Implementation and Evaluation of the Complex Streamed Instruction Set

Ben Juurlink[1] Dmitri Tcheressiz[2] Stamatis Vassiliadis[1] Harry A.G. Wijshoff[2]

[1]*Computer Engineering Laboratory*
Delft University of Technology
Delft, The Netherlands

[2]*Leiden Institute of Advanced Computer Science*
Leiden University
Leiden, The Netherlands

Abstract

An architectural paradigm designed to accelerate streaming operations on mixed-width data is presented and evaluated. The described Complex Streamed Instruction (CSI) set contains instructions that process data streams of arbitrary length. The number of bits or elements that will be processed in parallel is, therefore, not visible to the programmer, so no recompilation is needed in order to benefit from a wider datapath. CSI also eliminates many overhead instructions (such as instructions needed for data alignment and reorganization) often needed in applications utilizing media ISA extensions such as MMX and VIS by replacing them by a hardware mechanism. Simulation results using several multimedia kernels demonstrate that CSI provides a factor of up to 9.9 (4.0 on average) performance improvement when compared to Sun's VIS extension. For complete applications, the performance gain is 9% to 36% with an average of 20%.

1 Introduction

It is anticipated that multimedia applications such as JPEG and MPEG coders/decoders will become dominant workloads in the near future [7, 14]. Multimedia codes typically process small data types (for example, 8-bit pixels or 16-bit audio samples) and thus are not well-suited for common general-purpose systems which are optimized for processing word-size data (32 or 64 bits). Many vendors have, therefore, extended their instruction set architecture (ISA) with instructions targeted to multimedia applications. These instructions exploit SIMD parallelism at the subword level, i.e., they operate concurrently on, e.g., eight bytes or four halfwords packed in one 64-bit register. Examples of such media ISA extension are MMX [21, 22], VIS [27], MDMX [18], MAX [15, 16], and AltiVec [9].

Although it has been shown that these extensions improve the performance of many multimedia applications (see, e.g., [1, 19, 24]), they have several limitations. One is that the number of bits or elements that are processed in parallel, which is equal to the multimedia register size, is visible to the programmer. This implies that when the width of the SIMD datapath is increased in order to process more elements in parallel, either the ISA has to be changed implying that existing codes have to be recompiled or rewritten, or the issue width has to be increased. Another limitation is overhead for data reorganization. This includes pack/unpack instructions and alignment-related instructions. Furthermore, SIMD-style instructions are most effective if data is stored consecutively. Multimedia applications, however, often operate on sub-blocks of a large matrix, which implies that there is a gap between consecutive rows.

In this paper we present and evaluate an ISA extension called CSI (*Complex Streamed Instructions*) that addresses these problems as well as several others. CSI instructions process two-dimensional data streams. There is no architectural (i.e., programmer-visible) constraint on the length of the streams. Instead, the hardware is responsible for dividing the streams into sections which are processed in parallel. CSI also eliminates the need for pack/unpack instructions, because conversion between different packed data types (if required) is performed internally in hardware. Furthermore, because streams are two-dimensional, loop overhead instructions associated with updating of pointers and branching are also avoided.

This paper is organized as follows. Section 2 describes the limitations of current media ISA extensions and explains how the CSI paradigm solves these problems. The CSI ISA extension and its implementation are described in Section 3. Section 4 describes the benchmarks, the modeled architectures, and presents the experimental results. Related work is discussed in Section 5, and concluding remarks and topics for future research are given in Section 6.

2 Motivation

In this section we list some of the limitations of current media ISA extensions and describe how they are solved in the CSI architecture.

0-7695-1363-8/01 $10.00

Architectural Constraint on Section Size. All current media ISA extensions as well as most vector architectures have an architectural (i.e., programmer-visible) fixed section size. For example, MMX and VIS instructions operate on 64-bit registers which can be treated either as 8 bytes, 4 halfwords, or 2 words. Because of this, the section size appears explicitly in the code. This, however, means that if the width of the SIMD datapath is increased in order to exploit more parallelism, the ISA may have to be changed to reflect this. This implies that (parts of) the application may have to be recompiled or even rewritten in order to benefit from the wider datapath. For example, if MMX would operate on 128-bit registers, existing MMX codes must be adapted.

Another way to increase parallelism is by increasing the issue width so that more SIMD instructions can be processed in parallel. However, it is generally accepted that increasing the issue width requires a substantial amount of hardware and may negatively affect the cycle time [10, 20].

In CSI these problems are avoided because CSI instructions process data streams of arbitrary length. The implementation is responsible for dividing the data streams into sections which are processed in parallel. Therefore, the number of elements that is processed in parallel does not appear explicitly in the code.

Increasing Parallelism. A problem related to the previous is the following. Although it may be possible to increase the width of the datapath and the register size, it may not always be beneficial because many multimedia applications operate on sub-blocks of a large matrix (representing, e.g., an image), and the vector length in both directions is rather short (typically 8 or 16 bytes). Consider, for example, Figure 1 which shows a C-function taken from an MPEG encoder. The rows of the `pred` and `cur` blocks are not stored consecutively in memory. Consequently, as was also observed in [6], the amount of parallelism that can be exploited by a SIMD extension is restricted to a single row. In order to be able to exploit more parallelism, CSI instructions operate on two-dimensional streams.

