improvement*. If a video file processing is finished in N clock cycles on unoptimised libavcodec and in M cycles on the optimised one, the performance improvement (in %) is calculated as follows:

$$performance\_improvement(\%) = (1 - M/N) \times 100$$

The value defines how much faster the optimised libavcodec processes a file than the unoptimised one. To obtain the number of clock cycles needed for performing a task, it is simulated with the *tm2270sim* TriMedia simulator. The simulator is given a parameter defining the memory of 128 MB. For benchmarking FFmpeg performs decoding of video and audio streams in 4 MPEG2 and 6 MPEG4 files and stores them in raw video (yuv420p, resolution 160×128, 25.00 fps) and audio (WAVE) format. Each MPEG2 file has MPEG Audio Layer 2 audio stream, the video stream of the resolution 720×480 or 720×576, and the frame rate 30 frames per second (fps). Each MPEG4 file has MPEG Audio Layer 3 audio stream, the video stream of the resolution 640×480, and the frame rate 25 fps. The run time includes the I/O overhead.

In Table I the execution time (in clock cycles) needed by libavcodec to decode the video files is given, together

with the processor clock frequency required to perform the task in real time. The table shows the results before and after applying the manual optimisations to libavcodec. The performance improvement is visualised in Figure 2. On average, the MPEG2 decoder became approximately 41% faster and MPEG4 30% faster.

One of the most effective automatic optimisations applied was function inlining. When the inline level was set to default, the TriMedia compiler ignored the *inline* keyword in the declarations of many small functions. It was obvious from the profiling results, they included the functions that were supposed to be inlined (these functions did not have to appear in the function list after the compilation). The inline level has been set to the maximum, and after that FFmpeg performed 17.2% faster in average.

Another considerable contribution to the performance improvement (7.5% for MPEG2 and 3% for MPEG4 decoding) was given by the set of optimisations applied to the unoptimised C version of the IDCT algorithm. These optimisations mainly involved the use of SIMD TriMedia custom operations to increase the instruction level parallelism, branch elimination, and the use of restricted pointers. An example of the optimisations applied to the functions implementing the IDCT algorithm is given in Section III-C.

A tendency was noticed that usually an optimisation gives more performance improvement if it is applied together with some other optimisations. In other words, if an optimisation A gives $a\%$ and an optimisation B provides $b\%$ performance improvement of the unoptimised code, than applying both optimisations A and B gives a (possible) performance improvement which is more than $(a + b)\%$. This could be explained by a better scheduling achieved because of the two optimisations.

The MPEG4 decoder is less affected by the optimisations applied to the IDCT algorithm than MPEG2. The average execution time decrease after applying the optimisations to the IDCT algorithm is 7.5% for MPEG2 and 3% for MPEG4 decoder. This affects the results of all the applied optimisations: the MPEG2 decoder became on average 41%, and MPEG4 – 30.3% faster.

E. Automatic Transformation of MMX to TriMedia Code

In order to make porting of MMX-optimised software to the TriMedia fast and easy, Philips is developing a tool that automatically translates MMX macros to TriMedia-compilable code. The MMX-to-TriMedia tool is a set of C header files defining the MMX macros. A macro converts one MMX instruction to one or more TriMedia instructions and/or custom operation(s) trying to cause the least possible overhead in the execution time. Since in software the MMX code is often integrated into inline assembly code written for the x86 architecture, Philips is developing another tool that converts inline assembly in Intel's syntax to macros. This tool produces regular assembly macros as well as MMX macros, so the MMX-to-TriMedia tool also supports some basic x86 assembly instructions.

