# TEMPORAL VIDEO UP-CONVERSION ON A NEXT GENERATION MEDIA-PROCESSOR

Jan-Willem van de Waerdt*+, Stamatis Vassiliadis+, Erwin B. Bellers*, and Johan G. Janssen*
*Philips Semiconductors
San Jose, CA, USA

+Delft University of Technology – Computer Engineering Department
Delft, the Netherlands, (Stamatis@duteppO.et.tudelft.nl)

## ABSTRACT

This paper evaluates the performance of a temporal video up-conversion algorithm on a next generation media-processor, the TM3270. Temporal up-conversion adapts the temporal frequency of a video signal to the temporal frequency of a display device. In order to improve performance, the TM3270 provides architectural enhancements over previous TriMedia processors. We quantify the speedup of *new operations* to temporal up-conversion performance. We show that the new operations improve performance by 41%. Furthermore, we quantify the speedup of *data prefetching* and an *allocate-on-write-miss policy*. We show that data prefetching can improve performance by more than 20%. An allocate-on-write-miss policy significantly improves performance in the presence of a high latency memory subsystem. By applying all TM3270 architectural enhancements, we show that the temporal up-conversion algorithm can be performed in 17.8% of the available processor performance.

**KEYWORDS**

Media processor, temporal video up-conversion, software implementation.

## I. INTRODUCTION

Media-processors can be used in many video processing applications. Their programmability provides flexibility, which can be exploited in various ways. It enables algorithmic changes after design, a higher level of adaptation to changes in the video, multiple algorithms can be mapped to the same architecture, faster time-to-market, etc. When enough performance is available, media-processors provide an interesting alternative to fixed dedicated hardware solutions. Furthermore, when a single programmable platform can address multiple markets, its design costs can be shared, providing a cost-efficiency advantage over fixed dedicated hardware solutions.

Video codecs, such as MPEG2, MPEG4, and H.264/AVC [1], rely on standardization to ensure inter-operability, and therefore, these algorithms will generally not change over time. Proprietary video processing applications do generally change more often, i.e. due to algorithmic improvements, and as such have a stronger need for an implementation platform that provides a sufficient level of programmability.

Many proprietary video processing applications can be found in today's television sets, line noise reduction, de-interlacing, sharpness enhancement, film judder removal, scaling, etc. Since video quality is still the main driver in the high-end television market, the ability to improve quality before the competition, may determine the success of a solution in the television market. Many of these video processing applications have until recently only been available in a dedicated hardware design, i.e. ASICs. However, as the cost of new designs and integration is increasing dramatically, and programmable media-processors offer increased performance, we reach a point at which it becomes attractive to pursue a re-usable, programmable platform.

In this paper, we evaluate the performance of a high quality temporal up-converter on the next generation TriMedia processor, the TM3270. The TM3270 media-processor provides architectural enhancements over previous TriMedia processors. These enhancements improve performance in the video-processing domain. We show that a real-time software implementation of the temporal up-converter is achievable, with enough performance headroom to perform other video processing applications. We quantify the contribution of collapsed load operations, and two-slot operations to processor performance. Furthermore, we quantify the speedup of data prefetching and an allocate-on-write-miss data cache policy.

The remainder of this paper is organized as follows. In Section II, we describe the temporal up-conversion algorithm, which is used to evaluate processor performance. In Section III, we define our performance evaluation environment. In Section IV, the TM3270 architecture is presented. In Section V, we present six software implementations of the algorithm as described in Section II. In Section VI, we present and discuss the performance measurement results. Finally, in Section VII, we present our conclusions.

## II. TEMPORAL UP-CONVERSION

### A. Introduction

Temporal up-conversion adapts the temporal frequency of a video signal to the temporal frequency of a display device. An example is the conversion of a 60 Hz standard definition (SD) NTSC video signal to a display frequency at 85 Hz. Figure 1 illustrates the basic principle. A simple approach is to repeat the last available image. Although this is okay for stationary images, it introduces film judder for moving image parts (2:2 and 3:2 pull down). A better approach is to apply motion-compensated (MC) temporal up-conversion. It attempts to eliminate the film judder by accurate portrayal of the motion in the video signal.