Minimizing Overhead. The execution of a multimedia kernel typically consists of the following steps: (1) load operand data into registers, (2) rearrange data so that operand elements are stored consecutively, (3) convert data from storage to computational format, (4) perform the actual computation, (5) convert the results to their storage format, (6) rearrange the results and, finally, (7) store them in memory. ISA extensions like MMX and VIS have explicit instructions to perform data conversion and rearrangement, and, for example, [24] showed that on average, they constitute 41% of the VIS instructions. In CSI this overhead is eliminated by performing data conversion and rearrangement implicitly in hardware and by pipelining it with the actual computation. This is described in detail below.

```
static void add_pred(pred,cur,lx,blk)
unsigned char *pred, *cur;
int lx;
short *blk;
{
  int i, j;

  for (j=0; j<8; j++){
    for (i=0; i<8; i++)
      cur[i] = clp[blk[i] + pred[i]];
    blk+= 8;
    cur+= lx;
    pred+= lx;
  }
}
```

Figure 1. C code for saturating add.

a. Non-Unit Strides. SIMD extensions are most effective if the vector elements are stored consecutively, for otherwise, they need to be reordered. In some multimedia applications, however, consecutive stream elements are stored at a fixed but non-unit stride. This happens, for example, in JPEG's color conversion routine where the Red, Green and Blue components are stored at a stride of 3. In the upsampling/downsampling phases of JPEG, data is also accessed with a non-unit stride. In CSI, consecutive stream elements pertaining to the same row do not have to be stored in consecutive memory location. Thus, we allow any stride between two consecutive row elements, as well as between consecutive rows. The hardware implementation is responsible for aligning them properly. This is one of the differences with MOM [6], which allows an arbitrary stride between consecutive rows but requires a unit-stride between consecutive row elements.

b. Computing with Different Formats. When we consider Figure 1 again, we observe that one of the blocks consists of 16-bit (short) elements, whereas the other consists of 8-bit elements. When these two blocks are added using SIMD instructions, the `pred` block must be unpacked (or *promoted*) to a 16-bit format. Data promotion may also be required when the input elements have the same size, because the result may not be representable by this format. This incurs a performance penalty of at least a factor of 2, due to the reduced parallelism and the overhead caused by pack/unpack operations. Because of this, many media ISA extensions have instructions that automatically saturate to the smallest or largest value the data type can represent. (In VIS, saturation is performed while packing.)

In CSI, these problems are resolved as follows. When packing/unpacking is necessary because the input streams have different formats (as in the example shown in Figure 1), it is performed internally in hardware. No special opcodes are needed to specify that, for example, one of the input streams consists of 16-bit elements and the other of 8-

bit elements, because a control register associated with each stream specifies the element width. If packing/unpacking is not required, the programmer can specify that saturation arithmetic should be performed instead of "wrap-around arithmetic" by setting a bit in another control register. This is another difference with MOM [6], which can also perform saturation arithmetic, but which still requires packing/unpacking if the input streams have different formats.

c. Data Alignment and Loop Control. There are other instructions besides packing and unpacking that contribute to the overhead. This includes alignment-related and loop control instructions. For example, VIS needs alignment instructions if a vector is not stored at an 8-byte aligned address. Loop control instructions are the instructions required for breaking the data stream into fixed-size sections which are processed in parallel. This includes instructions needed to advance the pointers to the next sections, instructions that compute the loop termination condition, and branch instructions. In the CSI architecture, these functions are also replaced by a hardware mechanism. The hardware generates aligned addresses and is responsible for extracting the bytes that belong to the data stream. Furthermore, since CSI instructions process streams of arbitrary length, no loop control instructions are needed.

Memory-to-Memory Operation. CSI is a memory-to-memory architecture for two-dimensional streams. The decision not to use registers for stream data was motivated by the intention not to have an architectural constraint on the section size, and by the observation that vector registers are not always able to exploit data reuse in multimedia applications while caches are. For example, during the motion estimation phase of `mpeg2encode` the best matching block is searched by comparing it with a number of candidates in the reference frame. The new candidate block is usually shifted just a few pixels from the previous one. Candidate blocks, therefore, largely overlap. This results in high data reuse (and L1 hit rate), but there is no obvious way to exploit this using traditional vector registers. Another example can be found in the JPEG decoder. There, data from a large (several Kbytes) buffer flows in a pipelined fashion through several kernels. There is not much data reuse within the kernels themselves and the buffer is too large to be stored in a vector register. The cache, on the other hand, is likely to be large enough to store the buffer. We remark that when a kernel performs several operations on each stream element, one has to introduce temporaries to store intermediate results. However, we observed that writing and reading these intermediate streams did not stress the memory bandwidth because they have unit stride and usually were stored and retrieved from the L1 cache without going to lower levels of the memory hierarchy.

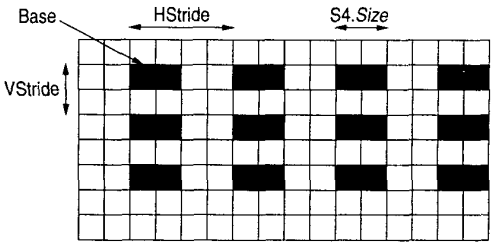

Figure 2. Two-dimensional stream format. Each box represents a byte. Filled boxes are stream elements.

3 Architecture and Implementation

In this section we present the CSI multimedia ISA extension. A possible implementation is also described.

3.1 CSI Overview

CSI is a memory-to-memory architecture. Most CSI instructions load two large data input streams from memory, operate on them element-wise, and write the resulting stream back to memory. There is no architectural constraint on the stream length. As illustrated in Figure 2, streams are two-dimensional. Each stream consists of an arbitrary number of rows, and the row elements are stored at a fixed stride which will be referred to as **HStride** (short for horizontal stride). There is also a fixed stride between consecutive rows, which will be referred to as **VStride**.

Each stream is specified by a *set of stream control registers* (SCR-set), which consists of the following 32-bit registers, each of which is addressed by a number between 0 and 5.

