Motion Estimation Performance of the TM3270 Processor

Jan-Willem van de Waerdt*+
Gerrit A. Slavenburg^
Jean-Paul van Itegem*
Stamatis Vassiliadis+

*Philips Semiconductors
San Jose, CA, USA
{Jan-Willem.van_de_Waerdt,
JeanPaul.vanItegem}
@philips.com

^NVIDIA Corporation
Santa Clara, CA, USA
GSlavenburg@nvidia.com

+TU Delft
Electrical Engineering Dept.
Delft, The Netherlands
Stamatis@dutepp0.et.tudelft.nl

ABSTRACT

Motion estimation constitutes a significant computational part of video standards such as MPEG2, MPEG4, and H264/AVC. This paper evaluates the performance of a motion estimation algorithm on the TM3270, a low-cost media-processor. In order to improve performance, the TM3270 processor provides architectural enhancements over previous TriMedia processors. We quantify the speedup of the proposed *new operations* to motion estimation performance. We show that the new operations incorporated in the TM3270 improve performance by a factor between 3 and 4. Furthermore, we quantify the speedup of *data prefetching*. We show that prefetching can improve performance up to 30%. By applying all TM3270 architectural enhancements, we show that standard resolution motion estimation can be performed in less than 5% of the available processor performance.

Categories and Subject Descriptors

D.1.1 [**Processor Architectures**]: Single Data Stream Architectures – *pipeline processors, RISC/CISC, VLIW architectures*. I.4.0 [**Image Processing and Computer Vision**]: General.

General Terms

Design, Measurement, Performance.

Keywords

Media processor, motion estimation, software implementation.

1. INTRODUCTION

Media-processors are used in the domain of video processing. Their programmability allows for a flexible implementation of video processing algorithms. When enough computational performance is available, they provide an interesting alternative to fixed dedicated hardware solutions. The time-to-market of a programmable solution, from algorithm conception to market introduction, can be kept short, since no lengthy hardware design cycle is required. Furthermore, a single programmable platform may address multiple markets. As a result, its development costs may be shared, providing a cost-efficiency advantage over fixed dedicated hardware solutions.

Motion estimation constitutes a significant computational part of *standard video codecs*, and finds application in *proprietary video enhancement algorithms*. Standard video codecs include MPEG2, MPEG4, H264/AVC [1], etc. Their functionality is pre-scribed to allow for inter-operability, and documented by international standardization committees. In this domain, flexibility allows for fast market introduction, or improved implementations of functionality after market introduction. Proprietary video algorithms are typically company specific. In the video display markets, these algorithms implement the key features with which the set makers distinguish themselves. Philips provides Natural Motion image enhancement, which includes motion based de-interlacing and temporal up-conversion. In a new and rapidly growing market like LCD-TV, fast time-to-market may determine the success of a solution based on these video algorithms. This paper evaluates the performance of a motion estimation algorithm on the TM3270. We quantify the speedup from new operations and from data prefetching. Furthermore, we evaluate the impact of memory latency to processor performance. We introduce the concepts of two-slot operations, and collapsed load operations.

The remainder of this paper is organized as follows. In Section 2, we introduce the motion estimation algorithm, which is used to evaluate processor performance. In Section 3, we define our performance evaluation environment. In Section 4, we present the TM3270 architecture. In Section 5, we present six software implementations of the algorithm as described in Section 2. In Section 6, we present and discuss performance measurement results. Finally, in Section 7, we present our conclusions.

2. MOTION ESTIMATION ALGORITHM

Many block-based motion estimation algorithms exist, and the suitability of a particular algorithm depends on the application at hand. This paper does not intend to introduce a new and better motion estimator, but rather to evaluate the performance of our processor on an existing algorithm. We decided upon the 3-D Recursive Search (3DRS) algorithm [2]. This algorithm provides a high quality result, at a relatively low computational complexity, making it an attractive candidate for a software implementation. Furthermore, it has found successful application in commercial ICs [3].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'05, March 13-17, 2005, Santa Fe, New Mexico, USA.
Copyright 2005 ACM 1-58113-964-0/05/0003…$5.00.

The 3DRS algorithm is a spatial-temporal algorithm; i.e. the candidate motion vectors for a block are derived from the motion vectors of surrounding blocks in both space and time. We will give a brief overview of the algorithm.

