The Computer Journal Advance Access published January 17, 2008

doi:10.1093/comjnl/bxm099

A Review of SIMD Multimedia Extensions and their Usage in Scientific and Engineering Applications

M. HASSABALLAH[1,*], SALEH OMRAN[1] AND YOUSSEF B. MAHDY[2]

[1]*Mathematics Department, Faculty of Science, South Valley University, Qena, Egypt*
[2]*Faculty of Computers and Information Sciences, Assuit University, Assuit, Egypt*
**Corresponding author: m.hassaballah@mailer.svu.edu.eg*

The volume and complexity of data processed by today's personal computers are increasing exponentially, placing incredible demands on the microprocessors. In the meantime, computing performance that can be achieved by increasing the clock speed of a microprocessor is reaching to physical limits thus making the architectural solutions more prominent. Due to this an important architectural feature is added to recent microprocessors, single instruction multiple data (SIMD), which is a set of instructions that can speed up an application performance by allowing basic operation to be performed on multiple data elements in parallel with fewer instructions. The SIMD computational technique was introduced in the IA-32 Intel® architecture with MMX technology and then further enhanced with Intel's introduction of streaming SIMD extensions (SSE), SSE 2 (SSE2) and SSE 3 (SSE3). Although programming using these SIMD extensions enables software to achieve higher performance, several exiting scientific applications are not affected. This paper gives an overview of SIMD multimedia extensions. The features of these extensions are introduced. Available methods for programming with multimedia instruction sets are discussed. It also reviews recent trends to use multimedia extensions to accelerate many applications such as multimedia, scientific and engineering applications, and argues for further use in other significant computationally intensive applications.

Keywords: high performance; SIMD; multimedia extensions; MMX technology; streaming SIMD extensions; instructions

Received 6 January 2007; revised 27 October 2007

1. INTRODUCTION

A variety of multimedia processing algorithms are used in media processing environments for capturing, manipulating, storing and transmitting multimedia objects such as text, handwritten data, 2D/3D graphics and audio objects. The increasing number of multimedia applications produces a commensurate increase in demand for cost-effective multimedia processing. Traditionally, media processing was implemented in expensive custom hardware specialized for specific applications (e.g. speech, video and graphics) [1, 2]. Advances in conventional microprocessor design now permit offloading some functionality to a general-purpose processor, possibly sacrificing performance in return for reduced cost. The key is to minimize this performance degradation, potentially by adding architectural support for media processing.

Multimedia instruction set architecture (ISA) extensions to modern microprocessor have been developed to allow some basic operations to be performed simultaneously on multiple items in such a set, which means that achieving higher performance by processing more data with fewer instructions. This is done by supporting single instruction multiple data (SIMD) parallel processing across multiple data elements within specially enhanced processor registers. By providing this support, these extensions attempt to capture some of the potential speed-up due to the parallel nature of these multimedia algorithms.

Beginning with the Pentium II and Pentium with Intel MMX technology processor families, four extensions have been introduced to the IA-32 architecture (Intel architecture) to permit IA-32 processors to perform SIMD operations. These extensions include the MMX technology, streaming

SIMD extensions (SSE) extensions, streaming SIMD extensions 2 (SSE2) extensions and streaming SIMD extensions 3 (SSE3) extensions. Each of these extensions provides a group of instructions that perform SIMD operations on packed integer and/or packed floating-point data elements contained in the 64-bit MMX or the 128-bit XMM registers.

These SIMD multimedia extensions are not a technology limited to the Intel x86 architecture. Other vendors offer similar and possible binary compatible technologies (Table 1). For example, AMD offers SIMD instructions via its 3DNow! Instruction set. IBM and Apple Computer offer a floating-point and integer SIMD instruction set in AltiVec technology. A thorough survey and comparison among all these multimedia instruction sets are reported by Slingerland and Smith [3]. They compare the instruction set design in detail (first comparing integer, then floating point and finally data-type-independent functionality).

Just as many other technologies, multimedia ISA extensions are not yet used to their full potential, even though a new generation is going to appear soon. Recently, some papers [4, 5] were written to argue scientists, software developers and professionals to use these instructions technologies in their implementation to get high benefit of using modern CPUs. Since the programs that use SIMD instructions can run much faster than their scalar counterparts [6]. In the following sections, the SIMD model will be discussed in more details, as well as, the most common supporting technologies such as MMX, its extensions (SSE, SSE2 and SSE3), *3DNow*! and *AltiVec*, with the main focus will go to the Intel's multimedia ISA extensions (MMX, SSE, SSE2 and SSE3) and their usage to improve the performance of other applications rather than those were intended for.

Table 1. Microprocessors with SIMD technology

Manufacturer	Microprocessor	Name of the technology
Intel	Pentium MMX/II	MMX (MultiMedia eXtensions)
	Pentium III/Xeon	MMX, SSE (Streaming SIMD Extensions)
	Pentium IV	MMX, SSE, SSE2 (Streaming SIMD Extensions 2)
	PXA	XScale
	Pentium IV with HT technology	MMX, SSE, SSE2, SSE3 (Streaming SIMD Extensions 3)
HP	PA-RISC	MAX-2 (MultiMedia Acceleration eXtensions)
AMD	K6/K6-2/K6-III	MMX/3DNow!
	Athlon	Extended MMX/3DNow!
Compaq (digital)	Alpha	MVI (Motion Video Instruction)
Motorola	PowerPC G4/G5	Velocity Engine (AltiVec)
SGI	MIPS	MDMX (MIPS Digital Media eXtensions)
Sun	SPARC	VIS (Visual Instruction Set)
ARM	ARMv6	NEON
	PPC970	VMX
IBM	P6	VMX (with some extensions)
	BG/L and BG/P	Double hummer extensions (two-way float/double SIMD)
Sony/Toshiba	Cell (PPE)	AltiVec

This review paper, in contrast to other survey papers, is the first review that introduces the Intel's SIMD extensions in more detail and that does a detailed overview of the recent research efforts to use these extensions. The applications that have recently benefited from these multimedia extensions are reported. It discusses the problems that prevent of using the multimedia extensions extensively such as the lack of compilers support. Some solutions for these problems and future perspectives to improve the performance of many other computationally intensive applications are also suggested in this paper.

The rest of the paper is organized as follows: Section 2 gives an overview of SIMD technique with detailed information about current supporting technologies. Section 3 covers the state-of-the-art of using Intel's SIMD multimedia extensions to speed up the computations of several applications. Considerations for code implementation and programming environment are outlined in Section 4. Future research directions to improve the performance of other applications based on SIMD extensions are introduced in Section 5 and finally, the conclusions are given in Section 6.

2. SIMD TECHNIQUE

SIMD mode represents one of the earliest styles of parallel processing. It is the simplest method of parallelism and now becoming the most common. SIMD aptly encapsulates the parallel processing model. In most cases, the SIMD means the same as vectorization [7]. The basic idea is to operate the same instruction sequence simultaneously on large number of discrete data sets. Figure 1 shows a typical SIMD computation [8]. Two sets of four packed data elements (X0, X1, X2 and X3 and Y0, Y1, Y2 and Y3) are operated on in parallel with the same operation being performed on each corresponding pair of data elements. The results of the four parallel computations are stored as a set of four packed data elements. In this way, computation with SIMD enables processors supporting SIMD technique to execute one instruction on multiple data points concurrently, which increase the amount of data that can be processed in a given time interval.

Even though SIMD techniques have not found their way into ubiquitous use, they have not completely died out. Because, SIMD architectures still make a lot of sense for special applications, which are inherently parallelizable tasks

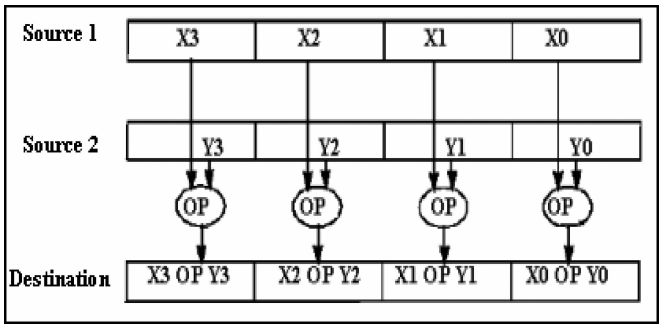

FIGURE 1. SIMD execution model.

and require a great deal of independent data computation. These applications include 3D graphics, image processing, speech recognition, scientific applications, database searches and any other applications having such inherent parallelism. In the last few years, many graphics processing units supporting SIMD technique have been designed to enhance the performance of these applications [9–11].

2.1. MMX technology

The MMX technology [12] was introduced in the later fifth-generation Pentium processors as a kind of add-on that improves image manipulation, encryption, video compression/decompression and I/O processing. It was originally designed to accelerate the multimedia and communication applications. Since it exploits the parallelism inherent in many of these applications using SIMD technique, yet maintains full compatibility with all existing IA microprocessors, operating systems and applications. MMX technology usually delivers 1.5 to 2 times speedup for the multimedia and communications applications in comparison to running on the same processor but without using MMX technology [13, 14]. MMX technology defines a simple and flexible SIMD execution model to handle 64-bit packed integer data [15, 16]. This model adds the following features to IA-32 architecture, while maintaining backwards compatibility with all IA-32 applications and operating-system code:

(i) Eight 64-bit data registers, called MMX registers (MM0–MM7).
(ii) Four MMX data types (packed bytes, packed words, packed double words and quad word).
(iii) Fifty-seven MMX instructions.

2.1.1. MMX registers

The eight 64-bit general-purpose registers of the MMX architecture can be directly addressed within the assembly by designating the register names MM0–MM7 in MMX instructions. These registers are used to hold MMX data only, and cannot be used to hold addresses nor are they suitable for calculations involving addresses. Although MM0–MM7 appear as separate registers in the Intel Architecture, the Pentium processors alias these registers with the floating point unit's (FPU) registers (ST0–ST7). Each of the eight MMX 64-bit registers is physically equivalent to the lower order 64-bits of each of the FPU's registers as shown in Fig. 2. The MMX registers overlay the FPU registers in much the same way that the 16-bit general-purpose registers overlay the 32-bit general-purpose registers.

2.1.2. The MMX data types

The MMX technology defines four different packed data types: an 8-byte array, a four-word array, a two-element double word array and a quadword object. Each element within the packed data type is a fixed-point integer. The decimal point of the fixed-point values is implicit and is left for the user to control for maximum flexibility. An MMX register processes one of these four data types as shown in Fig. 3.

As an example, graphics pixel data are generally represented in 8-bit integers, or bytes. With MMX technology, eight of these pixels are packed together in a 64-bit quantity and moved into an MMX register. When an MMX instruction executes, it takes all eight of the pixel values at once from the

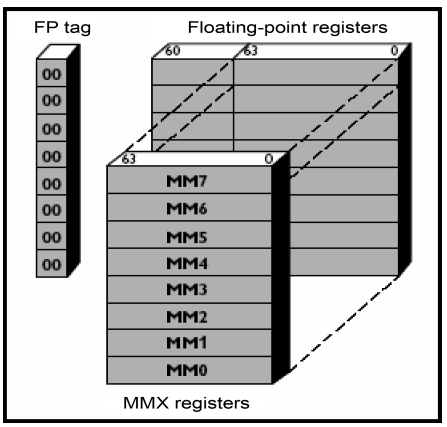

FIGURE 2. MMX and FPU register aliasing [12].

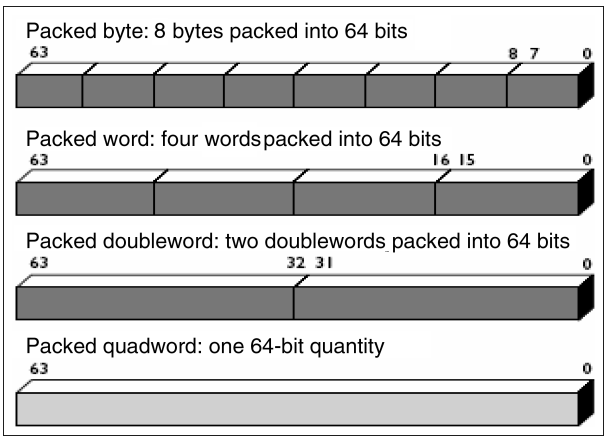

FIGURE 3. The MMX™ technology packed data types.

