The CSI Multimedia Architecture

Dmitry Cheresiz, Ben Juurlink, *Senior Member, IEEE*, Stamatis Vassiliadis, *Fellow, IEEE*, and Harry A. G. Wijshoff

***Abstract*—An instruction set extension designed to accelerate multimedia applications is presented and evaluated. In the proposed *complex streamed instruction* (*CSI*) set, a single instruction can process vector data streams of arbitrary length and stride and combines complex memory accesses (with implicit prefetching), program control for vector sectioning, and complex computations on multiple data in a single operation. In this way, CSI eliminates overhead instructions (such as instructions for data sectioning, alignment, reorganization, and packing/unpacking) often needed in applications utilizing MMX-like extensions and accelerates key multimedia kernels. Simulation results demonstrate that a superscalar processor extended with CSI outperforms the same processor enhanced with Sun's VIS extension by a factor of up to 7.77 on key multimedia kernels and by up to 35% on full applications.**

***Index Terms*—Computing, high performance, image-processing, video-processing.**

I. Introduction

MULTIMEDIA applications, such as audio and video compression/decompression and two-dimensional (2-D) and three-dimensional (3-D) graphics, provide new and highly valuable and appealing services to the consumer. Consequently, they form a new important workload for the general-purpose workstation and desktop processors. In order to meet the computational requirements of these applications, traditionally they have been implemented using general-purpose processors applying DSPs and/or ASICs to accelerate time-critical computations. General-purpose processors, however, are preferable to special-purpose media systems because they are easier to program, have higher performance growth, and are less costly [1]–[3]. Many microprocessor vendors have, therefore, extended their instruction set architecture (ISA) with instructions targeted at multimedia applications (e.g., [4]–[7]).

These ISA extensions exploit two characteristics exhibited by multimedia applications. First, multimedia codes typically process narrow data types (for example, 8-b pixels or 16-b audio samples). Second, data-level parallelism (DLP) is inherent in almost all multimedia applications. Accordingly, media instructions exploit SIMD parallelism at the subword level, i.e., they operate concurrently on, e.g., eight 8-b or four 16-b values packed in a 64-b register.

```
void Add_Block(unsigned char *rfp, short *bp, int iincr)
{  int i,j;
   for (i=0; i<8; i++) {
       for (j=0; j<8; j++)
           *rfp++ = Clip[*rfp + *bp++];
       rfp += iincr;
   }
}
```

Fig. 1. *C* code for saturating add.

It has been shown that these extensions can improve the performance of many multimedia kernels and applications (see, e.g., [1], [8], and [9]). Nevertheless, they have several limitations which can be summarized as follows.

- Because the size of the multimedia registers is visible at the architectural level, loops have to be strip mined at the length of these registers. This, however, implies that when the register size is increased to exploit more data parallelism, existing codes have to be modified to benefit from the wider datapath. Furthermore, increasing the register size may not be beneficial, because media kernels often operate on submatrices and the vector length in both directions is rather small.
- If the multimedia extension is implemented next to a superscalar core, a second option to exploit more parallelism is to add more multimedia functional units and to increase the issue width. However, it is generally accepted that this requires a substantial amount of hardware and may negatively affect the cycle time [10], [11].
- Another limitation is the overhead for data conversion and reorganization. Because the *storage format* (how data is stored in memory) is often too small for intermediate computations to occur without overflow, data needs to be converted (*unpacked*) to a wider *computational format*. In addition, alignment-related instructions are required if data is not stored at an aligned address and rearrangement instructions are needed if data is not stored consecutively.
- In addition, codes implemented using SIMD instructions typically incur loop overhead instructions needed for managing address and induction variables and branching.

The *C*-function depicted in Fig. 1 illustrates some of these limitations. This function adds two blocks of pixels and is taken from an MPEG decoder. The array `Clip` is used to saturate the result of the addition to the minimum/maximum value representable by an unsigned byte.

In order to bring the data in a form amenable to SIMD processing, a large number of instructions must be executed. First, alignment instructions are required because the `bp` and `rfp` pointers might not be 8-B aligned. Second, the data needs to be loaded and the data within the registers needs to be rearranged so

Manuscript received August 29, 2003; revised March 30, 2004.

D. Cheresiz was with the Computer Engineering Laboratory, Department of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology, 2628 CD Delft, The Netherlands. He is now with the Department of Information and Software Technology, Philips Research Laboratories, 5656 AA Eindhoven, The Netherlands (e-mail: dmitry.cheresiz@philips.com; cheresiz@ce.et.tudelft.nl).

B. Juurlink and S. Vassiliadis are with the Computer Engineering Laboratory, Department of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology, 2628 CD Delft, The Netherlands (e-mail: benj@ce.et.tudelft.nl; stamatis@ce.et.tudelft.nl).

H. A. G. Wijshoff is with the Leiden Institute of Advanced Computer Science (LIACS), Leiden University, 2300 RA Leiden, The Netherlands (e-mail: harryw@liacs.nl).

Digital Object Identifier 10.1109/TVLSI.2004.840415

1063-8210/$20.00 © 2005 IEEE

that the elements of `bp` and `rfp` are at corresponding positions. After that, the elements of `rfp` need to be unpacked to 16-b values. Only then the addition can be performed. Thereafter, the results need to be rearranged and packed again before they can be written back to memory. We implemented this kernel using Sun's *Visual Instruction Set* (VIS) [6]. Our implementation is available at http://ce.et.tudelft.nl/~benj/csi. It shows that in the worst case 32 instructions are needed to compute 8 pixels of the result. So, assuming perfect instruction and data caches and no dependencies, a four-way VIS-enhanced superscalar processor requires at least $8 \times 32/4 = 64$ cycles. If the `bp` and `rfp` pointers are 8-B aligned, 15 instructions are needed.

In this paper, we propose an ISA extension called complex streamed instructions (CSI) that addresses the limitations described above. CSI instructions process 2-D data streams stored in memory. There is no architectural (programmer-visible) constraint on the length of the streams, since the hardware is responsible for *sectioning*, i.e., for dividing the streams into sections which are processed in a SIMD manner. A single CSI instruction performs address generation and alignment, data loading and reorganization, packing and unpacking, as well as the operation that needs to be performed. In addition, CSI provides some special-purpose instructions that provide performance benefits on key multimedia kernels.

This paper is organized as follows. Related work is discussed in Section II. The CSI architecture is described in Section III and a possible implementation is given in Section IV. The CSI extension is experimentally validated and compared to VIS and SSE in Section V. Conclusions are drawn in Section VI.

II. Related Work

There are many media processing approaches, varying from general-purpose processors (GPPs) extended with SIMD media instructions to dedicated hardware implementations. Since CSI belongs to the former class, we restrict ourselves to a discussion of general-purpose processors enhanced with media instructions and some programmable media processors based on vector architectures.

As mentioned before, many GPPs have been extended with SIMD instructions, e.g., MMX [4], SSE [5], VIS [6], MVI [12], and AltiVec [7]. SIMD media instructions pack multiple, small-data elements into a wide (typically 64- or 128-b) register and process all elements (or *subwords*) in parallel. Fig. 2 illustrates a SIMD operation that adds two vector registers, each of which contains four 16-b values. The main differences between the various SIMD extensions are the location of the media or vector registers, the size of these registers (which determines the number of subwords that can be processed simultaneously), and the number of instructions supported. MMX and VIS, for example, provide 64-b wide SIMD operations and the media registers correspond to the floating-point registers. SSE and AltiVec, on the other hand, provide a separate file of 128-b wide media registers. The number of SIMD instructions supported varies significantly, from 13 in MVI to 121 in VIS.

Slingerland and Smith [13] study the performance of various SIMD instruction sets on several multimedia kernels. Interestingly (and independently), they also found two factors

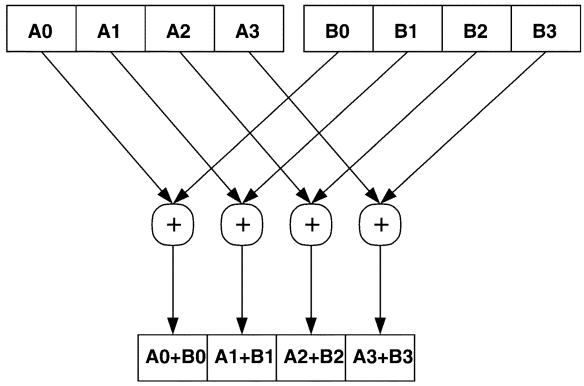

Fig. 2. Packed addition of two 64-b registers containing four 16-b values.

that limit the performance of current media ISA extensions: storage formats which are insufficient for computation (necessitating conversion overhead), and nonunit strides. They, therefore, propose a SIMD architecture that implicitly unpacks while loading, implicitly packs while storing, and provides strided load and store instructions. They do not evaluate the performance of the proposed architecture, however, and their proposal requires overhead for managing address and loop control variables. Moreover, in their proposal the vector length is architecturally visible.

The matrix oriented multimedia (MOM) extension [14] contains instructions that can be viewed as vector versions of SIMD instruction, i.e., they operate on matrices and each matrix row corresponds to a packed data type. MOM allows an arbitrary stride between consecutive rows but requires a unit stride between consecutive row elements and also requires explicit (un)packing if the storage format is inappropriate for computation. Furthermore, MOM does not provide floating-point SIMD instructions and has only limited support for conditional execution.

