# A New Look at Exploiting Data Parallelism in Embedded Systems

Hillery C. Hunter
Center for Reliable and High-Performance Computing
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
hhunter@crhc.uiuc.edu

Jaime H. Moreno
IBM Research Division
T.J. Watson Research Center
Yorktown Heights, NY
jhmoreno@us.ibm.com

## ABSTRACT

This paper describes and evaluates three architectural methods for accomplishing data parallel computation in a programmable embedded system. Comparisons are made between the well-studied *Very Long Instruction Word (VLIW)* and *Single Instruction Multiple Packed Data (*SIM $_p$D*)* paradigms; the less-common *Single Instruction Multiple Disjoint Data (*SIM $_d$D*)* architecture is described and evaluated. A taxonomy is defined for data-level parallel architectures, and patterns of data access for parallel computation are studied, with measurements presented for over 40 essential telecommunication and media kernels. While some algorithms exhibit data-level parallelism suited to packed vector computation, it is shown that other kernels are most efficiently scheduled with more flexible vector models. This motivates exploration of non-traditional processor architectures for the embedded domain.

## Categories and Subject Descriptors

C.1.1 [**Processor Architectures**]: Single Data Stream Architectures; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures; C.1.3 [**Processor Architectures**]: Other Architecture Styles; C.3 [**Processor Architectures**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*; C.4 [**Processor Architectures**]: Performance of systems—*Design studies*

## General Terms

Design

## Keywords

Data-Level Parallelism, DLP, ILP, SIMD, Sub-word Parallelism, VLIW, Embedded, Processor, DSP, Telecommunications, Media, Architecture


*CASES'03,* Oct. 30–Nov. 1, 2003, San Jose, California, USA.


## 1. INTRODUCTION

Demand for personal device functionality and performance has made the ability to perform multiple computations per cycle essential in the digital signal processing (DSP) and embedded domains. These computations can be gleaned from multiple forms of program parallelism. *Instruction-level parallelism* (ILP) occurs at the operation level when two or more operations are data-independent from one another and may be executed concurrently. *Data-level parallelism* (DLP) occurs when the same operation is executed on each member of a set of data. When the elements of data parallel computations are narrower than a standard data width (*e.g.* 8 instead of 32 bits), *sub-word parallelism* (SWP) is present. *Superscalar* machines detect instruction parallelism in hardware, but other approaches require ILP, DLP, and SWP to be explicitly exposed by a compiler or programmer.

Broad application parallelism categories exist (*e.g.* numerical applications are highly data parallel and control applications have little ILP), but most complete applications contain both instruction-level and data parallelism. In the embedded domain, repetition of computations across symbols and pixels results in particularly high amounts of data parallelism. This paper is motivated by the breadth of DSP and embedded architectures currently available for *2.5G* and *Third-Generation* (3G) wireless systems, so examples and analysis will focus on telecommunication and media kernels.

This paper qualitatively and quantitatively analyzes methods for performing data parallel computation. Three primary architecture styles are evaluated: *Very Long Instruction Word* (VLIW), *Single Instruction Multiple Packed Data* (SIM $_p$D), and *Single Instruction Multiple Disjoint Data* (SIM $_d$D). Section 3 presents a taxonomy of architectures and Section 4 describes implementations of these architectures. Comparisons in Section 5 include ease of implementation and programming; performance; and the ability to match inherent algorithmic patterns. In Section 6, kernels are grouped according to data access patterns, and the amount of irregular patterns is quantified. The results of this analysis motivate definition and exploration of alternative forms of data access.

## 2. RELATED WORK

In the uni-processor domain, previous research on data parallel computation deals primarily with architectures using *packed* vector data, meaning that multiple data elements are joined together into a single register. In the literature,

this is commonly referred to simply as SIMD – Single Instruction Multiple Data. For each commercial SIMD extension, there are many papers which evaluate speedups realized for particular domains. A thorough survey of these performance studies on the Sun VIS, Intel MMX and SSE, and IBM PowerPC AltiVec extensions is given in [1]. Most studies focus on general-purpose video and matrix kernels and applications. In addition, [2] compares TI C62x VLIW performance to signal processing and multimedia computation on a Pentium II with and without MMX support; and [3] compares execution of EEMBC telecommunication and consumer benchmarks on a proposed vector processor (VIRAM) to commercial VLIW and superscalar implementations.

Aside from performance studies, there are several bodies of work related to overcoming bottlenecks within the packed vector format. Proposals to increase multimedia performance include: the *MediaBreeze* architecture which improves address generation, loop handling, and data reordering [4]; the *MOM* (Matrix Oriented Multimedia) ISA extension for performing matrix operations on two-dimensional data; and use of in-order processors with long vectors instead of superscalar architectures [1].

Data alignment requirements of the packed vector data paradigm have also received some attention in the form of alternate permutation (data re-arrangement) networks and instruction set modifications. Yang *et al.* discuss the need for flexible permutation instructions in packed vector architectures, and propose instructions which use a butterfly permutation network to provide rearrangement of subword data elements [1].

Little work has considered the extent to which media algorithms are truly or only partially suited to packed vector computation. Bronson's work [5] was evaluated in a multi-processor context, but its premise is that there are algorithms which do not map fully to a SIMD computational model, and which benefit from a mixed SIMD/MIMD (Multiple Instruction Multiple Data) system. Faraboschi *et al.* [6] advocate a mixed VLIW/SIMD system in which SIMD components are programmed by hand, and ILP is scheduled on VLIW units by the compiler. They discuss tradeoffs between VLIW and packed vector architectures in terms of implementation and code generation.

Previous work assumes a packed vector approach is the only vector method for handling DLP on an embedded system. Algorithmic focus has been on video kernels, which are generally assumed to fit well into a packed vector model. This work provides a new taxonomy which includes the SIM$_d$D architecture style (Section 3); extends previous analyses to cover telecommunication kernels; and quantifies suitability to various data-level parallel architecture styles (Section 6).

# 3. TAXONOMY

In a *load-store* architecture, computation operations use only register values as sources and destinations, and `load` and `store` operations specify transfers of data between registers and memory. This paradigm, however, leaves room for architectural design of register usage and the memory interface. The following taxonomy of instruction composition and register specification presents a unique perspective on data access.

If instructions are viewed as the controllers of data usage and production, the most general architectural model will allow for simultaneous execution of an arbitrary combination of instructions and a similarly flexible use of data registers. However, to produce useful work, instructions will inherently have some degree of dependence on one another, and must therefore be sequenced or combined as dictated by program control flow. In contrast, data access is not generally restricted by program flow, but by hardware simplifications made at the time of architecture design.

In [7], Stokes illustrates a *Single Instruction Single Data* (SISD) architecture as consisting of a single sequential instruction stream which operates on a single data (register) stream, and produces a single output stream. An adaptation of Stokes' one-wide, in-order processor representation is shown in Figure 1(a).

If a statically-scheduled architecture allows simultaneous and different processing of multiple independent data (register) streams, the result is a *Very Long Instruction Word* (VLIW) architecture. The instruction stream in Figure 1(b) is a combination of multiple operations, grouped into an instruction word which specifies multiple computations. Individual operations may use any registers, with the restriction that their destinations must generally be unique so as to avoid write-back ambiguity within a single cycle of execution. This is the most general of the architecture types discussed in this paper.