Our version of the 3DRS algorithm evaluates 11 candidate motion vectors per 8x8 block of pixels *b*. Let $\vec{b} = \begin{pmatrix} b_x \\ b_y \end{pmatrix}$ denote the *block position*, such that $\begin{pmatrix} 8b_x \\ 8b_y \end{pmatrix}$ and $\begin{pmatrix} 8b_x + 7 \\ 8b_y + 7 \end{pmatrix}$ identify the upper left and lower right *pixel positions* of block b in the image. Let $mv(\vec{b}, n)$ denote the best motion vector for block $\vec{b}$ in image *n*, based on a cost function. The motion estimation vector candidates are taken from the candidate sets *CS_zero* (the zero vector), *CS_spatial* (motion vectors of already processed blocks in the current image), *CS_temporal* (motion vectors of blocks in the previous image), and *CS_noise_spatial* (noise updated motion vectors of already processed blocks in the current image):

$$CS_zero = \left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right\}$$

$$CS_spatial = \left\{ \begin{array}{l} mv(\vec{b} + \begin{pmatrix} -1 \\ 0 \end{pmatrix}, n), \\ mv(\vec{b} + \begin{pmatrix} -1 \\ -1 \end{pmatrix}, n), \\ mv(\vec{b} + \begin{pmatrix} 1 \\ -1 \end{pmatrix}, n) \end{array} \right\}$$

$$CS_temporal = \left\{ \begin{array}{l} mv(\vec{b}, n-1), \\ mv(\vec{b} + \begin{pmatrix} 1 \\ 0 \end{pmatrix}, n-1), \\ mv(\vec{b} + \begin{pmatrix} 2 \\ 0 \end{pmatrix}, n-1), \\ mv(\vec{b} + \begin{pmatrix} 1 \\ 1 \end{pmatrix}, n-1), \\ mv(\vec{b} + \begin{pmatrix} -1 \\ 1 \end{pmatrix}, n-1), \end{array} \right\}$$

$$CS_noise_spatial = \left\{ \begin{array}{l} mv(\vec{b} + \begin{pmatrix} -2 \\ 0 \end{pmatrix}, n) + \vec{u}_1, \\ mv(\vec{b} + \begin{pmatrix} 0 \\ -1 \end{pmatrix}, n) + \vec{u}_2 \end{array} \right\}$$

Noise vectors $\vec{u}_1$ and $\vec{u}_2$ are cyclically selected from the list of noise vectors *NS*:

$$NS = \left\{ \begin{array}{l} \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} -1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ -1 \end{pmatrix}, \begin{pmatrix} 2 \\ 0 \end{pmatrix}, \begin{pmatrix} -2 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 2 \end{pmatrix}, \begin{pmatrix} 0 \\ -2 \end{pmatrix}, \\ \begin{pmatrix} 4 \\ 0 \end{pmatrix}, \begin{pmatrix} -4 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 4 \end{pmatrix}, \begin{pmatrix} 0 \\ -4 \end{pmatrix}, \begin{pmatrix} 8 \\ 0 \end{pmatrix}, \begin{pmatrix} -8 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 8 \end{pmatrix}, \begin{pmatrix} 0 \\ -8 \end{pmatrix} \end{array} \right\}$$

Block matching uses the Sum-of-Absolute-Differences (SAD) cost function, which sums the absolute differences of corresponding values of source and reference blocks. These values can represent image pixels, fractional image pixels, or multiple image pixels (in case of spatial down-scaling).

3. PERFORMANCE EVALUATION

This section describes the performance evaluation environment. An accurate portrayal of the System-on-Chip (SoC) is important since the processor's performance heavily depends on its interaction with the rest of the SoC environment (Figure 1). The evaluation environment includes the TM3270 media-processor, a DDR memory controller, and off-chip DDR memory.

The TM3270 has an operating frequency of 450 MHz. We simulate with a DDR400 SDRAM memory; i.e. an operating frequency of 200 MHz. The SDRAM has a CAS latency of 3 cycles. The TM3270 provides an asynchronous clock domain transfer between the 200 MHz. memory, and the 450 MHz. processor clocks. We use the actual TM3270 Verilog HDL description as simulation model. The same description was used as input to synthesis and place&route tools. This ensures a cycle accurate portrayal of processor performance, including cache behavior. We use Cadence's NC-Verilog for Verilog HDL simulation. The path between the processor and the memory controller includes a delay block, which can be used to artificially delay the data traffic from/to the off-chip SDRAM memory. The artificial delay is used to mimic the processor observed memory latency in a SoC in which multiple on-chip IP devices share a unified off-chip memory.