The Imagine processor [15] has a load/store architecture for one-dimensional (1-D) streams of data records. It is suited for applications performing many operations on each element of a long, 1-D stream, but appears to be less suited when only a few operations on each record are performed or when the vector length is small.

The Vector IRAM (VIRAM) [16] is a register-to-register vector architecture supporting narrow data types. The elements in a vector register can be 16-, 32-, or 64-b wide. As in CSI, a control register specifies the element size (called the Virtual Processor Width or VPW in [16]). When the VPW is larger than the size of data in memory, vector load and store instructions imply a conversion between storage and computational format. For example, when the VPW is 16 b, the vector-load-byte instruction implicitly converts to 16-b values. VIRAM also supports strided and indexed addressing modes. These techniques reduce the overhead needed for managing address variables, loop control, and (un)packing. However, VIRAM seems less suited for algorithms that process 2-D submatrices.

CSI was originally presented in [17] and evaluated in [18]. Thereafter, Talla and John [19] also observed that the performance of SIMD-enhanced processors is limited by the overhead required for bringing data in a form suited for SIMD processing.

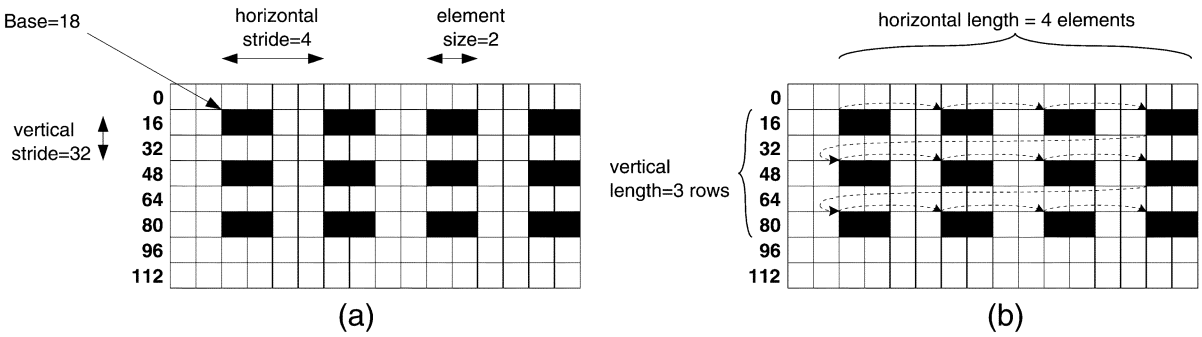

Fig. 3. Format of an arithmetic stream. Each cell represents a byte. Dark cells are stream data.

They proposed the MediaBreeze architecture, which contains instructions that support five levels of looping. CSI instructions support at least two levels and special-purpose CSI instructions can support more (that could exceed five levels of looping).

Another proposal more recent than CSI is the reconfigurable streaming vector processor (RSVP) [20]. The authors also observed that in SIMD-enhanced processors, the amount of parallelism is determined by the width of the programmer-visible media registers. In CSI, as well as the RSVP, the amount of parallelism is limited only by resource limitations and algorithm/data structure characteristics, which allows the same program to take full advantage of a wide range of implementations. The RSVP has a similar setup (i.e., host interaction) but requires additional programming based on data flow graphs to perform the CSI-like functions.

Finally, we remark that there have been several general-purpose vector architectures in the past, e.g., [21] and [22]. For such architectures, a single vector instruction could process only a limited number of elements, called the *section size* [21]. Processing a longer vector required several loop iterations and sectioning instructions. CSI differs from the mentioned approaches because it does not need explicit sectioning and, therefore eliminates associated overhead instructions required for loop control, address generation, and memory access. Additionally, CSI allows deterministic prefetching for general vector accesses. Finally, when operating on vectors with nonunit strides, CSI does not need instructions to rearrange elements as, for example, CDC Cyber 200 (Model 205) requires [21].

III. The CSI Architecture

In this section, we describe the CSI architecture, i.e., the structure and functionality of the processor visible at the (assembly) programming level.

A. CSI Streams

CSI is a memory-to-memory architecture for 2-D streams of arbitrary length. CSI streams are divided into two categories: *arithmetic* and *bit streams*. Elements of an arithmetic stream are 8-, 16-, or 32-b wide and represent fixed-point or floating-point data. Streams are located in memory following the (2-D) strided access pattern depicted in Fig. 3(a). We allow for an arbitrary stride between consecutive row elements as well as between consecutive rows. Commonly, consecutive row elements are stored sequentially, but nonunit strides can be found, e.g., in the color conversion phases of JPEG. As depicted in Fig. 3(b), stream elements are addressed in row-major order. This means, for example, that the sixth element is located in the second column of the second row.

Elements of a CSI bit stream are 1-b wide and should be stored contiguously in memory. Such streams are used for masked/conditional operations, and therefore, are also referred as *mask streams*. Bit streams need to be byte-aligned and the first element corresponds to the most significant bit of the byte located at the base address.

B. CSI Architecture State

Fig. 4 depicts all programmer-visible CSI registers. Rather than encoding all parameters that specify a stream in the instruction (which would necessitate very long instructions), each arithmetic stream is specified by a *set of stream control registers* (SCR-set). There are 16 of such sets, each of which consists of the following eight 32-b registers.

1) **Base**. This register contains the starting or base address of the stream. The only alignment restriction is that the address must be a multiple of the element size (in bytes). The base address of the stream depicted in Fig. 3 is 18.
2) **HStride**. The *horizontal stride*, i.e., the stride in bytes between consecutive stream elements in a row. In the example, $\mathbf{HStride} = 4$.
3) **HLength**. This register holds the number of stream elements in a row or the *horizontal length*. $\mathbf{HLength} = 4$ in the example.
4) **VStride**. The *vertical stride*, i.e., the distance in bytes between consecutive rows. $\mathbf{VStride} = 32$ in the example.
5) **VLength**. This register contains the *vertical length* or, equivalently, the number of rows. $\mathbf{VLength} = 3$ for the example stream.
6) **Format**. This register consists of various fields which mainly specify the storage and computational formats and how conversion between these formats is performed. For the example stream it specifies that each element consists of 2 B, whether the elements are signed or unsigned, and what the computational format is.
7) **CurrRow**. The number of the row to which the current element belongs. It is used for interrupt handling.
8) **CurrCol**. The position of the current element within its row. Also used for interrupt-handling. For example, if six

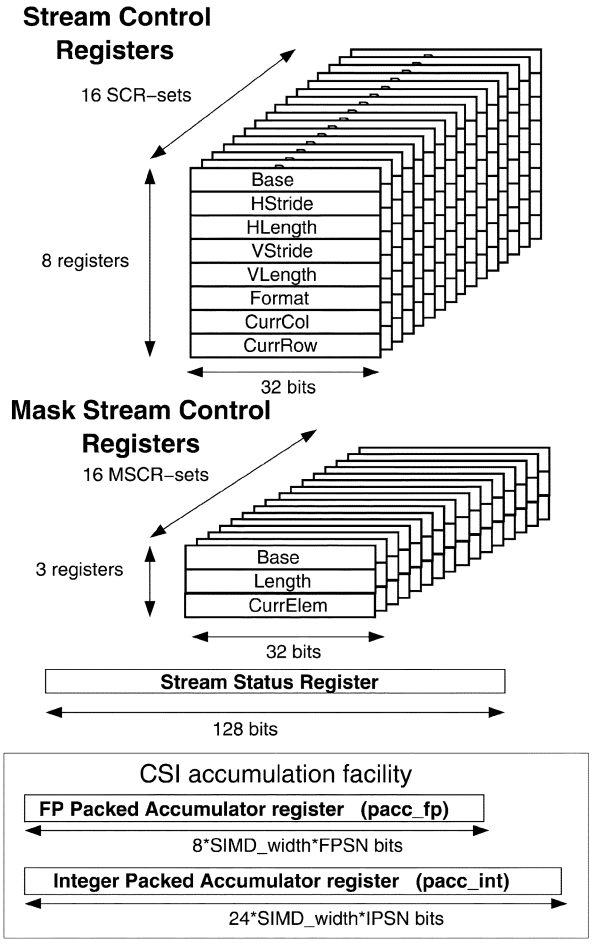

Fig. 4. CSI register space.

elements of the stream depicted in Fig. 3 have been completely processed, $\mathbf{CurrRow} = 1$ and $\mathbf{CurrCol} = 2$ (rows and columns are numbered from zero).

As indicated in this list, each stream control register in an SCR-set has a number between 0 and 7 by which they are addressed by the instructions that move data to/from them. For example, the instruction `csi_mtscr SCRS1, 0, r4` copies the address contained in general-purpose register `r4` to the **Base** register of SCR-set `SCRS1`.

Similarly, a CSI mask stream is specified by a *set of mask stream control registers* (MSCR-set). There are 16 such sets, each consisting of three 32-b registers: **Base**, **Length**, and **CurrElem**, which contain the base address, the number of elements, and the number of the element that is currently being processed, respectively.

The 128-b stream status register (SSR) contains control information. If the *mask bit* (bit 0) is set masked versions of instructions are executed. The *sequential mode bit* (bit 1) controls if stream elements are processed one by one or in parallel. This is useful during debugging since it allows to identify the element that caused an exception. The other three fields in this register are used for interrupt handling. They identify the stream that caused the exception, designate the type of exception, and contain a copy of the instruction that caused the exception. Some bits in the SSR are currently unused. They are provided for possible future extensions.