If the VLIW instruction stream is modified to consist of single, sequential operations, the result is a Single Instruction Multiple Data (SIMD) architecture. Here single operations are specified in the instruction stream, but each specified operation is performed on multiple data elements. For the general SIMD case, these data elements may come from, and be written back to, disjoint locations. This separation of input and output data streams will be indicated by the notation SIM$_d$D (Single Instruction Multiple Disjoint Data), and is pictured in Figure 1(c).

Current SIMD implementations, however, do not allow this level of data flexibility. Instead, multiple data elements for SIMD operations are packed into a single register, often called a *SIMD vector register*. Each instruction causes an operation to be performed on all elements in its source registers, and there is only one input data stream. This is depicted in Figure 1(d), and will be discussed in further detail in Section 4.2 as SIM$_p$D (Single Instruction Multiple Packed Data).

When implemented as pictured in Figure 1(c), there are many similarities between the SIM$_d$D and VLIW architectures. An alternate implementation, *indirect*-SIM$_d$D, pictured in Figure 2, provides access to disjoint data values via *vector pointers*. Rather than explicitly specifying vector elements in the SIM$_d$D instruction word, indirect-SIM$_d$D instructions have vector pointer source and destination fields, wherein each vector pointer specifies multiple indices. Data elements are accessed indirectly through statically specified pointers, and physical vectors are composed at execution-time based upon dynamic values in vector pointer registers.

Table 1 enumerates the architectural design space spanned by the variables *Operations per Instruction*, *Data Count*, and *Data Flexibility*. These variables are binary: instruction words may contain *one* or *multiple* operations, which operate on *one* or *multiple* data inputs either independent of one another (*disjoint*), or *joined* in a register or other archi-

Table 1: Taxonomy of access to data values for statically-scheduled architectures.

| # | Operations per Instruction | Data Count | Data Flexibility | Architectural Style |
|---|---|---|---|---|
| 1 | One | One | Disjoint | SISD (Figure 1(a)) |
| 2 | One | One | Joined | n/a |
| 3 | One | Multiple | Disjoint | Flexible SIMD (*e.g.* SIM $_d$D) (Figure 1(c)) |
| 4 | One | Multiple | Joined | SIM $_p$D (Figure 1(d)) |
| 5 | Multiple | One | Disjoint | Compound operations (CISC) |
| 6 | Multiple | One | Joined | n/a |
| 7 | Multiple | Multiple | Disjoint | VLIW (Figure 1(b)) |
| 8 | Multiple | Multiple | Joined | SIM $_p$D across multiple processors/clusters |

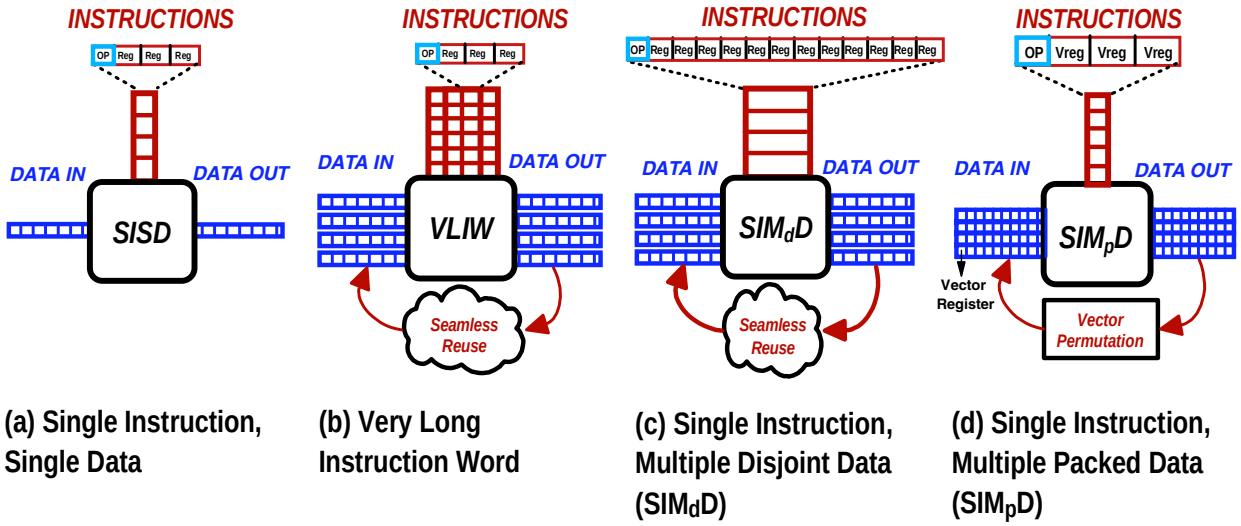



Figure 1: Taxonomy of architectures.

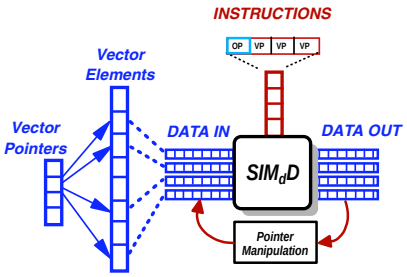



Figure 2: Indirect-SIM $_d$D implementation using vector pointers.

tecture construct. Styles 1, 3, 4, and 7 have been described above; styles 2 and 6 are not meaningful since a "single" datum cannot be "joined" to others. The closest technique to 5 is microcode created to perform multiple operations on a single data stream in *Complex Instruction Set Computers* (CISC). Architecture 8, SIM $_p$D across multiple processors, is indicative of the fact that at higher dimensions, the possibilities for combination of architectural techniques are numerous: homogeneous or heterogeneous combinations of VLIW, SIM $_p$D, and SIM $_d$D may be formed across clusters or processors.

The focus of this paper will be VLIW (style 7), general and indirect-SIM $_d$D (style 3), and SIM $_p$D (style 4) architectures. Characteristics and commercial implementations of each of these styles are discussed in the following section. Analysis of kernel access patterns and their correspondence to these architectures is presented in Section 6, motivating exploration of new architecture design space. Vector (or array) processors in which vector elements are processed sequentially through a pipeline typically require a different programming model, so are not included in this discussion.

# 4. IMPLEMENTATIONS OF DATA PARALLELISM

## 4.1 VLIW architectures

A statically-scheduled architecture with issue width greater than one operation per cycle is the simplest extension to traditional processors which issue one instruction per cycle. The power and performance advantages of this technique have motivated the adoption of VLIW techniques for signal processing applications. Available products include those from Philips [8], Infineon [9], Texas Instruments (TI C6x), and StarCore [10].

VLIW instruction words specify multiple operations on different source data, and thus inherently represent *MIMD* (Multiple Instruction Multiple Data) computation. This is illustrated at the top of Figure 1(b), where the sample instruction contains four opcodes (OP), and three registers

for each operation (one destination and two sources). Data parallelism is accomplished in the VLIW paradigm by specifying separate execution of the same operation on multiple data items, so the same opcode must be duplicated for each operation.