0. **Base**. This register contains the starting or base address of the stream. For example, if the matrix in Figure 2 is stored in row-major order and its base address is 8000, the base address of the stream is 8018.
1. **RLength**. This register holds the number of stream elements in a row (the number of elements belonging to the stream, *not* the row length of the enveloping matrix). In the example, **RLength**=4.
2. **SLength**. This register contains the stream length. In the example illustrated in Figure 2, **SLength**=12.
3. **HStride**. The stride in bytes between consecutive stream elements in a row.
4. **VStride**. The distance in bytes between consecutive rows.
5. **S4**. This register consists of four fields: ***Size***, ***Scale factor***, ***Sign*** and ***Saturate***. The first field consists

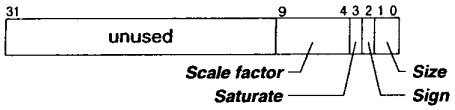

Figure 3. S4 register format

of two bits and specifies the size of the stream elements, where 00 corresponds to bytes, 01 half-words, 10 words, and 11 double-words. The ***Sign*** field is a flag that specifies if the stream elements are signed or unsigned values. The ***Saturate*** field is also a flag that specifies if saturation or modular arithmetic should be performed. If this bit is set and the result cannot be represented by the number of bytes indicated by the ***Size*** field, the result is clipped to the minimum or maximum value. If this bit is not set, the result simply "wraps-around". The function of the ***Scale factor*** field is identical to the ***Scale factor*** field of the Graphics Status Register of the VIS architecture [30]. It determines the number of bits by which the result is shifted to the left before the least significant bits (LSBs) are rounded and discarded. The number of bits discarded depends on the size of the operands and the result. For example, when packing from 16 to 8 bits, the 7 LSBs are discarded. This mechanism allows to specify the number of fractional bits. The format of the **S4** register is depicted in Figure 3.

The CSI instruction set is divided in two categories:

1. **CSI arithmetic and logical instructions**. These instructions have the following formats:
   - `op SCRSi, SCRSj, SCRSk`
     Such instructions process two data input streams and produce a data output stream. The streams are specified by the corresponding SCR-sets. Examples are pairwise addition, subtraction and multiplication of two data streams.
   - `op SCRSi, SCRSj, GPRk`
     These instructions are similar to the previous ones but the second operand is not a data stream but a scalar value. An example of such an instruction is the multiplication of a data stream by a scalar.
   - `op GPRi, SCRSj, SCRSk`
     These instructions process two input streams and produce a scalar result. Two examples are `csi_sad` and `csi_dotprod`, which compute the sum of absolute differences and dot product of two data streams, respectively.
2. **CSI auxiliary instructions**. These instructions set the individual stream control registers.
   - `csi_mtscr SCRSi, j, GPRk`
     `mtscr` stands for *move to stream control register*. This instruction loads SCR `j` of SCR-set `SCRSi` with the contents of the general purpose register `GPRk`. The stream control registers are numbered as above. For example, the base address of SCR-set `SCRS2` can be loaded from `GPR4` using `csi_mtscr SCRS2, 0, GPR4`.
   - `csi_mtscri SCRSi, j, imm`
     This instructions also loads a stream control register but with an immediate value.

Note that since the element size and the ***Sign*** and ***Saturate*** bits are set using control registers, the CSI ISA extension is quite compact and actually smaller than SIMD extensions such as VIS and MMX. For example, seven MMX instructions `padd[b,w,d]` (add with wrap-around on [byte, word, double-word]), `padds[b,w]` (add signed with saturation) and `paddus[b,w]` (add unsigned with saturation) correspond to just one CSI instruction `csi_add` which can add streams of signed as well as unsigned bytes, halfwords (words in Intel terminology) and words, and which also performs saturation if the ***Saturate*** bit is set.

As an example, Figure 4 shows the CSI code for the `add_pred` routine depicted in Figure 1. For each data stream, 6 instructions are needed to set the control registers, after which the `csi_add` instruction is triggered. These 18 instructions needed to set the SCRs might seem significant at first but mostly they are negligible for the following reasons. First, these instructions are executed only once and their number is still very small compared to the number of instructions that must be executed by a scalar processor. In this example, 64 iterations (with about 10 scalar instructions each, depending on the compiler) are replaced by 18 instructions that manipulate the SCRs and a single `csi_add` instruction. Second, in many cases, not all SCRs have to be reset to initiate a new CSI instruction. For example, the `add_pred` routine is executed on many different blocks. This means that after all SCRs are set the first time, only the base addresses have to be reset. The overhead is, therefore, amortized over many instructions.

We remark that the structure of the CSI architecture allows some powerful instructions to be constructed. For example, the arithmetic average of two 8-bit pixel streams can be calculated using the `csi_add` instruction by setting the ***Scale factor*** field of the SCR-set corresponding to the destination stream to 6. So, the results of the `csi_add` instruction are shifted to the left by 6 bit positions, after which the 7 least significant bits are rounded and then discarded.