The two accumulation registers $\mathbf{pacc\_fp}$ and $\mathbf{pacc\_int}$ are used by accumulation-related CSI instructions. They are described in Section III-C2.

C. CSI Instruction Set

Table I summarizes the CSI instruction set. We use, e.g., the notation $AS \times R \rightarrow AS$ to denote that an instruction takes an arithmetic stream and a scalar register as input and produces an arithmetic stream. For reasons of space, not all 47 CSI instructions are included in this table. Detailed descriptions of all instructions are available in [23].

The CSI instruction set is divided into the following categories: basic arithmetic and logical instructions, accumulation instructions, special-purpose instructions, stream reorganization instructions, and auxiliary instructions. In this section, these instruction categories are described.

1) Basic Arithmetic and Logical Instructions: These instructions perform pairwise addition, multiplication, bitwise AND, and other elementary operations. Examples of such instructions are `csi_add SCRS1, SCRS2, SCRS3`, which adds corresponding elements of the streams described by `SCRS2` and `SCRS3` and writes the results to the stream designated by `SCRS1`, and `csi_mul_reg SCRS1, SCRS2, r1`, which multiplies the stream specified by `SCRS2` with the scalar value contained in the integer register `r1`.

2) Accumulation Instructions: Accumulations are very sensitive to latency because every accumulation needs the previous value as input. Our solution for this problem is common in traditional vector architectures (e.g., [21]) and a similar solution has been proposed for MOM [24]. Let SIMD_width denote the number of bytes the CSI execution unit processes in parallel. Furthermore, let $n = \text{SIMD\_width}/4$ be the number of single-precision FP values that can be processed in parallel and let the *floating-point partial sum number* (*FPSN*) be the ratio of the latency of an FP addition L_{fadd32} to T_{fadd32}, the reciprocal of the throughput. The accumulator register $\mathbf{pacc\_fp}$ consists of $n \cdot \text{FPSN}$ 32-b elements $(\mathbf{pacc\_fp}_0, \ldots, \mathbf{pacc\_fp}_{n \cdot \text{FPSN}-1})$. Accumulation is performed in two stages. In the first, $n \cdot \text{FPSN}$ partial sums are produced as follows: the first n elements are added in parallel to $(\mathbf{pacc\_fp}_0, \ldots, \mathbf{pacc\_fp}_{n-1})$, T_{fadd32} cycles later the next n elements are added to $(\mathbf{pacc\_fp}_n, \ldots, \mathbf{pacc\_fp}_{2n-1})$, and so on. By the time the last n elements of the $\mathbf{pacc\_fp}$ register have been reached, the first n elements are available again and so the computation "wraps-around." In this way, the pipelines are fully utilized. This stage is carried out by the instruction `csi_acc`. When all elements have been processed, the `csi_acc_psum` instruction accumulates the partial sums and places the result in a general-purpose register.

Integer accumulations are performed similarly but employ the integer packed accumulator register $\mathbf{pacc\_int}$. This register is $3 \cdot \text{SIMD\_width} \cdot \text{IPSN}$ bytes wide, where IPSN is the *integer partial sum number*. Employing such a wide accumulator avoids having to promote the operands to a wider format. Similar accumulators are employed in DSP architectures as well as MDMX [25].

TABLE I
OVERVIEW OF THE CSI INSTRUCTION SET

Operation	Mnemonic	Masked	Data Types	Operands
Arithmetic and Logical				
add	csi_add[_reg]	yes	*all arith.*	$AS \times AS \rightarrow AS$, $AS \times R \rightarrow AS$
subtract	csi_sub[_reg]	yes	*all arith.*	$AS \times AS \rightarrow AS$, $AS \times R \rightarrow AS$
multiply	csi_mul[_reg]	yes	*all arith.*	$AS \times AS \rightarrow AS$, $AS \times R \rightarrow AS$
multiply-add	csi_mul_add[_reg]	yes	*all arith.*	$AS \times AS \times AS \rightarrow AS$ $AS \times R \times AS \rightarrow AS$
compare	csi_cmp[_reg]	no	*all arith.*	$AS \times AS \rightarrow BS$, $AS \times R \rightarrow BS$
maximum	csi_max	yes	*all arith.*	$AS \rightarrow R$
count ones	csi_cnt_ones	no	*b1*	$BS \rightarrow R$
bitwise AND	csi_and[_reg]	yes	*u8, u16,u32*	$AS \times AS \rightarrow AS$, $AS \times R \rightarrow AS$
	csi_and_bitstr	no	*b1*	$BS \times BS \rightarrow BS$
bitwise OR	csi_or[_reg]	yes	*u8, u16,u32*	$AS \times AS \rightarrow AS$, $AS \times R \rightarrow AS$
	csi_or_bitstr	no	*b1*	$BS \times BS \rightarrow BS$
Accumulation				
accumulate	csi_acc	yes	*all arith.*	$AS \rightarrow AccR$
acc. partial sums	csi_acc_psum	no		$AccR \rightarrow R$
accumulate section	csi_acc_section	yes	*all arith.*	$AS \times R \rightarrow AS$
Special-purpose				
SAD	csi_sad	no	*s8, u8, s16, u16*	$AS \rightarrow R$
Paeth predict	csi_paeth	no	*s8, u8*	$AS \times AS \times AS \rightarrow AS$
IDCT	csi_idct	no	*s16*	$AS \rightarrow AS$
Stream reorganization				
Extract stream	csi_extract	no	*all arith.*	$AS \times BS \rightarrow AS$
Insert stream	csi_insert	no	*all arith.*	$AS \times BS \rightarrow AS$
Auxiliary				
Move to SCR	csi_mtscr	no	*all arith.*	$R \rightarrow SCR$
Move to SCR immediate	csi_mtscri	no	*all arith.*	*imm*16 $\rightarrow$ *SCR*

Legend:

Data Types		Operands	
u8, s8	(un)signed 8-bit integer	AS	CSI Arithmetic stream
u16, s16	(un)signed 16-bit integer	BS	CSI Bit stream
u32, s32	(un)signed 32-bit integer	R	Scalar (integer or floating-point) register
f32	32-bit floating-point	*AccR*	CSI accumulation register
b1	single bit	*SCR*	CSI stream control register
all arith.	all types except *b1*	*imm*16	16-bit immediate
		Masked	instruction under mask-mode control

In several media applications, every n consecutive elements of a long stream of N elements need to be accumulated producing an output stream of N/n elements. n is usually small and since each CSI instruction incurs a certain startup cost, using separate `csi_acc` to accumulate every n consecutive elements would be inefficient. CSI, therefore, provides an "accumulate section" instruction `csi_acc_section SCRSi, SCRSj, rk` that (implicitly) divides the input stream specified by `SCRSj` into sections of `rk` consecutive elements, accumulates the elements within each section, and stores the obtained sums to the stream specified by `SCRSi`. This operation was found useful for, e.g., the modeling and projection stages of the 3-D geometry pipeline, where a small (3×3 or 4×4) matrix is multiplied with a long sequence of small (3- or 4-element) vectors.

3) Special-Purpose Instructions: We found that many kernels can be implemented using one or a few elementary CSI instructions. There are also kernels, however, that perform more complex operations and so need to be synthesized using multiple basic CSI instructions. It has been shown that many of these complex operations can be implemented in an area comparable to that of one or a few ALUs and do not require more cycles than basic operations (see, e.g., [26] and [27]). The huge performance benefits provided by such special-purpose instructions warrant their implementation.

We provide three examples of such CSI instructions. The `csi_sad` instruction computes the sum of absolute differences of two streams. It is used to implement the most time-consuming routine in MPEG-2 encoding, motion estimation. The`csi_paeth` instruction performs Paeth prediction, encoding, and decoding that is used in the PNG standard [28]. Finally, the `csi_dct` and `csi_idct` perform 1-D (inverse) discrete cosine transform on every 8 consecutive elements of a stream and are used in various image and video codecs.

4) Conditional Execution and Stream Reorganization Instructions: Several multimedia kernels, in particular 3-D graphics kernels, contain loops with if-then or if-then-else statements in the loop body. Without proper architectural support, these constructs prohibit the use of vector instructions. A solution commonly employed by conventional vector processors is *masking*. First, a *mask vector* consisting of single bit elements is produced. Thereafter, a *masked* instruction is executed that only writes its result if the corresponding bit in the mask vector is set. CSI employs the same technique, with mask streams used to control conditional execution. To save opcode space, the *mask bit* of the SSR determines if the masked or nonmasked version of an instruction should be executed. For example, if the mask bit is set, the instruction `csi_mul SCRSi, SCRSj, SCRSk, MSCRSl` is executed under control of the mask stream `MSCRSl`. If it is not set, the mask stream is ignored and the instruction is executed unconditionally. Mask streams are usually generated by the `csi_cmp` instruction.

The disadvantage of masked execution is that when many masks are 0, the corresponding operations turn into *no-ops*, i.e., they are performed but their results are discarded thereby reducing the efficiency. A solution to this problem is to split a data stream into shorter ones depending on the mask value

[15]. For this purpose CSI provides stream reorganization instructions. For example, `csi_extract` extracts elements that correspond to the nonzero bits in the mask stream. The obtained stream can then be processed without no-ops, and the results can be inserted back to the appropriate positions by means of the `csi_insert` instruction.

