

Efficient Orchestration of Sub-Word Parallelism in Media Processors

John Oliver, Venkatesh Akella, and Frederic Chong
University of California at Davis

ABSTRACT

Communication and multimedia applications with increased data rates and enhanced functionality continuously raise the bar for the computational requirements of future microprocessors. In order to meet these computational demands it is necessary to exploit sub-word parallelism efficiently. We propose to make sub-word data movement a first-class operation in microprocessor architectures by introducing a Sub-word Permutation Unit (SPU) in the execution pipeline. The SPU is evaluated in the context of the MMX media co-processor for the Intel Pentium architectures, but our results can be extended to any processor that supports sub-word parallelism. We find that the SPU allows us to orchestrate sub-word data placement prior to computation, thus allowing the MMX functional units to concentrate on performing calculations. Furthermore, we introduce a decoupled SPU control mechanism at the basic block level which allows static optimization to eliminate data-movement overhead in tight loops, where most media and signal processing occurs. We demonstrate that anywhere from 4% to 20% improvement can be obtained on key media and signal processing kernels with as little as 1% increase in hardware resources.

Categories and Subject Descriptors

C.1.2 [Processor Architectures]: Multiple Data Stream Architectures

General Terms

Design, Performance

Keywords

Sub Word Parallelism, Media Processors, Decoupled Control

1. INTRODUCTION

As information technology proliferates, next-generation microprocessors face increasingly demanding media processing workloads. Performance is key, but energy efficiency and

code size will also become important. These applications operate on smaller data types than a typical register word size (e.g.: 6 bit, 8-bit, 10-bit, 12-bit are common in Viterbi decoding, FIR filters, FFT, LDPC decoders) and have a high degree of *data parallelism* [2, 7].

To exploit this data parallelism, modern processors (both general-purpose and digital signal processors) have embraced sub-word parallelism that allows for more efficient processing of lower precision data [15, 12, 19, 14]. Standard word-precision data units are sub-divided into smaller data units called sub-words, but still share the same data paths. Since sub-words are packed into word-sized registers, the register file does not need additional ports to support sub-word computation. Any operation that is executed on the regular data unit can be executed on each individual sub-word, with a few minor modifications. Adders and multipliers need to have their carry chains optionally broken at sub-word boundaries, and control logic needs to be added to support the new sub-word instructions. To enable efficient control, Single Instruction Multiple Data (SIMD) techniques are typically employed.

However, sub-word parallelism can be awkward for instruction sets and microprocessors designed for standard word computations. Sub-word parallelism introduces some non-orthogonality¹ into the instruction set architecture. This is in the form of *restrictions* on which sub-words in a register file can be used for a given computation. We classify these restrictions into two categories: (a) inter-word restrictions and (b) intra-word restrictions. Inter-word restrictions are created because sub-words may only come from a limited number of registers, typically two registers for most load-store machines. Intra-word restrictions imply that each functional unit can only operate upon sub-words located at the same bit position – e.g. both sub-word operands must be bit-aligned within their source registers.

The result of these restrictions is that *data permutation* instructions must be inserted between SIMD operations to *align* sub-word operands properly as described in [15]. The number of data permutation instructions can be quite significant. For example, dynamic instruction counts of the EEMBC consumer benchmarks running on the Philips Tri-Media processor [6] show that over 23% of instructions are data alignment instructions such as pack/merge bytes (16.8%) and pack/merge half words (6.5%) [8].

These figures strongly suggest that *data alignment* should be supported in the micro-architecture directly as a part of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SPAA'04, June 27–30, 2004, Barcelona, Spain.
Copyright 2004 ACM 1-58113-840-7/04/0006 ...\$5.00.

¹we use the term orthogonality in the same sense as in Wulf's classic paper [24].

every sub-word computation. The trick is to do it efficiently and at the same time not modifying the instruction set drastically. That is the basic motivation for the proposed work.

We present a novel hardware structure called the Sub-word Permutation Unit (SPU), that addresses both inter-word and intra-word restrictions transparently with sub word computations. In order to make the SPU an efficient addition to the instruction set architecture to any sub-word capable processor the SPU uses a decoupled control unit. This control unit allows zero-overhead control of sub-word permutations in critical loop structures where performance is the most critical.

In the remainder of this paper, we present our case study of implementing the SPU in the context of the Pentium MMX instruction set architecture. Section 2 begins with a short description of the MMX architecture and its inter-word and intra-word restrictions. Section 3 presents the SPU architecture and describes its approach to removing sub-word restrictions from critical loops and its associated decoupled control mechanism. Section 4 describes the SPU programming interface and Section 5 presents our evaluation of the SPU. Section 6 discusses how our results apply to other signal processing and media architectures. Finally, Section 7 discusses related work and Section 8 presents our conclusions.

2. CASE STUDY: MMX

We motivate the SPU with the most common SIMD architecture available, the MMX media co-processor architecture for the Intel Pentium family [19]. MMX operates on 64-bit vectors whose sub-words are either 8, 16, 32 or 64 bits in precision. There are eight special purpose 64-bit registers which are mapped on top of the x87 floating point registers named MM0 through MM7. All MMX instructions operate in a single cycle, except multiply instructions which have a three cycle latency. In the implementation of the MMX as described in [18] there are two integer pipes called the U and V pipes. Both pipes support arithmetic and logic functions, but only one instruction can be a multiply instruction and only one instruction can be a permutation or shift instruction. The U pipe is shared with Floating Point operations and is used for all instructions that access memory. Up to four MMX registers can be read at any given time and three can be written. The destination register for the instructions in the U and V pipes should not be the same register. No read-after-write or write-after-read dependencies must exist between the two pipes. Essentially, the function of the two pipes allows the MMX to execute two MMX instructions in a given clock cycle.

There are three instructions that support sub-word computations across registers in the MMX. They are *packed multiply-add*, *packed add* and *pack/unpack* instructions.

Figure 1 shows how the packed multiply-add and packed add instructions operate on 16-bit data. The packed multiply-add (*Pmaddwd*) first multiplies the 16-bit values in the same bit positions in both input operand registers as specified by the instruction (the uppermost MM0 and MM1 in Figure 1), resulting in four 32-bit products. Then, the top and bottom 32-bit products are added, resulting in a pair of 32-bit sum-of-products (the lower most MM0 in Figure 1).