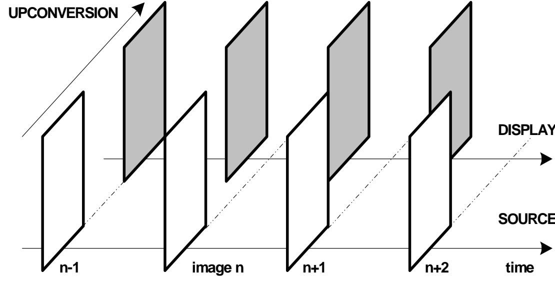



Fig. 1. Temporal up-conversion, the basic principle. A sequence of image at the display frequency (grey color) is derived from a sequence of images at the source frequency (white color).

New images are derived using the two temporally neighboring images in the source material. MC temporal up-conversion relies on the availability of a motion vector field [2], [3]. A motion vector $mv(\vec{b}, n)$ is assumed to be available for every 4x4 block of image pixels *b*, representing the motion of the block from source image *n-1*, to source image *n*. Let $\vec{b} = \begin{pmatrix} b_x \\ b_y \end{pmatrix}$ denote the block address, such that $\begin{pmatrix} 4b_x \\ 4b_y \end{pmatrix}$ and $\begin{pmatrix} 4b_x + 3 \\ 4b_y + 3 \end{pmatrix}$ identify the upper left and lower right pixels of block *b* in the image.

### B. Enhanced Cascaded Median Algorithm

This paper does not intend to introduce a new and better temporal up-converter, but rather to evaluate the performance of our processor on an existing algorithm. We use the *cascaded median* up-converter [4] as the basis for our MC up-converter algorithm.

The cascaded median up-converter uses non-motion compensated (static) and motion compensated (dynamic) pixels (Figure 2) to calculate the pixels in the up-converted image. Static pixels are taken from colocated positions in the temporally neighboring source images *n-1* and *n*. These pixels are represented by *st_left* and *st_right*. Dynamic pixels are taken from motion compensated positions in the temporally neighboring source images *n-1* and *n*. These pixels are represented by *dyn_left* and *dyn_right*. We support motion vectors at ¼ pixel resolution in both horizontal and vertical direction. Motion compensated pixels at fractional positions are calculated by means of linear interpolation of up to four spatially surrounding integer pixels. The parameter *p* is a fraction between 0 and 1, and represents the temporal position of the up-converted image in between the two neighboring source images *n-1* and *n*.

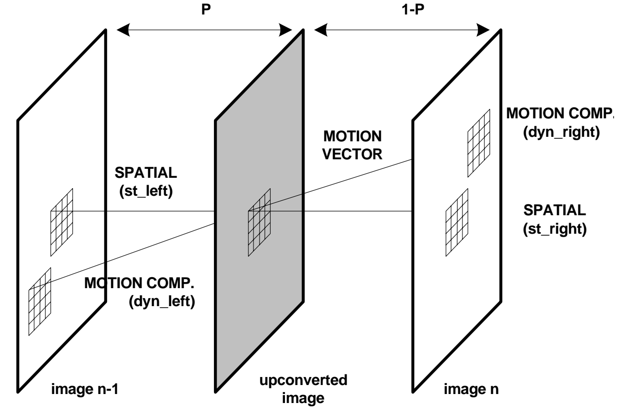



Fig. 2. Temporal up-conversion, spatial and motion compensated predictors.

The cascaded median algorithm combines *a static median, a dynamic median, and a mix filter*. This

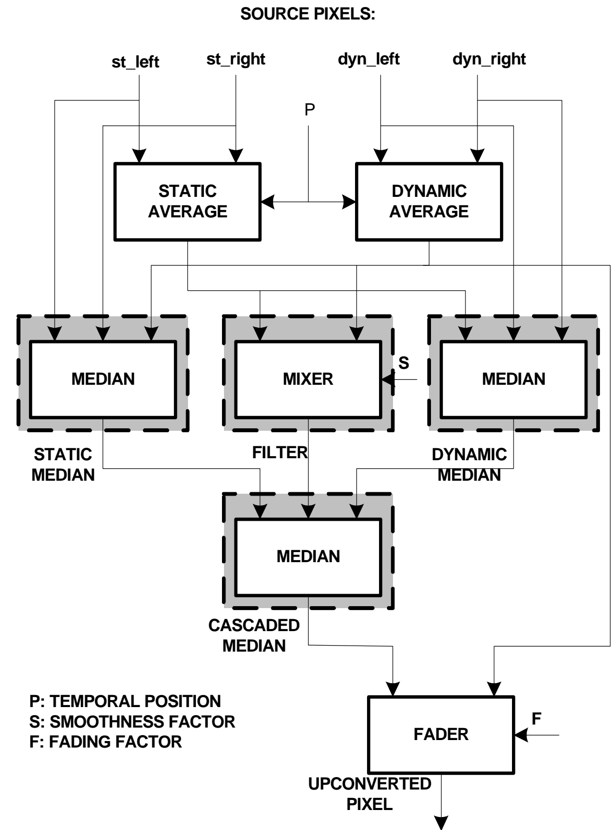



Fig. 3. Enhanced cascaded median.