5) Auxiliary Instructions: These instructions are used to move data to or from individual stream control registers, accumulation registers, and the SSR. For example, the instruction `csi_mtscr SCRSi, j, rk` (*move to stream control register*) copies the content of general-purpose register `rk` to SCR `j` of SCR-set `SCRSi`. The instruction `csi_mtscri SCRSi, j, imm16` is similar but moves a 16-b immediate to the SCR. These instructions are commonly used to initialize the SCRs.

D. Example

The CSI code for the `Add_Block` kernel depicted in Fig. 1 is available at http://ce.et.tudelft.nl/~benj/csi. It shows that 12 setup instructions need to be executed to initialize the stream control registers. After that, the nested loop is substituted by a single `csi_add` instruction. We estimate the performance of this kernel, assuming that the base processor is a four-way superscalar, perfect instruction and data caches, and that the datapath of the CSI execution unit is 128 b wide (the same assumptions as in Section I). Since the setup instructions are independent and processed by the superscalar core, it takes $12/4$ cycles to execute them. Furthermore, the pipelined datapath presented in the next section shows that there are eight stages, each of which is assumed to take 1 cycle. It, therefore, takes 8 cycles before the first 8-B result is produced, and after that, 8 B of the result are produced every cycle. So, in total 18 cycles are required. Compared to the 64 cycles needed by a four-way VIS-enhanced superscalar processor, this corresponds to a speedup by a factor of 3.56. If all pointers would be aligned, the speedup would still be 1.67.

IV. Implementation

In this section, we sketch a possible datapath for the CSI execution unit and discuss some other implementation issues.

A. CSI Datapath

A CSI instruction such as `csi_add` not only performs pairwise addition but also loading and storing, packing and unpacking, etc. Since these operations are independent, they can be pipelined. The CSI execution unit is, therefore, organized as a pipeline consisting of eight stages, as depicted in Fig. 5. For clarity, some parts have been omitted.

In the first stage, the source streams address generators AG1 and AG2 generate addresses aligned at cache-block-boundaries. In addition, they generate a *position mask* that indicates the bytes in the cache block that contain stream data. The aligned addresses are appended to the load queue.

In the second stage, the addresses at the front of the load queue are used to fetch blocks containing stream data from the L1 data cache. We decided to interface the CSI execution unit to the L1 cache rather than the L2 cache or main memory for the following reasons: first, Ranganathan *et al.* [1] as well as

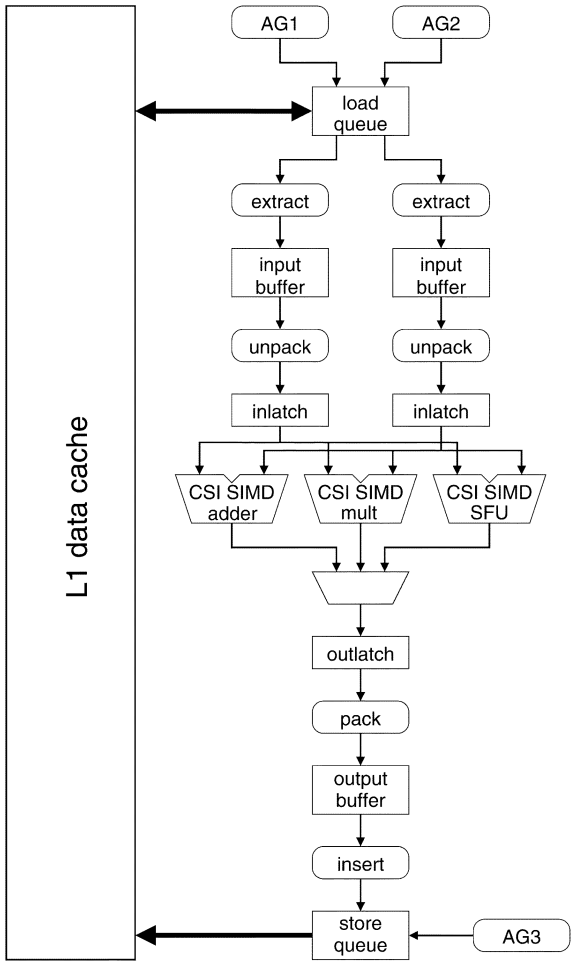

Fig. 5. Datapath of the CSI execution unit.

Slingerland and Smith [29] observed that multimedia applications exhibit high L1 data cache hit rates; second, since the L1 cache is on-chip, it is not expensive to implement a wide path between the cache and the CSI execution unit, so that a whole cache block can be transferred in a single access; and third, this organization keeps the cache coherent with memory. Since a two-ported cache is assumed, the cache ports need to be shared between the load and store queues. We remark that data is implicitly prefetched in two ways. First, data is loaded before it is needed because the load queue fetches entire cache blocks but the CSI processing units process only part of them. Second, since the load queue has eight entries and attempts to access the nonblocking L1 data cache each cycle, the memory latency can be hidden.

In the third stage, the bytes that contain stream data are extracted based on the masks produced by the address generators and placed consecutively in one of the input buffers. The extract unit is similar to a *collapsing buffer* [30].

In the fourth stage, provided the input buffers contain sufficient data, the data is unpacked from storage to computational format. Signed values are sign-extended and unsigned values are padded with zeroes. These operations are controlled by the **Format** register of the corresponding SCR-sets.

In the fifth stage, the CSI SIMD processing units perform packed operations on the data contained in the input latches.

The inputs of these units are $\mathrm{SIMD\_width}$ bytes wide, so they process either $\mathrm{SIMD\_width}$ bytes, $\mathrm{SIMD\_width}/2$ 16-b values, or $\mathrm{SIMD\_width}/4$ 32-b values in parallel. The outputs are twice as wide as the inputs, so no overflow occurs during computation. In Fig. 5, three SIMD units are shown: 1) a SIMD adder that performs packed additions, subtractions, and logical operations; 2) a SIMD multiplier that performs packed multiply and divide operations; and 3) a special functional unit (SFU) that implements the special-purpose instructions.

In the sixth stage, the data contained in the output latch is converted from computational to storage format, controlled by the **Format** register of the destination stream.

In the seventh stage, the insert unit performs the inverse operation of the extract unit, i.e., it "scatters" the stream elements so that they are placed in their correct positions.

Finally, in the eighth stage, the store queue writes a cache block of data to the L1 data cache. It performs a partial store operation, similar to the VIS partial store instruction. The address generator AG3 has already generated the cache-block-aligned address for the destination stream.

It is assumed that each stage except the fifth stage takes one cycle. The latency of each SIMD unit is taken to be equal to the latency of the corresponding VIS or scalar unit. The extract (insert) unit may require more than one cycle, but since the unpack (pack) unit is very simple, it is reasonable to assume that they take 2 cycles collectively. We finally remark that in [31] we presented a detailed description of the CSI execution unit with a three-stage pipelined implementation of the CSI address generators. We have performed experiments assuming a 3-cycle latency instead of single-cycle one and found that this increases the execution time by at most 5%.

B. In-Order Execution

It is important to realize that CSI instructions that access memory are executed in-order, even though the base processor is superscalar. For such instructions it needs to be ensured that there is no memory dependency with other CSI instructions or scalar load/store instructions. Furthermore, we do not speculatively execute memory-to-memory CSI instructions. Instructions that set control registers, however, are executed by the superscalar core. This helps to keep the startup cost low. We, therefore, took the following conservative approach. When a CSI instruction that accesses memory is detected in the instruction stream, the pipeline is stalled until all instructions prior to the CSI instruction are completed. After that, the instruction is issued for execution. Instruction fetching resumes when the CSI instruction is completed.

C. Interrupts and Context Switching

All CSI instructions can be interrupted during execution. For multimedia applications, however, one does not always want to detect arithmetic exceptions, because small differences in accuracy are acceptable as long as they are visually imperceptible. CSI, therefore, allows interrupts to be disabled and when they are, a bit in the **Format** control register signifies if wrap-around or saturation arithmetic should be performed.

If arithmetic exceptions are enabled or if another type of exception occurs, they are handled as follows. The execution of a CSI instruction is represented as a sequence of *units of operation* (*UOP*s). In each UOP, a fixed number of consecutive stream elements are processed. This number is determined by the width of the CSI SIMD units and the size of the elements during computation. If a UOP has been performed successfully (i.e., the elements have been loaded, the SIMD operation has been performed, and the results have been stored without exceptions), the **Base**, **CurrCol**, and **CurrRow** control registers are advanced to the address of the first element to be processed by the next UOP, and the row and column position of this element, respectively. If an exception occurs during the current UOP, the control registers are not updated. This allows to restart the instruction from the current UOP. The **CurrCol** and **CurrRow** are reset when the instruction is completed, so they do not have to be initialized explicitly.

On a context switch, all SCR-sets need in principle to be saved and restored. In order to reduce the cost of context switching, valid and dirty bits can be associated with each SCR-set [21]. The valid bits indicate the SCR-sets that are live, i.e., that will be used again. They must be cleared by the compiler or programmer every time an SCR-set is released. The dirty bits indicate which SCR-sets have changed since the last context switch. Since, usually, only a few SCR-sets are active, this reduces the amount of data that needs to be saved and restored on a context switch.