Since implementations may differ significantly, for the purposes of comparison with other architectures, the term *VLIW* will be used in the following sections to refer to a statically-scheduled 32-bit load-store architecture with an instruction issue width of four. Function unit mix is assumed to be uniform across the issue slots, including availability of four memory ports.

## 4.2 SIM$_p$D architectures

The first SIMD machine was the ILLIAC IV [11], with sixty-four 64-bit processors operating in parallel. Following the ILLIAC IV were commercial SIMD machines produced by ICL (Distributed Array Processor, DAP) and Goodyear (Massively Parallel Processor, MAP) [12]. These machines were used for scientific, military, and air traffic control problems which required high computational throughput. SIMD entered the desktop processing domain in the 1990s in order to accelerate graphics and gaming processes. Today, SIMD is finding new applications as current embedded performance falls short of emerging 3G wireless and consumer media requirements.

Support for single instruction, multiple data computation is most commonly provided in the form of special instructions which operate on wide, multi-element *vector* registers (shown in Figure 1(d)). For the generic case, the SIM$_p$D vector file consists of $r$ registers of $n$ *elements*, where each element is of size $b$ bits. Instructions require $log_2(r)$ bits to specify each source and destination register. When an instruction is issued, the operation is performed on all $n$ elements in the SIM$_p$D vector register. This imposes strict placement requirements for the elements in the registers, since every element needed for a data-parallel computation must reside in the same vector register.

For purposes of comparison with other architectures, this paper will assume a SIM$_p$D approach in which $n = 4$ and $b = 32$. This eliminates the need for discussion of special-purpose instructions used for computation on 8- and 16-bit quantities.

**General-purpose SIM$_p$D extensions**

As mentioned, most general-purpose architectures currently incorporate some form of SIMD to support high-speed graphics. These extensions generally have their own vector computation units and vector register file which sit beside the standard integer and/or floating point processing paths. The registers themselves are commonly 64 to 256 bits in length, and hold subword quantities of 8, 16, or 32 bits. The SIM$_p$D unit as a whole generally serves as an on-chip accelerator. Computation kernels which are data parallel are *vectorized* and their processing occurs in the processor's SIM$_p$D unit.

The AltiVec architecture is the most flexible of the general-purpose SIM$_p$D implementations. Its instructions exploit data parallelism at a granularity of 8, 16, or 32 bits; data-reordering, combining, and masking capabilities are also provided. Specifically, AltiVec's `vector permute` instruction provides alignment and element merging capabilities which are absent from MMX and VIS. The flexibility of AltiVec makes its use in domains other than graphics and gaming plausible, so it is the primary SIM$_p$D ISA referenced in this work.

SIM$_p$D **Embedded/DSP processors**

SIM$_p$D processing has become popular for DSPs intended particularly for consumer media processing, *i.e.* video, teleconferencing, image processing, and broadband applications. Among designs which include SIM$_p$D instructions are Analog's ADSP-21161, Equator's MAP-CA Broadband Signal Processor (BSP), 3DSP's UniPHY, Tensilica's Xtensa Vectra extension [10], and Philips' TriMedia CPU64 [8]. In addition, special ISA support is provided on general embedded processors: Motorola's Signal Processing Engine (SPE) [13] is a SIM$_p$D processing unit with special support for traditional DSP operations; Wireless MMX extensions to Intel's XScale architecture provide 64-bit versions of MMX, SSE, and other multimedia instructions for the embedded market [14]; and MIPS-3D instructions are 2-way SIM$_p$D operations intended for three-dimensional graphics processing on a MIPS64 architecture [15].

## 4.3 SIM$_d$D architectures

A *Single Instruction Multiple Disjoint Data* (SIM$_d$D) architecture implements SIMD computation without the alignment and placing restrictions of SIM$_p$D approaches. As pictured in Figure 1(c), each instruction specifies a single computation to be performed on more than one datum, but the source registers of the data elements are arbitrary. The instruction shown indicates that it is possible to accomplish this flexibility using very long instructions which contain fields for each source and destination data element. For four-wide SIMD, this requires eight source and four destination register fields, making the instructions longer as compared to SIM$_p$D. The authors are not aware of any general SIM$_d$D implementations.

An alternative method of data specification uses *vector pointers* to specify the elements used for computation. Figure 2 shows the level of indirection this adds to a general SIM$_d$D architecture. Each vector pointer has $n$ indices which point to vector elements to be used for computation, and instructions specify two source vector pointers and a destination vector pointer. Pointers may be set explicitly or loaded from memory, as described in [16]. An example of an indirect-SIM$_d$D architecture implemented using vector pointers is IBM's eLiteDSP [17].

For the remainder of this paper, an indirect-SIM$_d$D architecture will be assumed to have vector pointers which specify access to four elements at a time, with a data element size of 32 bits. These four elements are obtained and processed simultaneously.

# 5. ARCHITECTURE TRADEOFFS

The following sections discuss hardware, code generation, and performance tradeoffs associated with each architecture type. Specific implementations provide features to accelerate particular computations, overcome the memory bottleneck, or re-order data for particular applications. To the extent possible, this analysis abstracts away from these differences which are not fundamental to the VLIW, SIM$_p$D, or SIM$_d$D computation models. In some cases, however, the eLiteDSP processor will be discussed separately from general SIM$_d$D characteristics, since it is the only known SIM$_d$D implementation, and some of its indirect-SIM$_d$D features are relevant to the comparisons made. In each tradeoff

subsection, when an architecture has no particular benefits/drawbacks associated with it, it will not be mentioned.

## 5.1 Impact on hardware implementation

Data and instruction memory typically consume 40% of an embedded processor's real estate, and are predicted to dominate by 2005 [18]. Sizing of both memory types can be significantly affected by an architecture's data access and storage methods.

**Data memory size**

The most commonly-cited drawback of programming for current SIM$_p$D architectures is the requirement that data be aligned to vector-length boundaries in memory. This occurs because vector memory operations are specified using a single address, from which $n$ sequential vector data elements are loaded into a single vector register. Since there is usually no unaligned addressing support, the programmer must ensure that all data is aligned to the architected vector length, or else multiple vector loads and registers may be needed to compose a single vector from memory. This induces memory waste if a programmer/compiler is forced to pad data in order to meet alignment requirements and minimize on-chip re-arrangement of loaded/stored data.

**Instruction memory size**

With regard to instruction storage, if a single loop iteration does not contain enough parallelism to fully utilize a processor's execution resources, multiple loop iterations must be co-scheduled. Loop unrolling is the most straightforward technique to accomplish iteration co-execution, but incurs significant code size expansion. Modulo scheduling has become a common embedded processing technique to increase the throughput of statically scheduled loops, while keeping code size manageable. "Generic" modulo scheduling requires *modulo variable expansion* (MVE) which uses extra registers and instructions to keep unique variable copies. Code expansion incurred for loop unrolling and MVE increases the required instruction memory size.

If an architecture provides *rotating registers*, code size may be reduced because the burden of register renaming is moved from the compiler to hardware. While some general-purpose VLIW-like implementations (*e.g.* Itanium) include rotating register support, this is not yet a feature in embedded VLIW processors, and current SIM$_p$D implementations do not include the ability to rotate vector registers (though this might be beneficial to many algorithms).