combination takes advantages of the individual filters' strengths, and reduces the impact of the individual filters' weaknesses. A detailed description of the algorithm, and an evaluation of its quality can be found [4]. The quality of the cascaded median algorithm was further enhanced by the addition of a fader as illustrated in Figure 3.

The "static average" and "dynamic average" functions in Figure 3 perform a weighted average operation based on the temporal position *p*. The smaller the temporal position *p*, the higher the weight of the pixels of the source image on the left (image *n-1*).

$$Average\ (input1, input2, p) = (1-p) * input1 + p * input2$$

The "mixer" function performs a weighted average operation based on a smoothness factor *s*. The up-conversion algorithm calculates a smoothness factor based on the current block's motion vector, and the motion vectors of the surrounding blocks. The smaller the sum of the differences between the current motion vector and the surrounding motion vectors, the higher the relative weight of the dynamic average input to the mixer. The smoothness factor represents the local continuity/smoothness of the motion vector field.

$$Mixer\ (input1, input2, s) = (1-s) * input1 + s * input2$$

The "fader" function performs a weighted average operation based on a fading factor *f*. The up-conversion algorithm calculates a fading factor based on the current block's motion vector, and the motion vectors of the surrounding blocks. A large difference between the current motion vector and one of the surrounding motion vectors (a large block to block motion vector discontinuity), results in a small relative weight for the dynamic average input to the fader.

$$Fader\ (input1, input2, f) = (1-f) * input1 + f * input2$$

The "median" functions perform a three input median operation:

$$Median\ (input1, input2, input3) = Min\ (Max\ (Min\ (input1, input2), input3), Max\ (input1, input2))$$

## III. PERFORMANCE EVALUATION ENVIRONMENT

This section describes the performance evaluation environment. An accurate portrayal of the System-on-Chip (SoC) is important since the processor's video performance heavily depends on its interaction with the rest of the SoC environment (Figure 4). 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 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.

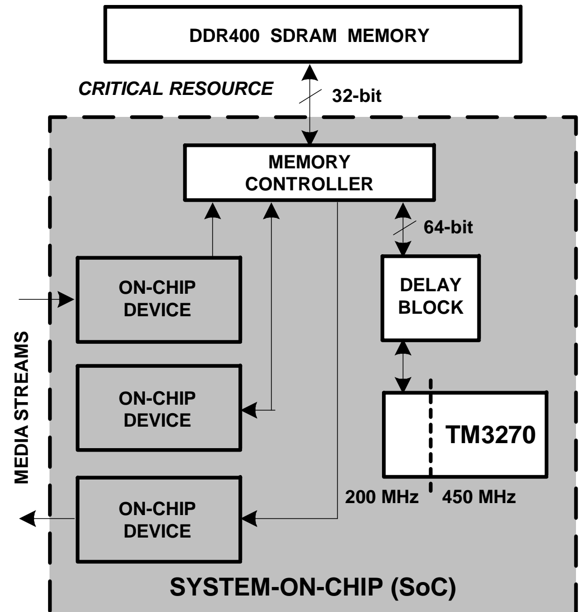


Fig. 4. Performance evaluation environment. The dotted line indicates a clock domain transfer.

TABLE I
TM3270 OVERVIEW

| Architectural feature | Quantity |
|---|---|
| Architecture | 5 issue slot VLIW<br>guarded RISC-like operations |
| Pipeline depth | 7-13 stages |
| Address width | 32-bit |
| Data width | 32-bit |
| Register-file | Unified<br>128 32-bit registers |
| Functional units | 31 |
| IEEE-754 floating point | yes |
| SIMD capabilities | 1 x 32-bit<br>2 x 16-bit<br>4 x 8-bit |
| Instruction cache | 64 Kbyte<br>8 way set-associative, LRU replacement<br>128 byte lines |
| Data cache | 128 Kbyte<br>4 way set-associative, LRU replacement<br>128 byte lines |

## IV. TM3270 ARCHITECTURE

This section gives an overview of the TM3270 media-processor. The TM3270 is backward source code compatible with the TriMedia architecture. An overview of the TriMedia architecture can be found in [7], [8], and [9]. 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 8.1 $mm^2$. Table I gives an overview of the main architectural features.