MMX register, performs the arithmetic or logical operation on all eight elements in parallel, and writes the result into an MMX register.

2.1.3. MMX ISA

MMX defines a set of instructions (57 MMX instructions) that perform parallel operations on multiple data elements packed into 64-bits. These 57 instructions cover several functional areas including:

(i) basic arithmetic operations such as add, subtract, multiply, arithmetic shift and multiply-add;
(ii) comparison operations;
(iii) conversion instructions to convert between the new data types: pack data together, and unpack from small to larger data types;
(iv) logical operations such as AND, AND NOT, OR and XOR;
(v) shift operations;
(vi) data instructions for MMX register-to-register transfers, or 64 and 32-bit load/store to memory and
(vii) state instruction to handle MMX to floating point transitions.

Arithmetic, comparison and shift instructions are designed to support the different packed integer data types; these instructions have a different opcode for each supported data type. As the MMX registers overlay the FPU registers, the FPU and MMX instructions cannot be mixed in the same computation sequence (i.e. concurrently). Executing of an MMX instruction sequence can be start at any time. In addition, to return the FP (floating point) stack to a sane state after MMX operations, an exit MMX machine state (EMMS) instruction must be used. This instruction resets the FPU so a new sequence of FPU calculations may be begun. The CPU does not save the FPU state across the execution of the MMX instructions; executing EMMS clears all the FPU registers. Because saving FPU state is very expensive, and the EMMS instruction is quite slow, it is not a good idea to frequently switch between the MMX and FPU calculations. Instead, the execution of MMX and FPU instructions should be at different times during program's execution. Moreover, all MMX instructions, except the EMMS instruction, reference and operate on two operands: the source and the destination operand. The first operand is the destination and the second operand is the source. The instruction overwrites the destination operand with the result. Complete coverage of all these instructions can be found in [17, 18].

An interesting feature of the architecture is *saturation arithmetic* [19]. During the normal addition of unsigned integers, an overflow condition typically results in a truncated value (often called wraparound arithmetic). This means, adding two large values may give a result smaller than the addends. In applications such as computer graphics, image processing and data compression, this may create anomalies: adding two darker shade values may result in a lighter shade value; subtracting two lighter shade values may result in a darker shade value! To fix this problem, saturation arithmetic is used which sets the result to the largest value in the range of the data type in case of overflow. Similarly, during an underflow, the result is set to the smallest value in the range of the data type. The MMX instructions support signed and unsigned saturation arithmetic in addition to the traditional, wraparound arithmetic. Simple examples demonstrating the essentials of SIMD programming using MMX instructions are given in [20]. Also, Peleg and Weiser [12] provide a comprehensive discussion on the rationale, design and applications of the MMX technology.

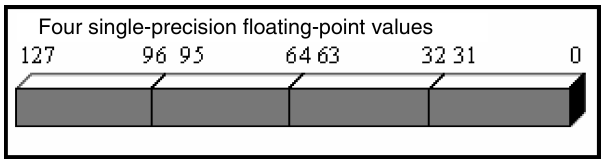

FIGURE 4. 128-bit packed single-precision floating-point data type.

2.2. Streaming SIMD Extensions (SSE)

SSE was introduced by Intel in Pentium III processor family. These extensions are an update to the MMX technology. Therefore, processors supporting SSE also support the original MMX instructions. This means that standard MMX-enabled applications run as they did on MMX-only processors. SSE extensions expand the SIMD execution model by adding facilities for handling packed and scalar single-precision floating-point values contained in 128-bit registers.

2.2.1. SSE data types

SSE extensions introduced one data type, the 128-bit packed single-precision floating-point data type, to the IA-32 architecture as shown in Fig. 4. This data type consists of four IEEE 32-bit single-precision floating-point values packed into double quadword. This 128-bit packed single-precision floating-point data type is operated on in the XMM registers or in memory. More information can be found in [8].

2.2.2. SSE programming environment and instruction set

SSE extensions add some other registers to the execution environment over those of MMX as shown in Fig. 5, these registers are:

(i) *XMM registers*: These eight 128-bit registers are used to operate on packed or scalar single-precision floating-point data. These registers can be accessed directly using the names XMM0–XMM7; and they can be accessed independently from x87 FPU, MMX registers and from the general-purpose registers. They can only be used to perform calculations on data; and cannot be used to address memory.

(ii) *MXCSR registers*: This is a 32-bit register and contains control and status information for SIMD floating-point operations.
(iii) *MMX registers*: These eight registers are used to perform operations on 64-bit packed integer data. They are also used to hold operands for some operations performed between the MMX and XMM registers.
(iv) *General-purpose registers*: As mentioned before, the MMX and XMM registers cannot be used to address memory, eight 32-bit general-purpose registers are introduced to SSE mode to address operands in memory. The general-purpose registers are also used to hold operands for some SSE instructions.
(v) *EFLAGS register*: This 32-bit register is used to record the result of some comparison operations.

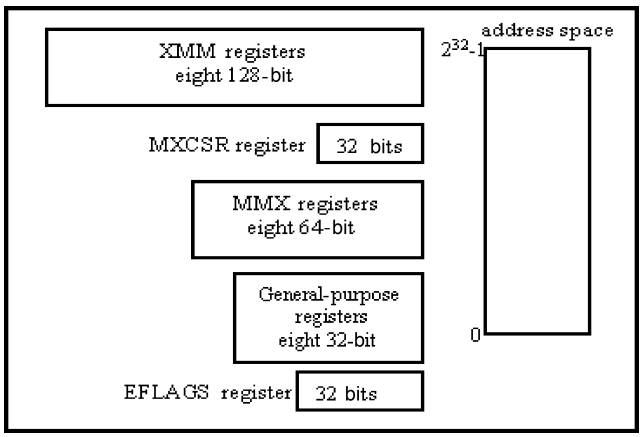

FIGURE 5. SSE execution environment.

The SSE extensions consist of 70 new instructions that can operate on the XMM registers, MMX registers and/or memory. Intel Corporation [17, 18] provides a detailed description of these instructions, which can be divided into four categories:

(i) SIMD single-precision floating-point instructions that operate on packed and single-precision floating-point values located in XMM registers and/or memory.
(ii) MXCSR state management instructions that allow saving and restoring the state of the MXCSR control and status register.
(iii) The 64-bit SIMD integer instructions that perform additional operations on packed byte, words or doubleword contained in MMX registers.
(iv) Cacheability control, prefetch and instruction ordering instructions, which provide control over the caching of non-temporal data when storing data from the MMX and XMM registers to memory.

SSE extensions are fully compatible with all software written for IA-32 processors. Recently, most software companies writing graphics and sound-related software have updated those applications to be SSE-aware applications and use the feature of SSE. For example, the high-powered graphics application such as Adobe Photoshop supports SSE instructions for higher performance on processors equipped with SSE. Microsoft included the support for SSE in its DirectX 6.1 and later video and sound drivers, which included with Windows 98, Me, 2000, NT and XP.

2.3. Streaming SIMD extensions 2

SSE2 was introduced into the IA-32 architecture in the Pentium IV and Intel Xeon processors. SSE2 allowed the ability to perform more computations in parallel, and extended those instructions introduced in MMX technology and SSE extensions. Notably, SSE2 introduces SIMD computations on two double-precision floating-point data elements. SSE2 extensions add the following features to the IA-32 architecture, while maintaining backward compatibility with all existing IA-32 Processors, applications and operating systems:

(i) Six data types.
(ii) Instructions to support the additional data types and extend existing SIMD integer operations.
(iii) Modifications to existing IA-32 instructions to support SSE2.

These new features provide the ability to perform SIMD operations on pairs of packed double-precision floating-point values. This permits higher precision computations to be carried out in XMM registers, which enhance the processor performance in scientific and engineering applications. They also provide the ability to operate on 128-bit packed integer (bytes, words, doublewords and quadwords) in XMM registers. This provides greater flexibility and greater throughput when performing SIMD operations on packed integers. Using the full set of SIMD registers, data types and instructions provided with the MMX technology and SSE/SSE2 extensions, programmers can develop algorithms that finely mix the packed single- and double-precision floating-point data on 64-and128-bit packed integer data.

No new registers or other instruction execution states are defined with SSE2 extensions. SSE2 instructions use the XMM registers, MMX registers and/or IA-32 general-purpose registers. SSE2 extensions are fully compatible with all software written for IA-32 processors. All exiting software continues to run correctly, without modification, on processors that incorporate SSE2 extensions, as well as in the presence of applications that incorporate these extensions. For more information about SSE2, see [8, 17, 18]. Also, Intel Corporation [21] gives guidelines for integrating the SSE and SSE2 extensions into an operating system environment.

2.4. Streaming SIMD extensions 3

SSE3 was introduced into the IA-32 architecture in the Pentium IV processor, which supports Hyper-Threading technology. SSE3 extensions include 13 new instructions. Ten of these 13 instructions support the SIMD execution model used with SSE/SSE2 extensions. One SSE3 instruction accelerates x87 style programming for conversion of a floating-point value to integer. The remaining two instructions accelerate the synchronization of threads. SSE3 does not introduce new data types and its programming environment is unchanged from that shown in Fig. 5. For detailed information about SSE3 and its instructions, see [8, 17, 18, 21]. In the following two sections, we will briefly discuss other two well-known technologies supporting SIMD multimedia ISA, namely 3DNow! and AltiVec.

2.5. 3DNow! technology

The 3DNow! technology was designed by AMD [22–24] as an extension to MMX/SSE to support single-precision floating-point arithmetic. The AMD-K6-2 microprocessor is the first implementation of 3DNow! technology. It is also implemented on the AMD-K6-III, and AMD Athlon™ processors. The AMD Athlon processor implements five new 3DNow! technology instructions that add streaming and digital signal processing (DSP) technologies.

3DNow! technology is a group of instructions, which opens the traditional processing bottlenecks for floating-point-intensive and multimedia applications. It uses the MMX registers but with 45 new floating-point instructions that can operate on one or two single-precision floating-point values at a time. 3DNow! supports addition, subtraction, multiplication, division, conversion to and from integers, negation, comparison, absolute, data prefetching and reciprocal square root. It is also possible to compute division and reciprocal square root to just 12-bit precision for extra speed. Using these instructions, applications can implement more powerful solution to create a more productive PC platform [25]. Just as with SSE, 3DNow! also supports single-precision floating-point SIMD operations and enables up to four floating-point operations per cycle. 3DNow! floating-point instructions can be mixed with MMX instructions with no performance penalties.

According to AMD, 3DNow! provides approximately the same level of improvement to MMX as did SSE, but in fewer instructions with less complexity. Although similar in capability, they are not compatible at the instruction level so that software specifically written to support SSE will not support 3DNow!, and vice versa. Also such as SSE, 3DNow! is well supported by software including Microsoft Windows 9x, Windows NT 4.0 and all newer Microsoft operating systems. Application programming interfaces such as Microsoft's DirectX 6.x API and SGI's Open GL API have been optimized for 3DNow! technology, as have the drivers for many leading 3D graphic accelerator suppliers, including 3Dfx, ATI and Matrox.

Examples of the type of improvements that 3DNow! enables are faster frame rates on high-resolution scenes, near theater-quality audio, much better physical modeling of real-world environments, sharper and more detailed 3D imaging and smoother video playback [25].