The eLiteDSP SIM$_d$D implementation includes an *auto-update* mechanism for vector pointers, which allows a form of register rotation more flexible than VLIW implementations [17]. Register rotation and pointer update mechanisms significantly reduce loop code sizes, and thus processor memory requirements.

Program code size is impacted not only by operation counts, but also by the size of individual operation encodings. As evident from the instruction fields in Figures 1 and 2, VLIW and general SIM$_d$D implementations have the largest instruction sizes. SIM$_p$D and indirect-SIM$_d$D operations specify multiple computations much more compactly. Additionally, since each SIM$_p$D register holds $n$ elements, for a given element count, $e$, there will be $r = e/n$ vector registers for a SIM$_p$D implementation, as opposed to the $e$ registers need for a VLIW. This reduces the number of bits needed to specify source and destination registers to $log_2(r = \frac{e}{n})$ in SIM$_p$D operations, from $log_2(e)$ for VLIW/general SIM$_d$D.

Inclusion of pointer auto-update in an indirect-SIM$_d$D implementation does not necessarily require additional opcode bits, if an update *stride* is associated with each vector pointer (see Section 6); several additional opcode bits may be necessary to support more complex auto-update mechanisms.

**Data memory ports**

As previously described, elements of SIM$_p$D and indirect-SIM$_d$D registers are stored in sequential memory locations. While restrictive from a data ordering perspective, this means, however, that if one vector `load/store` is allowed per cycle, only one memory port and address computation unit are needed to obtain $n$ data elements. In contrast, a VLIW or general SIM$_d$D architecture would need $n$ address computations and $n$ memory ports to obtain the same number of data elements, since elements are not necessarily sequential.

**Register files**

A register file's size and power consumption is impacted by its number of ports. VLIW and SIM$_d$D register files will be the largest: three times the processor issue width, and three times the vector width, respectively. SIM$_p$D ports will be the widest (since multiple data elements are obtained through a single register), but there need only be three ports.

All architectures will have a data element register file, but indirect-SIM$_d$D requires an additional file for vector pointers. This will likely be small and fairly simple, but if pointers and data must be accessed in the same pipeline stage, this may necessitate extention of the indirect-SIM$_d$D processor cycle. More likely, however, pointers will be accessed in a pipeline stage separate from data, so general and indirect-SIM$_d$D architectures should have register access times similar to those for a VLIW. A SIM$_p$D register access should be faster, since the file is smaller.

**Functional units**

For a VLIW architecture, DLP computations are performed using the same resources which produce ILP results. In contrast, SIM$_p$D capability is commonly added to a processor as a separate DLP unit which operates beside units for standard integer code and control. Even if integer or floating point registers and function units are used to achieve SWP computation, SIM$_p$D architectures will generally need a unit to re-arrange data elements. `Permute` and `merge` operations are logically crossbars, so this additional unit will generally require additional hardware. An indirect-SIM$_d$D implementation also requires an additional functional unit: a vector pointer manipulation unit capable of pointer loads, stores, and simple arithmetic.

While the VLIW paradigm is most flexible due to the ease with which arbitrary data are accessed, using 32- or 64-bit integer or floating point registers and computation units is wasteful for exploiting the small data type DLP commonly available in telecommunication and media applications (often 8, 13, or 16 bit granularity).

**Architecture extensibility**

Whereas a SIM$_p$D architecture requires $log_2(r)$ bits to specify each register, operand specification in an indirect-SIM$_d$D architecture is independent of the number of vector registers, and only depends on the number of architected vector pointers. This allows expansion of the number of data registers without sacrificing code compatibility. This is unique because VLIW and SIM$_p$D register counts must be architected, and are not extensible across processor implementations within the same architecture family.

## 5.2 Impact on code generation

**Compiled code**

Code for architectures with vector data types must first be *vectorized* into DLP computations which match the architected vector length. *Auto-vectorization* is the process of automatically generating code which expresses data parallelism. This process has been well-explored for numerical Fortran code, which contains clearly parallelizable `do` loops. Signal processing code, however, when not written in assembly language, is almost exclusively written in C. This presents several difficulties to a vectorizer. First, pointer use gives rise to *aliasing*. If compiler alias analysis is unable to prove independence of two arrays, some loops which use these arrays may not be vectorized. Second, use of function calls or `break` statements in C program `for` loops also causes difficulties for auto-vectorizers. Additionally, global arrays are often used in C with little attention paid to data alignment. Most auto-vectorizers target the SIM$_p$D paradigm, which requires memory alignment for `vector load` and `store` operations, so the compiler must understand correlations between array indices and alignment. If a loop's array accesses cannot be determined to be aligned, the loop will not generally be vectorized because the overhead of manual vector alignment is prohibitive. Loops in telecommunication and media kernels generally contain high amounts of DLP, but the language characteristics described above present many potential pitfalls to compilers auto-vectorizing code for a DLP processor.

The premise of VLIW architecture is to move complexity from a processor's hardware to its software, so VLIW architectures are generally quite regular, and thus good compiler targets. Unlike SIM$_p$D or SIM$_d$D architectures, they do not require vectorization. If a compiler is unable to fully resolve array dependences or a loop contains a cross-iteration dependence, available ILP may still be exploited. This can be a distinct performance advantage for automated compilation.

**Programmer effort**

Because vectorization is difficult to achieve automatically, but essential to SIM$_p$D performance, the burden of SIM$_p$D code generation is often placed on the code developer, even when using the C language. For the AltiVec ISA extensions, Motorola has defined pragmas and intrinsics which programmers use to describe vectorization in AltiVec pseudo-assembly [19]. All vector assignments and computations are stated explicitly in the C code, and their translation to AltiVec assembly instructions is nearly one-to-one.

For algorithms with irregular data patterns (see Section 6), it is sometimes possible to manipulate their computations in such a way as to regularize DLP patterns. This is generally tedious (extends software development time) and may require programmer understanding of the mathematical underpinnings of an algorithm. While expensive at development time, algorithmic modifications generally also cost run-time cycles and register resources. Given VLIW or SIM$_d$D data access flexibility, algorithmic modification is generally unnecessary.

While auto-vectorization is difficult for a compiler, assembly programmers generally find VLIW code hard to schedule because it is difficult to track many in-flight instructions related in irregular ways. Paradoxically, vectorized code generation may be easier for assembly programmers, once an algorithm is converted to regular data access patterns.

## 5.3 Execution impact

Telecommunication and media applications can generally be considered loop-dominated, and can be viewed as sets of performance-critical loops connected by control code. Given the same number of function units and equivalent operation types, actual data computation should proceed in any architecture at roughly the same rate. However, the manner in which data is obtained and arranged for computation may have a significant impact on performance, since it may not always be possible to supply function units with a constant stream of productive operations. In this section, the impact of the three described architectural styles on a program's execution time will be evaluated for the dominant application phases of loop prologue/epilogues and loop computation kernels.