The packed addition instruction (*Padd*), also shown in Figure 1, completes the sum-of products by adding the two 32-bit values stored in a register (the lower most MM0 in

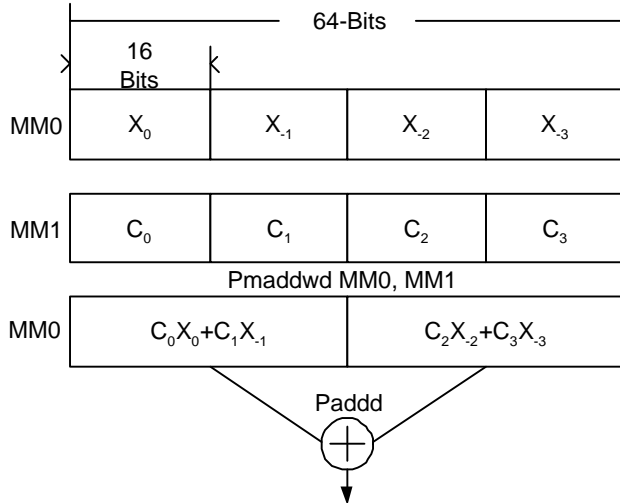


Figure 1: 16-bit packed operations

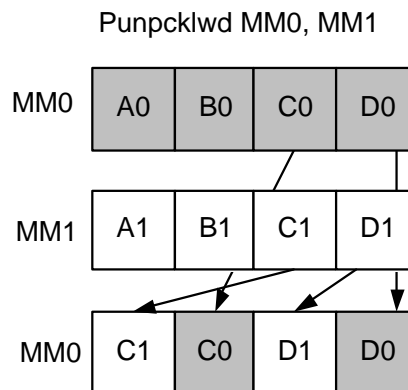


Figure 2: MMX unpack instruction.

Figure 1), resulting in a single 32-bit sum-of-products. It is interesting to note that these two operations are the core computations needed to perform a four-tap FIR filter.

Figure 2 shows the unpack instruction for the MMX. There is a corresponding pack instruction that supports saturation logic which is not shown in this figure. The pack instruction is vital to ensure proper data alignment for those MMX instructions that operate on data in the same bit-positions (all non packed multiply add and packed add computations).

2.1 Intra-Word Restrictions

We illustrate the impact of the intra-word restrictions on performance with a small but representative example from media processing. Consider the computation of the determinant of a 2x2 matrix as shown below:

$$\det \begin{bmatrix} a & b \\ c & d \end{bmatrix} \equiv \begin{vmatrix} a & b \\ c & d \end{vmatrix} \equiv ad - bc$$

Lets assume that a, b, c and d are 32-bit values that are organized as follows. 64-bit MMX register MM0 contains a and b and the 64-bit register MM1 contains c and d . To multiply a with d and b with c , we first need to swap the bit positions of the sub-words contained within MM0 or MM1, as the MMX does not support a non-bit aligned sub-word

multiply instruction. This swapping of data can either be accomplished through the use of a *unpack* instruction or a move instruction, but requires one execution cycle. This restriction occurs quite frequently in other common applications such as FIR filtering. Again, this is simply one instance of an intra-word restriction as it applies to the MMX media extension, but applies equally to almost every sub-word capable architecture.

2.2 Inter-Word Restrictions

Inter-word restrictions manifest themselves when the desired sub-words for a computation are spread across more registers than that can be addressed by a single computational instruction. A simple example of an algorithm that shows the inter-word restrictions of SIMD architectures is the matrix transpose, where the sub-words are packed in memory either in row or column ordering.

Figure 3 shows a 4x4 matrix transpose as calculated on the MMX, where each row is contained within a vector register in the MMX register file. A 4x4 transpose is shown for simplicity on 16-bit sub-words, but a similar set of operations can be used for an 8x8 matrix transpose for 8-bit sub words, or matrices of many more elements. The original source matrix is shown in the top-middle of the diagram and the transposed matrix is shown directly beneath it. Using MMX, a four-by-four matrix transpose can be calculated by employing a succession of eight merge instructions as shown in Figure 2 if the source matrix is contained in the MMX register file. For example, to create the first row in the transposed matrix, two 16-bit unpack instructions (*punpckhwd*) are used on the upper halves of the four MMX registers that contain the source matrix, creating the 2x4 matrix on the left of Figure 2. Likewise, two more unpack instructions (*punpcklwd*) are employed to create the 2x4 matrix on the right of Figure 2. These two half matrices form an intermediate matrix that can then be operated on by another four pack instructions (*punpckhdq* and *pupckldq*) to create the transpose of the original 4x4 matrix.

If the inter-word restrictions did not exist, we would be able to fetch any sub-word contained within any of the registers in the register file. The result would be the ability to transform any given column into a row of data in a single cycle, which would allow us to do a matrix transpose in four instructions on the MMX (one instruction for each column). However, since we are restricted to fetching sub-words from two MMX registers, we need to create the intermediate 2x4 matrices. The consequence of the inter-word restriction is that a 4x4 matrix transpose takes eight instructions instead of four. Typically, inter-word restrictions occur in multi-dimensional signal processing that involves matrix manipulations like transposing a matrix or multiplying a matrix with a vector. It is important to note, inter-word restrictions do not appear only on the MMX architecture, but can be found on most sub-word parallel machines. The MMX architecture is simply used as an example of one of the most widely familiar sub-word architectures.

3. THE SPU ARCHITECTURE

We would like to create a mechanism that allows us to address both intra-word and inter-word restrictions without having to issue an explicit permutation instruction. This would in turn allow our media processor to concentrate on processing streaming data on the SIMD computation units.