2.6. AltiVec™ technology

AltiVec technology [26–28] is Freescale's high-performance SIMD expansion to PowerPC®RISC processor architecture. It extends the PowerPC® architecture through the addition of 128-bit vector execution unit. This engine operates concurrently with the existing PowerPC's scalar integer and floating units and enables highly parallel operations-up to 16 operations in a single clock cycle. There is virtually no performance penalty for mingling integer, FPU and AltiVec technology operations [29]. Unlike many other extensions, which have supported media processing by leveraging existing functionality from the integer or floating-point data paths, AltiVec devotes a significant portion of the chip area to the new features and emphasizes the growing role of multimedia. AltiVec is a 128-bit wide extension with its own dedicated register file. It requires 32 registers of 128-bit width in its implementation as compared with only eight registers of same width in SSE, SSE2 and SSE3. Each value within an AltiVec register is a vector that is made up of elements. It consists of 162 floating-point and integer SIMD instructions. AltiVec instructions perform simultaneous operations on all elements within an AltiVec vector register. Depending on data size, vectors are 4, 8 or 16 elements long. These instructions offer support for:

(i) 16-way parallelism for 8-bit signed and unsigned integers;
(ii) 8-way parallelism for 16-bit signed and unsigned integers;
(iii) 4-way parallelism for 32-bit signed and unsigned integers and IEEE floating-point numbers.

There are a few instructions supporting bit-wise operations as well as making it possible to treat 128 bit of data at once in a single instruction. The target applications for AltiVec included IP telephony gateways, multi-channel modems, speech processing systems, echo cancellers, image and video processing systems, scientific array processing systems as well as network infrastructure such as Internet routers and virtual private network servers. AltiVec can also accelerate many of time consuming traditional computing and embedded processing operations such as memory copies, string compares and page clears.

3. RELATED WORKS

As mentioned before, SIMD extensions improve the performance of applications characterized by: inherent parallelism, recurring memory access patterns, localized recurring operations performed on the data and data-independent control flow. Many applications have these characteristics and their performance can be improved using SIMD extensions. Unfortunately, with all the advantages that SIMD multimedia ISA extensions introduce, only few applications have benefited of using these extensions. This section gives a comprehensive overview of up to date related research that make use of the Intel's SIMD multimedia extensions on a single processor to enhance the performance of these few applications such as multimedia, data security, database and general scientific applications.

3.1. Multimedia processing

Multimedia computing presents challenges from the perspectives of both software and hardware. For example, multimedia standards such as MPEG-1, MPEG-2, MPEG-4, MPEG-7, H.263 and JPEG 2000 involve the execution of complex media processing tasks in real-time. The need for real-time processing of complex algorithms is further accentuated by the increasing interest in 3D image and stereoscopic video processing. Each media in a multimedia environment requires different processes, techniques, algorithms and hardware. This type of applications often presents data parallelisms. Therefore, using SIMD extensions can improve their performance significantly. Much work has been done to present multimedia applications implementations on general-purpose processors with the SIMD media ISA extensions [30–48].

There are a large number of other important studies to improve multimedia applications performance using SIMD extensions. The results of over 25 research groups or individual researchers that have presented video coding implementations using SIMD multimedia extensions on general-purpose processors are summarized by Lappalainen *et al.* [49].

The discrete wavelet transform (DWT) mainly used in image/video compression (especially in JPEG2000 and MPEG-4) has been implemented using SIMD extensions in many works [50–52] to reduce the execution time of 2D/3D wavelet transform. Recently, the performance comparison of SIMD implementations of the DWT has been introduced by Shahbahrami *et al.* [53]. In this paper, they focused on SIMD implementations of the 2D DWT. The transforms considered in their work are Daubechies' real-to-real method of four coefficients (Daub-4) and the integer-to-integer lifting scheme. Daub-4 is implemented using SSE and the lifting scheme using MMX, and their performance is compared to C implementations on a Pentium IV processor. The MMX implementation of the lifting scheme is up to 4x faster than the corresponding C program for an 1-level 2D DWT, whereas the SSE implementation of Daub-4 is up to 2.6 faster than the C version. It also has been shown that when 64 k aliasing occurs the speedups are significantly higher than when it does not occur. This is because with 64 k aliasing the programs are entirely memory-bound and MMX and SSE reduce the number of memory accesses by a factor of 4. If 64 k aliasing does not occur the processing time is not insignificant but the maximum speedup of 4 cannot be achieved due to overhead required for rearranging data, loop overhead and due to lack of spatial locality. On the other hand, some common wavelet filters (e.g. Haar, Biorthogonal 4/12, Biorthogonal 7/9 and Biorthogonal 7/9 with Lifting) were implemented in [54] using SIMD operations based on the 1-D transforms, producing reasonable speedups.

Padua *et al.* [55] used MMX technology to improve the processing time of large satellite images. Seven operations commonly present in many digital image-processing algorithms were implemented in both MMX assembly and C. The results suggested that the routines in MMX as being a good alternative in the construction of image processing systems. In spite of the apparently difficulty in construct this routines, once they are written as a library their use can bring a great improvement.

Geometry processing is also an inherently parallel task, since each object vertex can be processed independently. Using SIMD instructions, operations on multiple vertices can be performed in one instruction [56]. Ma and Yang [57] evaluated the performance impact of using Intel SSE for 3-D geometry processing. They used SIMD-PF to improve the computational throughput by processing four vertices in parallel. Their experimental results showed that the Intel SSE provides significant speedup for geometry pipeline. The speedup ranges from 3.0x to 3.8x. The layout of vertices in memory is crucial for the effectiveness of SIMD-FP. Also, prefetching shows significant performance improvement for lighting. However, for transformation, it shows little performance benefit. Sometimes, the prefetching overhead even outweighs the benefit.

Optimizing of multimedia application performance using SIMD extensions was also investigated in many other works [58–62]. For example, improving fast Fourier transform performance using SIMD extensions was studied in detail in [63–65], desirable improvements were achieved based on the SIMD extensions. Another example is H.263/H.264, an emerging video coding standard, which aims at compressing high-quality video contents at low-bit rates. The complexity of H.263/H.264 standard poses a large amount of challenges to implement the encoder/decoder in real-time via software on personal computers. On the basis of the SIMD extensions, some articles [66–70] discussed the problem. Chen *et al.* [71] analyze the software implementation of H.264 encoder and decoder on general-purpose processors with media instructions and multi-threading capabilities. Specifically, the

authors discussed how to optimize the algorithms of H.264 encoders and decoders on Intel Pentium IV processors. They first analyzed the reference implementation to identify the time-consuming modules, and presented optimization methods using media instructions to improve the speed of these modules. After appropriate optimizations, the speed of the codec improved by more than 3x. Nonetheless, the H.264 encoder is still too complicated to be implemented in real-time on a single processor. Thus, they studied how to partition the H.264 encoder into multiple threads, which then can be run on systems with multiple processors or multi-threading capabilities. The authors also analyzed different multi-threading schemes that have different quality/performance, and proposed a scheme with good scalability (i.e. speed) and good quality. Their encoder can obtain another 3.8x speedup on a four-processor system or 4.6x speedup on a four-processor system with Hyper-Threading technology.

An optimized method to quarter-pixel interpolations used in H.264/AVC using SIMD instructions is presented in [72]. The implementation of the proposed method is approximately six times faster than that of the JM reference software for the H.264/AVC quarter-pixel interpolation operation, which needs multiple bilinear and 6-tap filtering.

On the other hand, microprocessor vendors have provided special-purpose instructions to accelerate the sum-of-absolute differences (SAD) similarity measurement. The usefulness of these special-purpose instructions is limited except for the motion estimation kernel. The limitations of these special-purpose instructions such as ***psadbw*** and ***pdist*** in media SIMD extensions are discussed by Shahbahrami *et al.* [73]. In this paper, the authors design and evaluate a variety of SIMD instructions for different data types. They synthesize special-purpose instructions using a few general-purpose SIMD instructions. In addition, they employ the extended subwords technique to avoid conversion overhead and to increase parallelism. In this technique, there are four extra bits for every byte of register. These extra bits provide much more room for many operations to be performed without overflow and avoid packing/unpacking overhead instructions. Their results show that using different SIMD instructions and extended subwords achieve a speedup ranging from 10.39 to 14.57 over C performance for SAD, sum-of-squared differences (SSD) with interpolation, and SSD functions in the motion estimation kernel, whereas MMX achieves a speedup ranging from 4.61 to 7.42. Additionally, the proposed SIMD instructions improve the performance of similarity measurement for image histograms by a factor ranging from 8.69 (1-way) to 11.70 (4-way) over C, whereas for MMX speedup is between 2.90 (1-way) and 4.33 (4-way).

3.2. Data security

The current data security techniques and tools are not flexible and fast enough to be useful for the next generation information technologies, e.g. mobile personal communications, electronic commerce and the Internet. Cryptographic algorithms are often organized as an iteration of a common sequence of operations. In many applications, encryption and/or hashing forms a computational bottleneck, and an increased performance of these basic cryptographic primitives is often directly reflected in an overall improvement of the system performance [74]. Elliptic curve cryptosystems are considering a vital technology for cryptography because of their high security with shorter key-length and faster computation than existing other cryptographic schemes. To increase the performance of elliptic curve computations, Aoki *et al.* [75] proposed two techniques for parallel computing with SIMD instructions, which significantly enhance the speed of elliptic curve scalar multiplication. They implicitly assumed that the time for computing multiplication by a constant over the definition field is negligible. Their computation time for computing Elliptic Curve ADDition (ECADD) and doubling (ECDBL) is $5M + S$ and $2M + 3S$, where M and S are the computation time of a multiplication and a squaring of the definition field, respectively. They evaluated one of them based on a real implementation on a Pentium III, which incorporates the SIMD architecture. Their study showed that the proposed method, based on SIMD operations, is about 4.4 times faster than the conventional methods.

Efficient algorithms for assembling an ECADD, ECDBL and k-iterated ECDBL (k-ECDBL) with SIMD operation are also introduced in [76]. These algorithms are written for the proposed addition formulas using only basic operations of the definition field namely multiplications, squarings, additions and subtractions. If A is the computation time of an addition or a subtraction of the definition field, the proposed ECADD requires time of $4M + 2S + 6A$ with eight auxiliary variables. The proposed ECDBL requires $2M + 3S + 7A$ with seven auxiliary variables. And the computation time of the k-ECDBL formula requires $2kM + (2k + 1)S + 7kA$. Using the signed binary chain, a scalar multiplication can be computed $\sim$10% faster than the previously fastest known algorithm by Aoki *et al.* Combined with the sliding window methods or the width-w NAF window method, the authors achieved $\sim$10% faster parallelized scalar multiplication algorithms with SIMD operations.

In [77], four versions of Secure Hashing Algorithms (SHA), namely SHA-1, SHA-256, SHA-384 and SHA-512, are analyzed to determine possible performance gains that can be achieved using SIMD operations, and performed on integers. The author pointed out the appropriate parts of each algorithm, where SIMD instructions can be used, and showed that each SHA algorithm has a great potential to boost both its speed and throughput using SIMD technology.

An optimized implementation of Advanced Encryption Standard (AES) algorithm in software based on the Intel's SIMD architecture is described in [78]. In this implementation, the author has optimized AES by following a top-down

approach in which the use of SMD instructions is the final step in the optimization process. The study showed that the proposed technique yields a significant increase in the performance and thereby the throughout of AES. For the encryption benchmarks, the execution speed in clock cycles/byte is 88.56 and 61.13 without and with SIMD, respectively. On other hand, the execution speed for decryption benchmarks is 89.00 without using SIMD and 54.43 with SIMD instructions. It also demonstrated that AES is a good candidate for optimization using SIMD approach.

3.3. Database

As database technology becomes pervasive, database management systems have been deployed in a wide variety of applications. The rapid growth of data volume for the past decades has intensified the need for high-speed database management systems. Most database queries and, more recently, data warehousing and data mining applications, are very data- and computation-intensive and therefore demand high processing power. Few researchers have actively sought to design and develop architectures and algorithms for faster query execution using SIMD technique [10].

Special attention has been given to increase the performance of the most important database operations such as aggregation, indexed searches, joins and selection. Zhou and Ross [79] described SIMD implementation of many important database operations including sequential scans, aggregation, indexed searches and joins. To better utilize SIMD instructions, they assume that the underlying data is stored column-wise as a contiguous array of fixed-length numeric values. Considerable performance gains were achieved by exploiting the inherent parallelism of SIMD instructions and reducing branch mispredictions. Their study showed that using an SIMD parallelism of four, the CPU time for the new algorithms is from 10% to more than four times less than for the traditional algorithms. Also, superlinear speedups are obtained as a result of the elimination of branch misprediction effects.