**Prologue/epilogue overhead**

As previously described, SIM$_p$D implementations generally require data to be aligned to vector-length memory bounds. In addition to this restriction, SIM$_p$D input data must appear in memory in an order close to that used for calculation since they must reside in a single vector register when an operation is issued. When a loop needs data in an order different from that stored in memory, or when a kernel produces data in a pattern different from that needed by a subsequent kernel (such as FFT bit reversal), cycles are spent re-arranging data in the prologue or epilogue.

While SIM$_d$D loads also require data alignment in memory, sequentially fetched elements are not necessarily written into adjacent registers, since arbitrary, disjoint, elements may be specified either explicitly (general SIM$_d$D) or by means of a vector pointer (indirect-SIM$_d$D). There is therefore no need to spend cycles re-arranging elements before or after access to memory, and it is less important that data be aligned to particular memory boundaries. Also, register pressure may be eased because the need for permutations is eliminated, reducing register spills, and thus aiding execution time.

While the cost of SIM$_d$D memory access may improve pro/epilogue execution relative to SIM$_p$D architectures, indirect specification of source and destination operands for an indirect-SIM$_d$D implementation requires prologue pointer initialization instructions. For arbitrary initialization, the number of bits required to indicate any four separate data elements will likely necessitate several operations (or a single long operation with multiple immediate fields). For example, kernels vectorized for the eLiteDSP architecture were found to use an average of 5.1 (hand assembly) and 5.9 (compiled) vector pointers. Initializing these pointers extends kernel prologues beyond those for a VLIW or SIM$_p$D architecture. Figure 3 shows an example of prologue vector pointer set-up and subsequent loop execution for a standard FIR filter on an indirect-SIM$_d$D architecture. Two vector pointers are initialized: one whose members all point to a coefficient value, and a second pointer to indicate input data. As filter processing progresses, the vector values will be updated by the indicated stride (1), so that sequential coefficients and input values are used as inputs to the vector multiply-accumulate operations.

**Computation kernel overhead**

VLIW, SIM$_p$D, and SIM$_d$D differ in the ease with which recently computed data may be reused for subsequent computations. For a VLIW architecture, register data reuse is trivial: the destination of a previous computation is simply

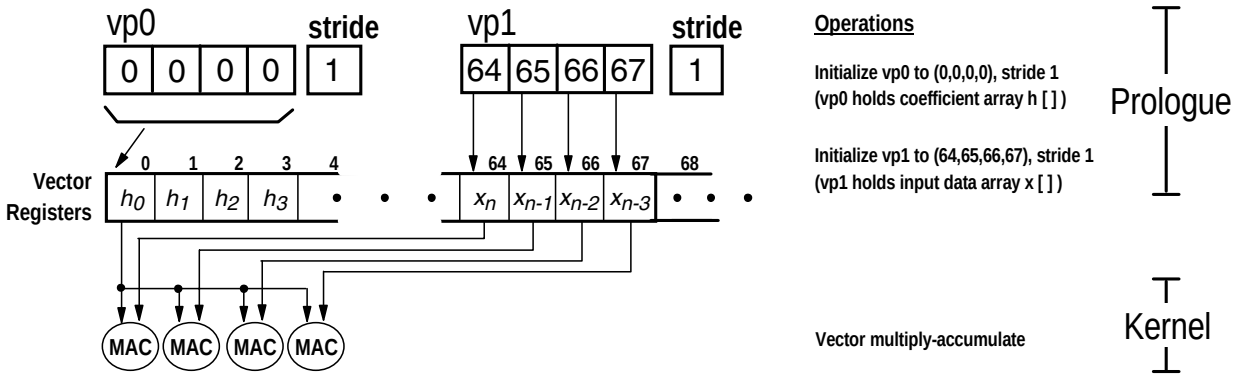



**Figure 3: Indirect-SIM $_d$D vector pointer set-up: FIR filter.**

**Table 2: Qualitative comparisons.**

| | **VLIW** | SIM $_p$D | SIM $_d$D | indirect-SIM $_d$D |
|---|---|---|---|---|
| **Data memory alignment** | Not needed; values loaded individually | Strict alignment to vector bounds | Alignment to vector bounds, but flexible reg. placement | Alignment to vector bounds, but flexible reg. placement |
| **Program size** | Large; better if rotating registers implemented | Moderate | Large | Small if pointer auto-update implemented; else Moderate |
| **Instruction size** | Large | Small | Large | Small |
| **Memory ports** | Large | Small | Large | Small |
| **Data register ports** | 3 * processor width | 3 wide ports | 3 * vector width | 3 * vector width |
| **Function Units** | Control logic, arithmetic | Control logic, arithmetic, vector permutation | Control logic, arithmetic | Control logic, arithmetic, pointer arithmetic |
| **Extensibility** | Vector size determined by VLIW size; instr. length dependent on reg. count | Vector size can be changed | Vector size can be changed; # vector elements changes instruction length | Vector size can be changed; instr. length independent of # vect. elements |
| **Code generation** | Compilation straightforward; assembly difficult | Vectorization needed; assembly straightforward if algorithm packs easily | Vectorization needed | Vectorization needed |
| **Data (re)use** | Arbitrary specification; no data movement required | Contiguous data: permutations required | Arbitrary specification; no data movement required | Arbitrary specification; no data movement required |
| **Prologue overhead** | None | Data align / permute | None | Pointer set-up |
| **Kernel overhead** | None | Permutations possible | None | Pointer arithmetic (but may be automated) |

specified as one of the sources for a subsequent operation. There is no need for data re-organization, regardless of how disperse data are in the register file. The same is true for general SIM $_d$D architectures.

Indirect-SIM $_d$D computation requires somewhat more overhead, but is still quite flexible. The programmer or compiler must keep track of the locations to which data are written (indicated by a *destination pointer*), and set a *source pointer* to access the newly produced data. If data are used in the same order as they are produced, it may be possible to simply copy the destination pointer to a separate source pointer in the loop prologue, thus avoiding a pointer set-up instruction sequence. In the ideal case, a modulus will have been established for the destination pointer which causes it to wrap around to the beginning of a data region once all data has been processed. This pointer can then become the source of subsequent instructions which iterate over the new results. In this way no additional instructions or cycles are required for data reuse.

In contrast to the relative ease with which VLIW and SIM $_d$D register data may be reused, SIM $_p$D presents obstacles to non-sequential data reuse. If computed data values do not reside in vector register locations which are exactly aligned to those of a second vector source register, then data must be re-organized. Not all current SIM $_p$D architectures provide meaningful `permute` instructions, so this characteristic of subword-parallel computation can cause performance problems. Flexible `permute` and `merge` operations allow arbitrary data reordering with a single register and between multiple registers, meaning that they provide for combination of vector elements from multiple registers into a single SIMD vector register. These can reduce the performance penalty of data reorganization to a single cycle. However, since the elements in each source register of a SIM $_p$D oper-

```
int real_32_hz (int ntaps,vector signed short *c,
                int ndat,vector signed short *x,
                vector signed short *y) {
   vector signed short x0, x1, coef;
   vector signed int tmp1, tmp2, tmp3, tmp4;
   vector signed int tmp5, tmp6, ac0, ac1, ac2;
   vector signed int ac3, ac4, ac5, ac6, ac7;
   int i, j, op, ndatavec, ncoefvec;

   op = 0;
   ndatavec = ndat >> 3;
   ncoefvec = ntaps >> 3;

   for (i = 0; i < ndatavec; i++) {
     x0 = x[i]; /* load next 8 input samples */
     /* clear accumulators (accs) */
     do { /* The j loop computes 8 outputs */
       coef = c[j];
       x1 = x[i+j+1];
       ac0 = vec_msums(coef, vec_sld(x0, x1, 2), ac0);
       /* Similar operations on ac1...ac7 */
       x0 = x1; /* x1 contains x0 val for next iter */
       j += 1;
     } while (j < ncoefvec);
     /* Round & sum across acc for each output */
     ac0 = vec_sums(ac0, ROUND);
     /* Same operation on ac1...ac7 */
     /* Copy high 16 bits of last elem.  of each acc
     to single output vector (containing 8 outputs) */
     tmp1 = vec_mergel(ac0, ac1);
     /* Merge ac2...ac7 */
     y[op] = vec_perm((vector signed short) tmp5,
                (vector signed short) tmp6, PR);
     op += 1;
   } /* for i */
   return op*8;
}
```