All CSI instructions can be interrupted during execution. However, we first observe that arithmetic overflow does not generate an exception, since either wrap-around or saturation arithmetic is performed. Other exceptions, such as page

```
# GPRi is denoted as $i
# We assume pred=$4, curr=$5, lx=$6, blk=$7
# Set SCRs for blk stream
csi_mtscr      SCRS1,0,$7     # Base
csi_mtscri     SCRS1,1,8      # RLength
csi_mtscri     SCRS1,2,64     # SLength
csi_mtscri     SCRS1,3,2      # HStride
csi_mtscri     SCRS1,4,16     # VStride
#scale=0,saturate=0,sign=1,size=01(halfword)
# So,constant to load in S4 is:
# 00101(base 2) = 5(base 10)
csi_mtscri     SCR1,5,5       # S4

# Set SCRs for pred stream
csi_mtscr      SCRS2,0,$4     # Base
csi_mtscri     SCRS2,1,8      # RLength
csi_mtscri     SCRS2,2,64     # SLength
csi_mtscri     SCRS2,3,1      # HStride
csi_mtscr      SCRS2,4,$6     # VStride
# scale=0, saturate=0, sign=0, size=00(byte)
# So,constant to load in S4 is 00000=0
csi_mtscri     SCRS2,5,0      # S4

# Set SCRs for curr stream
csi_mtscr      SCRS3,0,$5     # Base
csi_mtscri     SCRS3,1,8      # RLength
csi_mtscri     SCRS3,2,64     # SLength
csi_mtscri     SCRS3,3,1      # HStride
csi_mtscr      SCRS3,4,$6     # VStride
# scale=0, saturate=1, sign=0, size=00(byte)
# So,constant to load in S4 is 01000=8
csi_mtscri     SCRS3,5,8      # S4
# Trigger streamed operation csi_add
csi_add        SCRS3,SCRS2,SCRS1
```

Figure 4. CSI code for the add_pred routine.

faults, can be handled as in the IBM System/370 vector architecture [2]. A *stream interruption index* is maintained that indicates which stream elements are currently being processed. If the instruction is interrupted, this internal register marks the point that has been reached. If the instruction is reissued, execution resumes from that point.

3.2 Implementation

In this section we describe the hardware implementation of the CSI architecture, which will be referred to as the stream unit. The datapath of the experimental stream unit is depicted in Figure 5. Its main hardware entities are the stream control register sets (SCR-sets), the memory interface unit, the pack and unpack units, one or more CSI functional units which perform SIMD parallel operations, and the accumulator ACC. For clarity, some of the paths have been omitted. For example, one of the inputs of the CSI functional units can be a general-purpose register or an

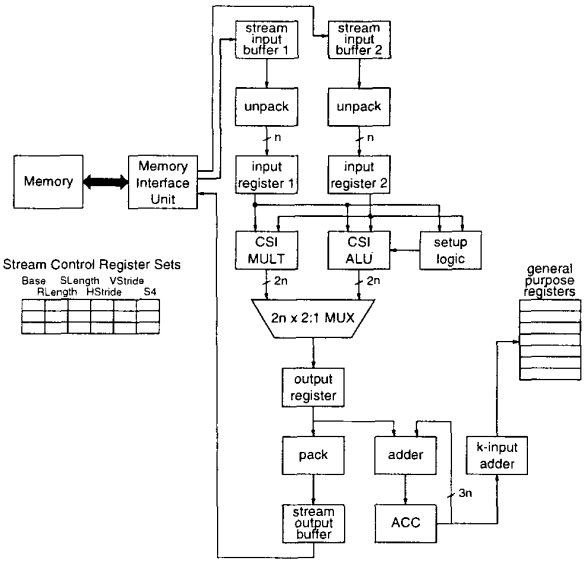

Figure 5. Datapath of the Stream Unit

immediate value, and there is also a path from the general-purpose registers to the stream control registers.

The memory interface unit is responsible for transferring data between the memory hierarchy and the stream buffers. In addition, if the data is not stored consecutively, it must also extract non-consecutive data from the cache and align them in the proper order. Its operation will be described in more detail below.

The unpack units convert stream data from storage format to computational format (if required). For this, they use the values of the ***Size*** and ***Sign*** fields of the SCR **S4**. For example, if one data input stream consists of unsigned bytes and the other consists of signed halfwords, the first is converted to 16-bit halfwords by padding with zeroes.

The CSI functional units perform subword parallel operations on the data contained in the input registers. Currently two CSI units are used: one CSI MULT unit that performs parallel multiplication and division, and one SIMD ALU that performs parallel addition and subtraction as well as the sum of absolute differences (SAD) operation. The setup logic is also used in the computation of the SAD operation. Following the scheme presented in [28], it determines the smallest of each pair of corresponding pixels contained in the input registers, and controls the CSI ALU so that it negates the smallest pixel of each pair. The size of the input registers is n, where n is implementation dependent, and the size of the output register is $2n$ so that no overflow occurs during computation.

From the output register, data flows either to the stream output buffer via the pack unit or to the accumulator. The pack unit converts the data from computational format to

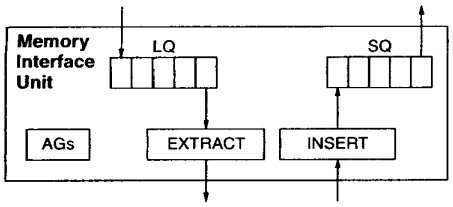

Figure 6. Memory Interface Unit

storage format. It also performs truncation and saturation, similar to the VIS pack instructions. For this, it uses ***Scale factor***, ***Sign***, ***Saturate*** and ***Size*** fields of the **S4** register corresponding to the output stream.

The accumulator is $3n$ bits wide. It is used in reduction operations such as the SAD and DOTPROD. It enables the accumulation of up to 2^n $n * n$ products without having to promote the operands to a larger format [18]. As mentioned, data promotion incurs a performance penalty due to the reduced parallelism and due to the cycles needed for executing pack/unpack instructions. Note that the stream unit performs data promotion only if the input streams have different formats, *not* when the result may become too large. Finally, the adder between the accumulator and the register file sums up the components contained in the accumulator. The accumulator can contain either $k = n/8$, $n/16$, $n/32$, or $n/64$ components.