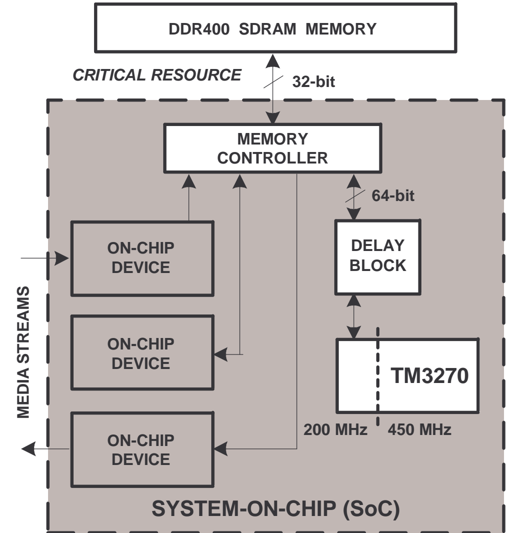

Figure 1. Performance evaluation environment. The dotted line indicates a clock domain transfer.

4. TM3270 ARCHITECTURE

This section gives an overview of the TM3270. The TM3270 media-processor is source code backward compatible with the TriMedia architecture. An overview of the TriMedia architecture can be found in [4], [5], and [6]. The processor has a fully synthesizable design using a standard-cell logic library and single-ported SRAMs, allowing for fast process technology mapping. Silicon area was one of the main design constraints, to allow for

an economically viable solution in the cost-driven consumer electronics market. The processor achieves a frequency of 450 MHz. in a .09 µm process technology, and measures around 8 mm^2. Table 1 gives an overview of the main architectural features.

Table 1. TM3270 architecture overview.

Architectural feature	Quantity
Architecture	5-issue slot VLIW, guarded RISC-like operations
Pipeline depth	7-13 stages
Address width	32-bit
Data width	32-bit
Register-file	Unified, 128 32-bit registers
Functional units	35
Floating point	IEEE-754
SIMD	1 x 32-bit 2 x 16-bit 4 x 8-bit
Instruction cache	64 Kbytes, 8 way set-associative, 128 byte lines
Data cache	128 Kbytes, 4 way set-associative, 128 byte lines

The TM3270 has a 32-bit VLIW architecture. A VLIW instruction may contain up to five operations. Each of these operations may be guarded; i.e. their execution can be made conditional on the value of a guard register. This allows the compiler to eliminate conditional jump operations, using if-conversion. SIMD arithmetic and shuffle operations allow for efficient manipulation and re-organization of 8-, and 16-bit data types. Floating-point operations comply with the IEEE-754 standard. Operations are grouped into functional units, and most functional units have multiple instantiations. Most functional units are fully pipelined, allowing for back-to-back issue of operations. The simple arithmetic functional unit has five instantiations, so up to five simple arithmetic operations can be issued every cycle. The floating point multiply and adder units have two instantiations each. The TM3270 provides some architectural enhancements over previous TriMedia processors; we mention those that impact motion estimation performance:

Two-slot operations. The TM3270 has function units that are located in two neighboring issue slots. As a result, the operations executed by these functional units can have up to 4 source and up to 2 destination operands. Table 2 gives the definition of a two-slot operation with 4 source operands and 1 destination operand: SUPER_ULCIP8ASR6MIX8UI.

Unaligned load/store operations. The load/store unit provides access to non-aligned data elements, without incurring processor stall cycles.

Collapsed load / reduction operations. The collapsing of multiple elementary arithmetic operations was introduced in [7]. The TM3270 instead combines the functionality of memory and reduction operations into a single operation. More specifically, support is provided for operations that combine the functionality of an ordinary load with the functionality of a reduction operation. These operations can improve processor performance, and have the additional benefit that they reduce register-pressure. The operations have two source operands: a memory address, and a 4-bit value that acts as a weight for a weighted average calculation. Table 2 gives the definitions of two of these operations: LD_FRAC8, and LD_PACKFRAC8. The LD_FRAC8 operation loads 5 byte elements from sequential memory addresses, and calculates a weighted average for the 1st and 2nd, the 2nd and 3rd, the 3rd and 4th, and the 4th and 5th byte elements. The LD_PACKFRAC8 operation loads 8 byte elements from sequential memory addresses, and calculates a weighted average for the 1st and 2nd, the 3rd and 4th, the 5th and 6th, and the 7th and 8th byte elements.

Data cache capacity. The TM3270 has a 128 Kbyte data cache, whereas previous TriMedia processors have a 16 Kbyte data cache. The cache contributes more than 1/3 of the area of the 8 mm^2 processor. The 128 Kbyte cache is able to capture the working set of most standard video codecs at standard definition (SD) resolution (720*480), and the working set of most of our proprietary video processing algorithms at either SD or HD resolution.