**Figure 4: AltiVec FIR filter in C with Motorola extensions.**

ation must be aligned to one another, it is commonly necessary to insert SIM$_p$D data permutations within loop bodies. In tight kernels, even one additional cycle can be costly. Some SIM$_p$D implementations provide instructions to rearrange data within a single register, but few allow data values to be combined from multiple registers. As previously mentioned, AltiVec includes `vector permute`, `vector shift`, and `vector select` instructions which provide complete element re-ordering capabilities. These instructions, however, only operate on two SIMD vector registers at a time, meaning that if a new register must be composed of elements from more than two vector registers, longer sequences of re-order operations may be required.

As an example of the kernel execution penalty of SIM$_p$D, Figure 4 shows excerpts of Motorola's sample FIR program [20]. For this code, `vec_perm`=*vector permute*; `vec_sld`=*vector shift left double*; and `vec_mergel`=*vector merge low*. As indicated in bold, data re-arrangement operations are necessary within both the inner and outer loops. In contrast, for the eLiteDSP FIR filter in Figure 3, both the coefficient (`vp0`) and input (`vp1`) vectors are initialized with an auto-update stride of 1. This eliminates the need for vector shifting within the kernel.

## 5.4 Tradeoff summary

Table 2 provides a summary of the qualitative comparisons made among the three primary architecture types in the previous sections. There are costs associated with each architecture type, but as will be shown in the next section, the flexibility afforded by some styles may more closely match fundamental algorithmic computation patterns, and thus warrant the additional hardware or programming cost for particular application domains.

# 6. DLP COMPUTATION PATTERNS

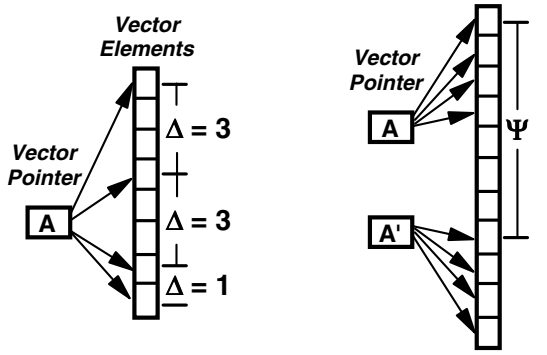
This section provides a quantitative description of DLP access patterns found in kernels essential to current and future telecommunication and media standards. These include filters of various lengths and types, vector and matrix operations, bit packing, an inverse discrete cosine transform (IDCT), and pitch estimation. While this study only examined kernels (as opposed to entire applications), they represent the major portion of many applications. Some kernels were written in assembly language by experienced programmers, based upon a C language specification (*Hand* kernels in Table 3); separate results are based upon code generated by the vectorizing compiler described in [17] (*Compiled* kernels). Execution traces were collected using the simulator described in [17].

While measurements were made on a particular architecture (the eLiteDSP), this analysis is not bound to that specific architecture type or implementation, and similar results should be observed if derived from code for other architectures. Since these kernels were written and compiled under the assumption of full indirect-SIM$_d$D architecture flexibility, it can fairly safely be assumed that register usage which follows SIM$_p$D conventions arises from the underlying algorithms, and corresponds to the manner in which code would have been vectorized for a SIM$_p$D architecture. The eLiteDSP ISA gives preference to SIM$_p$D-like pointer initializations, so where possible, SIM$_p$D patterns will generally be used.

Two metrics were used in deriving these results: $\Delta$ and $\psi$. As shown in Figure 5(a), $\Delta$ is the distance between elements located by a single indirect-SIM$_d$D vector pointer. The eLiteDSP ISA provides an automatic *pointer update* option, in which all elements of a vector pointer are incremented by a constant stride, $\psi$, each time the pointer is used (Figure 5(b)). Figure 6 demonstrates how use of R0 and then R2 in a SIM$_p$D context ($(\Delta, \psi) = (1, 8)$) correlates to a vector pointer change in the indirect-SIM$_d$D model.

For these experiments, if access occurs to a vector in which each element location is separated by a constant difference $\Delta$, the Vector Stride Register value $\psi$ is also recorded. These values are then assigned a DLP *pattern* name, according to the categories listed in Table 4. In general, preference is given to SIM$_p$D architectures, meaning that if a given $(\Delta, \psi)$ pair can be handled efficiently by multiple architectures, it is attributed to the SIM$_p$D pattern. For this reason, the case $(\Delta = 0, \psi = 0)$ is assigned to pattern SIM$_p$D, since it is easily handled in a SIM$_p$D architecture via a `splat` instruction which places a single value into all elements of a given vector register. The ROT pattern indicates accesses which would be easily handled with some form of automated register shifting, such as *rotating registers*. Any $(\Delta, \psi)$ pair not listed in Table 4 is assumed to require SIM$_d$D or VLIW flexibility and is categorized as a FLEX pattern.

Figures 7 and 8 show the measured distribution of DLP read access patterns for hand and compiled codes. Each




(a) Vector Element Distance (b) Vector Stride

**Figure 5: Meaning of $\Delta$ and $\psi$.**

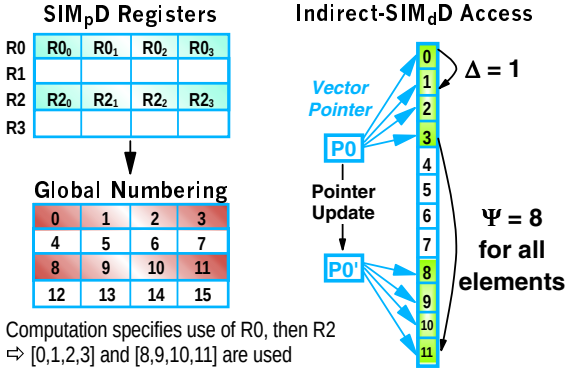



**Figure 6:** $\mathrm{SIM}_p\mathrm{D}$ **data access pattern and** $\mathrm{SIM}_d\mathrm{D}$ **corollary:** $(\Delta, \psi) = (1, 8)$**.**