Memory Interface Unit. We now describe the memory interface unit (MIU). One important issue is the following: should the MIU be connected to the level-1 (L1) cache, or should it bypass the L1 cache and go directly to the L2 cache or even main memory? In this study we decided to connect the MIU to a 3-ported (2 read , 1 write) L1 cache for the following reasons. First, Ranganathan et al. [24] observed that with realistic L1 cache sizes, multimedia applications achieve high hit rates. Our simulations support this observation. For example, with a 32K direct-mapped L1 data cache, all benchmarks exhibited hit rates over 99%. Another motivation is that since the L1 cache is on-chip, it will not be expensive to widen the path between the cache and the stream unit, so that a whole cache block can be brought to the stream unit in a single cycle. In the future, however, we intend to look at other memory organizations.

The memory interface unit is depicted in Figure 6. It consists of the following hardware entities: three address generators (AGs), a load queue (LQ) and a store queue (SQ), and extract and insert hardware.

The AGs generate the addresses of the cache blocks that must be fetched. After a CSI instruction has been issued, each AG aligns the **Base** address of its associated data stream to cache block boundaries, and inserts the aligned address into the load queue. Furthermore, with each LQ entry, a mask of *CBS* bits is associated, where *CBS* is the cache block size in bytes. This mask marks which bytes in the cache block belong to the stream. It is computed based on the values of the control registers **HStride**, **VStride** and **RLength**, and the ***Size*** field of the **S4** register. Each AG also updates some internal control registers in order to compute the address of the next block to fetch.

The load queue submits the load address to the cache read port. When the data arrives, it sets the ready flag of the corresponding entry. The store queue operates similarly.

The extract unit monitors the entry at the head of the LQ. When the ready flag of this entry is set, it extracts the useful bytes from it (based on the corresponding mask), and places them consecutively in an input stream buffer. It operates similar to a *collapsing buffer* [5]. The insert unit performs the inverse operation, i.e., it "scatters" the stream elements so that they are in their correct position, and places the cache block in the store queue. The SQ then performs a partial store, similarly to the VIS partial store instruction.

4 Evaluation

In order to evaluate the performance of the proposed ISA, we simulated a superscalar processor without a multimedia ISA extension, a superscalar processor with the VIS extension, and a processor extended with CSI instructions. We studied four benchmarks from the MediaBench [13] test suite: `mpeg2enc` (MPEG-2 encoder), `mpeg2dec` (MPEG-2 decoder), `cjpeg` (JPEG encoder), and `djpeg` (JPEG decoder). These programs are representative of video and image processing applications. For the MPEG benchmarks, we used the *test* bitstream, which consists of three 128 x 128 frames. For the JPEG benchmarks, the *rose* input was used, which is a 227 x 149 pixel image.

4.1 Simulation Methodology and Tools

We used the `sim-outorder` simulator of the SimpleScalar toolset (release 3.0) [3] to simulate a superscalar processor without and with VIS or CSI extensions. This is an execution-driven simulator that supports out-of-order issue and execution. Its architecture (Portable ISA or PISA) is derived from the MIPS-IV ISA [23]. CSI and VIS instructions were synthesized using annotations to instructions in the assembly files. We used a corrected version of the SimpleScalar memory model based on SDRAM specifications given in [8].

To our knowledge, there is no compiler that generates VIS code. We, therefore, had to write VIS (as well as CSI) code ourselves, but used code from the VIS Software Developer's Kit (VSDK) when possible. We remark that DCT routines are not available in the VSDK. They are available in the SUN mediaLib, but this library consists of bi-

Table 1. Processor configuration.

Clock rate	500MHz	
Issue width	4	
Register update unit size	16	
Load-store queue size	8	
Branch Prediction		
Bimodal predictor size	2K	
Branch target buffer size	2K	
Return-address stack size	8	
Functional unit types, number and cycles (latency, recovery)		
Integer ALU	4	(1/1)
Integer MULT	1	
multiply		(3/1)
divide		(20/19)
Cache ports	2	(1/1)
Floating-point ALU	4	(2/2)
Floating-point MULT	1	
FP multiply		(4/1)
FP divide		(12/12)
sqrt		(24/24)
VIS adder	2	(1/1)
VIS multiplier	2	
multiply and pdist		(3/1)
other		(1/1)

Table 2. Memory configuration.

Instruction cache	ideal
Data caches	
L1 line size	32 bytes
L1 associativity	direct-mapped
L1 size	32 KB
L1 hit time	1 cycle
L2 line size	128 bytes
L2 associativity	2-way
L2 size	1 MB
L2 replacement	LRU
L2 hit time	6 cycles
Main memory	
type	SDRAM
row access time	20 ns
row activate time	20ns
precharge time	20ns
bus frequency	100MHz
bus width	64bits

nary routines. The most time-consuming routines were first identified by profiling. After that, the functions that contained a substantial amount of data-level parallelism and whose key computation could be replaced by VIS and CSI instructions were rewritten manually. The loops were unrolled so that the loop bodies could be replaced by a set of equivalent VIS instructions. The selected kernels are: `Add_Block` (MPEG2 frame reconstruction), `Saturate` (saturation of 16-bit elements to 12-bit range in MPEG decoder), `dist1` (sum of absolute differences for motion estimation), `ycc_rgb_convert` and `rgb_ycc_convert` (color conversion between YCC and RGB color spaces in JPEG), and `h2v2_downsample` (2:1 horizontal and vertical downsampling of a color component in JPEG), and `idct` (inverse discrete cosine transform).

4.2 Modeled Architecture

It is important to note that the baseline architecture is SimpleScalar (i.e., PISA), not UltraSPARC. We have chosen VIS instead of the MIPS media ISA extension MDMX because, first, VIS is representative of many current media extensions [24], and, second, MDMX has no instruction that computes the sum of absolute differences (SAD). The SAD is used in motion estimation, which is the most time-consuming part of the MPEG encoder. With MDMX one should use the sum of squared differences [18] instead, but this would have required to modify the benchmarks.