3.4. Scientific applications

In contrast to the wide usage of SIMD extensions in accelerating multimedia applications, no significant research has yet reported into scientific applications-based SIMD extensions. Current research into using SIMD instructions in scientific applications is clearly at a very early stage, and it will be a long time before any generally useful systems based these instructions emerge [80]. There are few studies that investigate use of the SIMD extensions in different directions of scientific and engineering applications. These studies will be covered here.

The comparison and alignment of DNA and protein sequences are important tasks in molecular biology and bioinformatics. One of the most well-known algorithms to perform the string-matching operation present in these tasks is the Smith–Waterman (SW) algorithm [81]. However, it is a computation intensive algorithm, and many researchers have developed heuristic strategies to avoid using it, especially when using large databases to perform the search. There are some efficient implementations of the SW algorithm on general-purpose processors [82]. SW algorithm was implemented in [83] using Intel SIMD multimedia extensions (MMX and SSE). Six-fold speed-up relative to the fastest previously known SW implementation on the same hardware was achieved by an optimized 8-way parallel processing approach. A speed of more than 150 million cell updates per second was obtained on a single Pentium III 500 MHz microprocessor. A semi-heuristic database searching algorithm, namely ParAlign, specifically designed to exploit the advantages of the SIMD technology to perform both rapid and sensitive sequence database searches is introduced in [84]. Another implementation of the SW algorithm that combines fine grain and coarse grain parallelism and multi-level scheduling is presented in [85] achieving a speedup of 143 on a cluster of 16 dual-CPU Pentium IV Xeons.

Of course, matrix calculations form the kernel of many scientific applications especially mathematical algorithms. A faster matrix–matrix multiply immediately benefits these algorithms. A general matrix–matrix multiplication method using SIMD features of the Pentium III processor is presented in [86, 87], achieving 2.09 times faster than the leading public domain matrix–matrix multiply routines. Muezerie *et al.* [88] also evaluated the use of SIMD floating point instructions for matrix calculations. They proved that with a little effort the use of single-precision floating-point vector operations can speed up significantly computational intensive matrix calculations.

Another issue related to matrix calculations is solving linear system equations. Fung *et al.* [89] have presented the basic operations involved in utilizing the SSE features of the Pentium III processor to speed up LU decomposition algorithm that is commonly used in solution of linear system equations. In order to examine the effectiveness of SSE, a set of experiments was conducted with both the real and complex valued linear system equations of various dimensions. According to their results, approximately a speedup ratio of 2.5 can easily be obtained in the case of complex number algorithm. By utilizing special data structure and related intrinsics provided in the Intel's C compiler, the performance of an existing LU decomposition algorithm could be improved 80% on the average without any need for additional hardware.

Use of parallelization for polynomial root finding methods, which are also computationally intensive, has enormous effect on their execution time. Moslemi *et al.* [90] have chosen four widely used polynomial root finding methods namely, Newton's, Durand–Kerner's, Aberth–Ehrlich's and QD and implemented them using SIMD instructions with C ++ and

assembly language. Experiments showed that a speedup of three or higher can be achieved, depending on the order of polynomial, required accuracy and the method employed.

Scientific applications are various. For example, normalized cross correlation (NCC) is often the adopted similarity measure due to its robustness with respect to photometric variations. But with large size images and/or templates the matching process using NCC algorithm can be computationally very expensive. Speeding up the calculations of NCC algorithm using SIMD extensions has been studied in [91], achieving significant improvements over the brute force NCC algorithm.

Finding the minimum or maximum value in an array forms an important step in a variety of applications. Aart *et al.* [92] discuss vectorization schemes that take advantage of the streaming-SIMD-extensions in commonly used floating-point MIN and MAX reductions. Performance advantages of the presented vectorization schemes for various reduction kernels and applications are demonstrated on a 3 GHz Pentium IV processor with HT Technology and 2 GB main memory using the 9.0 Intel C++/Fortran compilers. The impact of the proposed schemes on applications as a whole can be substantial, as demonstrated with an application in the Polyhedron benchmarks. For example, for **GAS_DYN** application, a substantial speed-up of over 5.5 in total is observed and the other benchmarks exhibit satisfactory speed-ups from all the optimizations performed by the Intel Fortran compiler.

Software receivers have had a discernable impact on the GNSS research community. Often such receivers are implemented in a compiled programming language, such as C or C + +. A software receiver must emulate the DSP algorithms executed on dedicated hardware in a traditional receiver. The DSP algorithms, most notably correlation, have a high computational cost; this burden precludes many software receivers from running in real time. However, the computational cost can be lessened by utilizing SIMD operations. Gregory and James [93] demonstrate how C/C++ compatible code can be written to directly utilize the SIMD instructions. First, an analysis is carried out to demonstrate why the real-time operation is not possible when using traditional C/C++ code. Second, a tutorial outlines how to write and insert x86 assembly, with SIMD operations, into C/C++ code. Finally, a C/C++ compatible SIMD enabled arithmetic library is added to the global positioning system toolbox for use in software receivers.

4. CONSIDERATIONS FOR IMPLEMENTATION

4.1. Programming environment

Including SIMD extensions on all contemporary CPU designs is not in itself a solution to handling applications workloads. Apowerful SIMD extensions instruction set is worthless without the means to utilize it. Here, a brief overview of exiting compiler techniques that support SIMD operations and coding methodologies that may be used to implement an application using SIMD instructions will be given.

4.1.1. Compilers limitations

In spite of the fact that the SIMD extensions are present in most of current processors, today it is still difficult to find compilers that can generate efficient SIMD instructions-based code sequences, due to the difficulty of automatically extracting parallelism from conventional sequential C programs. In the mean time, the small number of commercial compilers that can automatically use these instructions are in most cases expensive and the results are unsatisfactory. This area is now a thriving field for research and development, with some reports of new compiler techniques have been recently appeared [94, 95]. One preferable previous research is an SIMD compiler experimentally developed for compiling the SIMD within a register (SWAR) [96]. This model is implemented for multiple target architectures: initially as compatible libraries, then as optimizing compilers accepting a simple high-level parallel language; namely SWARC (a data parallel C language mainly target at using SIMD instructions) [97]. However, as restrictions exist in the way of facilitating SIMD instructions, the success of a data parallel language approach greatly depends on the existence of effective parallel methods for the application fields and a well-defined parallel language for their succinct descriptions. The lack of assuming specific parallel methods at the language design stage of SWARC limits the applicability of the language.

To effectively utilize microprocessors with enhanced SIMD instructions for accelerating image processing tasks, Kyo *et al.* [98] described a method and an extended C language for parallel implementation and description of image filter tasks, as well as an SIMD compiler for the extended C language that generates efficient SIMD code sequences for image filters on Intel Pentium processors. First they showed that, based on a parallel method called row-wise method and the use of a data parallel C language called 1DC, a succinct and computational efficient description for symmetric image filters can be achieved. Then, they described the design and development of an SIMD compiler called 1DCC for translating 1DC descriptions into SIMD code sequences based on the generation and optimization of code blocks containing loop structures of SIMD instructions. The performance of code sequences for the Intel Pentium processors, generated by 1DCC from row-wise method-based 1DC descriptions of various image filter tasks, are compared with code sequences generated by the latest C compiler from conventional C descriptions. Benchmark results showed that 1DC codes outperform C codes up to four times for word operation dominant image filters, and up to seven times for a byte operation dominant image filter.

Some of the recent researches [99–101] on compilation techniques have reduced the complexity of the programming task with the goal of minimizing the effort for the programmer in learning a new language and to keep very high the performance of the compiled code. For instance, the complexity of the programming model of MMX and SSE is discussed by Conte *et al.* [102]. They introduce a programming methodology and the Aphelium compiler. The performance of Aphelium is based on optimizing the code for MMX and Intel P6 processor core, which is used in all Pentium processors since Pentium Pro. Thus, there is no support for other multimedia ISA extensions or processor cores.

Leupers [103] presents a novel code selection technique capable of exploiting SIMD instructions when compiling plain C source code. It permits to take the advantage of SIMD instructions while still using portable source code. His approach builds on the classical tree-based code selection paradigm, but it generates alternative covers. The detailed code selection is performed only later, when enough information for the generation of SIMD instructions for an entire data flow graph is available. This may be sufficient for current processors, but new SIMD architectures are bound to offer higher levels of parallelism. Larsen and Amarasinghe [104] propose a robust compiler algorithm for synthesizing SIMD instructions from the statements in single basic blocks instead of in loop nests only.

In some applications, higher parallelism could be achieved if compilers inserted permutation instructions that reorder the data in registers, using compiler-known functions. Each of such functions directly corresponds to a specific SIMD instruction. By inserting such functions in a program, the programmer instructs the compiler which instructions need to be selected, and the compiler finishes the job by performing register allocation and code scheduling. The permutation topic is discussed recently in some works. Kudriavtsev and Kogge [105] describe how SIMD instructions can be created from regular code, and how the ordering of individual operations in the SIMD instructions can be determined to minimize the number of permutation instructions. In their approach, individual memory operations are grouped into SIMD operations based on their effective addresses. The SIMD data flow graph is then constructed by following data dependences from SIMD memory operations. Then, the orderings of operations are propagated from SIMD memory operations into the graph. This approach is not tied to any particular architecture and can be relatively easily ported to any SIMD instruction set. The potential of this approach is demonstrated with Intel's SSE, because the architecture is wide spread and well known. This approach scales well with the number of operations in SIMD instructions (SIMD width) and can be used to compile a number of important kernels, achieving up to 35% speedup.

Another different issue related to compiler technology is the energy consumption. It is not obvious that SIMD operations can save any energy; if n operations are executed in parallel, each of them might consume the same amount of energy as if there were executed sequentially. The influence of compiler-generated code containing SIMD operations with respect to energy consumption is investigated for the first time by Lorenz *et al.* [106]. In this paper, the effects of SIMD operations on the energy consumption are shown for several benchmarks and MP3 applications. The study concluded that making use of SIMD operations leads to an average reduction of 72% in terms of energy and 76% in terms of performance.

4.1.2. Coding methodologies

With the lack of adequate compiler support for SIMD extensions, it has been clear that SIMD extensions still enhance applications performance. Today, SIMD multimedia instruction set can be utilized in three ways.

4.1.2.1. Assembly language. This is the most effective method because programming directly in assembly language for a target platform may produce the required performance gain, but it is also more tedious and error prone than any other methods. On other hand, assembly code itself is not portable across the different processor architectures. Slingerland and Smith [107, 108] used it to measure the performance of multimedia instruction sets. They study the performance of MMX/*3DNow*!, MMX/SSE, AltiVec and VIS on optimized kernels extracted from a broad multimedia workload. They compare the performance obtained with the assembly-optimized kernels to C-compiled kernels on each platform.

4.1.2.2. Shared libraries. These libraries are often available from microprocessor manufacturers, but they tend to only cover particular functions and for some particular class of microprocessors. For example, Intel's assembly libraries [109] provide the versions of many common signal processing, vector arithmetic and image processing kernels that can be called as C functions. Moreover, often there is a mismatch between the functions available in a library and what the target application requires to be efficient.