Libavcodec contains, among others, MMX optimisations. Most of the MMX code is in the form of GNU inline assembly. To employ the MMX-to-TriMedia tool for libavcodec, the GNU inline assembly syntax first had to be converted to Intel inline assembly and then to C code with MMX macros. First, the code is compiled on Windows using MSYS and MinGW[1]. After that the Windows executable file is disassembled with the command *objdump -d -mi386:intel* which produces Intel assembly code. Then the functions in libavcodec with GNU inline assembly are replaced with the corresponding disassembled code. When changing the GNU inline assembly code to the disassembled Intel's code, care must be taken of jumps, global and local variables. The absolute addresses should be changed to labels, constructs of the form *[ebp-X]* must be substituted by the names of the function's arguments, and constructs of the form *[ebp+X]* should be changed to the names of local (automatic) variables or eliminated. Currently these changes can only be made manually. When the sources with the Intel inline assembly are ready, they are processed with the tool that converts each instruction in the assembly code into a macro. Finally the code with macros can be compiled using the MMX-to-TriMedia tool.

Converting MMX code to TriMedia code did not succeed. At first, only the functions of the IDCT algorithm were compiled using the MMX-to-TriMedia tool to check the functionality and to compare the performance improvement obtained with that achieved by manual optimisation. The speedup given by the tool was comparable and sometimes even larger than that achieved by manual optimisation, but the output video stream was damaged. It is likely that the problem appeared while transforming the GNU inline assembly to macros. But since the MMX to TriMedia tool is in the development stage and has not been thoroughly tested, the problem can also be improper functionality of this tool. Discovering the reason(s) for the malfunction would take too much time, so further work with the MMX-to-TriMedia tool was cancelled.

The MMX-to-TriMedia tool introduces a certain overhead because there is no one-to-one match of MMX instructions to those of the TriMedia. Some MMX instructions have to be expanded into non-optimal sequences of TriMedia instructions. In other words, in theory a manu-

[1] MSYS and MinGW [1] provide a Linux environment with GNU tool sets allowing compilation of some Linux software on Windows.

ally optimised code must perform better than MMX code compiled with the tool. But in the case of libavcodec the IDCT algorithm implemented in C differs from that implemented using MMX. Hence, it is not correct to compare the performance improvement achieved by the conversion tool to that given by the manual optimisation.

If the problem in the IDCT code can be fixed without affecting the execution time, the MMX-to-TriMedia tool provides a good opportunity to improve the execution time of applications that have already been optimised for MMX.

Unfortunately, the process of transforming from GNU to Intel inline assembly took much more time and effort than expected. It appeared to be very time-consuming and error-prone and we question if it is better than to apply some manual optimisations spending the same amount of time.

In our opinion, the MMX-to-TriMedia tool is very useful for quick optimisation for the TriMedia, if MMX-optimised code in the form of macros or Intel inline assembly is available. If the conversion tool also supports GNU inline assembly, it will solve the problem of GNU to Intel inline assembly conversion.

IV. Parallelisation of Libavcodec for Wasabi

The Wasabi chip contains several CPUs. Although the final design has not been decided, it is anticipated that most processors will be TriMedias. To exploit the computational power of multiple TriMedias, libavcodec must be parallelised to distribute the workload among several processors.

In the original libavcodec the video encoding process is parallelised using data parallelism. Each thread executes the same code but on a different piece of data (for example, different areas of a video frame). It is implemented using the POSIX Pthreads [8] and Windows threads [4] interfaces. However, Wasabi supports the TriMedia Operating System Abstraction Layer (TM OSAL [17]) API. Therefore, a new interface for TM OSAL was created. The Win32 threads interface was chosen as a base for it.

The creation of TM OSAL interface involved a change of library function calls in the Windows interface by equivalent calls for TM OSAL and some more changes when an equivalent function could not be found. A new semaphore was created to signalise that a task has finished its work.

Below the experimental results are presented. They consisted of encoding video and audio from raw format. Video is encoded in MPEG4 and audio – in MPEG Audio Layer 2 format. The sample is 5 seconds long, the resolution is 720×480, and the frame rate is 29.97 frames per second. The bitrate is 2000 kbit/s.

The experiments show that the highest speedup is achieved when the number of threads created by FFmpeg is equal to the number of CPUs (TriMedias) in Wasabi. Hence below the number of CPUs determine the number of threads.