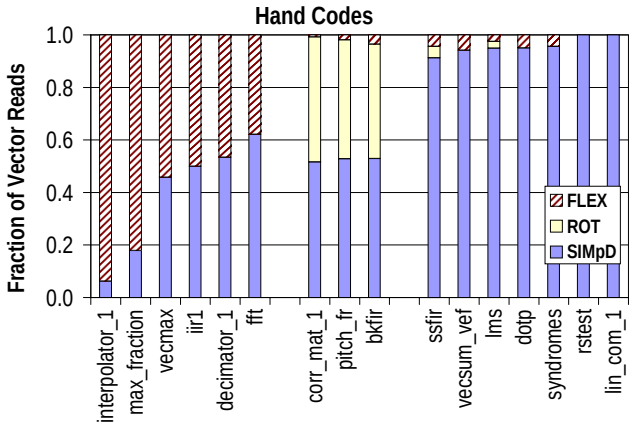



**Figure 7: Hand code: vector element distance and stride–$(\Delta, \psi)$ pairs.**

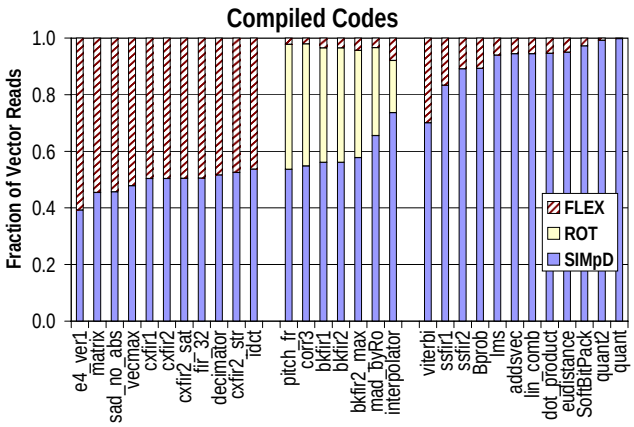



**Figure 8: Compiled code: vector element distance and stride–$(\Delta, \psi)$ pairs.**

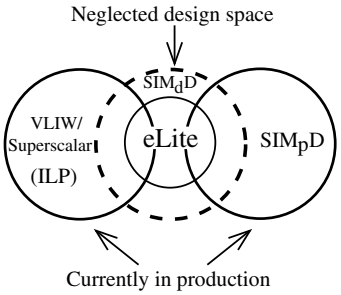



**Figure 9: Partial architecture design space.**

**Table 3: Benchmark kernels.**

| Benchmark | Description |
|---|---|
| **Hand-Generated Code** | |
| *bkfir* | Real block FIR filter; T=16 taps, N=40 (block size) |
| *corr_mat_1* | Autocorrelation matrix |
| *cxfir* | Complex block FIR filter |
| *decimator_1* | Decimator, 2:1 (M=2) |
| *divide* | Integer divide |
| *dotp* | Vector dot product |
| *fft* | 256-pt, rad-4 FFT |
| *iir1* | IIR filter |
| *interpolator_1* | Interpolator, 1:2 (L=2) |
| *lin_com_1* | Linear comb. of 2 vectors |
| *lms* | Least-Mean-Square filter |
| *max_fraction* | Find max of vector of rational fractions – from GSM-EFR codebook search |
| *pitch_fr* | GSM-FR pitch est. (vect. convolution) |
| *ssfir* | Single sample FIR |
| *syndromes* | Syndrome computation for (255,239) Reed-Solomon |
| *vecmax* | Vector maximum |
| *vecsum_vef* | Vector sum – VEF preloaded |
| **Compiler-Generated Code** | |
| *addsvec* | Add two vectors, one un-aligned; N=40 |
| *bkfir1* | Real block FIR filter; T=16, N=40 |
| *bkfir2* | Alternate coding style: real block FIR filter; T=16, N=40, no delay line |
| *bkfir2_max* | Real block FIR filter; T=16, N=40 |
| *Bprob* | B-probability implemented as squared Euclidean distance; N=40 |
| *corr3* | 3-vector correlation; N=64 |
| *cxfir1* | Complex block FIR filter; N=40, T=16 |
| *cxfir2_sat* | Alternate coding style: complex block FIR filter; N=40, T=16 |
| *cxfir2_str* | Alternate vectorization used: complex block FIR filter; N=40, T=16 |
| *decimator* | Decimator, 2:1; N=$\|h\|$=16, $\|x\|$=96 |
| *dot_product* | Vector dot product; N=40 |
| *e4_prld* | Synthetic test case |
| *e4_ver1* | Synthetic test; different compilation method |
| *eudistance* | Euclidian distance between two vectors; N=48 |
| *fir_32* | Dbl-precision FIR filter; N=40, T=16 |
| *idct* | 8x8 block 2D-IDCT |
| *interpolator* | Interpolator, 1:2; N=$\|h\|$=16, $\|x\|$=64 |
| *lms* | Least-Mean-Square filter; block size 40, T=32 |
| *lin_comb* | Linear combination of 2 vectors; N=40 |
| *mad_byBlk* | Minimum absolute distance (8x8 block); for motion estimation |
| *mad_byRow* | Minimum absolute distance (8x8 block); for motion estimation |
| *matrix* | Multiply 4x4 matrix by a vector |
| *quant* | H.263 quantization of 8x8 coefficient block |
| *quant2* | Quantize 15 bit (+sign) bit samples to 5 bit (+sign) |
| *sad_no_abs* | Sum of absolute distance calculations for H.263; N=16x16 |
| *ssfir1* | Single sample FIR filter; N=1, T=16 |
| *ssfir2* | Single sample FIR filter; T=16 |
| *SoftBitPack1* | Pack 6 soft bits into 32-bit vectors; 1 soft bit per vector |
| *vecmax* | Find max in an array and return index of first occurrence; N=40 |
| *viterbi* | Viterbi algorithm to decode convolutional codes in GSM full rate standard; N=189 |

**Table 4: DLP patterns.**

| Delta ($\Delta$) | Stride ($\psi$) | Description | Pattern Name |
|---|---|---|---|
| 0 | 0 | Constant | SIM$_p$D |
| 0 | $> 0$ | SIM$_p$D | SIM$_p$D |
| 1 | 0 | Constant | SIM$_p$D |
| 1 | 1 | Shift/Rot. | ROT |
| 1 | 2 | Shift/Rot. by non-unit stride | ROT |
| 1 | 3 | Shift/Rot. by non-unit stride | ROT |
| 1 | [4,8,12,...] | SIM$_p$D | SIM$_p$D |
| 2 | 1 | alternating even/odd | FLEX |
| 4 | 1 | 4x4 matrix col. traversal | FLEX |
| 8 | 1 | 8x8 matrix col. traversal | FLEX |
| 16 | 1 | 16x16 matrix col. traversal | FLEX |
| Non-const. | any | Random element access | FLEX |

set of kernels is grouped into categories: those requiring high, moderate and low amounts of flexibility in their data ordering. Kernels with the most rigid DLP patterns (right-hand side of the graphs; SWP = 90-100%) would be efficiently computed on a SIM$_p$D architecture with packed vector registers. Those with large percentages of FLEX patterns (*e.g.* interpolation or vector maximum search) are algorithms with inherently irregular computation patterns and would incur execution penalties on an architecture with strict data alignment requirements.