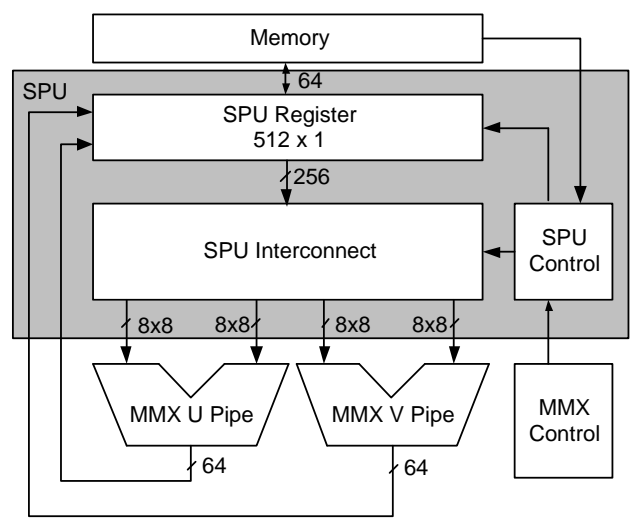


Figure 4: The SPU Block Diagram

However, we want to avoid a radical instruction set change.

The Sub-word Permutation Unit (SPU) is a circuit that controls the access to the SPU register through the SPU interconnect. It allows data permutations in existing software to be executed before the execution of any computational instructions by removing the permutation from the instruction stream and instead having the SPU controller schedule the permutation. This is done by allowing statically scheduled permutations of byte or half-word granularity to be completed on register data before execution in the MMX pipes. The net effect is to make key sub-word permutations transparent architecture while providing the programmer or compiler with first-class access to sub words.

A block diagram of the SPU is shown in the gray box as connected to the MMX architecture. The SPU consists of three major components, the SPU register, the SPU interconnect and the SPU controller. It resides between the memory and computational stages. The SPU register is simply a set of D Flip Flops that are grouped into bytes (the smallest sub-word supported by the MMX architecture). This unified register allows access to all sub-words within the register space of the MMX and eliminates inter-word restrictions. The SPU Interconnect is a full crossbar of byte granularity which connects the bytes from the SPU register to the MMX computational units. The SPU interconnect's purpose is to forward the appropriate sub-words to the ALUs in the correct byte location, thereby eliminating the inter-word restrictions of sub-word parallelism. The SPU controller decides which bytes are transferred from the SPU register to the MMX computational units. On each read of the SPU register, the entire register is read. On writes to the SPU register, only those bits that are overwritten are changed.

The SPU interconnect is a crossbar of byte granularity that allows the functional units to pick any sub-word from the register file and hence it overcomes the inter-word or intra-word restrictions. There is a trade-off between the amount of flexibility in the interconnect and the cost of the interconnect and we will explore this later in the paper.

The SPU controller, also shown in Figure 4 is a dynamically programmed state-machine that generates the required sub-word permutations for static code structures. It selects the desired sub-words for a computation from the SPU register by controlling the configuration of the SPU interconnect.

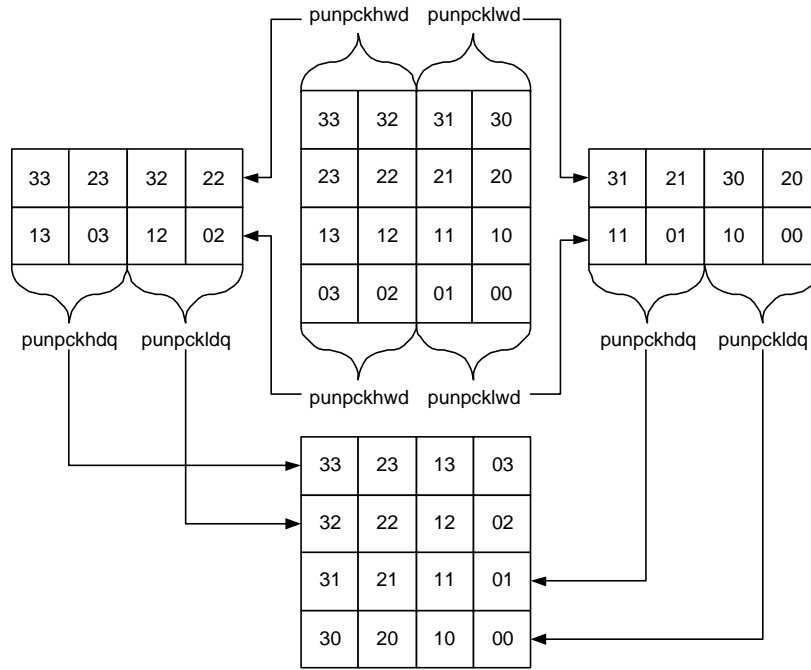


Figure 3: Inter-word restrictions in a 4×4 matrix transpose on the MMX using unpack instructions

Essentially, the SPU interconnect and the controller allows subword-level addressing of the register file. This could be done by adding six additional bits to each instruction to enable subword-level addressing modes and adding additional ports to the register file. But, that would change the instruction set architecture and increase the code size significantly. The innovation here is to achieve the same flexibility with a decoupled controller that is programmed separately, so that the instruction set architecture does not change significantly, but with the limitation that only a finite number of static instructions can be coupled with SPU permutations. Further details of the SPU controller are shown in Figure 8. It is comprised of a K-state state machine, where K is the number of states. This is a parameter that needs to be chosen for a particular micro-architecture based on the size of the core kernels. We fix the value of K at 128 for our experiments. The SPU can support several copies of the SPU control registers, allowing for fast context switching in cases where more than one configuration of the SPU is desired. Of course, more area would be required to support these extra contexts. In addition to the state machine the controller has a pair of counters that keep track of how many dynamic instructions have been executed. These are initialized prior to using the SPU. This allows the SPU controller to support data permutation patterns in up to three *nested loops*. Only the counter specified in each state is used in that particular state. When the specified counter for a particular state reaches zero, the loop of that particular state is exited. Further details of the programming of the SPU controller are shown in the next section. Finally, the SPU has control registers that are memory-mapped, hence the need for a connection to memory as shown in Figure 4. When the SPU is not active, data is transferred to the MMX computational units as it exists in the register file.