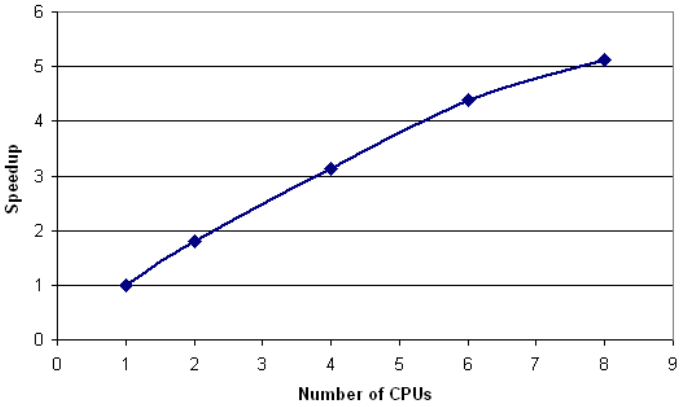

Fig. 3
Speedup as a function of the number of CPUs.

Table II shows the execution time for different numbers of processors. Figure 3 depicts the speedup as a function of the number of CPUs. It can be seen that the speedup is almost linear for up to 6 processors. After that it slightly levels off. But the scalability is not very impressive: on 2 CPUs the speedup is 1.8, on 4 CPUs it is 3.1, and finally on 8 CPUs it is only 5.1.

Number of CPUs	Number of cycles (millions)	Frequency needed for real-time processing
1	21 626	4.3 GHz
2	12 005	2.4 GHz
4	6 906	1.4 GHz
6	4 933	987 MHz
8	4 203	841 MHz

TABLE II
Number of cycles and frequency required to encode video and audio for different numbers of processors.

To explain this behaviour, Figure 4 depicts the thread activity during the encoding of one video frame. The rows represent the tasks (threads), the horizontal bars show when a processor was performing the task. A bar's colour indicates which CPU was executing the corresponding task. The top row represents the main thread *ROOT*, the next seven rows named as *IDLn* (where *n* is a number) are "idle" threads meaning that the processors were idle, and the following eight rows represent the tasks (threads) created by libavcodec. The figure shows thread activity while encoding a single video frame only, but the behaviour of the entire application is similar, except for the starting

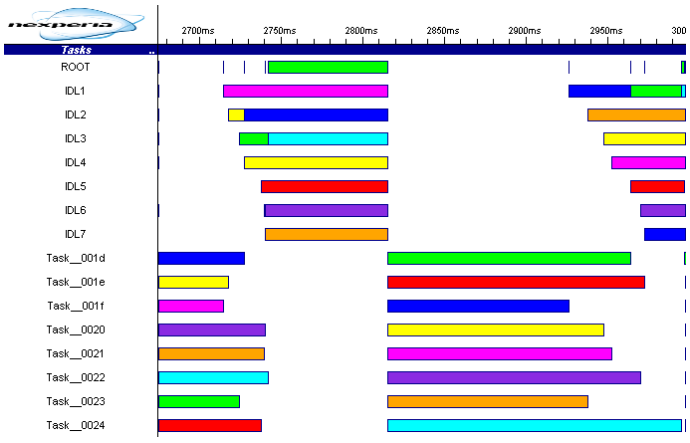

Fig. 4

THREAD ACTIVITY DURING THE ENCODING OF ONE VIDEO FRAME ON 8 PROCESSORS (THE IMAGE IS PROVIDED BY THE WASABI PERFORMANCE ANALYSIS TOOL TIMEDOCTOR).

point where the main thread performs some extra work (reading the input file etc.).