Prefetching. Prefetching reduces processor observed latency of the off-chip memory. By prefetching data from the off-chip memory into the processor's data cache before actual use of the data, performance is improved by eliminating stall cycles associated to cache misses.

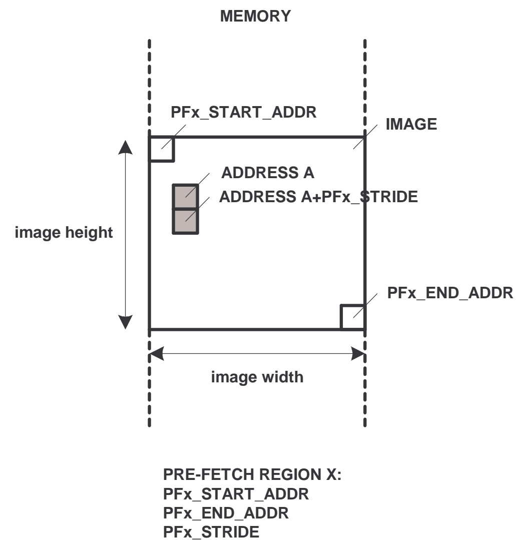

Figure 2. Memory region based prefetching.

Table 2. Some of the new TM3270 operations.

Operation	Description

SUPER_UCLIP8ASR6MIX8UI
rsrc1 rsrc2 rsrc3 rscr4 ->rdest1;
 	
Semantics: Weighted average of 8-bit unsigned integers (with rounding).
	temp = (rsrc1[31:24]*rsrc2[31:24] + rsrc3[31:24]*rsrc4[31:24] + 32) / 64;
	rdest1[31:24] = min (max (0, temp), 255);
	temp = (rsrc1[23:16]*rsrc2[23:16] + rsrc3[23:16]*rsrc4[23:16] + 32) / 64;
	rdest1[23:16] = min (max (0, temp), 255);
	temp = (rsrc1[15:8]*rsrc2[15:8] + rsrc3[15:8]*rsrc4[15:8] + 32) / 64;
	rdest1[15:8] = min (max (0, temp), 255);
	temp = (rsrc1[7:0]*rsrc2[7:0] + rsrc3[7:0]*rsrc4[7:0] + 32) / 64;
	rdest1[7:0] = min (max (0, temp), 255);

LD_FRAC8
rsrc1 rsrc2 -> rdest1;
 	
Semantics: Collapsed load; load combined with linear interpolation.
	data0 = Mem[rsrc1]; data1 = Mem[rsrc1 + 1]; data2 = Mem[rsrc1 + 2];
	data3 = Mem[rsrc1 + 3]; data4 = Mem[rsrc1 + 4];
	rdest1[31:24] = (data0*(16-rsrc2[3:0]) + data1*rsrc2[3:0] + 8) / 16;
	rdest1[23:16] = (data1*(16-rsrc2[3:0]) + data2*rsrc2[3:0] + 8) / 16;
	rdest1[15:8] = (data2*(16-rsrc2[3:0]) + data3*rsrc2[3:0] + 8) / 16;
	rdest1[7:0] = (data3*(16-rsrc2[3:0]) + data4*rsrc2[3:0] + 8) / 16;

LD_PACKFRAC8
rsrc1 rsrc2 -> rdest1;
 	
Semantics: Collapsed load; load combined with linear interpolation.
	data0 = Mem[rsrc1]; data1 = Mem[rsrc1 + 1]; data2 = Mem[rsrc1 + 2];
	data3 = Mem[rsrc1 + 3]; data4 = Mem[rsrc1 + 4]; data5 = Mem[rsrc1 + 5];
	data6 = Mem[rsrc1 + 6]; data7 = Mem[rsrc1 + 7];
	rdest1[31:24] = (data0*(16-rsrc2[3:0]) + data1*rsrc2[3:0] + 8) / 16;
	rdest1[23:16] = (data2*(16-rsrc2[3:0]) + data3*rsrc2[3:0] + 8) / 16;
	rdest1[15:8] = (data4*(16-rsrc2[3:0]) + data5*rsrc2[3:0] + 8) / 16;
	rdest1[7:0] = (data6*(16-rsrc2[3:0]) + data7*rsrc2[3:0] + 8) / 16;

Besides the support of software prefetch operations, the processor supports hardware based prefetching. Hardware based prefetching uses so-called prefetch memory regions, which allow for a prefetching pattern that reflects the access pattern of a data structure mapped onto a certain address space. The TM3270 supports four separate memory regions. The identification of these memory regions, and the required prefetch pattern is under *software control*, and defined by the following parameters (n = 0, 1, 2, 3):

- PFn_START_ADDR
- PFn_END_ADDR
- PFn_STRIDE

The first two parameters, PFn_START_ADDR and PFn_END_ADDR, are used to identify a memory region. The third parameter, PFn_STRIDE, is used to specify the prefetch pattern for the associated region.