The vast majority of the FLEX accesses in Figures 7 and 8 occur with constant stride ($\psi$) values, but a non-unit element separation ($\Delta > 1$) (see categories in Table 4). For example, it was advantageous in the 2:1 decimation calculation (the *decimator* kernel) to use every other data element as the input to vector multiplies, *i.e.* elements $(1, 3, 5, 7)$ then $(2, 4, 6, 8)$, etc. Assuming availability of an automatic pointer update mechanism (increment pointer values by a constant after use), these FLEX patterns represent data rearrangement which could occur with little or no penalty on the SIM$_d$D architecture, but would require permutations or data regularization on a SIM$_p$D architecture.

FLEX usage is also found in manipulation of complex numbers. If numbers $C_0, C_1, C_2, C_3$ are stored as sequential $(real, imag)$ pairs in locations (0...7), then accessing only real or only imaginary elements occurs with $\Delta = 2$, *i.e.* elements $(0, 2, 4, 6)$ are real, while $(1, 3, 5, 7)$ are imaginary. Accessing complex numbers in reverse order of their storage sequence requires a non-constant element separation. If four elements may be accessed at once, $C_3$ and $C_2$ will be used first, then $C_1$ and $C_0$. This corresponds to access of locations $(6, 7, 4, 5)$ and $(2, 3, 0, 1)$ together, which are irregular, or FLEX, access patterns.

Some FLEX usage results from pointer arithmetic operations available for the eLiteDSP. These include the capability to bit-reverse vector pointer values (used for the FFT) and to left-shift pointer values. Such instructions can be assumed to cost 1 cycle for an indirect-SIM$_d$D architecture, but would be performed on a different datapath than computations on vector elements, so could be performed in parallel with vector calculations. They are thus low-cost, but allow easy access to data values in patterns inherent to

important kernels. As an example, the SIM$_d$D eLiteDSP *vector pointer shift left* instruction is used for the *vec_max* and *max_fraction* benchmarks; its equivalent on SIM$_p$D AltiVec would be to issue a *load vector for shift [right,left]* operation, followed by *vector permute*.

It is apparent from these results that some computation kernels contain DLP patterns conducive to efficient programming on an architecture with packed data vectors. However, this is not a consistent property of all telecommunication and media kernels, and depending on a target application's combination of kernels, it appears beneficial to consider use of an alternate architecture style, such as indirect-SIM$_d$D, which would allow irregular data patterns to be exploited using the same vector mechanism capable of regular DLP operations.

# 7. CONCLUSIONS

Figure 9 depicts a subset of the uni-processor design space available to embedded processor architects. On the left, labeled as *VLIW/Superscalar*, are architectures which realize data parallel computation directly through instruction-level parallelism. The circle on the right represents SIM$_p$D architectures, which pack quantities into vector registers and use these aligned groups of data as inputs to computation. In the embedded and DSP domains, commercially available and published processor designs currently fall into the ILP or SIM$_p$D categories, or contain computation units for each of these styles. The SIM$_d$D class of architectures can be viewed as a bridge between these two design forms: SIM$_d$D provides flexible access to data elements (a likeness to VLIW); but also has separate, compact instructions for performing data parallel computations. The eLiteDSP's indirect vector access mechanism is a particular implementation of the general SIM$_d$D class which bridges the ILP→SIM$_p$D design space.

This study indicates that many telecommunication and media kernels contain opportunities for data parallel computation which do not map well to the SIM$_p$D paradigm. These computational patterns motivate exploration of embedded and DSP architectures whose design points lie outside the space currently implemented in the VLIW/Superscalar and SIM$_p$D domains.

# 8. ACKNOWLEDGMENTS

The authors would like to thank the reviewers and the eLiteDSP team for their contributions, in particular Amir Geva for his work on the simulator.

Hillery Hunter was partially supported by a fellowship from the National Science Foundation.

# 9. REFERENCES

[1] V. Lappalainen, T. Hamalainen, and P. Liuha, "Overview of research efforts on media ISA extensions and their usage in video coding," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 12, pp. 660–670, Aug. 2002.

[2] D. Talla, L. John, V. Lapinskii, and B. Evans, "Evaluating signal processing and multimedia applications on SIMD, VLIW and superscalar architectures," in *Proceedings of the 2000 Int'l Conf. on Computer Design (ICCD '00)*, pp. 163–172, 2000.

[3] C. Kozyrakis and D. Patterson, "Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks," in *Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO-36)*, pp. 283–293, Nov. 2002.

[4] D. Talla, L. K. John, and D. Burger, "Bottlenecks in multimedia processing with SIMD style extensions and architectural enhancements," *IEEE Transactions on Computers*, 2002.

[5] E. Bronson, T. Casavant, and L. Jamieson, "Experimental application-driven architecture analysis of an SIMD/MIMD parallel processing system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 195–205, Apr. 1990.

[6] P. Faraboschi, G. Desoli, and J. Fisher, "The latest word in digital and media processing," *IEEE Signal Processing Magazine*, pp. 59–85, Mar. 1998.

[7] J. Stokes, "3 1/2 SIMD architectures." http://www.arstechnica.com/cpu/1q00/simd/simd-1.html, Mar. 2000.

[8] J. van Eijndhoven, *et al.*, "TriMedia CPU64 Architecture," in *Proc. Int'l Conf. on Computer Design (ICCD '99)*, pp. 586–592, 1999.

[9] Infineon Technologies Corp., *CARMEL DSP Core Technical Overview Handbook*, 1.0 ed., 2000.

[10] R. Cravotta, "2003 DSP directory," *EDN Magazine*, Apr. 2003.

[11] G. Barnes, R. Brown, M. Kato, D. Kuck, D. Slotnick, and R. Stokes, "The ILLIAC IV computer," *IEEE Transactions on Computers*, vol. C-17, pp. 746–757, Aug. 1968.

[12] G. Fox, R. Williams, and P. Messina, *Parallel Computing Works!* San Francisco, CA: Morgan Kaufmann Publishers, 1994.

[13] K. Gala, "Signal processing engine architecture, instruction set, and programming model." Motorola Smart Networks Developer Forum (SNDF), July 2002.

[14] Intel Corporation, *Intel Wireless MMX Technology Product Brief*, Publication 251669-001, 2002.

[15] MIPS Technologies, Inc., *MIPS-3D Graphics Extension Product Brief*, Publication 0600/2k/rev0, 2000.

[16] J. Derby and J. Moreno, "A high-performance embedded DSP core with novel SIMD features," in *Proceedings of the 2003 International Conference on Acoustics, Speech, and Signal Processing (ICASSP'03)*, Apr. 2003.

[17] J. Moreno, *et al.*, "An innovative low-power high-performance programmable signal processor for digital communications," *IBM Journal of Research and Development*, vol. 47, pp. 299–326, March/May 2003.

[18] Semiconductor Industry Association, "International technology roadmap for semiconductors (ITRS)," 2001.

[19] Motorola Corporation, *AltiVec Technology Programming Interface Manual*, June 1999.

[20] Motorola Corporation, "Altivec code samples." http://www.motorola.com/SPS/PowerPC/AltiVec.