4. PROGRAMMING THE SPU

As discussed in the previous section, the SPU has to be programmed to control the SPU interconnect to have the appropriate configuration for media kernels. Figure 6 shows the structure of the SPU's program (the similarity with a horizontal microprogram structure is deliberate). The address space of the SPU is memory mapped as noted in the previous section and it has to be programmed before with the desired communication pattern before executing a computational loop that utilizes the SPU. The CNTRx bit selects one of two counters that will be used for that particular state to support zero overhead looping. The SPU Interconnect field contains the desired byte address from the SPU register for each input operand. The Next State0 field contains the address of the next state to execute in the SPU controller if the counter that is specified in the CNTRx field of that state is zero. Otherwise if the associated CNTRx field of the state is one the Next State1 contains the address of the next state. Not shown in Figure 6 are the two counters and the SPU configuration register. The counters should be initialized with the dynamic instruction count required for the computational loop. State 127 in the SPU controller is a special idle state - when the control reaches this states the SPU is automatically *disabled*. and the counters are reset to their initial values. The configuration register contains a GO bit that activates the SPU when it is written to.

A simple example is presented next to illustrate the programming of the SPU. Let us assume we have sub-words a,b,c,d and e,f,g,h in memory, where each of the sub-word is a 16-bit quantity and we want to compute the products: $a * c, e * g, b * d, f * h$, to realize the dot-product for example.

First we look at how the MMX would implement this dot product, assuming the ordering as shown on the left of Figure 5. After loading these two vectors into MMX registers MM0 and MM1, the data would need to be re-ordered so

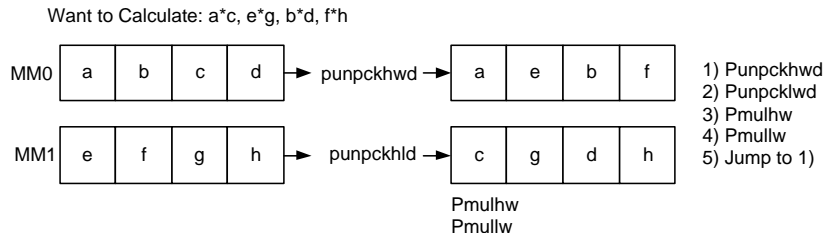


Figure 5: Example SPU operation

CNTRx	Output to SPU		
	Interconnect	Next State0	Next State1
State0	1	192	7
State1			
		⋮	⋮
State127			

Figure 6: The SPU program structure

that the sub-words that we are interested in multiplying together share the same bit alignment in registers MM0 and MM1. To accomplish this reordering, the MMX typically employs the packing instructions (instructions 1 and 2) as shown in Figure 5. After the re-ordering is complete, we can then proceed to multiply the sub-words (instructions 3 and 4). The pseudo-MMX assembly code for this would look like:

```

Loop:  punpckhwd
      punpcklwd
      pmulhw
      pmullw
      jump Loop

```

Utilizing the SPU, we can complete the two packing instructions by modifying the SPU interconnect to fetch and align the sub-words needed for each multiply instruction implicitly along with the multiply instructions. The result would be to remove the two permutations from the loop by having the SPU execute the permutation instructions transparently. The result would transform the five pseudo-code instructions from above into the following three pseudo-code instructions:

```

Loop:  pmulhw
      pmullw
      jump Loop

```

Shown in Figure 7 is the setup in the SPU controller that is needed to support the data patterns as required by the unpack instructions that were removed in our dot product example. Let us assume this loop needs to be executed 10 times. We would need to pick a loop counter and setup a loop counter to be equal to 10 times the number of static instructions inside the loop. For the example shown in Figure 7, we would setup CNTR0 to be equal to $10 * 3 = 30$

CNTRx	Output to SPU	NextState0	NextState1
state0	0 Byte position in SPU register of a,e,c,g	127	1
state1	0 Byte position in SPU register of b,f,d,h	127	2
state2	0 straight	127	0

1) pmulhw
2) pmullw
3) jump 1)

Figure 7: SPU register usage

(the total number of dynamic instructions executed in our loop). The NextState0 field in Figure 7 shows that the exit state is the IDLE state, which puts the SPU in sleep mode and reinitializes CNTR0 to 30.

To utilize the zero overhead loop capability of the SPU, we simply specify the CNTR value for each instruction that is associated with each individual loop. In the case of our dot product example we have set CNTR0 to our dynamic instruction count, and so each of the CNTRx entries for this example would contain a "0". Since the SPU automatically restores the CNTR value to its original programmed state after reaching zero, the SPU has zero overhead for maintaining up to 2 nested loops.

As we can see, the generation of the code for the SPU is systematic and can be automated. Additionally, a separate instruction set extension could be mapped to the SPU controller freeing the programmer from having to micro-code this engine. The startup cost of programming the SPU needs to also be considered carefully by either the programmer or a compiler. However, for the media applications where the workloads are well defined at compilation time, the startup cost should be easily scheduled. With the combination of the regularity of media applications and the ability to load multiple contexts into the SPU, the startup costs should be easily manageable. Finally, on an exception, we can either ensure that the exception handler disables the SPU by writing to the SPU control register, or switches to a free context of the SPU.

5. SPU EVALUATION

In this section, we present a micro-architectural and system-level evaluation of the SPU mechanism. We present area and delay estimates based upon a VHDL implementation and present performance results in the context of the MMX architecture. The findings show that the SPU derives significant performance benefits from all three of its design objectives: intra-word permutation, intra-word permutations, and decoupled control.

5.1 SPU Micro-architectural Evaluation

5.1.1 Intra-word speedups

The SPU was modeled in VHDL and simulated for functional correctness. The control logic shown in Figure 8 was