As an example, consider an application that is processing a two-dimensional image in memory (Figure 2). Assume the application processes all image pixels in a line-by-line fashion, in a top to bottom line direction. The memory region is set to include the image. The stride value, PFn_STRIDE, is set to reflect the image access pattern. By setting the stride value equal to the image width, the image line sequential to the one being processed is pre-fetched. When the off-chip memory latency exceeds the time needed to process an image line, prefetching may not complete in time. Therefore, it might be necessary to prefetch more than one image line ahead, by setting the stride value to a multiple of the image width.

Note that by setting the prefetch stride to the cache line size, traditional next-sequential cache line prefetching is implemented.

5. MOTION ESTIMATION IMPLEMENTATIONS

We evaluated six different software implementations of the 3DRS algorithm, as described in Section 2. The implementations differ in the extent to which they exploit the TM3270's new operations, and their quality level. The quality level is determined by the ability to support a fractional horizontal and/or vertical motion vector component. When supported, fractional pixels are calculated using linear interpolation at ¼ pixel image resolution.

All implementations operate on a SD image of 720*480 pixels, resulting in a total of 5400 8*8 blocks per image. All implementations use memory region based prefetching. For the current image, the memory region stride is set such that while processing a *pixel position* $\begin{pmatrix} c_x \\ c_y \end{pmatrix}$, the pixel at position $\begin{pmatrix} c_x \\ c_y + 8 \end{pmatrix}$, is prefetched. For the reference image, the memory region stride is set such that while processing a pixel at image position $\begin{pmatrix} r_x \\ r_y \end{pmatrix}$, the pixel at position $\begin{pmatrix} r_x \\ r_y + m \end{pmatrix}$, is prefetched (with m = 40, being the maximum value of the vertical component of the motion vector).

As the 3DRS algorithm iterates over a sequence of video images, it tends to converge to a smooth motion vector field. Although we implemented the 3DRS algorithm as presented, including the control overhead of tracking the best motion vector candidate, we decided not to rely on the spatial and temporal convergence of the algorithm. By doing so, our measurement results reflect worst case, rather than typical case execution behavior. The spatial

convergence is non-existent, since we operate on random initialized video images; i.e. image pixels have a random value between 0 and 255. The temporal convergence is non-existent, since we initialize the motion vectors $mv(\vec{b}, n\text{-}1)$, for all blocks *b* of the previous image *n-1*, to a random value. The horizontal component of the motion vectors are taken from the range [-128, 127 ¾], and the vertical component of the motion vectors are taken from the range [-32, 31 ¾] (video motion is typically more dominant in the horizontal direction). Note that the motion candidates from *CS_noise_spatial* include the addition of a noise vector, possibly extending the motion vector ranges to [-136, 135 ¾], and [-40, 39 ¾], respectively.

Our base line, *implementation A*, uses neither unaligned load/store operations, nor any new operations. No software support is provided for fractional motion vectors. *Implementation B* is intended to show the benefit of unaligned load/store operations. No support is provided for fractional motion vectors. Implementations A and B provide the same functionality; i.e. integer motion vector based motion estimation. *Implementation C* uses unaligned load/store operations. The SUPER_UCLIP8ASR6MIX8UI operation is used to support a horizontal fractional motion vector component. *Implementation D* uses the LD_FRAC8 operation to support a horizontal fractional motion vector component. Implementations C and D provide the same functionality; i.e. support for a horizontal fractional motion vector component. The support of a fractional horizontal motion vector component improves the quality of implementations C and D over implementations A and B. *Implementation E* uses the LD_FRAC8 operation to support a horizontal fractional motion vector component, and the UCLIP8ASR6MIX8UI operation to support a vertical fractional motion vector component. The support of a fractional component in both directions improves the quality of this implementation over the previous implementations. *Implementation F* uses the LD_PACKFRAC8 operation, to support horizontal down-sampling by a factor 2. Note that down-sampling is at ¼ pixel resolution and performed on the fly; i.e. input to the down-sampling are the pixels of the original image, and not the pixels of a down-sampled image. This approach does not have the potential advantage of a reduction in data working set size, but does provide a better quality result, and eliminates the need to perform a separate down-sampling pass on images. Down-sampling, and the fact that a vertical fractional motion vector component is not supported, degrades the quality of this implementation.