4.1.2.3. Vectorizing compilers. Ideally, high-level language compiler would be able to automatically identify parallelizable sections of code and generate appropriate SIMD instructions. There have been many proposed methods of automatic SIMD vectorization with limited success [95, 110]. As an example of vectorizing compilers that make use of the built-in functions, the Intel C ++ compiler [111]. It provides a set of intrinsics for MMX, SSE and SSE2 SIMD support, C ++ SIMD vector classes, and a loop vectorizer that extracts SIMD parallelism from code automatically [112]. It can deliver significant applications performance improvement for Microsoft Windows as well as Linux operating system environments. In the Windows environment, the Intel C ++ compiler

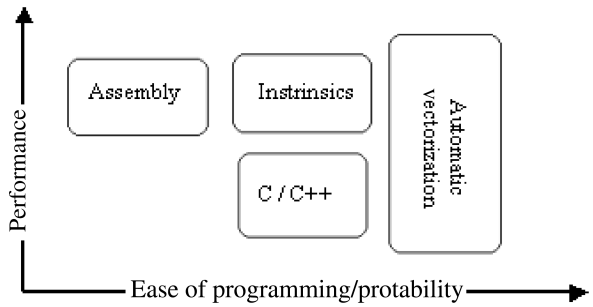

FIGURE 6. Application speed versus ease of the development for different environments.

is source and binary compatible with Microsoft Visual C ++ and plugs into the Microsoft.NET IDE, in Linux it is binary compatible with the corresponding version of gcc. On the other hand, the Portland group also offers the PGI® Workstation Fortran/C/C ++ compilers that support automatic usage of SSE/SSE2 extensions. The Codeplay™ announces the VectorC compiler for all x86 extensions and the Crescent Bay Software extends VAST to generate codes for AltiVec extension.

Today programming mechanisms via intrinsics library and C ++ class are easier to use than assembly language, particularly because programmers do not have to explicitly mange the media registers, and they especially make it easier to develop large applications. This comes at the cost of application speed. Code implemented using these mechanisms is fast, but not as fast as corresponding code written in assembly language. In some cases with a compiler that do a good job for register allocation and instruction scheduling, it is possible that the code written with intrinsics could be faster than assembly code. Recently, the Gnu compiler collection (gcc 4.2 and later versions) also does a good job [113, 114]. It supports intrinsic functions for several multimedia extensions including AtliVec, SSE2 and 3DNow!. Figure 6 illustrates the trade-offs involved in the performance of hand-code assembly versus the ease of programming and portability [115].

4.2. Alignment of data

Alignment is putting data and code in the memory in addresses that are more efficient for the hardware to access. In other word, alignment is a property of a memory address, expressed as the numeric address modulo a power of 2. For example, the address 0x0001103F modulo 4 is 3; that address is said to be aligned to $4n + 3$, where 4 indicates the chosen power of 2. The alignment of an address depends on the chosen power of 2. The same address modulo 8 is 7. An address is said to be aligned to X if its alignment is $Xn + 0$. CPUs execute instructions that operate on data stored in memory, and the data are identified by their addresses in memory. In addition to its address, a single datum also has a size. A datum is called naturally aligned if its address is aligned to its size and misaligned otherwise. For example, an 8-byte floating-point datum is naturally aligned if the address used to identify it is aligned to 8. In most processors, movement from and to memory must be word-aligned addresses. Specifically, a quadword is expected to be aligned on an 8-byte boundary [95].

Accessing a block of memory from a location that is not aligned on a natural vectorsize boundary is often prohibited or bears a heavy performance penalty. These memory alignment constraints raise problems that can be handled using the data reordering mechanisms [105]. Such mechanisms are costly, and usually involve with generating the extra memory accesses and special code for combining data elements from different vectors in each iteration of the loop. In order to avoid these penalties, techniques such as loop peeling and static and dynamic alignment detection [116, 117] can be used. Many CPUs, such as those based on Alpha, IA-64, IA-32, MIPS and SuperH architectures, refuse to read misaligned data. When a program requests that, one of these CPUs access data that is not aligned, the CPU enters an exception state and notifies the software that it cannot continue. On ARM, MIPS and SH device platforms, for example, the operating system default is to give the application an exception notification when a misaligned access is requested.

Data alignment is considered as one of the principal problems in writing a compiler that automatically uses the SIMD extensions. Namely, the vector registers are 128-bits/each and correspond to one cacheline containing 16 bytes. It is critical that data loaded into a 128-bit register are aligned beginning on a cacheline (block) boundary. For SIMD extensions such as AltiVec, if the data are not properly aligned, the address is truncated to the nearest block boundary and the loaded data are incorrect. For example, if 'float' elements **a[1]** through **a[4]** are to be loaded and the cacheblock begins at **a[0]**, the elements **a[0]** through **a[3]** will be loaded in the register. Management of these alignments makes auto-vectorization difficult and limits its scope [118]. In order to provide the best performance for memory accesses in the multimedia extensions that load or store consecutive subwords from/to memory, the memory access must be correctly aligned. This means that an n-byte transfer must be set on an n-byte boundary.

This alignment constraint can significantly impact the effectiveness of SIMD vectorization. For example, the addition of two arrays of size 1024×1024, whose addresses are either aligned or unaligned, aligned code is 1.47 times faster than unaligned code using SSE instructions [119]. If aligned access cannot be guaranteed, the programmer should consider the alignment in software using overhead instructions. This means that the data from two consecutive aligned addresses

must explicitly merge. Fridman [120] has explained three solutions for data alignment. First, maintaining multiple replications of coefficients. This approach used by Intel for MMX and SSE implementation of the FIR filter. Second method depends on the memory system support for misaligned accesses, as in SSE memory systems [60]. Third, accessing the aligned memory addresses before and after the misaligned memory address and providing misaligned subwords using rearrangement instructions. These techniques have some limitations. For example, the replication of the coefficients has two drawbacks. It needs large memory for replication of the n filter coefficients. Also, this method cannot be applied to algorithms, which use dynamic data. In the memory system that supports misaligned accesses, however, a single misaligned access is significantly slower than an aligned access. Moreover, using a permutation unit and shifter in the third method causes the extra execution time and larger code size of the program.

Larsen *et al.* [121] have concentrated on the detection of memory alignments and with techniques to increase the number of aligned references in a loop. They used loop peeling to align accesses. The loop peeling method is equivalent to the Eager-Shift policy with the restriction that all memory references in the loop must have the same misalignment. Their approach scales better with the SIMD width, and can be applied to many SIMD media ISA extensions. Eichenberger *et al.* [122] describe their approach to the problem of SIMD vectorization (SIMDization) with data alignment constraint. They have proposed a systematic method to simdizing loops with misaligned stride one-memory references for SIMD architecture with alignment constraints. In their method, data reorganization instructions are automatically generated during the simdization to align data in registers. These instructions are inserted into the simdized code to satisfy the actual alignment constraints. They have introduced a data reorganization operator, *vshiftstream*(*o1,o2*), which shifts all values of a register stream from offsets *o1* to *o2*. In general, they focused on generating the optimized SIMD codes in the presence of misaligned references.

Two important limitations of the alignment framework proposed by Eichenberger *et al.* [122], i.e. inefficiently handling of run-time alignment and a lack of support for length conversion are addressed in [123]. In this paper, Eichenberger *et al.* [122] propose a technique to efficiently shift arbitrary streams to an arbitrary offset, regardless whether the alignments or offsets are known at the compile time or not. This technique enables the application of the more advanced alignment optimizations such as Eager- and Lazy-Shift policies to run-time alignment. On a G5 machine with a 16-byte wide VMX/AltVec unit, their technique demonstrates a 19–34% improvement of performance over prior art on a benchmark stressing the impact of misaligned data. They address the second limitations by supporting length conversion in alignment handling. Speedup factors from 1.02 to 8.14 for real benchmarks are demonstrated over sequential execution. However, their algorithm can generate permutation instructions to align data for misaligned data references, but they do not address the general problem of generating permutations.

The alignment problem, the behavior of multimedia extensions on the aligned and unaligned memory accesses and cacheline split are discussed in detail by Shahbahrami *et al.* [119]. They evaluate the advantages and disadvantages of different techniques to avoid misaligned memory accesses such as replication of data in memory, padding of data structures, loop peeling and shift instructions. They show that the MMX implementation of the FIR filter using the replication of data is up to 2.20 times faster than the MMX implementation with misaligned accesses. Furthermore, the MMX and SSE implementations using the loop peeling technique are up to 1.45 and 1.66 faster than their implementation for addition of two arrays with different sizes, respectively. They also show that the unaligned memory accesses have a large performance penalty. For example, the MMX and SSE aligned codes for addition of two arrays are up to 2.26 and 2.72 times faster than their implementations using misaligned accesses on the Pentium III and IV processors, respectively.

Ivan *et al.* [124] have shown how the pointer alignment analysis can be used to improve code quality for multimedia processors with SIMD instruction sets. A method statically determines alignment information for program pointers have been described, and implemented using the OCE compiler framework. Initial experiments indicate that the method can significantly improve the quality of the code when compared with code, which uses dynamic alignment checking. By removing the dynamic checks, the code size can be reduced by a factor of as much as 4.5. The number of cycles can be reduced by the degree of SIMD parallelism.

Alvarez *et al.* [125] analyze the performance impact of extending the Altivec SIMD ISA with unaligned memory operations. Their results show that for several kernels in the H.264/AVC media codec, unaligned access support provides a speedup up to 3.8 times compared with the plain SIMD version, translating into an average of 1.2 times in the entire application. In addition to providing a significant performance advantage, the use of unaligned memory instructions makes programming SIMD code much easier both for the manual developer and the autovectorizing compiler.

4.3 Selection a suitable SIMD extension

As described earlier, SIMD extensions are available on any recent PC. If it is there, why not it be used? But, before coding an application, the following questions should be answered:

(i) Will the current code benefit by using MMX, SSE, SSE2 or SSE3 technology?
(ii) Is this code integer or floating-point?

(iii) What coding techniques should be used?
(iv) How the data types should be arranged and aligned?

Figure 7 provides a flowchart for the process of converting code to MMX technology, or SSE, or SSE 2.

Moreover, to use any of the SIMD multimedia ISA extensions optimally, the following situations must be evaluated:

(i) Fragments that are computationally intensive.
(ii) Fragments that are executed often enough to have an impact on performance.
(iii) Fragment that with little data-dependent control flow.
(iv) Fragments that require floating-point computations.
(v) Fragments of computation that can coded using fewer instructions.
(vi) Fragments that require help in using the cache hierarchy efficiently.

Using one of the commercial tools such as the Intel VTune performance analyzer [126], these tasks may be easier to evaluate. Intel's VTune is one of the standard performance analyzers for the x86 architectures. It uses Pentium on-chip performance monitoring hardware counters that keep track of many processor-related events. For each event type, VTune can count the total number of events during an