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. The TM3270 provides some architectural enhancements over previous TriMedia processors; we mention those that impact temporal up-conversion performance:

*Two-slot operations*. New operations have been added that can significantly contribute to processor performance on e.g. temporal up-conversion. Some of these operations are two-slot operations. These two-slot operations allow for up to 4 source, and up to 2 destination operands. Table II gives the definitions of two of these operations.

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

*Collapsed load and reduction operations*. The collapsing of multiple elementary arithmetic operations was introduced in [10]. 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 II gives the definitions of one of these operations: LD_FRAC8. The LD_FRAC8 operation loads 5 byte elements from sequential memory addresses, and calculates a weighted average for the $1^{st}$ and $2^{nd}$, the $2^{nd}$ and $3^{rd}$, the $3^{rd}$ and $4^{th}$, and the $4^{th}$ and $5^{th}$ byte elements.

*Data cache capacity.* The 128 Kbyte data cache is able to capture the working set of our proprietary video processing algorithms at either SD or HD resolution.

*Allocate-on-write-miss policy*. The processor allocates, rather than fetches a cache line when a write miss occurs, reducing the cache miss penalty and bandwidth to off-chip memory. Byte valid bits are maintained to track the validity of individual bytes in a cache line.

TABLE II
SOME OF TM3270 NEW OPERATIONS

| Operation | Description |
|---|---|
| SUPER_QUADUSCALEMIXUI<br>rsrc1 rsrc2 rsrc3 rscr4 ->rdest1;<br><br>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;<br>rdest1[31:24] = Min (Max (0, temp), 255);<br>temp = (rsrc1[23:16]*rsrc2[23:16] + rsrc3[23:16]*rsrc4[23:16] + 32) / 64;<br>rdest1[23:16] = Min (Max (0, temp), 255);<br>temp = (rsrc1[15:8]*rsrc2[15:8] + rsrc3[15:8]*rsrc4[15:8] + 32) / 64 ;<br>rdest1[15:8] = Min (Max (0, temp), 255);<br>temp = (rsrc1[7:0]*rsrc2[7:0] + rsrc3[7:0]*rsrc4[7:0] + 32) / 64;<br>rdest1[7:0] = Min (Max (0, temp), 255); |
| SUPER_QUADUMEDIAN rsrc1 rsrc2 rsrc3 -> rdest1;<br><br>Semantics: Median of 8-bit unsigned integers. | rdest1[31:24] = Min (Max (Min (rsrc1[31:24], rsrc2[31:24]), rsrc3[31:24]),<br>Max (rsrc1[31:24], rsrc2[31:24]));<br>rdest1[23:16] = Min (Max (Min (rsrc1[23:16], rsrc2[23:16]), rsrc3[23:16]),<br>Max (rsrc1[23:16], rsrc2[23:16]));<br>rdest1[15:8] = Min (Max (Min (rsrc1[15:8], rsrc2[15:8]), rsrc3[15:8]),<br>Max (rsrc1[15:8], rsrc2[15:8]));<br>rdest1[7:0] = Min (Max (Min (rsrc1[7:0], rsrc2[7:0]), rsrc3[7:0]),<br>Max (rsrc1[7:0], rsrc2[7:0])) ; |
| LD_FRAC8 rsrc1 rsrc2 -> rdest1;<br><br>Semantics: Collapsed load; load combined with linear interpolation. | data0 = Mem[rsrc1]; data1 = Mem[rsrc1 + 1]; data2 = Mem[rsrc1 + 2];<br>data3 = Mem[rsrc1 + 3]; data4 = Mem[rsrc1 + 4];<br>rdest1[31:24] = (data0*(16-rsrc2[3:0]) + data1*rsrc2[3:0] + 8) / 16;<br>rdest1[23:16] = (data1*(16-rsrc2[3:0]) + data2*rsrc2[3:0] + 8) / 16;<br>rdest1[15:8] = (data2*(16-rsrc2[3:0]) + data3*rsrc2[3:0] + 8) / 16;<br>rdest1[7:0] = (data3*(16-rsrc2[3:0]) + data4*rsrc2[3:0] + 8) / 16; |