The base system is a 4-way superscalar processor with out-of-order issue and execution based on the Register Update Unit (RUU). The processor parameters are listed in Table 1, and the parameters of the memory subsystem are listed in Table 2. Because the benchmarks used in this study have small instruction working sets, a perfect instruction cache is assumed. VIS instructions operate on the floating-point register file and have a latency of 1 cycle, except for the `pdist` (which computes the SAD) and the packed multiply instructions, both of which have a latency of 3 cycles. There are two VIS adders that perform partitioned add and subtract, merge, expand and logical operations, and two VIS multipliers that perform the partitioned multiplication, compare, pack and pixel distance operations. This is modeled after the UltraSPARC [27] with the following exceptions. In the UltraSPARC, the `alignaddr` instruction cannot be executed in parallel with other instructions [30] but this limitation is not present in the architecture we modeled. Furthermore, the UltraSPARC has only one 64-bit VIS multiplier. We assumed two because the width of the datapath of the stream unit is assumed to be 128 bits. The degree of parallelism of the VIS-enhanced and the CSI-enhanced architectures are, therefore, comparable.

All sub-units (i.e., pack/unpack, extract/insert, CSI adder etc.) of the stream unit require 1 cycle, except for the CSI multiplier, which requires 3 cycles but is fully pipelined. The datapath of the stream unit is 128 bits wide. So, the CSI functional units process either 16 bytes, 8 halfwords, 4 words, or 2 double-words in parallel. The input registers are therefore 128 bits wide, the output register 256 bits, and the accumulator 384 bits (cf. Figure 5).

Because one CSI instruction can replace two embedded loops, the requirements for the machine's fetch, decode and issue bandwidth will be greatly reduced. In order to evalu-

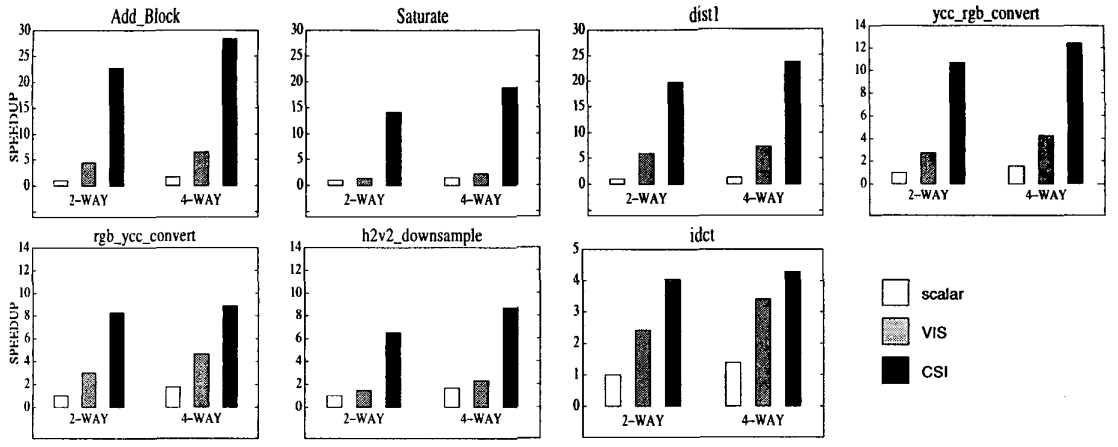

Figure 7. Kernel-level speedups (w.r.t. the 2-way superscalar processor).

ate this effect, we also simulated a 2-way superscalar processor in addition to a 4-way system.

Since VIS instructions are register-to-register and operate on the floating-point register file, they do not interfere with the existing processor pipeline. CSI instructions are memory-to-memory and require extra care. One must ensure that CSI instructions do not overlap with scalar memory instructions. We, therefore, took the following conservative approach: when a CSI instruction is detected, the pipeline is stalled until all memory instructions have committed. After that, the CSI instruction is issued and fetching resumes when it has finished.

4.3 Experimental Results

In this section we present the speedups attained by the two multimedia ISA extensions (VIS and CSI) considered. Speedups will be given with respect to the 2-way system. We first present results for several kernels from our benchmarks. After that, we analyze how kernel-level speedup translates to application speedup.

Figure 7 depicts the speedups attained for the seven kernels selected from the benchmarks. When the issue width is 2, the VIS-enhanced architecture achieves a speedup of 1.4 to 5.9 with an average of 3.1, whereas the CSI-enhanced architecture attains speedups ranging from 4.0 to 22.7 (12.3 on average). When the issue width is 4, the average speedup (w.r.t. to the 2-way system) of the VIS-enhanced architecture is 4.4 (2.3 to 7.2) and the average speedup of the CSI-enhanced architecture is 15.0 (4.2 to 28.3). So, CSI clearly outperforms VIS.

Especially on the `Saturate` kernel the CSI-enhanced architecture performs much better than the architecture extended with VIS instructions. Whereas the VIS-enhanced processor attains speedups of 1.43 (2-way issue) and 2.25 (4-way issue), the CSI-enhanced processor attains speedups of 14.1 and 18.8, respectively. The reason is that in this kernel 16-bit values have to be clipped to a 12-bit range and, simultaneously, the clipped values have to be accumulated. Because CSI instructions have saturation to any desired range as a feature (by setting the ***Saturate*** bit and adjusting the ***Scale factor*** field of the **S4** register), and because the accumulator accumulates all results, the body of the `Saturate` kernel is essentially replaced by one instruction. In the VIS-enhanced architecture, saturation and accumulation have to be performed explicitly in software.