Table 3 gives a summary of the six implementations.

6. MEASUREMENTS AND RESULTS

All six implementations were simulated in our cycle accurate SoC environment, as described in Section 3. For all implementations, the instruction working set fits within the instruction cache, so apart from initial compulsory cache misses, no instruction cache misses and associated stalls were encountered.

6.1 Comparing the implementations

To compare the performance of the implementations, we simulated them under the same SoC conditions: the delay block adds 10 cycles delay in the memory clock domain (22.5 cycles in the processor clock domain). Table 4 gives the simulation results.

All implementations show a CPI of close to 1.0, which is the theoretical optimal. Implementation A, however, suffers from low issue slot utilization (large amount of NOP operations), resulting in a large VLIW instruction count. The main reason is the inability of the compiler/scheduler to inline the SAD cost function. Function parameter passing through the stack requires a series of store and load operations. Issue slot constraints on load and store operations (a VLIW instruction may only contain one load and one store, or two store operations), result in low issue slot utilization. Furthermore, read-after-write dependencies between successive store and load operations result in additional stall cycles for implementation A. The implementations only utilize a small amount of available processor performance. Implementation A utilizes 18.4% of the processor's full potential. Implementation F improves the performance by a factor 4, resulting in a processor utilization of only 4.69%.

Implementation A does not support unaligned loads. Therefore, additional load and shuffle operations are required, to extract the required reference image pixels. This is reflected in the high VLIW instruction count. Implementation B shows the benefit of unaligned load/store operations. It provides tha same functionality as implementation A at significant lower VLIW instruction count. This confirms that media-processors should not only provide SIMD support for computational operations, but also for memory operations. Implementation C shows that the support for a horizontal fractional motion vector component adds a significant amount of VLIW instructions. Implementation D provides the same functionality as implementation C, but uses the LD_FRAC8 operation. This operation combines the horizontal fractional calculation with the loading of data elements from memory. The VLIW instruction count of implementation D is comparable to implementation B, which only supports an integer motion vector. Implementation E adds support for a vertical fractional motion

Table 3. Implementations A to F, their use of TM3270 architectural enhancements and their support of fractional horizontal and/or vertical motion vector components.

Implem.	Unaligned load/store support	Horizontal fractional motion vector	Vertical fractional motion vector	Horizontal down-sampling	Relative quality level
A	no	no	no	no	-
B	yes	no	no	no	-
C	yes	SUPER_UCLIP8ASR6MIX8UI	no	no	+
D	yes	LD_FRAC8	no	no	+
E	yes	LD_FRAC8	SUPER_UCLIP8ASR6MIX8UI	no	++
F	yes	LD_PACKFRAC8	no	LD_PACKFRAC8	-

Table 4. Performance results for implementations A to F (10 cycle delay block).

Implementation	Image average					MHz. req. for SD @ 30 images/sec.	% of 450 MHz. processor performance
	Cycles	VLIW instruction count	Stall cycles	Cycles/ VLIW instr.	Operations/ VLIW instr.		
Prefetching on							
A (-)	2761797	2556595	205202	1.08	1.66	82.9	18.4 %
B (-)	1175520	1086999	88521	1.08	3.46	35.3	7.8 %
C (+)	1962161	1907856	54305	1.03	3.96	58.9	13.1 %
D (+)	1239854	1173460	66394	1.06	4.04	37.2	8.3 %
E (++)	1697739	1637807	59932	1.04	4.37	50.9	11.3 %
F (-)	704411	617076	87335	1.14	4.43	21.1	4.7 %
Prefetching off							
A (-)	3092172	2556592	535580	1.21	1.66	92.7	20.6 %
B (-)	1481615	1086932	394683	1.36	3.44	44.4	9.9 %
C (+)	2300662	1907789	392873	1.21	3.96	69.0	15.3 %
D (+)	1565680	1173393	392287	1.33	4.03	47.0	10.4 %
E (++)	2028260	1637741	390519	1.24	4.36	60.8	13.5 %
F (-)	1027849	617069	410780	1.67	4.41	30.7	6.8 %

vector component; at the cost of a cycle budget increase of approximately 50% over implementation D. Implementation F has the best cycle budget. The LD_PACKFRAC8 operation is used to perform down-sampling. As a result, the amount of operations involved in the SAD calculation is halved.