Figure 4 clearly shows that there are two steps in the encoding of a single frame that have been parallelised, i.e., the encoding algorithm is divided into two computationally intensive parts that have been parallelised. The first part is the motion estimation algorithm for the P- and B-frames and the algorithm determining the spatial complexity for the frame rate control for I-frames. The second part performs the actual encoding. Between these two parts the application (the main thread) merges the contexts after the first part and performs some other work, including preparing the MPEG4 picture header. This work is not parallelised and this is the reason why there is a gap between the two parallelised parts. In this gap only the main thread is performing useful work on a single CPU. The other processors are idle. This is also the main reason for the lack of scalability. In addition, the work is slightly unbalanced: some threads finish earlier than others. The second parallelised part is performing quite well. Some CPUs finish their work earlier than others and stay idle, but not for very long.

Between the frames the main thread needs some time (very little compared to that inside a frame) to process one or more audio frames and switch to the next video frame.

V. LIBAVCODEC INTERFACE

The Wasabi chip will contain several TriMedias and one or more general purpose processors (GPPs). If the TriMedias run libavcodec and the GPP(s) with Linux application(s) using it (as depicted in Figure 5), it gives a wide

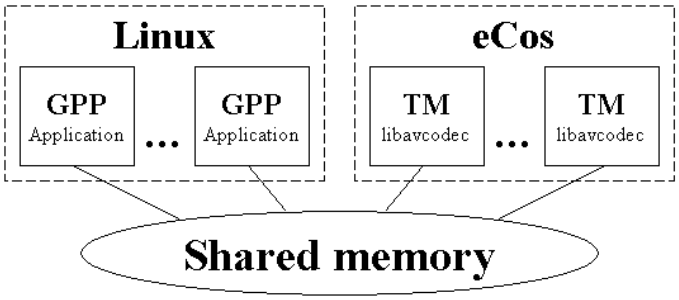

Fig. 5

MAPPING OF TASKS ON PROCESSORS IN WASABI CHIP

range of possibilities to speed up the system.

The idea of this part of the project is to develop a mechanism to communicate between applications running on Linux (on GPPs) and libavcodec on TriMedia(s). It must support multitasking, i.e., each GPP should be able to run several applications using libavcodec simultaneously. The user applications utilising libavcodec should be kept unchanged, it is certainly not desirable to require an application to deal with the interface. For an application, libavcodec is a dynamically loaded library, and it should remain the same.

A. System Structure

Since the interface provided to user applications should not be changed, a library called libavcodec on GPP with Linux can be made that is in fact a “wrapper” providing interface to TriMedia processors as shown in Figure 6.

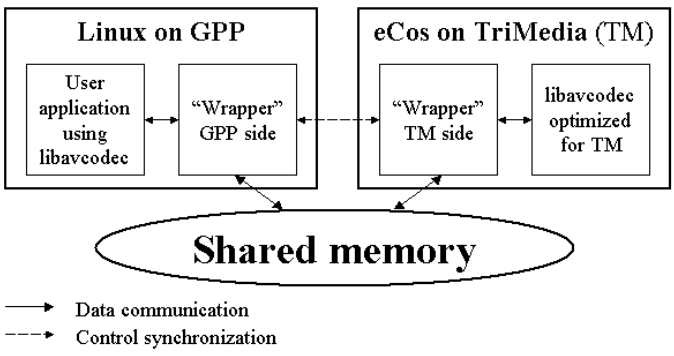

Fig. 6

LIBRARIES - “WRAPPERS”

This “wrapper” library must provide a set of libavcodec application programming interface (API) functions. When a function is called by an application, the “wrapper” library on the GPP side must pass the function name and its

parameters to the “wrapper” on the TriMedia side, which calls that function in the real libavcodec library. The result is passed by the “wrapper” on the TriMedia side to that on the GPP side, and the latter returns it to the application.

The Wasabi chip will support interrupts. Remote procedure calls can be implemented using them. The GPP-side “wrapper” raises interrupt when a function call happens, the TriMedia-side “wrapper” does it when a function returns. When an interrupt is sent, a special data structure in the shared memory is filled by the sender for the receiver.