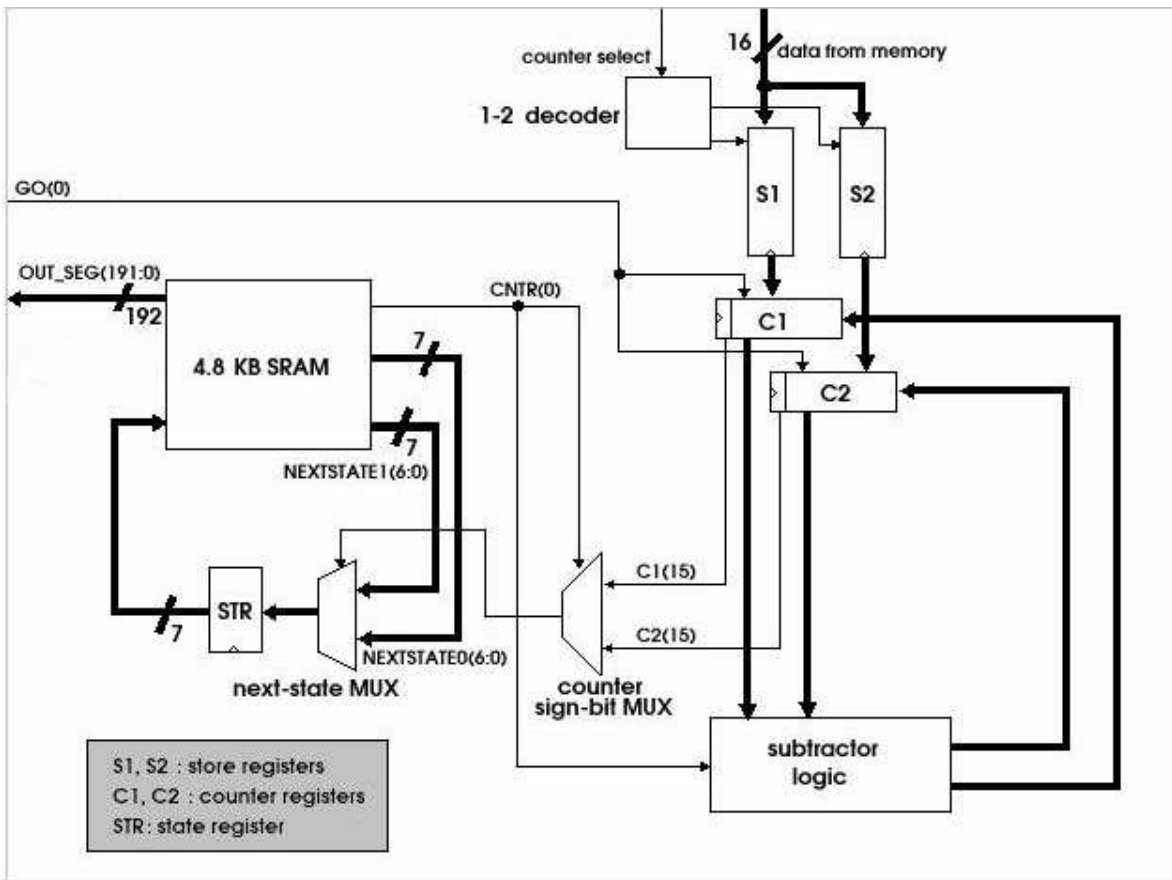


Figure 8: Shows the block diagram of the SPU controller modeled in VHDL. This controller supports full byte addressability of the SPU register.

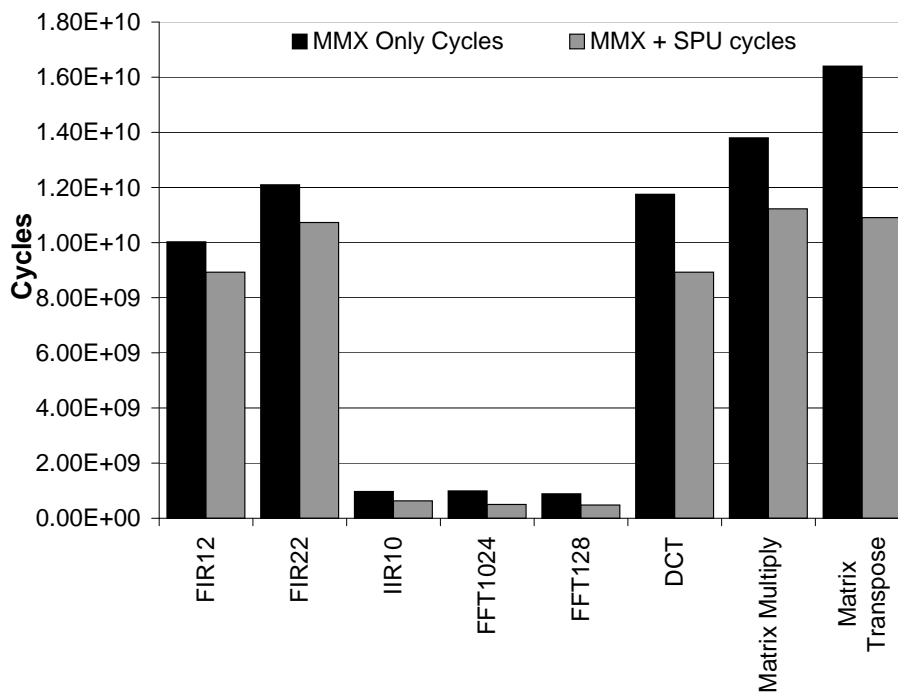


Figure 9: Cycles Executed on MMX and the MMX+SPU using the Intel IPP media routines. The dark bars show the MMX cycle count, and the gray bars show the MMX+SPU cycle count

generated from the VHDL description. Our results indicate that the SPU can be implemented with less than 1% area overhead in a 0.18μ $106mm^2$ Pentium III processor [1]. Additional contexts of the SPU control registers would cost additional area.

The performance and the area of the SPU are dominated by the size of the interconnect and the control memory. The area and delay for the memory and interconnect are estimated from the implementation and layout of the Princeton VSP (Video Signal processor) project described in [23, 3]. The Princeton work uses *folded* crossbar structures to optimize the layout area which is particularly applicable to the SPU interconnect. *The technology used in the Princeton work is 0.25um CMOS with 2 metal layers for routing.* Table 1 describes the area and delay trade-offs for four different configurations of the SPU. Note that the amount of control memory depends on the size of the crossbar used which in turn dictates the amount of *flexibility* in terms of the amount of inter-word parallelism that can be explored and the *granularity* in terms of the size of the data operands. The control memory size in our implementation is given by a simple formula $128*(15+K)$ where K is the number of addressable location and 128 is the number of states assumed in the controller. Configuration A represents full byte-level flexibility, which means any byte in the register file can be used in the computation. This will eliminate *all* inter-word and intra-word restrictions and make the sub-word parallelism fully orthogonal. Typically, full byte-level flexibility is not needed, and a restricted version of the crossbar can be used with corresponding benefits in area and delay. All the applications used in this paper can be realized with configuration D that utilizes a 16×16 crossbar, which has a very modest area requirement of 2.86 square mm in 0.25um 2 layer metal process.