V. Experimental Validation

In order to validate the proposed ISA extension, we compare the performance achieved by a superscalar processor extended with CSI to the performance attained by a VIS-enhanced processor using integer media benchmarks. Because VIS does not support floating-point SIMD instructions, we use Intel's SSE extension for the 3-D graphics benchmark.

A. Experimental Setup

1) Benchmarks and Simulation Tool: We attempted to cover a wide spectrum of media processing workloads: image compression and decompression (JPEG), 2-D image processing (the *add8*, *blend8*, *scale8*, and *convolve*3×3 kernels from Sun's VIS Software Developer Kit (VSDK) [32]), video compression (MPEG-2), and 3-D Graphics (SPEC's *Viewperf*). The JPEG and MPEG-2 codecs were taken from MediaBench [33].

We developed near cycle-accurate simulators of VIS-, SSE-, and CSI-enhanced superscalar processors by extending the `sim-outorder` simulator of SimpleScalar (release 3.0) [34]. A corrected version of SimpleScalar's memory model was used based on the SDRAM specifications given in [35].

2) Methodology: The most time-consuming routines were identified using the `sim-profile` tool. Subsequently, the kernels that contain a substantial amount of data-level parallelism and whose key computations can be replaced by VIS, SSE, or CSI instructions were coded in assembly. The identified kernels are *Add_Block* (MPEG2 frame reconstruction), *Saturate* (saturation of 16-b elements to 12-b range in MPEG decoder), *dist1* (sum of absolute differences for motion estimation in MPEG), *ycc_rgb_convert* and *rgb_ycc_convert* (color conversion in JPEG), *h2v2_downsample*

TABLE II
PROCESSOR PARAMETERS

Clock frequency	666 MHz
Issue width	4/8/16
Instruction window size	64/128/256
Load-store queue size	8-128
Branch Predictor	
Bimodal predictor size	2K
Branch target buffer size	2K
Return-address stack size	8

FU type	*Number*	*Latency/recovery (cycles)*
Integer ALU	4/8/16	1/1
Integer MULT	1/2/4	
multiply		3/1
divide		20/19
Floating-point ALU	4/8/16	2/2
Floating-point MULT	1/2/4	
FP multiply		3/1
FP divide		20/19
sqrt		24/24
VIS adder	2/4/8	1/1
VIS multiplier	2/4/8	
multiply and pdist		3/1
other		1/1
SSE unit	1/2/4	as corr. scalar/VIS FU
CSI execution unit	1	
datapath width	16/32/64B	
CSI SIMD units		as corr. scalar/VIS FU
CSI 1D IDCT SFU		10/1

and *h2v2_upsample* (2:1 down- and upsampling of a color component in JPEG), *Fast_idct* and *jpeg_idct_islow* (inverse discrete cosine transform in MPEG2 and JPEG), and *xform_points_4fv* and *gl_color_shade_vertexes_fast* (transform and lighting stages of the 3-D geometry stage in *Viewperf*). We had to code the kernels ourselves because, to our knowledge, there is no publicly available compiler that generates VIS or SSE code.[1] However, we based our implementations on vendor supplied codes [32], [37], [38] when possible. We remark that in our experience, coding kernels in assembly using CSI instructions is easier and less error-prone than using VIS or SSE instructions, because the programmer does not have to explicitly administer data promotion and demotion, address alignment, or data reorganization, etc.

3) Modeled Processors: The base processor is four-way superscalar but larger issue widths are also considered. The instruction window size [i.e., the number of entries in the register update unit (RUU)] was fixed at 64 because larger sizes provided no performance benefit. Table II summarizes the basic processor parameters and lists the number of FUs of each type and the instruction latencies.

The VIS-enhanced processor has two VIS adders that perform partitioned add, subtract, merge, expand, and logical operations, and two VIS multipliers that perform the partitioned multiplication, compare, pack, and pixel distance (SAD) operation. VIS instructions operate on the floating-point register file and have a latency of 1 cycle, except for the pixel distance and packed multiply instructions which have a latency of 3 cycles. This is modeled after the UltraSPARC [6] with two exceptions. First, in the UltraSPARC the `alignaddr` instruction cannot be executed in parallel with other instructions. This limitation is not present in the processor we simulated. Second, the UltraSPARC has only one 64-b VIS multiplier. We assumed two in order to perform a fair comparison between VIS- and CSI-enhanced processors, since the width of the CSI execution unit is 128 b. Any speedup of CSI over VIS should, therefore, not be attributed to different degrees of parallelism. We remark that a superscalar processor with two VIS adders and two VIS multipliers is, in fact, capable of processing 256 b in parallel, but only when packed additions and multiplications are perfectly balanced.

SSE instructions operate on a separate register file consisting of 128-b registers. The basic SSE-enhanced processor has one SSE unit that performs all packed floating-point operations. The latencies of SSE instructions are taken to be equal to those of the corresponding scalar instructions.

The datapath of the CSI execution unit is 128-b wide (16×8, 8×16, or 4×32 b). The latency of a CSI instruction is nondeterministic, since it depends on the stream length and the location of data in the memory hierarchy. However, the latencies of the CSI SIMD units are assumed to be equal to the latencies of their scalar or VIS counterparts. Since there are two multiplications and four additions/subtractions on the critical path of the implemented 1-D IDCT algorithm [39], the latency of the SFU that performs the 8-point 1-D IDCT is assumed to be $2 \times 3 + 4 \times 1 = 10$ cycles.

The processors are equipped with a 64 KB, four-way set-associative L1 data cache with a line size of 64 B, and a 256 KB, two-way set-associative L2 data cache with a line size of 128 B. Both caches employ LRU replacement. The L1 hit time is 1cycles and the L2 hit time is 6 cycles. Because the benchmarks have small instruction working sets, a perfect instruction cache is assumed. Furthermore, the number of cache ports is fixed at two. Since CSI instructions access up to four data streams, this means that the cache ports need to be shared. The main memory is implemented using SDRAM with a row access, row activate, and precharge time of 2-bus cycles. The 64-b wide memory bus has a frequency of 166 MHz and the ratio of CPU frequency to memory bus frequency was set to four, resulting in a CPU frequency of 666 MHz.

B. Performance of Image and Video Benchmarks

Fig. 6 depicts the speedups achieved by the base four-way, CSI-enhanced superscalar over the same processor extended with VIS for VSDK and JPEG/MPEG kernels. The behavior of *rgb_ycc_convert* is similar to *ycc_rgb_convert* (labeled *ycc_rgb*) and *h2v2_upsample* is comparable to *h2v2_downsample* (labeled *h2v2*) and have, therefore, been omitted.

It can be seen that the processor extended with CSI clearly outperforms the VIS-enhanced processor. The speedup varies from 0.97 to 7.77. There are two cases where CSI is not much more effective than VIS. The first is IDCT. This kernel can be implemented using the special-purpose `csi_idct` instruction. Since VIS does not provide an IDCT instruction, we did not use the `csi_idct` instruction in order to make a fair comparison. Instead, the CSI version is based on the standard definition of the IDCT as two matrix multiplications. The VIS version of this kernel, on the other hand, is based on a highly optimized IDCT algorithm [39]. Therefore, the CSI version of *idct* executes much

[1]A compiler that translates loops to code that uses the SIMD extensions to the Intel architecture has recently been described in [36].

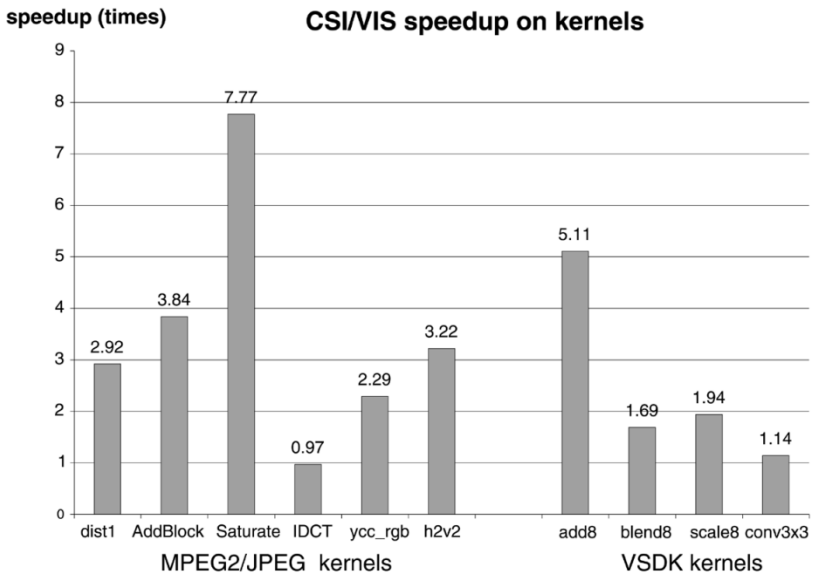

Fig. 6. Speedup of CSI over VIS for several kernels.