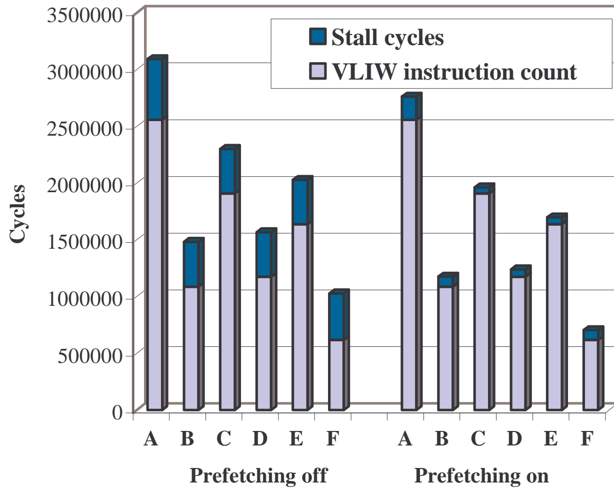

Figure 3. Performance results for implementations A to F, with prefetching off and prefetching on.

6.2 Influence of prefetching

By prefetching data from the off-chip memory into the processor's data cache before actual use of the data by the processor, performance is improved by eliminating stall cycles associated to cache misses. To quantify the benefit of prefetching, we simulated all implementations with and without prefetching. Figure 3 gives the simulation results (numbers taken from Table 4). The benefit from prefetching is similar for all implementations. On average, the cycle count decreases by around 300.000 cycles when prefetching is on. Those implementations with the lower cycle count have the largest *relative* benefit from prefetching. For implementation F, prefetching decreases the cycle count by roughly 30%.

6.3 Influence of memory latency

To quantify the influence of SoC SDRAM memory latency on processor performance, we simulated all implementations with different memory delay cycles. The simulation with 0 delay cycles reflects a SoC in which only the processor requires off-chip memory bandwidth. Increasing the amount of delay cycles mimics a SoC in which off-chip memory bandwidth is consumed by other on-chip IP devices. Figure 4 gives the cycle counts for the simulation results.

As expected, all implementations suffer from increased memory latency. The implementations with a relatively low VLIW instruction count (B, D, and F) have a discontinuity in the increase in cycle budget as a function of memory latency. Implementation F becomes more latency dependent at approximately 30 additional memory delay cycles. Implementations B and D show similar behavior, at approximately 60 additional delay cycles. Due to the relatively low VLIW instruction count, implementations B, D, and F become memory bound. In the presence of high memory latency, it is no longer possible to overlap prefetching with computation. Implementations A and C have a relatively high VLIW instruction count, and have not become completely memory bound at 100 memory delay cycles. For these implementations is possible to overlap prefetching and computation. Implementations A and C offer better performance at 100 additional delay cycles with prefetching on, than at 10 additional delay cycles with prefetching turned off.

Higher memory latencies make an implementation more dependent on the ability to overlap prefetching with computation,