The total latency of the SPU is dictated by the latency of the crossbar and the control memory. These results could likely be significantly improved by optimizing the design for a specific processor by suitable transistor sizing and of course using more than two metal layers, as the crossbar design is dominated by wiring.

Taken in the context of the $106mm^2$ Pentium III die, and scaling to $.18\mu$ with 6-layers of metal, we expect the SPU can be implemented with less than 1% area overhead. To accommodate SPU latency without adversely affecting processor clock cycle, we propose to add a pipeline stage to the MMX for data motion within the SPU, if necessary. For other implementations of the SPU and other architectures, the granularity of the SPU interconnect can be traded for cycle time as needed. Additionally, for modern designs, additional pipelining may be necessary to ensure that the SPU's interconnect meets clock cycle requirements.

However, further pipelining the MMX should not degrade the performance on media benchmarks by very much. First, control dependencies are calculated in the scalar pipeline, not in the MMX. Lengthening the MMX pipeline will not increase branch mis-prediction penalties. Second, digital signal processing and video applications that are the focus of this paper (those that are candidates for exploiting sub-word parallelism) contain few conditional branch instructions [8] and exhibit large amount of data parallelism and therefore very few branch mis-predictions. The impact of additional pipelining may be more acutely felt for general purpose programs which are not examined in this paper, but should

certainly be considered depending upon your application.

We extracted the branch statistics of various programs used in our study using the VTune performance analysis tool from Intel [9]. These are shown in Table 2. As the branch statistics indicate, an additional pipeline stage is unlikely to be detrimental to the overall performance. If a single extra cycle penalty is added for each branch mis-predict, our results are essentially the same due to the low frequency of branch mis-predictions for media algorithms. The impact of the addition of an extra pipe stage to accommodate the SPU on another media architecture would have to be weighed on a case-by-case basis. However, due to the workloads of typical media processors, we believe similar results would be found.

5.2 SPU Performance

In this section, we evaluate the system-level performance impact of the SPU on a Pentium III with MMX. We describe our experimental methodology and present a breakdown of our performance results.

5.2.1 Methodology

The evaluation of the benefits of SPU had to be done carefully. One could write poor code that has many unnecessary data movements, which could give us a false estimate for the benefits of the SPU. So, we chose benchmarks that satisfied the following two requirements - (a) they should be highly optimized for the MMX architecture and (b) they should be from an independent source, so that the quality of the code and the fractions of MMX usage were not within our control. We used the The Intel Integrated Performance Primitives (IPP) [10] for our experiments. These are highly optimized libraries from Intel that have been hand tuned for the highest performance utilizing the MMX. Improvements shown in this paper are for code that has already been optimized without the knowledge of an existing SPU. This does introduce an interesting question, can the code be optimized further if the programmer or the compiler had the knowledge of the SPU. We believe that is possible, but do not address at this time.

The IPP consists of three types of algorithms, signal processing, image processing and matrix algebra. First, each of the benchmark is executed on the MMX unit of a Pentium III processor. Then, using a performance analysis tool developed by Intel called Vtune [9], we extract statistical information about the run-time performance of each algorithm. In particular, we can see what percentage of each algorithm's operations are MMX instructions, and what percentage of each algorithm's operations were packing or permutation instructions that are required for sub-word realignment. Then, each of the algorithms is re-coded to avoid utilizing the permutation instructions that can be addressed by the SPU unit. The algorithms are then re-run with the permutations instructions replaced with implicit SPU instructions and the performance is cataloged. The code is assumed to reside in L1 cache for all the experiments. Only a single context SPU was utilized in this study. It is assumed that most vector architectures are in-order machines, as out-of-order execution would not improve ILP beyond vectorization.

5.2.2 Performance Overview

Figure 9 summarizes the results from this study. Overall, speedups resulting from the SPU range from 4-20%. The figure shows both the performance of the MMX and the

SPU Configuration	Interconnect Area (mm^2)	Interconnect Delay (ns)	Control Memory Size(mm^2)	Description
A	8.14	3.14	1.35	64x32 crossbar with 8-bit ports
B	4.07	2.29	1.1	32x32 crossbar with 8-bit ports
C	4.72	1.95	0.6	32x16 crossbar with 16-bit ports
D	2.36	0.95	0.5	16 x16 crossbar with 16-bit ports

Table 1: Delay and Area for four possible SPU configuration in 0.25um 2 metal CMOS

Media Algorithm	Clocks Executed	Branches	Missed Branches	Missed Branches %	Benchmark Description
FIR12	1.51E+10	2.56E+09	1.43E+07	0.094%	12 TAP, 150 Sample blocks
FIR22	2.13E+10	2.05E+09	1.00E+07	0.046%	22 TAP, 150 Sample blocks
IIR	1.45E+10	8.98E+08	1.11E+07	0.076%	10 TAP, 150 Sample blocks
FFT1024	1.27E+10	4.19E+08	8.42E+06	0.066%	1024 Sample, Radix 2 Real FFT
FFT128	1.19E+10	7.41E+08	1.87E+07	0.157%	128 Sample, Radix 2 Real FFT
DCT	1.69E+10	2.75E+08	1.84E+04	0.000%	8x8 Kernel
Matrix Multiply	1.78E+10	3.53E+08	2.24E+04	0.000%	16x16 16b Matrix Multiply
Matrix Transpose	1.88E+10	1.57E+09	7.73E+06	0.041%	16x16 Matrix Transpose, 16-bits

Table 2: Branch Statistics for Some Media Algorithms on the MMX, showing the percentage of missed branches on the MMX architecture

MMX augmented with the SPU on the IPP media routines. The results include the overhead of the additional pipeline cycle needed for the SPU interconnect. The hashed portions of each bar indicate the percentage of execution cycles that the MMX engine is executing. As we can see, neither the FFT or IIR filter routines from the IPP package utilize the MMX efficiently. In these instanced, the SPU obviously does not impact the performance on these routines. The FIR filters for the MMX try to avoid many sub-word permutes that would normally be required by having multiple copies of the filter coefficients in the MMX registers where each copy of coefficients are offset by one sub word. This allows for the small range of sub-word permutations required by the FIR to be handled at the expense of register file pressure and additional memory requirements. Because of this effect, the SPU gives only a small eight percent speedup to MMX. However, if the code was reworked with the SPU in mind, register file pressure could be improved. In general it should also be noted the code that was used for this study was highly optimized code given the MMX architecture, and not necessarily the optimal code for an MMX that has been augmented with the SPU. Therefore, it is our belief that the improvements seen here represent a lower estimate of the true performance advantages of the SPU.