It can be observed that the smallest performance improvement of the CSI-enhanced architecture over the VIS-enhanced architecture occurs for the `idct` kernel. The reason is that the VIS version is based on the scalar version, which in turn is based on a highly optimized DSP algorithm proposed in [4]. However, this DSP algorithm does not operate on long vectors and can therefore not be efficiently implemented using CSI instructions. The CSI version of the `idct` is based on the standard definition of the IDCT as two matrix multiplications. Thus, the CSI version of `idct` executes many more operations than the VIS version, but nevertheless a speedup is obtained.

The results for complete applications are depicted in Figure 8. For a 2-way issue machine, the VIS-enhanced architecture achieves speedups of 1.42 (on the `djpeg` benchmark), 1.17 (`cjpeg`), 1.40 (`mpeg2dec`) and 1.93 (`mpeg2enc`), whereas the CSI-enhanced architecture attains speedups of 1.94, 1.28, 1.70 and 2.28, respectively. When the issue width is 4, the respective speedups are 2.08, 1.59, 2.13 and 2.37 for the VIS-enhanced processor, and 2.75, 1.74, 2.48 and 2.77 for the CSI-enhanced processor. Of course, due to Amdahl's Law, the speedups for complete programs are less impressive than those for kernels. Nevertheless, when the issue width is two, the CSI-enhanced

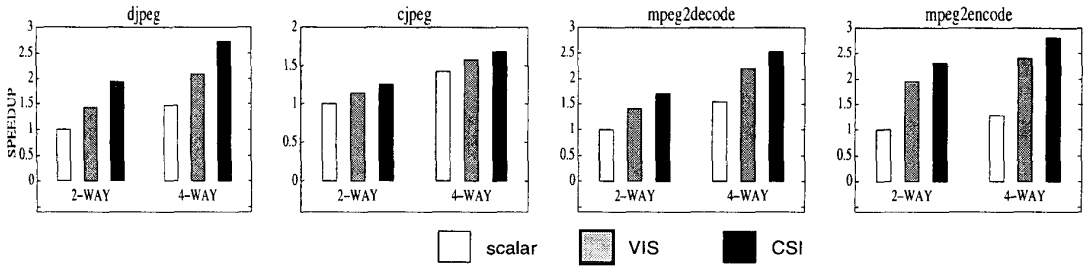

Figure 8. Application-level speedups (w.r.t. the 2-way superscalar processor).

architecture yields an average performance gain over VIS of 20% on average (range of 8% to 36%), and when the issue width is four, the average speedup of CSI over VIS is 18% (range of 8% to 32%).

Finally, we remark that when the issue rate is 2, the CSI-enhanced architecture attains higher speedups w.r.t. the VIS-enhanced architecture than when the base system is a 4-way processor. This means that the performance of the stream unit is rather insensitive to the processor issue width. This makes the CSI architecture highly suitable for embedded systems, where high issue rates and out-of-order issue and execution are too expensive. The same observation has been made in [6] for the MOM ISA extension.

5 Related Work

The CSI architectural paradigm was introduced in [29]. Since then we modified the architecture in order to accommodate more instructions using fewer opcodes. This is accomplished using the stream control registers. Furthermore, we significantly extend the work described in [29] by providing results for several other benchmarks, by including a detailed simulation of the memory hierarchy and by providing a comparison with VIS. In [29], just one benchmark application was considered (an MPEG encoder) and the effect of cache misses was imitated by varying the latencies of the CSI instructions.

CSI instructions eliminate the need for vector sectioning, i.e., bookkeeping instructions needed for processing vectors of arbitrary length in sections. An early proposal aimed at hiding the actual section size (which is implementation dependent) is the load vector count and update (VLVCU) instruction of the IBM/370 vector architecture [2].

There are many SIMD-style multimedia ISA extensions, for example, MMX [21, 22], VIS [27], MDMX [18], MAX [15, 16], AltiVec [9] and SSE [26]. They mainly differ in the number and types of the newly added instructions. All of these ISA extensions operate on 64-bit registers, except AltiVec and SSE, which operate on 128-bit registers. However, as was argued in this paper, it is questionable if increasing the register size further provides any benefit, because often, the number of stream elements stored consecutively in memory is rather small.

The differences between CSI and the Matrix Oriented Multimedia (MOM) extension [6] have been discussed previously. MOM instructions can be viewed as vector versions of subword parallel instructions, i.e., they operate on matrices where each row corresponds to a packed data type.

Another related proposal is the Imagine processor [12], which has a load/store architecture for one-dimensional streams of data records. It is centered around a large, 128KB stream register file, and consists of 48 functional units grouped in 8 arithmetic clusters. Imagine is suited for applications performing many arithmetic operations on each element of a long, one-dimensional stream. It seems less suited when only a few operations on each record are performed or when the vector length is small.

CSI is a memory-to-memory architecture, i.e., there are no programmer visible registers. There have been memory-to-memory vector architectures in the past (for example, the Texas Instruments' TI ASC and CDC's Star-100 [11]), but they suffered from high startup cost which was mainly due to long memory latency. Our experiments show, however, that all benchmarks considered exhibit very high L1 hit rates, and therefore, the startup cost is less of a problem.

6 Conclusions

In this paper we have presented an architectural paradigm designed to accelerate streaming operations on mixed-width data. The described Complex Streamed Instruction (CSI) set has been evaluated using four multimedia benchmarks. On a number of important kernels, we have observed average speedups of 4.0 relative to an architecture extended with VIS instructions. These local improvements have resulted in application speedups of up to 36%.

One of the distinct features of the CSI architecture is that the number of bytes which are processed in parallel (the section size of processing width) is not determined by

the architecture but solely by the implementation. This ensures that no recompilation is needed in order to benefit from a wider datapath. The CSI architecture also eliminates overhead associated with data alignment and conversion between storage and computational format.