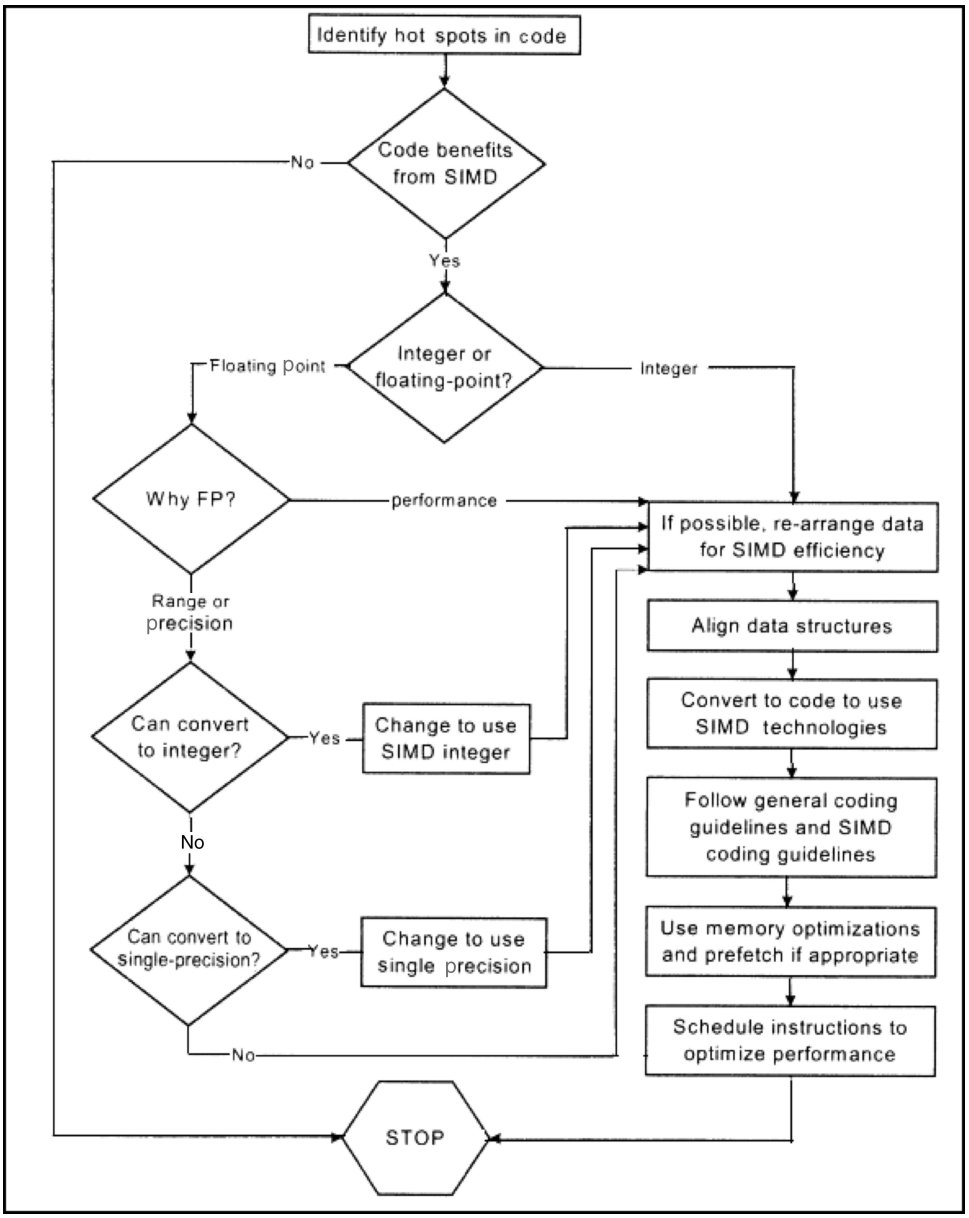

FIGURE 7. Converting to Streaming SIMD Extensions chart [115].

execution of a program and locate the spots in the program's source code where these events took place (with corresponding frequencies). Additionally, VTune also permits to perform a dynamic analysis (simulation) of a portion of a code. The simulation takes a lot of time and is therefore useful mainly for short segments of code. In other word, it helps in understanding the performance characteristics of software at all levels: the system, application, microarchitecture.

Generally, keep in mind that a good candidate code is the code that contains small-sized repetitive loops that operate on sequential arrays of integers of 8, 16 or 32 bits, single-precision 32-bit floating-point, double-precision 64-bit floating-point, using the smallest possible data type enables more parallelism with the use of a longer vector, and, careful management of memory operands can improve performance use.

5. FUTURE RESEARCH DIRECTIONS

The SIMD multimedia ISA extensions are very powerful, and can easily meet their goal of providing cheap performance for multimedia applications. Not only multimedia applications can benefit from this, but actually also many more computation intensive applications. Scientific applications with uniform structure map very well to SIMD architectures. Examples of such applications include molecular dynamics, seismic modeling with finite-grid methods and circulation patterns in the ocean and atmosphere. These applications typically have fixed data and calculations that are uniform over the entire data set.

Many mathematical methods have been developed and implemented in an iterative manner. These methods construct a sequence of approximations that converge to a certain object and repeat this procedure until the required accuracy is satisfied. Examples of such methods are: iterated function system for fractal image modeling [127], eigenvalue problems, solving large system of linear equations and finding zeros/minimum points by iterative methods [128]. When we are dealing with such iterative methods, one major problem is their convergence time, since most of these methods take a long time to converge. Some of such methods can be parallelized well, and the parallelization of them improves the convergence time to great extent. This suggests that the convergence times of such methods can be markedly reduced using an appropriate SIMD extension.

Also, re-implementation of SIMD in the current microprocessors allow even faster variations of the database operations and searching algorithms such as SW [81, 83] and ParAlign [84] algorithms, because these microprocessors include 128-bit wide registers which can be divided into 16 8-bit units. On the other hand, other algorithms for other database operations and queries such as sorting, join and indexed search must be developed. Data mining tasks such as cube roll up and drill-down, classification and clustering, which may benefit from SIMD instructions should be investigated.

Other interesting research topics include compiler technology for SIMD extensions, which is still in its infancy. Currently, to use multimedia instructions, programmers need to hand code the most time-consuming portion of each algorithm. So, how to efficiently support the multimedia instructions with high-level language compiler level is still an open question. Another interesting question that remains to be answered is to ask how well prefetching would continue to hide the memory bottleneck if the application memory access patterns were less spatially local. Although previous literature has found that multimedia applications are primarily of very small working sets, there are several applications that involve complex global interaction between data, such as image segmentation or shape from shading computer vision algorithms.

On the other hand, one drawback of the SIMD model used in general-purpose processors is that some instructions need to be added to support packing and unpacking of the registers. Thus, the performance increase due to the parallelization of the calculations is negated to some extent by the overhead of packing and unpacking. To improve the performance, some advanced data rearrangement may be performed by the compiler. Some studies [88, 129] stated that the speedup obtained for larger data trends to decrease. This is due to increasing data packing and unpacking overhead that eventually dominates the speedup gained by SIMD computations. To minimize this overhead even more, and thus improve the performance, more advanced transformations would also need to be performed. For example, to perform some operations on a subset of an array, one could first pack all relevant elements in a temporary array, perform the computation using SIMD instructions and finally put the result back into the original array.

The alignment information can serve as a basis for program transformations. One possibility is to use alignment information to cause the compiler to lay out arrays in memory differently. For example, if the alignment analysis shows that an array is unaligned for access by a loop where SIMD instructions would be appropriate, the array could be placed at a different address and thereby avoid the need for preloop code [124].

Finally, as multimedia standards become more complex, processors need to scale their SIMD extensions in order to provide the performance required to new applications as described in [130]. Scaling these extensions not only need to address performance issues, but also power consumption, design complexity and cost. The first way for scaling SIMD extensions consist of adding execution units to the SIMD pipeline. The advantage of this way is that it could improve the performance at no programming cost. The other approach of scaling is to increase the width of SIMD registers, i.e. from current 128-bit in SSE3 to 256-bit, 512-bit or more.

6. CONCLUSION

Although the state-of-the-art scientific and engineering applications are getting more complicated and demand more computational capabilities than before, the performance of personal computers has improved significantly due to the rapid growth of clock frequency as well as the enhancement of SIMD multimedia extensions. These extensions can dramatically improve the performance of today's applications, at the expense of development time. This paper gives a comprehensive overview of SIMD multimedia extensions, and its supporting microprocessors. The features of these multimedia extensions are addressed with main focus on common ones namely; the MMX/SSE/SSE2/SSE3 extensions of Intel Pentium processors. The recent use of these extensions in common applications such as multimedia, and scientific applications has been surveyed in details. Some considerations for compiler technology, programming environment and code implementation are reported in the paper. Also, this article aims to exploit these technologies in other significant scientific and engineering applications. For this purpose, several research directions for improving the performance of these applications using SIMD multimedia extensions are suggested in this paper.

REFERENCES

[1] Kuroda, I. and Nishitani, T. (1998) Multimedia processors. *Proc. IEEE*, **86**, 1203–1221.

[2] Pirsch, P., Demassieux, N. and Gehrke, W. (1995) VLSI architectures for video compression a survey. *Proc. IEEE*. **83**, 220–246.

[3] Slingerland, N.T. and Smith, A.J. (2005) Multimedia extensions for general purpose microprocessors: a survey. *Microprocess. Microsyst.*, **29**, 225–246.

[4] Ortiz, A. (2003) Teaching the SIMD execution model: assembling a few parallel programming skills. *Proc. 34th ACM SIGCSE'03, Technical Symp. on Computer Science Education*, Reno, NV, USA, February 19–23, pp. 74–78. ACM Press, USA.

[5] Krishnaprasad, S. (2004) SIMD programming illustrated using Intel's MMX instruction set. *J. Comput. Sci Coll.*, **19**, 268–277.

[6] Wadleigh, K.R. and Crawford, I.L. (2000) *Software Optimization for High Performance Computing*. Prentice-Hall PTR, Englewood Chiffs, NJ.

[7] Petersen, W.P. and Arbenz, P. (2004) *Introduction to Parallel Computing*. Oxford University Press, New York.

[8] Intel Corporation (2006) *Intel IA-32 Architecture Software Developer's Manual*, vol. **1**, Basic Architecture, Order Number 253665-018US.

[9] Cheema, M.O. and Hammami, O. (2006) Application-specific SIMD synthesis for reconfigurable architectures. *Microprocess. Microsyst.*, **30**, 398–412.

[10] Govindaraju, N.K., Lioyd, B., Wang, W., Lin, M. and Manocha, D. (2004) Fast computation of database operations using graphics processors. *Proc. ACM SIGMOD Int. Conf. Management of Data*, Paris, France June 13–18, , 215–226. ACM Press, USA.

[11] Bajaj, C., Ihm, I., Min, J. and Oh, J. (2004) SIMD optimization of linear expressions for programmable graphics hardware. *Eurograph. Comput. Graph. Forum*, **23**, 697–714.

[12] Peleg, A. and Weiser, U. (1996) MMX technology extension to the Intel architecture. *IEEE Micro*, **16**, 42–50.

[13] Lempel, O., Peleg, A. and Weiser, U. (1996) Intel's MMX^{TM} technology—a new instruction set extension. *Proc. 42nd IEEE Int. Computer Conf.*, San Jose, USA, February 23–26, pp. 255–259. IEEE Press.

[14] Bhargava, R., John, L.K., Evans, B.L. and Radhakrishnan, R. (1998) Evaluating MMX technology using DSP and multimedia applications. *Proc. 31st IEEE Int. Symp. Microarchitecture (MICRO-31)*, TX, USA, 30 November–2 December, pp. 37–46. IEEE Press, USA.

[15] Intel Corporation (1997) *MMX^{TM} technology overview*. Technical Report, Intel Corporation.

[16] Intel Corporation (1997) *The Complete Guide to MMX Technology*. McGraw-Hill, New York.

[17] Intel Corporation (2006) *Intel IA-32 Architecture Software Developer's Manual*, vol. **2A**, Instruction Set Reference, A–M, Order Number 253666-019US.

[18] Intel Corporation (2006) *Intel IA-32 Architecture Software Developer's Manual*, vol. **2B**, Instruction Set Reference, N–Z, Order Number 253667-019US.

[19] Lapsley, P., Bier, J., Shoham, A. and Lee, E.A. (1997) *DSP Processor Fundamentals: Architectures and Features*. IEEE Press Series on Signal Processing. Wiley-IEEE Press.

[20] Ortiz, A. (1999) An overview of Intel's MMX technology. *Linux Journal*, Issue 61.

[21] Intel Corporation (2006) *Intel IA-32 Architecture Software Developer's Manual*, vol. **3A**, System Programming Guide, Order Number 253668-019US.

[22] AMD Corporation (1999) *3DNow! Instruction Porting Guide*. Application Note, Issue #2261.

[23] AMD-22466B (1999) AMD Manual, Extension to 3DNow! and MMX Instruction Sets. AMD Corporation, Advanced Micro Devices.

[24] 21928G/0 Application Notes (2000) 3DNow! technology manual. 21928G/0. Application Note. http://www.amd.com/21928.pdf.

[25] Oberman, S., Favor, G. and Weber, F. (1999) AMD 3DNow! technology: architecture and implementations. *IEEE Micro*, **19**, 37–48.

[26] Phillip, M. (1998) AltiVec technology: accelerating media processing across the spectrum. *Proc. HOTCHIPS10. A Symp. High-Performance Chips*, Stanford University, Palo, Alto, CA, August 16–18.

[27] Gwennap, L. (1998) AltiVec vectorizes PowerPC forthcoming multimedia extensions improve on MMX. *Microprocessor. Report*, **12**, 6.

[28] Schmookler, M., Putrino, M., Roth, C., Sharma, M., Mather, A., Tyler, J., Nguyen, H.V., Pham, M.N. and Lent, J. (1999) A low-power, high-speed implementation of a PowerPC microprocessor vector extension. *Proc. 14th IEEE Symp. Computer Arithmetic*, April 14–16, pp. 12–20. IEEE Computer Society, USA.