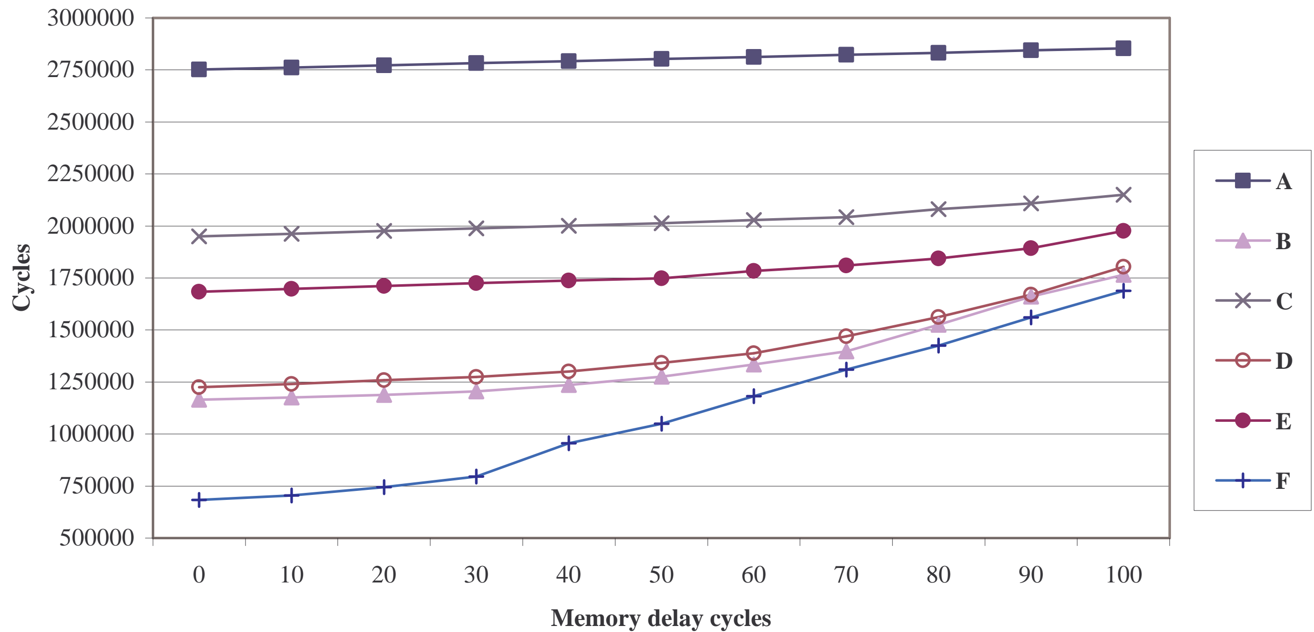

Figure 4. Performance results for implementations A to F, for different memory delay cycles (prefetching on).

and less dependent on the VLIW instruction count. As memory latency increases, the absolute cycle difference between different implementations decreases. One could draw the conclusion that the additional quality as provided by an implementation with a higher VLIW instruction count, becomes cheaper as the memory latency increases.

7. CONCLUSION

Although this paper does not intend to provide the best motion estimator for the TM3270, the simulation results show that real-time motion estimation utilizes only a fraction of the processor's computational abilities (4.7% for implementation F and 11.3% for the highest quality implementation, implementation E). The TM3270 provides architectural enhancements that allow for efficient implementation of other parts of video standards (IDCT/DCT, and standard compliant fractional pixel calculation), which makes SD multi-standard video encoding achievable.

The use of two-slot operations (SUPER_UCLIP8ASR6MIX8UI) and collapsed load operations (LD_FRAC8 and LD_PACKFRAC8) significantly reduces the cycle count. At a higher quality level, implementation E has a cycle count that is 39% lower than that of implementation A (1697739 versus 2761797 cycles for implementations A and E respectively). At a comparable quality level, implementation F has a cycle count that is 75% lower than that of implementation A.

Data prefetching improves performance of implementation F by 30%, at 10 memory delay cycles. As the amount of memory delay cycles increases, the relative speedup due to prefetching is decreasing.

Dedicated hardware solutions are typically considered to be smaller than programmable solutions. However, this conclusion may be premature. Unutilized processor performance may be used to implement other parts of a video encoder, and an audio encoder. When taking a system approach to the encoder functionality at large, a programmable solution can compete with dedicated hardware solutions, in terms of silicon area. The data cache contributes more than 1/3 of the area of the processor. The cache capacity was found necessary to limit the amount of off-chip memory bandwidth. It is likely, that a dedicated hardware solution requires a similar sized memory structure, to achieve comparable memory bandwidth requirements.

Since the motion estimation algorithm has a software implementation, improvements on the 3DRS algorithm or different motion estimation algorithms are easily realized.

8. REFERENCES

[1] Richardson, I.E.G. *H.264 and MPEG-4 video compresson, video coding for next-generation multimedia*, Wiley, 2003.

[2] de Haan, G. et al. True-motion estimation with 3-D recursive search block matching, In *ICCE Transactions on Circuits and Systems for Video Technology*, vol.3, pp. 368-379, October 1993.

[3] de Haan, G. IC for motion compensated deinterlacing, noise reduction and picture rate conversion, In *IEEE Transactions on Consumer Electronic*s, pp. 617-624, August 1999.

[4] Rathnam, S. and Slavenburg, G. An architectural overview of the programmable multimedia processor, tm-1, In *Proceedings of the COMPCON '96*, pp. 319-326, 1996.

[5] Halfhill, T. Philips powers up for video, In *Microprocessor Report*, http:/www.mpronline.com/, November 2003.

[6] Hennessy, J.L. and Patterson, D.A. *Computers Architecture: A Quantitative Approach, 3rd edition*, Morgan Kaufmann, 2003.

[7] Vassiliadis, S., Phillips, J. and Blaner, B. Interlock collapsing ALU's, In *IEEE Transactions on Computers*, vol. 42, issue 7, pp. 825-839, July 1993