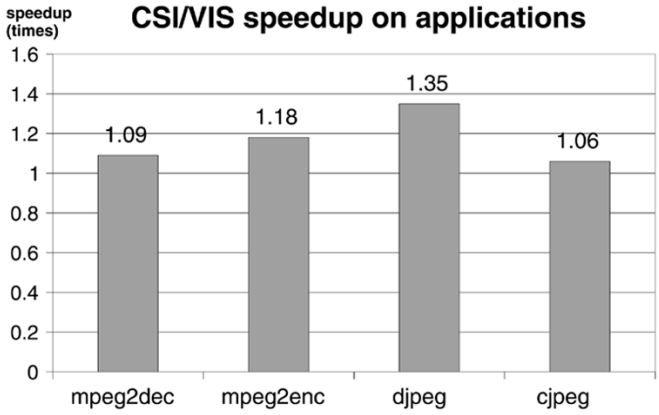

Fig. 7. Speedup of CSI over VIS for JPEG/MPEG codecs.

more operations than the VIS version. Nevertheless, its performance is comparable to that of the VIS implementation. If the `csi_idct` instruction is employed, CSI is faster than VIS by a factor of 2.48. The second kernel for which CSI is not much more efficient is *conv*3×3. The reason is that this kernel contains a balanced mix of packed additions and multiplications. As explained in Section V-A3, this means that the VIS-enhanced processor can process 32 B in parallel, whereas the CSI-enhanced CPU processes only 16 B, simultaneously. The largest speedup is obtained for the *Saturate* kernel. In this kernel, 16-b signed values are clipped to the range $[-2048, 2047]$ (the range of 12-b values). VIS can only clip to the range of 8- or 16-b values and, therefore, the required operation needs to be synthesized. CSI, on the other hand, allows saturation to an arbitrary range during the packing stage.

Fig. 7 shows how the kernel-level speedups translate to application-level speedups. Of course, due to Amdahl's law, the speedups for complete applications are smaller than for kernels. Nevertheless, CSI provides a performance improvement of up to 35%. The smallest performance improvement is obtained for the JPEG encoder *cjpeg*. This is because the kernels *rgb_ycc* and *h2v2* together consume only about 10% of the execution time on the VIS-enhanced processor. The largest speedup is achieved for *djpeg*. The reason is that the *ycc_rgb* and *jpeg_idct_islow* kernels account for a large part of the total execution time. The second kernel not only performs IDCT but also dequantization and clipping. While the CSI version of IDCT is 3% slower than the VIS version, dequantization and clipping are significantly faster.

C. Scalability

To exploit more parallelism in a VIS-enhanced superscalar, the issue width and the number of SIMD units have to be increased. For CSI, on the other hand, exploiting more parallelism does not involve issuing and executing more instructions but increasing the datapath width of the CSI execution unit. In this section the scalability of both approaches is investigated.

The amount of parallel execution resources is characterized by the number of bytes that can be processed in parallel. As before, we refer to this number as the *SIMD width*. For VIS, it is determined by the number of VIS adders and multipliers. For example, a processor with two VIS adders and two VIS multipliers can process 16 B in parallel (32 B with the right operation mix). For CSI, the SIMD width is determined by the width of the CSI datapath.

To determine the scalability of a VIS-enhanced superscalar processor, we consider issue widths of 4, 8, and 16, and scale the number of functional units of each type accordingly. The number of RUU entries is 64, 128, and 256, respectively, because larger RUUs provided hardly any benefit [40]. Let a VIS-enhanced superscalar processor with an issue width of x, a window size of y, and a SIMD width of z B be denoted by $\mathrm{VIS}(x, y, z)$. A similar notation is used for CSI-enhanced CPUs. Fig. 8(a) depicts the speedup of $\mathrm{VIS}(x, y, z)$ relative to

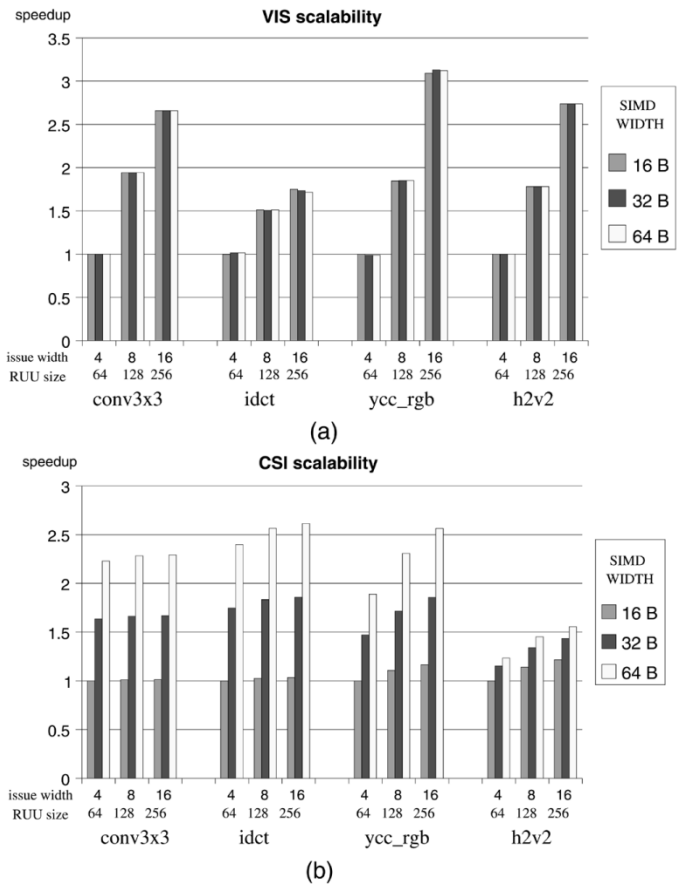

Fig. 8. Scalability of VIS and CSI with respect to the amount of parallel execution hardware.

$\mathrm{VIS}(4, 64, z)$ for various kernels. The speedups of $\mathrm{CSI}(x, y, z)$ relative to $\mathrm{CSI}(4, 64, z)$ are depicted in Fig. 8(b). All kernels exhibited similar behavior and we, therefore, only present results for four representative kernels.

It can be seen that when the issue width is fixed, increasing the number of VIS units does not provide any benefit. Contention for VIS resources is, therefore, not a problem in a VIS-enhanced processor. Fig. 8(b) shows that CSI, on the other hand, is able to utilize additional SIMD execution resources. The only case where increasing the SIMD width does not yield a significant performance improvement is the *h2v2* kernel. The reason is that this kernel is memory-bound. It incurs many cache misses (it processes new image scanlines each time it is executed) and, furthermore, the operation it performs is relatively simple. It can also be observed that the performance of the CSI-extended processors is rather insensitive to the issue width. This is expected since CSI does not exploit instruction-level parallelism and, therefore, does not need to issue instructions in parallel.

It may be argued that the performance of the VIS-enhanced processor does not scale with the number of VIS units because the RUU is not large enough to find sufficient independent instructions. Fig. 9 shows, however, that this is not the case. It depicts the instructions per cycle (IPC) attained by the VIS-enhanced CPUs, normalized with respect to the issue width. It can be seen that's attained by the four- and eight-way VIS-enhanced processors are close to ideal. These IPCs are within 78% to 90%

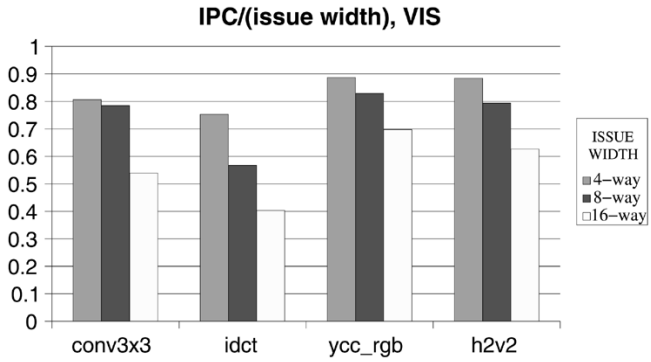

Fig. 9. Ratio of achieved IPC to issue width for several kernels for VIS-enhanced processors.

of the issue width, which means that performance cannot be improved by more than 11–28% $(1/0.9-1/0.78)$. Therefore, even when the IPC approaches the issue width, the performance of the VIS-enhanced processors will not approach that of the CSI-enhanced CPUs. This shows that the issue width together with the large number of instructions that need to be executed limit the performance of the VIS-enhanced processors.

D. Performance of a 3-D Graphics Benchmark

There are two important differences between 3-D graphics applications and the integer benchmarks studied in the previous section. First, the main data type is single precision floating-point. Because of the dynamic range of this data type, packing/unpacking is not required. Second, the lighting kernel of the geometry stage of the 3-D graphics pipeline contains *if-then-else* statements. In this section we, therefore, investigate if CSI also accelerates 3-D graphics processing.

Because VIS does not support floating-point SIMD instructions, we use SSE instead. We focus on the geometry stage of the 3-D graphics pipeline. The other two stages (database traversal and rasterization), are commonly performed by the CPU and a graphics card, respectively. Although, some modern cards also perform parts of the geometry computations, this dramatically increases their cost. We employ the industry-standard 3-D benchmark *Viewperf* from SPEC.

Since a 16-way processor is unlikely to be implemented in the near future, we only consider four- and eight-way processors. They were configured with instruction windows of 128 and 256 entries, respectively, which is twice as large as for the integer benchmarks. This was done to increase the possibility of finding independent instructions, since floating-point SIMD instructions take more time than integer SIMD instructions. Larger windows provided no significant improvements. As was done in the previous section, we also varied the SIMD width. The SSE-enhanced processor was configured with either one, two, or four SSE units, each of which can perform four single-precision floating-point operations in parallel. Accordingly, the CSI execution unit was configured with a datapath width of 16, 32, or 64 B. To investigate if the number of cache ports constitutes a bottleneck, we also consider a 4-ported cache in addition to a 2-ported cache.