[29] Diefendorff, K., Dubey, P.K., Houchsprung, R. and Scales, H. (2000) AltiVec extension to PowerPC accelerates media processing. *IEEE Micro*, **20**, 85–95.

[30] AP-528 Application notes (1998) *Using MMX Instructions in a Fast 1DCT Algorithm for MPEG Decoding*. Intel Corporation.

[31] AP-808 Application Notes (1999) *Split-Radix fast Fourier Transform Using Streaming SIMD Extensions*, Version 2.1. Intel Corporation.

[32] Order number 243631-004 (1999) *Streaming SIMD Extensions-3D Transform*. Intel Corporation.

[33] AN-942 (2000) *Using Streaming SIMD Extensions 2 in Motion Compensation for Video Decoding and Encoding*, version 2.0. Intel Corporation.

[34] AN-945 (2000) *Using streaming SIMD Extensions 2 to Implement an Inverse Discrete Cosine Transform*, version 2.0. Intel Corporation.

[35] Yen-Kuang, C., Matthew, H., Eric, D., Sergey, Z., Alexander, K., Stanislav, B., Roman, B. and Ishmael, S. (2002) Media applications on hyper-threading technology. *Intel Technol. J. Q1*, **6**, 1–11.

[36] Martins, F.M., Nickerson, B.R., Bostrom, V. and Hazra, R. (1999) Implementation of a real-time foreground/background segmentation system on the Intel architecture. *Proc. IEEE Int. Conf. Computer Vision (ICCV'99)*, Kerkyra, Greece, September, 20–27. IEEE Press, USA.

[37] Yam, C.Y. (2005) *Optimizing Video Compression for Intel Digital Security Surveillance Applications with SIMD and Hyper-Threading Technology*. White Paper, Intel Corporation.

[38] Lee, R.B. (1995) Accelerating multimedia with enhanced microprocessors. *IEEE Micro*, **15**, 22–32.

[39] Bhaskaran, V., Konstantinides, K., Lee, R.B. and Beck, J.P. (1995) Algorithmic and architectural enhancements for real-time MPEG-1 decoding on a general purpose RISC workstation. *IEEE Trans. Circuits Syst. Video Technol.*, **5**, 380–386.

[40] Lee, R.B. and Smith, M.D. (1996) Media processing: a new design target. *IEEE Micro*, **16**, 6–9.

[41] Lee, R.B. (1997) Multimedia extensions for general-purpose processors. *Proc. IEEE Workshop on VLSI Signal Processing*, Leicester, UK, November 3–5, 1–15. IEEE Press.

[42] Murata, E., Ikekawa, M. and Kuroda, C. (1998) Fast 2D 1DCT implementation with multimedia instructions for a software MPEG2 decoder. *Proc. IEEE Int. Conf. Acoustics, Speech and Signal Processing*, Seattle, WA, USA, May 12–15, vol. **5**, pp. 3105–3108. IEEE Press.

[43] Berekovic, M. *et al*. (1999) Instruction set extensions for MPEG-4 video. *J. VLSI Signal Process.*, **23**, 27–49.

[44] Tung, Y.-S., Chia-Chiang, H. and Ja-Ling, W. (1999) MMX-based DCT and MC algorithms for real-time pure software MPEG decoding. *Proc. IEEE Int. Conf. Multimedia Computing and Systems*, Florence, Italy, July 7–11, vol. **1**, pp. 357–362. IEEE Press.

[45] Lappalainen, V., Defee, P. and Hallapuro, A. (2000) Performance of an advanced video codec on a general-purpose processor with media ISA extensions. *Proc. IEEE Int. Conf. on Consumer Electronics*, Los Angles, CA, USA, June 13–15, vol. **45**, pp. 318–319. IEEE Press.

[46] Lee, R.B., Fiskiran, A.M., Shi, Z. and Xiao, Yang (2002) Refining instruction set architecture for high performance multimedia processing in constrained environments. *Proc. IEEE Int. Conf. Application-Specific Systems, Architectures and Processors*, July 17–19, pp. 253–264. IEEE Computer Society, USA.

[47] Paver, N.C., Khan, M.H. and Aldrich, B.C. (2004) Accelerating mobile multimedia using Intel wireless MMX technology. *Proc. 6th IEEE Int. Symp. on Multimedia Software Engineering*, Miami, FL, USA, December 13–15, pp. 491–498. IEEE Press.

[48] Waerdt, J.W. and Vassiliadis, S. (2005) Instruction set architecture enhancements for video processing. *Proc. 16th IEEE Int. Conf. Application-Specific Systems Architectures and Processors* (ASAP), Samos, Greece, July 23–25, pp. 146–153. IEEE Press.

[49] Lappalainen, V., Hamalainen, T.D. and Liuha, P. (2002) Overview of research efforts on media ISA extensions and their usage in video coding. *IEEE Trans. Circuits Syst. Video Technol.*, **12**, 660–670.

[50] Chaver, D., Tenllado, C., Pinuel, L., Prieto, M. and Tirado, F. (2002) 2-D wavelet transform enhancement on general-purpose microprocessors: memory hierarchy and SIMD parallelism exploitation. *Proc. Int. Conf. High Performance Computing*, Bangalore, India, December 18–21.

[51] Tenllado, C., Chaver, D., Pinuel, L., Prieto, M. and Tirado, F. (2003) Vectorization of the 2D wavelet lifting transform using SIMD extensions. *Proc. IEEE Int. Parallel and Distributed Processing Symp.*, Nice, France, April 22–26. IEEE Press, USA.

[52] Bernabe, G. and Garcia, J.M. (2005) Reducing 3D fast wavelet transform execution time using blocking and the streaming SIMD extensions. *J. VLSI Signal Process.*, **41**, 209–223.

[53] Shahbahrami, A., Juurlink, B. and Vassiliadis, S. (2005) Performance comparison of SIMD implementations of discrete wavelet transform. *Proc. 16 IEEE Int. Conf. Application-Specific Systems, Architecture Processors (ASSAP'05)*, July 23–25, pp. 393–398. IEEE Computer Society Press, USA.

[54] Kutil, R., Eder, P. and Watzl, M. (2005) SIMD parallelization of common wavelet filters. *Proc. Int. Workshop on Parallel Numerics (PARNUM'05)*, Portoroz, Slovenia, April 20–23, pp. 141–149.

[55] Padua, F.L.C., Pereira, G.A.S., de Queiroz Neto, J.P., Campos, M.F.M. and Fernandes, A.O. (2001) Improving processing time of large images by instruction level parallelism. *CD-ROM Proc. Chilean Computing Week 2001, Workshop*

on Parallel and Distributed Systems, Punta Arenas, Chile, November 5–9. IEEE Computer Society Press, Chile.

[56] Yang, C.-L., Sano, B. and Lebeck, A.R. (2000) Exploiting parallelism in geometry processing with general purpose processors and floating-point SIMD instructions. *IEEE Trans. Comput.*, **49**, 934–946.

[57] Ma, W.-C. and Yang, C.-L. (2002) Using Intel streaming SIMD extensions for 3D geometry processing. *Proc. 3rd IEEE Pacific-Rim Conf. Multimedia (PCM)*, Hsin-Chu, Taiwan, December 18–21, pp. 1080–1087. IEEE Press.

[58] Erol, B., Kossentini, F. and Alnuweiri, H. (2000) Efficient coding and mapping algorithms for software-only real-time video coding at low bit rates. *IEEE Trans. Circuits Syst. Video Technol.*, **10**, 843–856.

[59] Ranganathan, P., Adve, S. and Jouppi, N. (1999) Performance of image and video processing with general-purpose processors and media ISA extensions. *Proc. 26th Int. Symp. Computer Architecture*, Atlanta, GA, USA, May 2–4, pp. 124–135. IEEE Computer Society, USA.

[60] Thakkar, S. and Huff, T. (1999) Internet streaming SIMD extensions. *IEEE Comput. Mag.*, **32**, 26–34.

[61] Raman, S., Pentkovski, V. and Keshava, J. (2000) Implementing streaming SIMD extensions on the Pentium III processor. *IEEE Micro*, **20**, 47–57.

[62] Franchetti, F., Kral, S., Lorenz, J. and Ueberhuber, C. (2005) Efficient utilization of SIMD extensions. *Proc. IEEE*, **93**, 409–425.

[63] Nadehara, K., Miyazaki, T. and Kuroda, I. (1999) Radix-4 FFT implementation using SIMD multi-media instructions. *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing (ICASSP '99)*, Phoenix, AZ, USA, March 15–19, vol. **4**, pp. 2131–2134. IEEE Press, USA.

[64] Kral, S., Franchetti, F., Lorenz, J. and Ueberhuber, C. (2003) SIMD vectorization of straight line FFT code. *Proc. 9th Euro-Par'03 Int. Conf. Parallel Processing*, Klagenfurt, Austria, August 26–29. LNCS 2790, pp. 251–260. Springer, Berlin.

[65] Rodriguez, P. (2002) A radix-2 FFT algorithm for modern single instruction multiple data (SIMD) architectures. *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing (ICASSP '02)*, Orlando, FL, USA, May 13–17, vol. **3**, pp. III-3220–III-3223. IEEE Press, USA.

[66] Lappalainen, V. (1998) Performance analysis of Intel MMX technology for an H.263 video encoder. *Proc. 6th ACM Int. Conf. Multimedia*, Bristol, UK, September 13–16, pp. 309–314. ACM Press, USA.

[67] Ge, S., Tian, X. and Chen, Y.-K. (2003) Efficient multi-threading implementation of H.264 encoder on Intel hyper-threading architectures. *Proc. IEEE Pacific-Rim Conf. Multimedia*, December 15–18, vol. **1**, pp. 469–473. IEEE Press, USA.

[68] Zhou, X., Li, E.Q. and Chen, Y.-K. (2003) Implementation of H.264 decoder on general-purpose processors with media instructions. *Proc. SPIE Conf. Image Video Communication and Processing*, vol. **5022**, pp. 224–235. SPIE Press.

[69] Li, E.Q and Chen, Y.-K. (2004) Implementation of H.264 encoder on general-purpose processors with hyper-threading technology. *Proc. SPIE Conf. Visual Communication and Image Processing*, vol. **5308**, pp. 384–395. SPIE Press.

[70] Iverson, V., McVeigh, J. and Reese, B. (2004) Real-time H.264/AVC codec on Intel architectures. *Proc. IEEE Int. Conf. Image Processing (ICIP '04)*, Singapore, October 24–27, vol. **2**, pp. 757–760. IEEE Press.

[71] Chen, Y.-K., Li, E.Q., Zhou, X. and Ge, S. (2006) Implementation of H.264 encoder and decoder on personal computers. *J. Vis. Commun. Image Represent.* **17**, 509–532.

[72] Chae-Bong, S. and Hye-Jeong, C. (2006) An efficient SIMD-based quarter-pixel interpolation method for H.264/AVC. *Int. J. Comput. Sci. Netw. Secur.*, **6**, 85–89.

[73] Shahbahrami, A., Juurlink, B. and Vassiliadis, S. (2006) Limitations of special-purpose instructions for similarity measurements in media SIMD extensions. *Proc. Int. Conf. Compilers, Architecture and Synthesis for Embedded Systems*, Seoul, Korea, October 23–25, pp. 293–303. ACM Press, USA.

[74] Bosselaers, A., Govaerts, R. and Vandewalle, J. (1996) Fast hashing on the Pentium. In Koblitz, N. (ed.), *Advances in Cryptology. Proc. Crypto'96*, LNCS 1109, pp. 298–312. Springer-Verlag, Berlin.

[75] Aoki, K., Hoshino, F., Kobayashi, T. and Oguro, H. (2001) Elliptic curve arithmetic using SIMD. *ISC20001*, LNCS 2200, pp. 235–247. Springer-Verlag.