Most of the algorithms tested show intra-word restrictions which are addressed by the flexible interconnect. Intra-word restrictions impact code performance in algorithms for a few reasons. Some algorithm's basic data block length simply does not match the computational width of the processor, thus requiring some sub-word movement to support efficient usage of the sub-word computational blocks. Other algorithms, like the FIR filter, can handle the bulk of computations without sub-word restrictions, but have sub-word processing such as shifting the delay line by a single sample. One interesting pattern that has emerged from this study is that intra-word restrictions appear to account for less than 10% of cycles lost for most media applications. The low intra-word speedup is due to the fact that most media applications contain enough parallelism so they may

Media Algorithm	Cycles Overlapped	% MMX Instr	Total Instr
FIR12	1.12E+09	11.20%	07.42%
FIR22	1.38E+09	11.40%	06.48%
IIR	9.11E+08	93.63%	06.28%
FFT1024	4.98E+08	50.30%	03.92%
FFT128	4.26E+08	48.08%	03.58%
DCT	2.83E+09	23.98%	16.75%
Matrix Multiply	2.58E+09	18.70%	14.49%
Matrix Transpose	3.33E+09	20.12%	17.55%

Table 3: Cycles overlapped through decoupled control.

be parallelized sufficiently to avoid some intra-word effects. One technique to avoid intra-word restrictions is to make duplicate, sub-word shifted copies of data in the register file to avoid having to perform software permutations. This is done in the IPP FIR filter code, however the replication of coefficients in the register file would be unnecessary with a SPU-enabled MMX. As a result, less pressure would be placed on the register file.

5.2.3 Inter-word speedups

Inter-word restrictions appear in primarily matrix or imaging applications. Additionally, the next generation of communications applications utilizing smart antenna arrays may also require multi dimensional filtering. The unified SPU register allows most inter-word restrictions to be eliminated. By removing the intra-word restrictions, the speedups are quite a bit more impressive, as shown by the DCT, matrix multiply and matrix transpose kernels.

5.2.4 Overlap through decoupled control

Table 3 illustrates the significant benefits of the decoupled SPU controller. Between 11% and 93% of MMX permutation instructions are off-loaded to the SPU controller. This results in a total instruction savings between 3.58% and 17.55%, accounting for a substantial portion of our overall

speedups from Figure 9. In general, we believe that this decoupled control is an important mechanism for enabling efficient sub-word parallelism for relatively little expense.

6. DISCUSSION

Most SIMD machines support a large variety of permutation instructions, and some architectures can even efficiently schedule permutations in parallel with the computational stream in DSP applications. Such support, however, comes at significant cost and suffers from inter-word restrictions.

For example, the AltiVec vector media co-processor for the PowerPC architecture has a permutation unit that is one of four integer pipelines that are fed by a dynamic dispatch unit [12]. The AltiVec's permutation unit can generate any permutation of data from two input registers of up to byte wide granularity and can be scheduled quite well for most media codes even without aggressive software pipelining. Although this solution provides high performance, it comes at significant cost and complexity. First, most architectures in the DSP domain exploit the regularity of applications by avoiding the complexity and power consumption of dynamic scheduling. Second, the permutation unit can only access a small number of registers at a time, resulting in serious limitations on inter-word permutations such as those we found useful for matrix and DCT applications.

The Texas Instruments C64x VLIW DSP has two of eight execution units that handle a subset of permutations [20]. However, using aggressive software pipelining and intelligent data alignment, most permutations can be executed in parallel with the other six computational units for most algorithms. Both the AltiVec and C64x implementations require the permutations to be included in the software as separate explicit instructions, resulting in greater demands on instruction bandwidth and cache. Another DSP architecture, Analog Device's eight-wide SIMD static super-scalar TigerSHARC DSP [5] supports a small sub-set of permutation directly in each computation, thus hiding the need for the majority of explicit permutation instructions. This solution is appealing, but limited in both inter-word and intra-word capabilities.

One reason we chose the MMX (other than its wide acceptance) for our study was that it provides a clean basis for evaluating the SPU mechanism. Qualitatively, the SPU is a good design alternative to existing DSP mechanisms, enabling more general sub-word orchestration with low complexity and cost. Quantitatively, it is difficult to remove such mechanisms from existing designs in order to provide an accurate numerical comparison. The MMX architecture provided a clean platform for a quantitative analysis.

We note that the MMX architecture works with a relatively small register set when compared to architectures such as the AltiVec. Providing general inter-word permutations across a large register set would require the SPU to have significantly more interconnect and register bandwidth. Design trade-offs would include restricting permutations to a subset of registers, pipelining the SPU interconnect into multiple cycles, and using a multi-stage interconnect instead of a crossbar. Although challenging, we believe that the SPU design can be scaled to large register sets and provide significant performance and efficiency advantages, especially in the coming era of up to a billion transistors on a chip.

As a last point, the SPU implemented in this study is relatively simple, allowing only equal sub-word access to all

sub-words. However, additional modes could be added to the SPU, like sign extension, negation, or even more complex operations.

7. RELATED WORK

The predominant solution to overcome the restrictions of subword parallelism is to provide special instructions like permute, mix, pack, unpack and variations of it as described in [15, 19]. Modern VLIW and super-scalar processors have found a way to hide this overhead by providing a special functional unit that can perform *data rearrangement* in parallel with regular computation. This is often accomplished by loop unrolling and scheduling the data reordering of one iteration with the computation of a previous iteration. Examples of such processors include the AltiVec co-processor of PowerPC [12] and the flagship Texas Instruments DSP based on the VelociTI architecture [20]. So, in effect the prevalent solution is to perform data orchestration in *software* with additional instructions, which obviously increases the code size and wastes expensive resources on the processor like the instruction fetch and decode mechanism. This is especially detrimental (as our results indicate) when the programs have an abundance of inter-word restrictions like DCT which is a critical kernel in many multimedia and compression applications.