Fig. 10 depicts the speedups attained by the SSE- and CSI-enhanced processors over the four-way SSE-enhanced processor

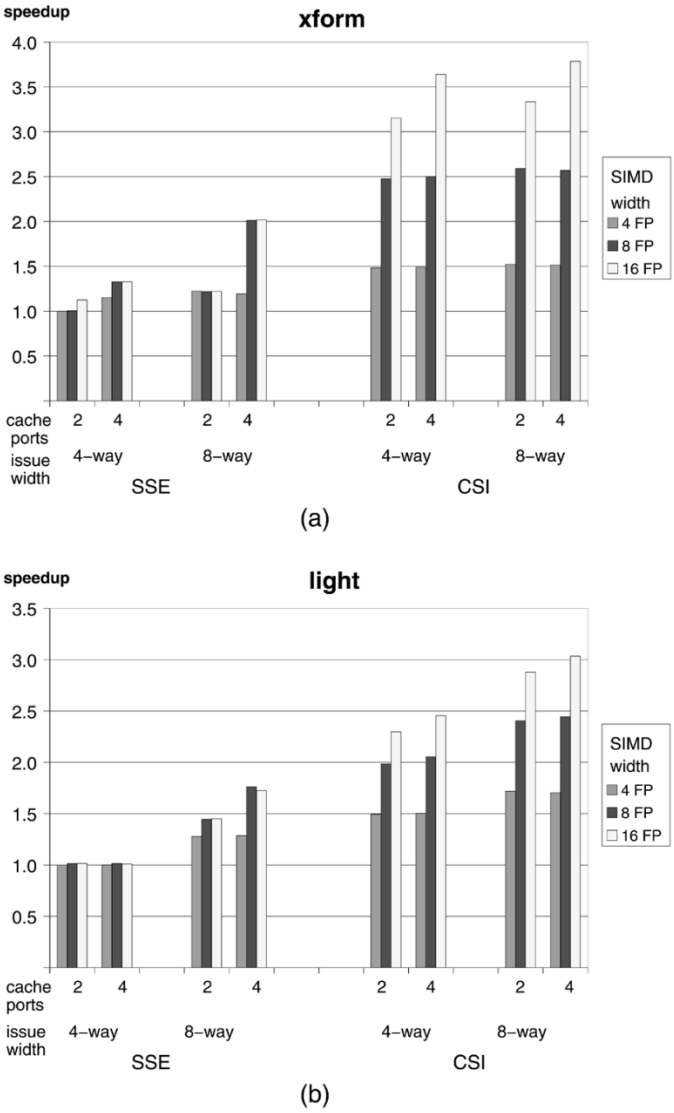

Fig. 10. Speedups of the SSE- and CSI-enhanced processors over the four-way SSE-enhanced processor with a SIMD width of 16 B.

with a SIMD width of four single-precision FP values. Fig. 10(a) shows the results for the *xform_points_4fv* kernel and Fig. 10(b) depicts the results for the *gl_color_shade_vertexes_fast* kernel. For brevity, these kernels are referred to as *xform* and *light*.

It can be observed that although these kernels incur less overhead due to the dynamic range of FP data, the CSI extension provides significant performance gains. For example, on the *xform* kernel the four-way CSI-enhanced processor that can perform 16 FP operations in parallel outperforms the four-way SSE-enhanced processor with the same processing capabilities by a factor of 2.8. There are two reasons for this. First, CSI eliminates the sectioning overhead (i.e., the overhead associated with managing address and loop induction variables, branch instructions, etc.). Although, SSE also reduces this overhead compared to a conventional superscalar, it is not negligible. Second, the results show that the number of cache ports constitute a bottleneck for the SSE-enhanced processor. This is more significant for the eight-way than for the four-way SSE-enhanced processor, because the number of cache accesses per cycle is smaller when the issue width is 4. There is also more cache port contention in the *xform* kernel than in the *light* kernel. The reason is that the frequency of load/store instructions is higher in the *xform* kernel than in the *light* kernel. The performance of the CSI-enhanced processor, on the other hand, is independent of the number of cache ports. The reason is that the CSI execution unit accesses complete cache lines but processes only parts of them (because the SIMD width is smaller than the line size of 64 B). So, the CSI execution unit does not need to perform multiple cache accesses per cycle. Contention for cache ports could be avoided if SSE would provide an instruction that loads multiple consecutive SSE registers. We also remark that CSI performance again scales well with the amount of parallel execution hardware and does not require increasing the issue width. SSE, on the contrary, requires such increases in order to utilize more parallel hardware.

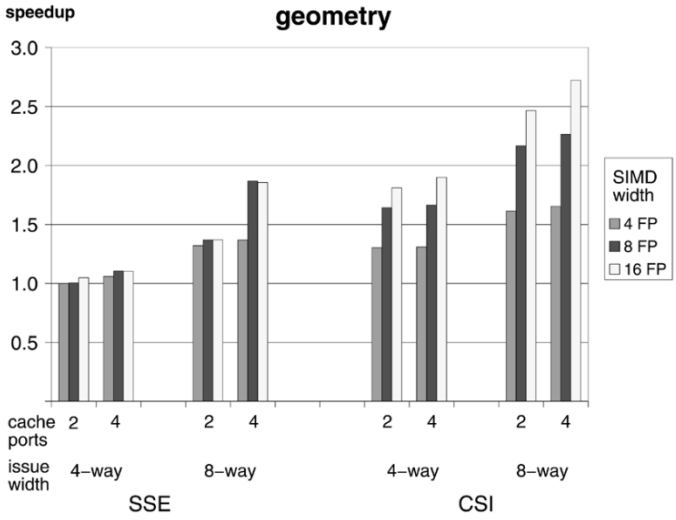

Fig. 11. Speedups of the SSE- and CSI-enhanced processors over the four-way SSE-enhanced processor with a SIMD width of 16 b for the *geometry* benchmark.

The speedups attained for the complete geometry stage of the 3-D graphics pipeline are depicted in Fig. 11. Of course, because large parts of the application have not been rewritten, partly because of project time limitations and partly because other kernels do not contain substantial amounts of data-level parallelism, the application speedup is smaller than the kernel speedups. Nevertheless, CSI provides a performance boost of 22% to 80%. We also observe that even though increasing the issue width of the SSE-enhanced processor provided no performance benefit for the *xform* and *light* kernels, it does speedup the full application. This is because other code sections benefit from a larger issue width.

VI. Conclusion

In this paper, we have proposed a novel media ISA extension called the CSI instruction set. We have found that typically a multimedia kernel must pass through seven steps: 1) generate and align addresses; 2) load data; 3) align data; 4) convert (unpack) data from storage to computational format; 5) process data; 6) convert (pack) data back to storage format; and finally 7) store data. A single CSI instruction carries out all of these

tasks. It has been shown that CSI significantly outperforms current multimedia extensions such as VIS and SSE. For example, a four-way CSI-enhanced superscalar processor with a CSI datapath width of 16 B outperforms a four-way VIS-enhanced machine with similar execution resources by factors of up to 7.77 on 2-D imaging and JPEG/MPEG kernels. These kernel-level speedups translate to application speedups ranging from 6% to 35% on image and video codecs. There are two main reasons why CSI achieves higher performance than VIS and SSE. First, CSI practically eliminates the overhead needed to bring the data in a form so that it can be processed in a SIMD manner, such as address and data alignment, (un)packing, and managing address and loop induction variables. Second, even though floating-point SIMD extensions incur less overhead due to the dynamic range of floating-points, CSI reduces contention for cache ports because it fetches complete cache blocks. As a result, a CSI-enhanced superscalar processor accesses the cache less often than processors extended with VIS or SSE. An additional advantage of CSI is that performance can be improved by simply increasing the datapath width of the CSI execution unit without having to increase the issue width. Furthermore, since the code for the CSI architecture is independent of the number of bits or elements that are processed in parallel, the same program can run on different implementations, thereby facilitating code maintenance.

Since CSI instructions that process streams incur a certain startup cost, they are not very efficient when the streams are short. However, because CSI processes 2-D streams, short streams are commonly not encountered. Furthermore, there are two reasons why the startup cost is usually tolerable. First, the instructions that set the SCRs are executed by the superscalar core. Second, CSI instructions often process many different subblocks. This means that only for the first subblock all SCRs need to be initialized. Thereafter, only the base address needs to be changed. Only for one kernel (*idct*) we observed that VIS was slightly more efficient than CSI. However, this was because the CSI implementation was based on an $O(n^3)$ algorithm, whereas the VIS version was based on an $O(n^2 \log n)$ algorithm. If the `csi_idct` instruction is employed, CSI is faster than VIS by a factor of 2.48.

There are several directions for future research. First, we are currently investigating if CSI can be used to improve the performance of scientific and engineering workloads. Second, we intend to investigate if multiple CSI SIMD units can be chained together to avoid having to write temporary streams to the L1 cache. Finally, we plan to develop a VHDL model of the CSI execution unit in order to estimate the area, power, and timing requirements of the CSI execution unit.

REFERENCES

[1] P. Ranganathan, S. Adve, and N. Jouppi, "Performance of image and video processing with general-purpose processors and media ISA Extensions," in *Proc. Int. Symp. Computer Architecture*, 1999, pp. 124–135.

[2] K. Diefendorff and P. Dubey, "How multimedia workloads will change processor design," *IEEE Computer*, vol. 30, pp. 43–45, Sep. 1997.