The structure of the data passed from GPP-side “wrapper” to that on the TriMedia side depends on several system properties. If the “wrappers” on both sides can address memory in the same way (share the Linux page tables), than the communication data structure is very simple since it does not have to transfer the data pointed by the functions’ arguments. If the data must be transfered, than the communication data structure depends on the communication buffer size. If there is a small communication buffer which cannot hold all the data pointed by the arguments, all the data must be transfered through a FIFO. The communication data structure should contain in any case the function name, arguments of the integral type, identifier of the calling process (if the system is multitasking), and synchronisation fields. If the virtual address space is shared between the GPP and TriMedia, the function arguments of the pointer type can be passed as they are. Otherwise the data must be transfered in one or more data buffers. The sender writes the data into the first, then into the second buffer, etc. When all the buffers are filled in, the sender rewrites the buffers in the same order, first checking if the receiver has read the buffer it is going to write into. Splitting the data into several buffers is used to speed up the data transfer: while the sender is writing the second buffer, the receiver can read the first one etc. If there is only one large buffer, the receiver has to wait while the sender writes it completely before reading it.

In the case if all the data must be transferred from GPP to the TriMedia, the “wrappers” are responsible for managing all the pointers, which is a very complicated task. It must convert the Linux virtual addresses into physical addresses understandable for the TriMedia side and convert them back when receiving data from the TriMedia. These addresses can be not only the pointers among a function’s arguments, but also in the data structures pointed to by them. Some structures contain function pointers that can be invoked from libavcodec, and than the wrappers must be able to organise it. Fortunately, all the function pointers needed seen so far have default implementations inside libavcodec and thus this problem can be avoided by passing *NULL* instead of them (but than the application must be changed if it uses these function pointers).

A solution to these complications is that both the “wrapper” sides work in the same address space, and only pointers are passed from GPP to TriMedia. This is possible only if both the GPP and TriMedia sides run Linux which works with the same (shared) page tables.

B. Memory Allocation

If the TriMedia side does not run Linux which is not currently the case, special care must be taken when transferring data from the GPP to TriMedia side. There is one major problem with memory allocation for data transfer: on the GPP-side, the “wrapper” operates on a Linux system. Usually applications running on Linux do not have to bother with physical memory since Linux provides a sophisticated memory management mechanism which allows any application to have its own large linear virtual memory space. However, the “wrapper” on the GPP cannot pass a virtual address to the “wrapper” on the TriMedia, since it is meaningless for the latter: to translate a virtual address to a physical one, it has to access the Linux page tables. The “wrapper” on the GPP-side must provide physical addresses, and if the buffer it allocates is larger than one page, it must occupy contiguous pages in physical memory, which is usually not the case when the space is allocated in Linux with *malloc* since the memory is fragmented. In addition, Linux should not swap the allocated page(s).

B.1 “Wrapper” Structure on the GPP Side

The GPP-side “wrapper” needs to work with kernel functions that are available only for kernel modules. The proposed structure is shown in Figure 7:

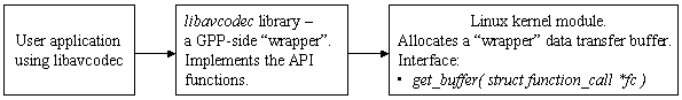

Fig. 7
GPP-SIDE “WRAPPER” STRUCTURE

The Linux kernel module is loaded with *insmod* command (see [13] and [14]). It allocates the buffer of the required size with *kmalloc* (can allocate up to 128 KByte) or *get_free_pages* (can allocate maximum around 2 MB on most systems with Linux 2.0 and later) function (see [12]). The GPP-side “wrapper” libavcodec library calls the interface function *get_wrapper_buffer_addr* (it must be checked that the name does not contradict with other global Linux kernel symbols). The module returns the virtual address

of the buffer that can be used by the library to transfer the data.

As the system works, the memory becomes more and more fragmented; it is possible that there is no contiguous space in memory of the size the module requests and Linux cannot solve this problem by swapping/rearranging the allocated pages. In this case the module will fail to allocate memory. To avoid this situation, the memory must be allocated as early as possible after the system boot.