TABLE III
IMPLEMENTATIONS A TO F, THEIR USE OF NEW OPERATIONS.

| Impl. | Unaligned load/store support | Horizontal fractional position | Vertical fractional position | Average, mixer and fader implementation | Median implementation |
|---|---|---|---|---|---|
| A | no | - | - | - | - |
| B | yes | - | - | - | - |
| C | yes | LD_FRAC8 | - | - | - |
| D | yes | LD_FRAC8 | SUPER_QUADUSCALEMIXUI | - | - |
| E | yes | LD_FRAC8 | SUPER_QUADUSCALEMIXUI | SUPER_QUADUSCALEMIXUI | - |
| F | yes | LD_FRAC8 | SUPER_QUADUSCALEMIXUI | SUPER_QUADUSCALEMIXUI | SUPER_QUADUMEDIAN |

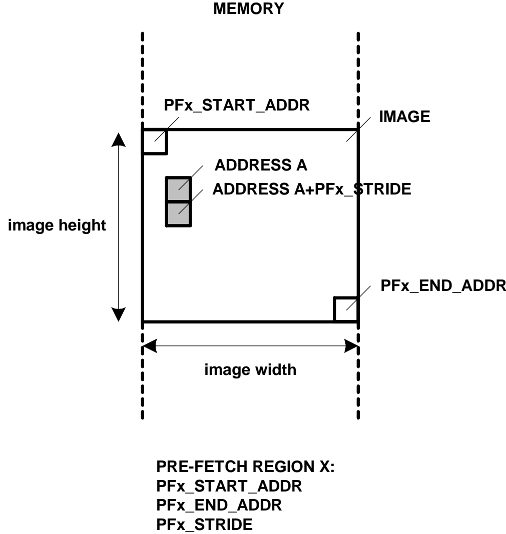



Fig. 5. Memory region based prefetching

*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. 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 5). 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 prefetched. 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 stride value to the cache line size, traditional next-sequential cache line prefetching is implemented.

## V. UP-CONVERSION IMPLEMENTATIONS

We evaluated six different software implementations of the temporal up-conversion algorithm, as described in Section II. The implementations all provide the same functionality, but they differ in the extent to which they exploit the TM3270's architectural enhancements. This allows us to quantify the contribution of the individual enhancements.

All implementations operate on SD NTSC images of 720*480 pixels at 60 Hz, resulting in a total of 21600 4x4 blocks per image. A temporal up-conversion to 85 Hz was used in the experiments. We restrict ourselves to the up-conversion of luminance vales. All implementations use memory region based prefetching. For the two source images, 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 = 24, being the maximum value of the vertical component of the motion vector).

Since video objects typically cover multiple 4x4 blocks, the motion vector field tends to show a certain continuity (motion vectors of neighboring blocks are often similar in size and direction). We decided not to rely on this characteristic; we use a random motion vector field, rather than an extracted motion vector field from a video sequence. As a result, our performance measurement results reflect worse case, rather than typical case, execution behavior. We support motion vectors at ¼ pixel resolution in both horizontal and vertical direction. The horizontal component of the

TABLE IV

PERFORMANCE RESULTS FOR IMPLEMENTATIONS A TO F (PREFETCHING ON, ALLOCATE-ON-WRITE-MISS POLICY, 0 DELAY CYCLE S).

| Implementation | Image average | | | | | MHz. req. for SD @ 85 images/sec. | % of 450 MHz. processor performance |
|---|---|---|---|---|---|---|---|
| | Cycles | Stall cycles | VLIW instructions | Cycles/ VLIW instr. | Operations/ VLIW instr. | | |
| A | 1592693 | 73203 | 1519490 | 1.05 | 4.64 | 135 | 30.1 |
| B | 1344255 | 73766 | 1270489 | 1.06 | 4.64 | 114 | 25.4 |
| C | 1150785 | 73855 | 1076930 | 1.07 | 4.63 | 98 | 21.7 |
| D | 1091897 | 74366 | 1017531 | 1.07 | 4.71 | 93 | 20.6 |
| E | 1040331 | 77281 | 963050 | 1.08 | 4.75 | 88 | 19.7 |
| F | 943700 | 72938 | 870762 | 1.08 | 4.79 | 80 | 17.8 |

motion vectors is taken from the range [-128, 127 ¾], and the vertical component of the motion vectors is taken from the range [-24, 23 ¾] (video motion is typically more dominant in the horizontal direction).