[3] R. Lee and M. Smith, "Media processing: A new design target," *IEEE Micro*, vol. 16, pp. 6–9, Aug. 1996.

[4] A. Peleg, S. Wilkie, and U. Weiser, "Intel MMX for multimedia PCs," *Comm. ACM*, vol. 40, no. 1, pp. 24–38, 1997.

[5] S. Thakkar and T. Huff, "The internet streaming SIMD extensions," *Intel Technol. J.*, pp. 26–34, May 1999.

[6] M. Tremblay, J. O'Conner, V. Narayanan, and L. He, "VIS speeds new media processing," *IEEE Micro*, vol. 16, pp. 10–20, Aug. 1996.

[7] L. Gwennap, "AltiVec Vectorizes Powerpc,", vol. 12, Microprocessor Report, 1998.

[8] R. Bhargava, L. John, B. Evans, and R. Radhakrishnan, "Evaluating MMX technology using DSP and multimedia applications," in *Proc. Int. Symp. Microarchitecture*, 1998, pp. 37–45.

[9] H. Nguyen and L. John, "Exploiting SIMD parallelism in DSP and multimedia algorithms using the altivec technology," in *Proc. Int. Conf. Supercomputing*, 1999, pp. 11–20.

[10] J. Hennessy and D. Patterson, *Computer Architecture—A Quantitative Approach*, 3rd ed. New York: Elsevier, 2002.

[11] S. Palacharla, N. Jouppi, and J. Smith, "Complexity-effective superscalar processors," in *Proc. Int. Symp. Computer Architecture*, 1997, pp. 206–218.

[12] P. Rubinfeld, B. Rose, and M. McCallig, "Motion video instruction extensions for Alpha," *White Paper*, 1996.

[13] N. Slingerland and A. Smith, "Measuring the performance of multimedia instruction Sets," *IEEE Trans. Computers*, vol. 51, pp. 1317–1332, Nov. 2002.

[14] J. Corbal, M. Valero, and R. Espasa, "Exploiting a new level of DLP in multimedia applications," in *Proc. Int. Symp. Microarchitecture*, 1999, pp. 72–81.

[15] 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*, vol. 21, pp. 35–47, Mar./Apr. 2001.

[16] C. Kozyrakis and D. Patterson, "Vector Vs. superscalar and VLIW architectures for embedded multimedia benchmarks," in *Proc. Int. Symp. Microarchitecture*, 2002, pp. 283–293.

[17] S. Vassiliadis, B. Juurlink, and E. Hakkenes, "Complex streamed instructions: Introduction and initial evaluation," in *Proc. EUROMICRO Conf.*, 2000.

[18] B. Juurlink, D. Tcheressiz, S. Vassiliadis, and H. Wijshoff, "Implementation and evaluation of the complex streamed instruction set," in *Proc. Int. Conf. Parallel Architectures and Compilation Techniques*, 2001, pp. 73–82.

[19] D. Talla and L. John, "Cost-effective hardware acceleration of multimedia applications," in *Proc. IEEE Int. Conf. Computer Design*, 2001, pp. 415–424.

[20] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi, "The reconfigurable streaming vector processor (RSVP™)," in *Proc. Int. Symp. Microarchitecture*, 2003, pp. 141–150.

[21] A. Padegs, B. B. Moore, R. M. Smith, and W. Buchholz, "The IBM system/370 vector architecture: Design considerations," *IEEE Trans. Computers*, vol. 37, no. 5, pp. 509–520, May 1988.

[22] K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing*, 2nd ed. New York: McGraw-Hill, 1984.

[23] D. Cheresiz. (2003) Complex Streamed Media Processor Architecture. Leiden University, Leiden, The Netherlands. [Online]. Available: http://ce.et.tudelft.nl/publicationfiles/682_5_Dmitry_bw_13-02_changed.pdf

[24] J. Corbal, R. Espasa, and M. Valero, "On the efficiency of reductions in μ-SIMD media extensions," in *Proc. Int. Conf. Parallel Architectures and Compilation Techniques*, 2001, pp. 83–94.

[25] MIPS Extension for Digital Media with 3-D [Online]. Available: ftp.yars.free.net/pub/doc/SGI/Tech_Manuals/isa5_tech_brf.pdf

[26] E. Hakkennes and S. Vassiliadis, "Multimedia execution hardware accelerator," *J. VLSI Signal Processing*, vol. 28, no. 3, pp. 221–234, 2001.

[27] M. Sima, S. Cotofana, J. van Eindhoven, S. Vassiliadis, and K. Vissers, "8×8 IDCT implementation on an FPGA-augmented trimedia," in *Proc. IEEE Symp. FPGA's for Custom Computing Machines*, Rohnert Park, CA, 2001.

[28] G. Roelofs, *PNG: The Definitive Guide*. Sebastopol, CA: O'Reilly and Associates, 1999.

[29] N. Slingerland and A. Smith, "Cache performance for multimedia applications," in *Proc. Int. Conf. Supercomputing*, 2001, pp. 209–217.

[30] T. Conte, K. Menezes, P. Mills, and B. Patel, "Optimization of instruction fetch mechanisms for high issue rates," in *Proc. Int. Symp. Computer Architecture*, 1995, pp. 333–344.

[31] D. Cheresiz, B. Juurlink, S. Vassiliadis, and H. Wijshoff, "Implementation of a streaming execution unit," *J. Syst. Architecture*, vol. 49, pp. 599–617, 2003.

[32] VIS Software Development Kit. [Online] Available: http://www.sun.com/processors/vis/vsdk.html
[33] C. Lee, M. Potkonjak, and W. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communication systems," in *Proc. Int. Symp. Microarchitecture*, 1997.
[34] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," *IEEE Computer*, vol. 35, pp. 59–67, Feb. 2002.
[35] M. Gries, "The impact of recent DRAM architectures on embedded systems performance," in *Proc. EUROMICRO Conf.*, 2000, pp. 282–289.
[36] A. Bik, M. Girkar, P. Grey, and X. Tian, "Automatic intra-register vectorization for the intel architecture," *Int. J. Parallel Programming*, vol. 30, no. 2, pp. 65–98, 2002.
[37] Streaming SIMD Extensions—Matrix Multiplication, Application Note AP-930 [Online]. Available: http://developer.intel.com/design/pentiumiii/sml/245 045.htm
[38] Diffuse-Directional Lighting, Application Note AP-596 [Online]. Available: http://cedar.intel.com/cgi-bin/ids.dll/topic.jsp?catCode=DEC
[39] W. Chen, C. Smith, and S. Fralick, "A fast computational algorithm for the discrete cosine transformation," *IEEE Trans. Commun.*, vol. COM-25, pp. 1004–1009, Sep. 1977.
[40] D. Cheresiz, B. Juurlink, S. Vassiliadis, and H. Wijshoff, "Performance scalability of the multimedia instruction set extensions," in *Proc. Euro-Par*, 2002, pp. 678–686.

Dmitry Cheresiz graduated (*cum laude*) from Novosibirsk State University, Novosibirsk, Russia, in 1995, and received the Ph.D. degree in computer engineering in 2003 from Delft University of Technology, Delft, The Netherlands, and Leiden University, Leiden, The Netherlands, research collaboration.

In 2003, he worked as a Postdoctoral Researcher at the Computer Engineering Laboratory of Delft University of Technology. Currently, he is a Research Scientist at Philips Research Laboratories, Eindhoven, The Netherlands. His research interests include computer architecture, vector and multimedia processing, processor design, modeling, and evaluation.

Ben Juurlink (M'01–SM'04) received the M.S. degree in computer science, from Utrecht University, Utrecht, The Netherlands, in 1992, and the Ph.D. degree also in computer science from Leiden University, Leiden, The Netherlands, in 1997.

In 1998, he joined the Department of Electrical Engineering, Mathematics, and Computer Science at Delft University of Technology, The Netherlands, where he is currently an assistant professor. His research interests include instruction-level parallel processors, application-specific ISA extensions, low power techniques, and hierarchical memory systems.

Stamatis Vassiliadis (M'86–SM'92–F'97) was born in Manolates, Samos, Greece, in 1951.

He is currently a Chair Professor in the Electrical Engineering Department of Delft University of Technology (TU Delft), Delft, The Netherlands. He has also served in the electrical engineering faculties of Cornell University, Ithaca, NY, and the State University of New York (S.U.N.Y.), Binghamton, NY. He worked for a decade with IBM where he had been involved in a number of advanced research and development projects. For his work, he received numerous awards including 24 publication awards, 15 invention awards and an outstanding innovation award for engineering/scientific hardware design. His 72 USA patents rank him as the top all time IBM inventor.

Dr. Vassiliadis received an Honorable Mention Best Paper Award at the ACM/IEEE MICRO25, and the Best Paper Awards in the IEEE CAS in 1998 and 2002, IEEE ICCD in 2001 and at the PDCS in 2002.

Harry A. G. Wijshoff received the M.S. (*cum laude*) and Ph.D. degrees from Utrecht University, Utrecht, The Netherlands.

From 1987 to 1990, he was a visiting Senior Computer Scientist at the Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign. Currently, he is a professor of computer science at the Leiden Institute of Advanced Computer Science, Leiden University, The Netherlands. His research interests include performance evaluation, sparse matrix algorithms, programming environments for parallel processing, and optimizing compiler technology.