If media processing with libavcodec is one of the major tasks of the system, the different technique for memory allocation is advised. To get a large buffer of physically contiguous memory, we can allocate it by requesting memory at boot time. Allocation at boot time is the only way to retrieve consecutive memory pages without the limits on the size imposed by *get_free_pages*, both in terms of maximum allowed size and limited choice of sizes. To use this technique, the module must be linked directly in the kernel image; that is, the kernel must be rebuilt with it. With the kernel 2.3.23 and later, the functions *alloc_bootmem_pages* and others can be used. The way appropriate also for older kernels is reserving the top of RAM by passing the *mem=* argument to the kernel at boot time, reserving a part of memory from kernel's usage. The memory allocated in this way can never be freed until the system reboot (see [12], chapter 7 and 13 for details).

VI. Conclusions

The process of porting libavcodec to the TriMedia took (too) much time. Many other open-source applications, which may also be ported to the TriMedia, are also written for the latest GNU compiler and would have the same porting problems. These problems can be solved by extending the GNU compiler with a back-end for the TriMedia. This solution is better than extending the front-end of the TriMedia compiler to support the new features of the GNU compiler because it requires less effort. The back-end of the GNU compiler has to be extended only once. After that, new extensions of the front-end of the GNU compiler will not require additional effort on the back-end for the TriMedia. On the contrary, if the approach to extend the front-end of the TriMedia compiler to support the syntax of the GNU compiler is taken, it should be updated every time the GNU compiler's front-end is enhanced with new features. This becomes more and more expensive in time. Another approach is to create a "C-to-C" compiler which will produce code supported by the TriMedia compiler from the code for the GNU compiler. This solution has the same drawback as extending the front-end of the TriMedia compiler: the "C-to-C" compiler must be updated every time the GNU compiler is extended. Besides that, a "C-to-C" compiler will produce a more or less unreadable C code for the TriMedia compiler. If Philips wants to incorporate the TriMedia extension into an open-source project, i.e. submit the TriMedia extension to the project maintainers and propose to include them in the project, an unreadable code can prevent the project maintainers from accepting it. This conclusion is based on the requirements for the code style that are posed by the maintainers of libavcodec for those who want to submit some extensions.

The manual optimisations applied demonstrate considerably good results. When the time constraints permit, it is always advisable to apply manual optimisations to the most time-consuming parts of applications.

Libavcodec supports many optimisations. As many other multimedia applications (for example the XviD codec), libavcodec supports MMX optimisations. The automatic MMX-to-TriMedia conversion tool described in Section III-E tries to exploit the existing MMX optimisations for a fast and easy optimisation for the TriMedia. The results of this work show that this approach is not fast and easy with the current MMX-to-TriMedia tool in case the MMX optimisations are in the form of GNU inline assembly. The reason is that the MMX-to-TriMedia tool expects the instructions in the form of macros which can be obtained with another tool, but the latter accepts only the Intel inline assembly syntax. Manual code transformation from GNU to Intel inline assembly syntax is very time-consuming and error-prone. This problem can be solved by extending the inline assembly to macros conversion tool with GNU inline assembly support. Furthermore, these tools can be incorporated into the TriMedia compiler to make the conversion process transparent to the user.

The parallelisation process performed in this work proved that Wasabi is well prepared for a quick parallelisation of applications that already have been parallelised. With the advantages of hyperthreaded processors and chip multiprocessors, it is expected that more and more open source applications such as libavcodec will support multithreading. The only way to improve this feature of Wasabi even further is to extend its libraries with support for POSIX Pthreads, because parallelised open-source applications for Linux will most probably utilise POSIX Pthreads.

