Modern microprocessors need to meet the high performance/throughput requirements of the increasingly complex applications that operate over huge datasets. In addition, they have to provide such high performance under a very stringent power envelope. Computer architects meet the performance requirements by extracting different kinds of parallelism (Instruction/Data/Thread Level Parallelisms) through different architectural and micro-architectural extensions. One of these extensions, namely Single Instruction Multiple Data (SIMD) extensions, has become so popular that it can be found in processors from a diverse range of computing domains ranging from general purpose and embedded processors to digital signal processors and gaming consoles. One of the reasons for their popularity is that they not only boost performance but do so in an energy efficient manner due to their hardware simplicity. Moreover, their replicated functional units and simple control mechanism make them amenable to scaling to higher vector lengths.

However, code generation for SIMD accelerators has been a challenge from the days of their inception. Compilers generate vector code conservatively to ensure correctness. As a result they lose significant vectorization opportunities and fail to extract maximum benefits out of SIMD accelerators. The problem further deepens at higher vector lengths because it becomes difficult to find enough independent instructions, performing the same operation, to fill the wider vector/SIMD paths. Furthermore, SIMD accelerators without any leakage control mechanism might become a major source of leakage energy if not utilized judiciously.

My current research targets these issues in SIMD accelerators and this document provides a brief summary of my contributions towards this end.

**Speculative Dynamic Vectorization** [HiPC 2013, PACT 2012, extended version under review at TOCS]

Even though SIMD accelerators are very simple from the hardware perspective, code generation for them has always been a challenge. Static compile time vectorization loses significant vectorization opportunities due to conservative memory disambiguation analysis. A recent evaluation of vectorizing compilers by S. Maleki et al. shows that the modern compilers including GNU GCC, IBM XLC, and Intel ICC are limited in extracting available vectorization opportunities from the vectorizable applications. One of the main problems in static compiling that they discovered is compilers inability to do accurate interprocedural pointer disambiguation and interprocedural array dependence analysis. Furthermore, J. Holewinski et. al. showed that static vectorization fails to extract significant vectorization opportunities especially in pointer-based applications. They vectorize array- and pointer-based version of Digital Signal Processing (DSP) kernels from UT DSP benchmark suit. Their results show that the compiler is able to extract significant parallelism from array based version of the kernels, whereas for pointer based version it fails to extract any vectorization opportunity.

We propose to complement the static vectorization with a speculative dynamic vectorizer. Static vectorization applies several complex and time consuming loop transformations to make a loop vectorizable. However, due to conservative memory disambiguation analysis it loses significant vectorization opportunities, especially in pointer rich applications. We propose to have a speculative dynamic vectorizer to handle these cases at runtime. The proposed dynamic vectorizer speculatively assumes that a pair of ambiguous memory accesses will never alias. This speculative assumption gives more freedom in instructions reordering and hence the dynamic vectorizer is able to discover more vectorization opportunities. During execution, the hardware checks for any memory dependence violations caused by the speculative vectorization. If any violation is detected, the hardware rolls back to a previously saved check-point and executes a non-speculative version of the code.

**Vectorizing for Wider Vector Units** [HPCC 2013]

Due to their hardware simplicity SIMD accelerators are relatively easy to scale to higher vector lengths. As a result, SIMD accelerators grow in size with each new generation e.g. from 64-bit Intel’s MMX to 512-bit Intel Xeon Phi. However, generating efficient vector code for wider SIMD accelerators is not straightforward. The problem lies in the fact that different applications have different natural vector length. The applications with low natural vector length cannot benefit from wider vector units. There are applications for which compilers just need to unroll loops with a higher unroll factor to fill the wider vector paths. However, there is another category of applications that does not have enough parallelism for vectorization at higher vector lengths. Generating code for these applications for wider vector units becomes a challenge. We discover that there are two key factors that thwart the performance at higher vector lengths: 1) Reduced dynamic instruction stream coverage for vectorization and 2) Huge number of permutation instructions.

We propose Variable Length Vectorization (VLV) to increase the dynamic instruction stream coverage. Compilers generate vectorized code only when it is possible to fill the entire vector path, or in other words
when there are enough independent scalar operations to occupy all the vector lanes. If this condition is not met, all the instructions are left in the scalar form. VLV starts by vectorizing for the given physical (maximum) vector length. After vectorizing the code for the physical vector length, the vectorizer reduces the logical vector length iteratively. At lower logical vector lengths, code is vectorized even if the resulting vector instructions have fewer operations and cannot occupy all the vector lanes. Mask registers are used to indicate which vector lanes to enable.

To tackle the problem of permutation instructions, we propose Selective Writing. Permutation instructions are needed when the input operands of a vector instruction are not available in a single vector register or are not in the correct order. Selective Writing allows the scalar SIMD instructions to write to any element of the vector register instead of always writing to the lowest element. Therefore, scalar instructions can write their results in a vector register in an order needed by the vector consumer, hence avoiding the need of permutation instructions.

**Dynamic Selective Devectorization** [SBAC-PAD 2013, TACO 2014]

Leakage energy is the static energy consumption of a circuit when it is idle. Functional units of microprocessors are responsible for a major fraction of leakage energy. Even though SIMD accelerators are an energy efficient way of improving performance, they become the main source of leakage energy due to their wider datapaths, for the applications lacking Data Level Parallelism (DLP) and in the absence of leakage control mechanism. Therefore, it is of prime importance to shrink the leakage energy of SIMD accelerators when they cannot be utilized efficiently due to lack of DLP.

Power gating is a widely used technique to reduce leakage energy consumption of functional units. However, it has an energy and performance overhead associated with it that has to be paid every time a power gated function unit is awakened to perform some operation. This overhead is unjustifiable especially if a functional unit like SIMD accelerator is needed to be awakened only for few cycles. Power gating benefits can be increased if the functional units can be kept off during these intervals of low activity.

Dynamic Selective Devectorization (DSD) is a technique to efficiently power gate higher SIMD lanes. DSD dynamically profiles the applications to find higher lanes usage pattern. If a period of low activity, when the higher lanes are used scarcely, is detected, DSD devectorizes the corresponding piece of code. The devectorized code is executed only on the lowest SIMD lane. As a result, the higher SIMD lanes can be power gated for longer time intervals without intermittent awakening. Therefore, DSD enables power gating to save more leakage energy. Moreover, since only the code corresponding to low activity periods of higher lanes is devectorized, its effect on the performance is minimal.