Table III gives a summary of the six implementations. Our base line, *implementation A*, uses neither unaligned load/store operations, nor any other new operations. *Implementation B* uses unaligned load/store operations. Unaligned memory access reduces the need to reorganize 8-bit video pixel elements. Note that for the calculation of horizontal fractional pixels, five horizontal neighboring pixels are used, requiring two load operations. *Implementation C* uses the collapsed LD_FRAC8 operation. This operation loads five horizontal neighboring pixels and performs an interpolation between two neighboring pixels. The operation allows for an efficient implementation of the horizontal fractional pixel calculation of motion compensated pixels. Furthermore, the operation halves the amount of load operations. *Implementation D* uses the two-slot SUPER_QUADUSCALEMIXUI operation. Its use is restricted to the implementation of vertical fractional pixel calculation of motion compensated pixels. Compared to an implementation using original TriMedia architecture operations, a reduction of 50% in the issue slot bandwidth utilization is achieved. *Implementation E* uses the two-slot SUPER_QUADUSCALEMIXUI operation. It extends the use of this operation beyond that of implementation D, the operation is used to efficiently implement the average, mixer, and fader functionality. *Implementation F* uses the SUPER_QUADUMEDIAN operation. It is used to efficiently implement the median functionality. A 3-taps median filter is defined as follows:

$$Median\ (a, b, c) = Min\ (Max\ (Min\ (a, b), c), Max\ (a, b))$$

The original TriMedia architecture supports the “4 x 8-bit” QUADUMAX and QUADUMIN operations. Both operations are executed on DSPALU functional units (3 instantiations are available), and have a latency of 2 cycles. A 3-taps median filter on “4 x 8-bit” operands occupies 4 issue slots, and produces a compiler schedule with a compound latency of 6 cycles. The TM3270 supports the two-slot SUPER_QUADUMEDIAN operation. The operation is executed on a SUPER-DSPALU functional unit. The operation occupies two neighboring issue slots (a reduction of a factor 2), and has a latency of 2 cycles (a reduction of the compound latency by a factor 3).

## VI. MEASUREMENTS AND RESULTS

All six implementations were simulated in our cycle accurate SoC environment, as described in Section III. 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.

### *A. Comparing the implementations*

To compare the performance of the implementations, we simulated them under the same SoC conditions: prefetching is turned on, an allocate-on-write-miss policy is used, and the delay block adds 0 cycles delay in the memory clock domain. Table IV gives the simulation results.

The implementations have a cycles-per-VLIW ratio (CPI) in the range of [1.05, 1.08] (a CPI of 1 is the theoretical optimum). The low percentage of stall cycles is mainly due to the efficiency of prefetching. The issue slot utilization is close to the theoretical maximum of 5 (TM3270 has a 5 issue VLIW architecture), reflecting high operation scheduling efficiency for the up-conversion algorithm.

Implementation B shows that the support of unaligned load operations significantly improves processor performance. Implementation C uses the LD_FRAC8 operation. This operation not only accelerates the horizontal fractional pixel calculation, but also halves the amount of load operations required to calculate a motion compensated 4x4 block of pixels. Additional support of two-slot operations further improves performance. Enabling all of the new TM3270 operations improves performance by roughly 41% (implementation A vs. implementation F).