[76] Izu, T. and Takagi, T. (2002) Fast elliptic curve multiplications with SIMD operations. *ICISC20002*, LNCS 2513, pp. 217–230. Springer-Verlag, Berlin.

[77] Aciicmez, O. (2004) *Fast Hashing on Pentium SIMD Architecture*. MSc Thesis, School of Electrical Engineering and Computer Science, Oregon State University.

[78] Godbole, P. (2004) *Optimizing the Advanced Encryption Standard on Intel's SIMD Architecture*. MSc Thesis, School of Electrical Engineering and Computer Science, Oregon State University.

[79] Zhou, J. and Ross, K.A. (2002) Implementing database operations using SIMD instructions. *Proc. ACM SIGMOD Int. Conf. Management of Data (SIGMOD'02)*, Wisconsin, USA, June 4–6, pp. 145–156. ACM Press, USA.

[80] Hurbain, I. and Silber, G.-A. (2006) An empirical study of some x86 SIMD integer extensions. *Proc. 12th Int. Workshop on Compilers for Parallel Computers (CPC2006)*, Spain, January 9–11.

[81] Smith, T.F. and Wateman, M.S. (1981) Identification of common molecular subsequences. *J. Mol. Biol.*, **147**, 195–197.

[82] Friman, S., Esther, S., Alex, R. and Mateo, V. (2005) Parallel processing in biological sequence comparison using general-purpose processors. *Proc. IEEE Int. Symp. Workload Characterization*, Austin, TX, USA, , October 6–8, pp. 99–108. IEEE Press, USA.

[83] Rognes, T. and Seeberg, E. (2000) Six-fold speed-up of Smith-Waterman sequence database searches using parallel

processing on common microprocessors. *Int. J. Bioinf.*, **16**, 699–706.

[84] Rognes, T. (2001) ParAlig: a parallel sequence alignment algorithm for rapid and sensitive database searches. *Int. J. Nucleic Acids Res.*, **29**, 1647–1652.

[85] Meng, X. and Chaudhary, V. (2006) Optimised fine and coarse parallelism for sequence homology search. *Int. J. Bioinf. Res. Appl.*, **2**, 430–441.

[86] Aberdeen, D. and Baxter, J. (2000) General matrix-matrix multiplication using SIMD features of the PIII. *Proc. 6th Int. Euro-Par Conf. Parallel Processing (Euro-Par2000)*. LNCS 1900, 980–983, Springer-Verlag, Berlin.

[87] Aberdeen, D. and Baxter, J. (2001) EMMERALD: a fast matrix-matrix multiply using Intel's SSE instructions. *J. Concurrency Comput.: Pract. Exp.*, **13**, 103–119.

[88] Muezerie, A., Nakashima, R.J., Travieso, G. and Slaets, J. (2001) Matrix calculations with SIMD floating point instructions on x86 processors. *Proc. 13th Symp. Computer Architecture and High Performance Computing*, Pirenópolis-GO, Brazil, September 10–12, pp. 50–55.

[89] Fung, Y.F., Ercan, M.F., Ho, T.K. and Cheung, W.L. (2002) A parallel solution to linear systems. *Microprocess. Microsyst.*, **26**, 39–44.

[90] Moslemi, M., Ahmadi, H. and Sarbazi-Azad, H. (2005) Efficient polynomial root finding using SIMD extensions. *Proc. 11th IEEE Int. Conf. Parallel and Distributed Systems (ICPADS'05)*, July 20–22, vol. **2**, pp. 529–533. IEEE Computer Society, USA.

[91] DeStefano, L., Mattoccia, S. and Tombari, F. (2005) Speeding-up NCC-based template matching using parallel multimedia instructions. *Proc. 17th IEEE Int. Workshop on Computer Architecture for Machine Perception (CAMP'05)*, July 4–6, pp. 193–197. IEEE Computer Society, USA.

[92] Aart, J.C.B., Xinmin, T. and Milind, B.G. (2006) Multimedia vectorization of floating-point MIN/MAX reductions. *J. Concurrency Comput.: Pract. Exp.*, **18**, 997–1007.

[93] Gregory, W.H. and James, L.G. (2006) SIMD correlator library for GNSS software receivers. *GPS Solut. J.*, **10**, 269–276.

[94] Krall, A. and Lelait, S. (2000) Compilation techniques for multimedia processors. *J. Parallel Program.*, **28**, 347–361.

[95] Sreraman, N. and Govindarajan, R. (2000) A vectorizing compiler for multimedia extensions. *J. Parallel Program.*, **28**. 363–400.

[96] Fisher, R.J. and Dietz, H.G. (1998) Compiling for SIMD within a register. *Proc. 11th Int. Workshop on Languages and Compilers for Parallel Computing*, August 7–9, LNCS 1656, pp. 290–304. Springer-Verlag, London, UK.

[97] Fisher, R.J. and Dietz, H.G. (2000) The SCC compiler: Swaring at MMX and 3DNow! *Proc. 12th Annual Workshop Languages and Compilers for Parallel Computing*, August 4–6, LNCS 1863, pp. 399–414. Springer-Verlag, London, UK.

[98] Kyo, S., Okazaki, S. and Kuroda, I. (2003) An extended C language and SIMD compiler for efficient implementation of image filters on media extended micro-processors. *Proc. Int. Conf. Advanced Concepts for Intelligent Vision Systems (ACIVS'03)*, Belgium, September 2–5, pp. 234–241.

[99] Lonardo, A., Panizzi, E. and Proietti, B. (2000) C++ programming language for an abstract massively parallel SIMD architecture. *CoRR CS. PL/0005023 V1*, May 19.

[100] Patricio, B. and Veselko, G. (2003) An extended ANSI C for processors with a multimedia extension. *J. Parallel Program.*, **31**, 107–136.

[101] Cockshott, P. and Michaelso, G. (2006) Orthogonal parallel processing in vector Pascal. *J. Comput. Lang. Syst. Struct.*, **32**, 2–41.

[102] Conte, G., Tommeasani, S. and Zanichelli, F. (2000) The long and winding road to high-performance image processing with MMX/SSE. *Proc. 5th IEEE Int. Workshop on Computer Architectures for Machine Perception*, Padova, Italy, September 11–13, pp. 302–310. IEEE Computer Society, USA.

[103] Leupers, R. (2000) Code selection for media processors with SIMD instructions. *Proc. ACM Conf. Design, Automation and Test in Europe*, Paris, France, March 27–30, pp. 4–8. ACM Press, USA.

[104] Larsen, S. and Amarasinghe, S. (2000) Exploiting superword level parallelism with multimedia instruction sets, *ACM SIGPLAN Not.*, **35**, 145–156.

[105] Kudriavtsev, A. and Kogge, P. (2005) Generation of permutations for SIMD processors. *Proc. ACM SIGPLAN/SIGBED Conf. Languages, Compilers, and Tools for Embedded Systems*, Chicago, IL, USA, June 15–17, pp. 147–156. ACM Press, USA.

[106] Lorenz, M., Drager, T. and Leupers, R. (2004) Compiler based exploration of DSP energy saving by SIMD operations. *Proc. Asia and South Pacific Design Automation Conf.: Electronic Design and Solution Fair (ASP-DA'04)*, Yokohama, Japan, January 27–30, pp. 838–841. IEEE Press, USA.

[107] Slingerland, A.J. and Smith, A.J. (2001) Performance analysis of instruction set architecture extensions for multimedia. *Proc. 3rd Workshop on Media and Stream Processors*, TX, USA, December 1–2.

[108] Slingerland, A.J. and Smith, A.J. (2002) Measuring the performance of multimedia instruction sets. *IEEE Trans. Comput.*, **51**, 1317–1332.

[109] Intel Cooperation. *Performance Library Suite.* http://developer.intel.com/vtune/perflibst/index.htm

[110] Bik, A.J.C., Girkar, M., Grey, P.M. and Tian, X.M. (2002) Automatic intra-register vectorization for the Intel (R) architecture. *Parallel Program.*, **30**, 65–98.

[111] Intel Cooperation. *Intel® C/C++ Compiler User's Guide*. Order Number 693500. http://www.intel.com/software/products/compilers.

[112] Wolf, J.H. (1999) Programming methods for the Pentium III processor's streaming SIMD extensions using the VTune performance enhancement environment. *Intel Technol. J. Q2*.

[113] Naishlos, D. (2004) Autovectorization in GCC. *GCC Developers' Summit*, June 28–30, pp. 105–117. http://www.gccsummit.org/

[114] Naishlos, D., Biberstein, M. and Zaks, A. (2002) *Compiler Vectorization Techniques for Disjoint SIMD Architectures*. Research Report H-0146. Haifa Research Laboratory, Haifa, Israel.

[115] Intel Corporation (2006) *IA-32 Intel Architecture Optimization Reference Manual*. Order Number 248966-013US, April.

[116] Andreas, K. and Sylvain, L. (2000) Compilation techniques for multimedia processors. *J. Parallel Program.*, **28**, 347–361.

[117] Jaime, H.M. *et al*. (2003) An innovative low-power high-performance programmable signal processor for digital communications. *IBM J. Res. Dev.*, **47**, 299–326.

[118] Gang, R., Peng, W. and David, P. (2003) A preliminary study on the vectorization of multimedia applications for multimedia extensions. *Proc. 16th Int. Workshop of Languages and Compilers for Parallel Computing*, TX, USA, October 2–4. (LNCS), Vol. 2958/2004, pp. 420–435.

[119] Shahbahrami, A., Juurlink, B. and Vassiliadis, S. (2006) Performance impact of misaligned accesses in SIMD extensions. *Proc. 17th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC2006)*, Veldhoven, The Netherlands, November 23–24, pp. 334–342. EEMCS EPrints Service, the Netherlands.

[120] Fridman, J. (1999) Data alignment for sub-word parallelism in DSP. *Proc. IEEE Workshop on Signal Processing Systems*, October, pp. 251–260. IEEE Press.

[121] Larsen, S., Witchel, E. and Amarasinghe, S. (2002) Increasing and detecting memory address congruence. *Proc. 7th Int. Conf. Parallel Architectures and Compilation Techniques*, VA, USA, September 22–25, pp. 18–29.

[122] Eichenberger, A.E., Wu, P. and O'Brien, K. (2004) Vectorization for SIMD architectures with alignment constraints. *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, Washington, DC, USA, June 9–11, vol. **39**, pp. 82–93. ACM Press, USA.

[123] Wu, P., Eichenberger, A.E. and Wang, A. (2005) Efficient SIMD code generation for runtime alignment and length conversion. *Proc. Int. Symp. Code Generation and Optimization (CGO'05)*, March 20–23, pp. 153–164. IEEE Computer Society, USA.

[124] Ivan, P., Andreas, K. and Nigel, H. (2007) Compiler optimizations for processors with SIMD instructions. *J. Softw. – Pract. Exp.*, **37**, 93–113.

[125] Alvarez, M., Salami, E., Ramirez, A. and Valero, M. (2007) Performance impact of unaligned memory operations in SIMD extensions for video codec applications. *Proc. IEEE Int. Symp. Performance Analysis of Systems & Software*, San Jose, CA, USA, , April 25–27, pp. 62–71. IEEE Press, USA.

[126] Intel Cooperation. *VTune Performance Analyzer*. http:// developer.intel.come/Vtune/

[127] Fisher, Y. (1994) *Fractal Image Compression Theory and Application.* Springer-Verlag, New York.

[128] Stoer, J. and Bulirsch, R. (2002) *Introduction to Numerical Analysis* (3rd edn), Springer-Verlag, New York.

[129] Talla, D., John, L.K. and Burger, D. (2003) Bottlenecks in multimedia processing with SIMD style extensions and architectural enhancement. *IEEE Trans. Comput.*, **52**, 1015–1031.

[130] Sanchez, F., Alvarez, M., Salami, E., Ramirez, A. and Valero, M. (2005) On the scalability of 1- and 2-dimensional SIMD extensions of multimedia applications. *Proc. IEEE Int. Symp. Performance Analysis of Systems and Software (ISPASS-2005)*, TX, USA, March 20–22, pp. 167–176. IEEE Press, USA.