There are several important research issues regarding the memory subsystem. An interesting option to investigate is the direct interfacing of the CSI unit to the main memory. In recent work [25] we have shown that in many media applications, the main memory bandwidth is the critical performance factor while latency is less important. Contemporary DRAM devices such as SDRAM and RAMBUS can potentially supply data at high bandwidth. A controller is responsible for issuing requests to utilize this bandwidth potential. The design and evaluation of such a controller for streaming accesses was performed by McKee et al. [17, 31]. These studies showed that a controller with simple heuristics and modest hardware cost can supply data at high bandwidth and efficiently exploit the potential of contemporary DRAM devices. Such a stream controller fits well into the CSI paradigm because all information about the data streams is available in the stream control registers and can be easily communicated to the controller.

Acknowledgment. Thanks to Matthias Gries for providing us with his corrected version of the SimpleScalar memory model.

References

[1] R. Bhargava, L. John, B. Evans, and R. Radhakrishnan. Evaluating MMX Technology Using DSP and Multimedia Applications. In *MICRO 31*, pages 37–46, 1998.

[2] W. Buchholz. The IBM System/370 Vector Architecture. *IBM Systems Journal*, 25(1):51–62, 1986.

[3] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report 1342, Univ. of Wisconsin-Madison, Comp. Sci. Dept., 1997.

[4] W. Chen, C. Smith, and S. Fralick. A Fast Computational Algorithm for the Discrete Cosine Transformation. *IEEE Transactions on Communications*, Sept. 1977.

[5] T. Conte, K. Menezes, P. Mills, and B. Patel. Optimization of Instruction Fetch Mechanisms for High Issue Rates. In *ISCA'95*, pages 333–344, 1995.

[6] J. Corbal, M. Valero, and R. Espasa. Exploiting a New Level of DLP in Multimedia Applications. In *MICRO 32*, 1999.

[7] K. Diefendorff and P. Dubey. How Multimedia Workloads Will Change Processor Design. *IEEE Computer*, 30(9):43–45, 1997.

[8] M. Gries. The Impact of Recent DRAM Architectures on Embedded Systems Performance. In *EUROMICRO 26*, 2000.

[9] L. Gwennap. AltiVec Vectorizes PowerPC. *Microprocessor Report*, 12(6), 1998.

[10] J. Hennessy and D. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 2nd edition, 1996.

[11] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, 2nd edition, 1984.

[12] B. Khailany, W. Dally, U. Kapasi, P. Mattson, J. Namkoong, J. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: Media Processing With Streams. *IEEE Micro*, 21(2):35–47, 2001.

[13] C. Lee, M. Potkonjak, and W. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems. In *MICRO 30*, 1997.

[14] R. Lee and M. Smith. Media Processing: A New Design Target. *IEEE Micro*, 16(4):6–9, 1996.

[15] R. B. Lee. Accelerating Multimedia with Enhanced Microprocessors. *IEEE Micro*, 15(2):22–32, 1995.

[16] R. B. Lee. Subword Parallelism with MAX-2. *IEEE Micro*, 16(4):51–59, August 1996.

[17] S. McKee, W. Wulf, J. Aylor, R. Klenke, M. Salinas, S. Hong, and D. Weikle. Dynamic Access Ordering for Streamed Computations. *IEEE Micro*, 49(11):1255–1271, 2000.

[18] MIPS Extension for Digital Media with 3D. Document available via http://www.mips.com/Documentation/isa5_tech_brf.pdf.

[19] H. Nguyen and L. John. Exploiting SIMD Parallelism in DSP and Multimedia Algorithms Using the AltiVec Technology. In *ICS'99*, pages 11–20, 1999.

[20] S. Palacharla, N. Jouppi, and J. Smith. Complexity-Effective Superscalar Processors. In *ISCA'97*, 1997.

[21] A. Peleg and U. Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, 16(4):42–50, 1996.

[22] A. Peleg, S. Wilkie, and U. Weiser. Intel MMX for Multimedia PCs. *Communications of the ACM*, 40(1):24–38, 1997.

[23] C. Price. *MIPS IV Instruction Set, revision 3.1*. MIPS Technologies, Inc., Mountain View, CA, 1995.

[24] P. Ranganathan, S. Adve, and N. Jouppi. Performance of Image and Video Processing with General-Purpose Processors and Media ISA Extensions. In *ISCA 26*, pages 124–135, 1999.

[25] D. Tcheressiz, B. Juurlink, S. Vassiliadis, and H. Wijshoff. Performance of the Complex Streamed Instruction Set on Image Processing Kernels. In *Euro-Par'01*, 2001. To appear.

[26] S. Thakkar and T. Huff. The Internet Streaming SIMD Extensions. *Intel Technology Journal*, May 1999.

[27] M. Tremblay, J. M. O'Conner, V. Narayanan, and L. He. VIS Speeds New Media Processing. *IEEE Micro*, 16(4):10–20, 1996.

[28] S. Vassiliadis, E. Hakkennes, J. Wong, and G. Pechanek. The Sum-Absolute-Difference Motion Estimation Accelerator. In *EUROMICRO 24*, pages 559–566, 1998.

[29] S. Vassiliadis, B. Juurlink, and E. Hakkenes. Complex Streamed Instructions: Introduction and Initial Evaluation. In *EUROMICRO 26*, 2000.

[30] VIS Instruction Set User's Manual. Document available via http://www.sun.com/microelectronics/vis/, March 2000.

[31] L. Zhang, J. Carter, W. Hsieh, and S. McKee. Memory System Support for Image Processing. In *Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 1999.