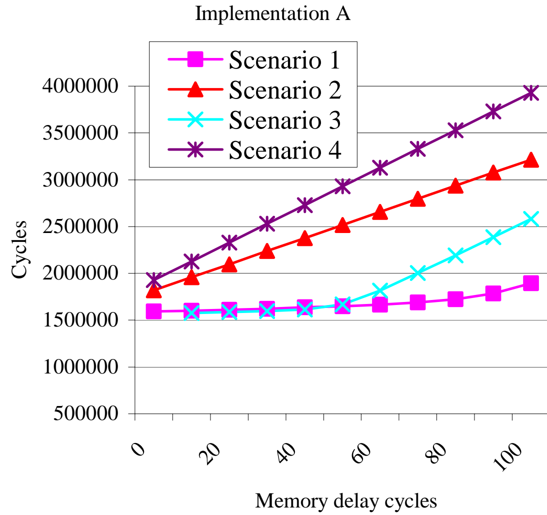
### *B. Influence of prefetching*

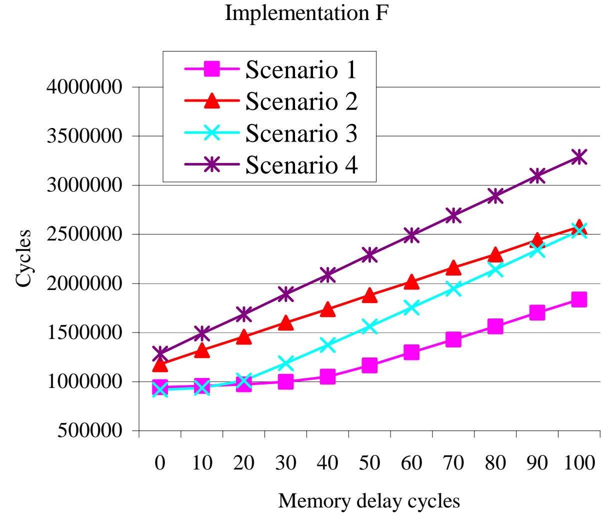
To quantify the contribution of prefetching to processor performance, we simulated implementations A and F with prefetching turned on (scenario 1) and prefetching turned off (scenario 2). Table V gives the simulation results. When prefetching is turned off, the amount of data cache misses increases, degrading processor performance. For 0

TABLE V

PERFORMANCE RESULTS FOR IMPLEMENTATIONS A AND F (DIFFERENT DELAY CYCLES, PREFETCH AND WRITE-MISS POLICIES).

| **Implementation** | Memory delay cycles | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| Scenario 1: Prefetching, allocate-on-write-miss policy | | | | | | | | | | | |
| A | 1592693 | 1600697 | 1609906 | 1621620 | 1636600 | 1649298 | 1665309 | 1690936 | 1724163 | 1786625 | 1893428 |
| F | 943700 | 955382 | 971631 | 998285 | 1051851 | 1166124 | 1296844 | 1428693 | 1563878 | 1700985 | 1836183 |
| Scenario 2: No prefetching, allocate-on-write-miss policy | | | | | | | | | | | |
| A | 1819102 | 1959327 | 2097728 | 2238262 | 2377077 | 2517321 | 2656075 | 2796328 | 2935369 | 3075623 | 3214557 |
| F | 1177986 | 1321405 | 1457453 | 1601024 | 1738244 | 1880860 | 2017763 | 2161344 | 2295430 | 2442317 | 2576074 |
| Scenario 3: Prefetching, fetch-on-write-miss policy | | | | | | | | | | | |
| A | 1570562 | 1577825 | 1586476 | 1597652 | 1615853 | 1668595 | 1816605 | 2003844 | 2192863 | 2387618 | 2580891 |
| F | 917729 | 935667 | 1011365 | 1187154 | 1372551 | 1561989 | 1755179 | 1948117 | 2143462 | 2340515 | 2536786 |
| Scenario 4: No prefetching, fetch-on-write-miss policy | | | | | | | | | | | |
| A | 1927048 | 2128510 | 2327918 | 2528756 | 2728536 | 2929260 | 3128990 | 3330039 | 3529292 | 3730352 | 3929539 |
| F | 1285505 | 1489392 | 1686914 | 1892374 | 2088612 | 2292813 | 2490184 | 2694473 | 2891830 | 3096706 | 3290896 |




additional memory delay cycles, the performance of scenario 2 is 20% worse than the performance of scenario 1 (implementation F). As the amount of additional memory delay cycles increases, the cache miss penalty increases, degrading processor performance even further.

The effectiveness of prefetching depends on the processor's ability to overlap prefetching with computation. Beyond the point at which both can be overlapped, both scenario 1 and scenario 2 show a similar dependency on additional memory delay cycles. This is reflected by the similar steepness of the right hand side of the performance curves for scenario 1 and 2.

### *C. Influence of cache write-miss policy*

To quantify the contribution of the data cache write-miss policy to processor performance, we simulated implementations A and F with both an allocate-on-write-miss policy (scenario 1 and 2), and a fetch-on-write-miss policy (scenario 3 and 4). The allocate-on-write-miss policy allocates, rather than fetches, a cache line, when a write miss to the line occurs. As a result, data cache stall cycles related to write misses are reduced. Furthermore, memory bandwidth requirements are reduced.