Hardware solutions for parallel data orchestration seem attractive. In fact, the earliest solution to this problem can be found in the Burroughs Scientific Processor [13], where two full crossbars are provided between the memory and the functional units for data alignment. A restricted version of this was proposed in the Analog Devices' flagship TigerSHARC DSP in the form of a data alignment buffer [4]. Similar data alignment problems were addressed in massively parallel SIMD computers such as the CM2 [11]. The work in the area of hardware support for memory access reordering such as [17] is also similar though it is not just restricted to overcome subword parallelism.

The key innovation of this paper is to provide a hardware mechanism for addressing the data alignment problem (to overcome both inter-word and intra-word restrictions) between the register file and the functional units, without having to redesign the instruction-set architecture of existing machines. Furthermore, we show how this can be integrated within a execution pipeline in a complex processor state-of-the-art processor such as the Pentium with MMX. This is accomplished by decoupling the data orchestration from the instruction execution with a specialized low overhead controller.

The SPU solution can also be viewed as a form of data forwarding unit that is programmer controlled. The programmer can make the data appears in front of the functional units in the correct order by configuring the SPU controller appropriately. In this sense the concept of SPU reinforces the key insight behind the recent work in MIT [22] and Stanford [16] in supporting data communication *explicitly*. The SPU makes sub-word data re-ordering explicit and visible to the compiler yet invisible to the execution stream. The work reported in [21] shares the same motivation as ours but they focus more on reducing the loop overhead with a hardware programmable loop engine. They mention data alignment but do not present any details or implementation results.

8. CONCLUSION

Sub-word orchestration will become increasingly important in future microprocessors as media applications demand higher data rates. Media processors, due to the regular nature of media applications, will continue to answer this challenge with larger amounts of parallel resources, increasingly relying on sub-word parallelism.

The SPU demonstrates that a carefully placed interconnect can significantly accelerate signal processing applications with minimal overhead by removing some of the restrictions of sub-word parallelism. In particular, the SPU achieves high performance through an efficient, decoupled controller to support both inter-word and intra-word sub-word permutations. The SPU also pipelines data orchestration latency by exploiting the regular behavior of signal processing applications. We believe that the SPU is an elegant solution to the sub-word orchestration problem and represents an attractive alternative to more ad-hoc methods currently in practice.

9. ACKNOWLEDGMENTS

This work is supported by NSF ITR grants 0312837 and 0113418. Erik Czernikowski contributed to the VHDL effort.

10. REFERENCES

- [1] Virtual press kit: Intel Pentium 4 processor. http://www.intel.com/pressroom/archive/photos/p4_photos.htm.
- [2] K. Diefendorff and P. Dubey. How multimedia workloads will change processor design. *IEEE Computer*, 30(9):43–45, sept 1997.
- [3] S. Dutta, K. Connor, W. Wolf, and A. Wolfe. A Design Study of a 0.25 μ m Video Signal Processor. *IEEE Transactions on Circuits and Systems for Video Technology*, 8:501–519, august 1998.
- [4] J. Fridman. Subword parallelism in digital signal processing. *IEEE Signal Processing Magazine*, 17(2):27–35, march 2000.
- [5] J. Fridman and Z. Greenfield. The TigerSHARC DSP Architecture. *IEEE Micro*, pages 66–76, 2000.
- [6] S. R. Gerrit Slavenburg and H. Dijkstra. The TriMedia TM-1 PCI VLIW Media Processor. In *Proceedings of the HotChips 8: A Symposium on High Performance Chips*, august 1996.
- [7] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*, 2002.
- [8] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*, 2002. Figure 2.37, page 142, Third Edition.
- [9] V. Intel. Vtune performance analyzers. <http://www.intel.com/software/prodcuts/vtune/>.
- [10] IPP Intel. Intel Integrated Performance Primitives for Intel Pentium Processors and Intel Itanium Architectures. <http://www.intel.com/software/prodcuts/ipp/ipp30/>.
- [11] S. L. Johnsson and C.-T. Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Transactions on Computers*, 38(9):1249–1268, September 1989.
- [12] P. D. Keith Diefendorff, R. Hochsprung, and H. Scales. AltiVec extension to powerpc accelerates media processing. *IEEE Micro*, pages 85–96, march 2000.
- [13] D. J. Kuck and R. A. Stokes. The Burroughs Scientific Processor (BSP). *IEEE Transaction on Computers*, 31:363–376, may 1982.
- [14] R. B. Lee. Subword parallelism with MAX-2 — accelerating media processing with a minimal set of instruction extensions supporting efficient subword parallelism. *IEEE Micro*, 16(4):51–59, 1996.
- [15] R. B. Lee. Multimedia extensions for general-purpose processors. In *IEEE Workshop on Signal Processing Systems*, pages 9–23, november 1997.
- [16] P. Mattson, W. Dally, S. Rixner, and J. Owens. Communication Scheduling. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, november 2000.
- [17] S. A. McKee, A. Aluwihare, B. H. Clark, R. H. Klenke, T. C. Landon, C. W. Oliver, M. H. Salinas, A. E. Szymkowiak, K. L. Wright, W. A. Wulf, and J. H. Aylor. Design and evaluation of dynamic access ordering hardware. In *International Conference on Supercomputing*, pages 125–132, 1996.
- [18] D. O. Michael Kagan, Simcha Gochman and D. Lin. MMX microarchitecture of Pentium processors with MMX technology and Pentium II microprocessors. (Q3):8, 1997.
- [19] A. Peleg and U. Weiser. MMX technology extension to Intel architecture. *IEEE Micro*, 16(4):42–50, 1996.
- [20] N. Seshan. High Velocity Processing. *IEEE Signal Processing Magazine*, pages 86–101, march 1998.
- [21] D. Talla. Architectural techniques to accelerate multimedia applications on general-purpose processors, 2001.
- [22] M. Taylor, W. Lee, S. Amarsinghe, and A. Agarwal. Scalar operand network: On-chip interconnect for ilp in partitioned architectures. In *HPCA*, february 2003.
- [23] A. Wolfe, J. Fritts, S. Dutta, and E. Fernandes. Datapath Design for a VLIW Signal Processor. In *Proceedings of HPCA-3, 1997*, february 1997.
- [24] W. Wulf. *Compilers and Computer Architecture*. *IEEE Computers*, pages 41–48, July 1981.