The interface suggested in this work is considerably complex. The primary reason of the complexity is that the application runs in the Linux domain and libavcodec works in the TriMedia domain which uses another, less sophisticated operating system. Because the Memory Management Unit (MMU) creates the virtual memory address space and the memory segmentation on Linux, the interface must copy all data from the Linux virtual address

space to contiguous buffers in the physical memory. This copying has a large negative effect on the performance. Furthermore, it complicates the interface significantly because of nested pointers in the data structures that should be copied. The only solution to these problems is to run Linux on the TriMedias.

VII. Future Work

As future work on optimisation of libavcodec for Wasabi, further TriMedia-style optimisations are highly recommended. The optimisations targeted at the increase of the instruction level parallelism are expected to significantly improve the performance of the tasks running on the TriMedia processors, because currently most parts of libavcodec do not employ the features offered by the VLIW architecture and a large amount of a useless code (NOPs) is executed.

The scalability of libavcodec's video encoder execution time when the number of CPUs is increased can be improved by parallelising the work that is performed by the main thread while the other tasks are idle. Another approach is to improve this work using the TriMedia-style optimisations so that its effect is less important. Furthermore, the video and audio decoders and audio encoder can be parallelised. The potential of this parallelisation should first be investigated, if it makes sense in terms of the introduced synchronisation and communication overhead in the system.

Finally, the interface between the general-purpose processor and the TriMedia can be designed in a greater detail and implemented.

References

[1] *MinGW - Home*, http://www.mingw.org/.

[2] *MPEG Pointers and Resources*, http://mpeg.org.

[3] *The FFmpeg project*, http://ffmpeg.sourceforge.net/.

[4] *Windows threads*, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/multiple_threads.asp.

[5] Inc. Advanced Micro Devices, *3DNow! Technology Manual*, (2000), Available at http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/21928.pdf.

[6] Bill Homer, *Restricted Pointers*, (1995), WG14/N448, X3J11/95-049. Available at ftp://ftp.dmk.com/DMK/sc22wg14/c9x/aliasing/.

[7] Ze-Nian Li and Mark S. Drew, *Fundamentals of Multimedia*, Oct 2003.

[8] Andrae Muys, *POSIX Pthreads Tutorial*, http://www.cs.nmsu.edu/~jcook/Tools/pthreads/pthreads.html.

[9] Alex Peleg, Sam Wilkie, and Uri Weiser, *Intel MMX for Multimedia PCs*, Communications of the ACM **40** (1997), no. 1, 24–38.

[10] Philips, *TM1100 Preliminary Data Book*, March 1999.

[11] ———, *TriMedia Compilation System 4.5 - User Manuals*, vol. 3 - Compilation Tools, Sept 2004.

[12] Alessandro Rubini and Jonathan Corbet, *Linux Device Drivers*, second ed., June 2001, Available at http://www.xml.com/ldd/chapter/book/.

[13] Peter Jay Salzman, Michael Burian, and Ori Pomerantz, *The Linux Kernel Module Programming Guide*, 2.6.1 ed., Sept 2005, Available at http://tldp.org/LDP/lkmpg/2.6/html/.

[14] ———, *The Linux Kernel Module Programming Guide*, 2.4.0 ed., Sept 2005, Available at http://tldp.org/LDP/lkmpg/2.4/html/.

[15] Paul Stravers and Jan Hoogerbrugge, *Homogeneous Multiprocessing and the Future of Silicon Design Paradigms*, Proc. of International Symposium on VLSI Technology, Systems, and Applications (VLSI-TSA), Apr 2001.

[16] Jos van Eijndhoven, Jan Hoogerbrugge, M.N. Jayram, Paul Stravers, and Andrei Terechko, *Cache-coherent heterogeneous multiprocessing as basis for streaming applications*, vol. 3, pp. 61–80, 2005.

[17] Arjan van Lankveld, *User Documentation – OSAL*, Philips Semiconductors RTG/PID/2003/0086 (2003).

[18] John Watkinson, *The MPEG Handbook. MPEG-1, MPEG-2, MPEG-4*, second ed., Nov 2004.