For scenario 3 prefetching is on. We use memory region based prefetching to retrieve data from the address region of the up-converted image (the associated data elements are overwritten by the algorithm), to reduce write miss stall cycles. The prefetch memory region stride is set such that while computing a pixel at image position $\begin{pmatrix} c_x \\ c_y \end{pmatrix}$, the pixel at position $\begin{pmatrix} c_x \\ c_y + 4 \end{pmatrix}$, is prefetched (with '4' being the height of our 4x4 block of pixels). For scenario 4 prefetching is off.

The fetch-on-write-miss policy increases the memory bandwidth requirements of the algorithm. This is because an additional image has to be retrieved from memory. For scenario 3, the increased bandwidth is responsible for a higher cycle count as soon as prefetching and computation cannot be overlapped. For implementation A this point is reached at approximately 50 additional memory delay cycles, for implementation F this point is reached at 20 additional cycles.

When the amount of additional memory delay cycles is small, scenario 3 outperforms scenario 1. This is explained as follows. For scenario 1, the allocate-on-write-miss policy is used to resolve a write miss. The

allocation of a cache line takes a few cycles to complete and may introduce stall cycles. Scenario 3, however, will not incur any write misses when prefetching is performed in time. The ability to perform prefetching in time is dependent on the ability to overlap prefetching with computation. This ability decreases when the memory latency increases. As a result, scenario 3 only outperforms scenario 1 when the amount of additional memory delay cycles is small.

The additional bandwidth requirement for scenario 4 (over scenario 2) is reflected by the steepness of the performance curves: for scenario 4 the curve is steeper than for scenario 2. A similar behavior is observed when comparing the right hand side of the curves for scenario 3 and scenario 1.

### *D. Influence of memory latency*

To quantify the influence of SoC SDRAM memory latency on processor performance, we simulated implementations A and F 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. Table V gives the simulation results.

As expected, increased memory latency decreases performance. The amount of performance degradation is not only dependent on the additional memory delay, but also on the ability of the processor to overlap prefetching and computation.

For implementation F (scenario 1), we can observe a discontinuity in the performance curve, as a function of additional memory delay. Up to roughly 30 additional delay cycles the performance degradation is limited, above 30 delay cycles performance degradation is more severe. The discontinuity represents the point beyond which the processor is no longer able to efficiently overlap prefetching with computation. Note that for implementation A (scenario 1), a similar discontinuity can be observed at around 80 additional memory delay cycles. Since implementation A uses more VLIW instructions, the ability to overlap prefetching with computation is present for a larger amount of additional memory cycles than for implementation F.

For scenarios 2 and 4 prefetching is turned off. As the amount of additional memory delay cycles increases, the cache miss penalty increases, resulting in an almost linear dependency between memory delay cycles and processor performance.

## VII. CONCLUSION

The simulation results show that real-time temporal video up-conversion of a SD NTSC video signal to a display frequency of 85 Hz can be performed in 17.8% of the available processor performance (implementation F, scenario 1, 0 delay cycles). Recently, improvements to our up-conversion algorithm were described to address artifacts in so called occlusion areas [6]. These algorithmic improvements can relatively easily be mapped to the proposed architecture, due to the flexibility of the architecture and the available processor performance.

The use of new operations significantly reduces the cycle count. We found a speed up of 41%, when comparing implementations A and F, for scenario 1, at 0 additional memory delay cycles. As the memory delay increases, the algorithm becomes memory bound, and the benefit of the new operations decreases.

Data prefetching improves performance of implementation F by 20%, at 0 additional delay cycles (scenario 1 vs. scenario 2). As the memory delay increases, the benefit of prefetching increases.

The allocate-on-write-miss policy reduces memory bandwidth requirements. Its effect is most noticeable for large memory delays. Implementation F shows an improvement of 28%, at 100 addition delay cycles (scenario 1 vs. scenario 3).

## REFERENCES


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

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

[3] J.W. van de Waerdt, J.P. van Itegem, G. Slavenburg and S. Vassiliadis, "Motion estimation performance of the TM3270 processor", *ACM Symposium on Applied Computing*, March 2005.

[4] O. Ojo and G. de Haan, "Robust motion-compensated video up-conversion", *IEEE Transactions on Consumer Electronics*, vol. 43, No. 4, pp. 1045-1056, November 1997.

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

[6] R.B. Wittebrood, G. de Haan and R. Lodder, "Tackling occlusion in scan rate conversion systems", *Digest of the ICCE'03*, pp. 344-345, June 2003.

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

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

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

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