Optimizing SIMD Execution in HW/SW Co-designed Processors

Rakesh Kumar
Department of Computer Architecture
Universitat Politècnica de Catalunya

Advisors:

Alejandro Martínez
Intel Barcelona Research Center
Antonio González
Intel Barcelona Research Center
Universitat Politècnica de Catalunya

A thesis submitted in fulfillment of the requirements for the degree of

Doctor of Philosophy / Doctor per la UPC

ABSTRACT

SIMD accelerators are ubiquitous in microprocessors from different computing domains. Their high compute power and hardware simplicity improve overall performance in an energy efficient manner. Moreover, their replicated functional units and simple control mechanism make them amenable to scaling to higher vector lengths. However, code generation for these 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.

This thesis proposes to vectorize the program binary at runtime in a speculative manner, in addition to the compile time static vectorization. There are different environments that support runtime profiling and optimization support required for dynamic vectorization, one of most prominent ones being: 1) Dynamic Binary Translators and Optimizers (DBTO) and 2) Hardware/Software (HW/SW) Co-designed Processors. HW/SW co-designed environment provides several advantages over DBTOs like transparent incorporations of new hardware features, binary compatibility, etc. Therefore, we use HW/SW co-designed environment to assess the potential of speculative dynamic vectorization.

Furthermore, we analyze vector code generation for wider vector units and find out that even though SIMD accelerators are amenable to scaling from hardware point of view, vector code generation at higher vector length is even more challenging. The two major factors impeding vectorization for wider SIMD units are: 1) Reduced dynamic instruction stream coverage for vectorization and 2) Large number of permutation instructions. To solve the first problem we propose Variable Length Vectorization that iteratively vectorizes for multiple vector lengths to improve dynamic instruction stream coverage. Secondly, to reduce the number of permutation instructions we propose Selective Writing that selectively writes to different parts of a vector register and avoids permutations.

Finally, we tackle the problem of leakage energy in SIMD accelerators. Since SIMD accelerators consume significant amount of real estate on the chip, they become the principle source of leakage if not utilized judiciously. Power gating is one of the most widely used techniques to reduce leakage energy of functional units. However, power gating has its own energy and performance overhead associated with it. We propose to selectively devectorize the vector code when higher SIMD lanes are used intermittently. This selective devectorization keeps the higher SIMD lanes idle and power gated for maximum duration. Therefore, resulting in overall leakage energy reduction.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Microprocessor design has traditionally been driven by higher performance requirements. Several microarchitectural techniques have been used to extract different kinds of parallelism from the applications. Starting with the introduction of pipeline, processor microarchitecture has gone through numerous improvements like superscalar and VLIW execution, out-of-order execution, speculative execution, multithreaded architectures, application specific accelerators, etc. Different microarchitectural extensions target different kinds of parallelism available in the applications in order to boost performance. Broadly, the parallelism available in an application falls under one or more of the following categories.

1) Instruction Level Parallelism (ILP)
2) Data Level Parallelism (DLP)
3) Thread Level Parallelism (TLP)

Microarchitectural techniques like instruction pipelining, superscalar and VLIW execution, out-of-order execution, register renaming, branch prediction, etc. are all used to exploit ILP. Single Instruction Multiple Data (SIMD) accelerators and vector processors are specifically designed to extract DLP from data parallel applications. Finally, techniques like multithreading and architectures like chip multiprocessors (CMP) target TLP. This thesis focuses on extracting data level parallelism through SIMD accelerators.

SIMD accelerators are one of the most widely used microarchitectural extensions because these are particularly effective in exploiting DLP. Applications from multimedia, scientific, and throughput computing domains are the main targets of these accelerators since they provide significant amount of DLP. Data parallel applications perform the same operation on multiple pieces of data. As a result, SIMD accelerators just need to have duplicated functional units with a very simple control mechanism. The performance boosting ability of SIMD accelerators and their relatively low complexity has led to their incorporation in processors from all the computing domains: general purpose processors, digital signal processors, gaming consoles, as well as embedded architectures. Intel´s

MMX, SSE and AVX extensions, AMD´s 3DNow!, PowerPC´s Altivec, and ARM Neon are prominent examples of SIMD extensions.

Apart from microarchitectural innovations, code optimizations have played an important role in boosting performance. Significant amount of work has been done in compiler optimizations to generate optimized binaries. Traditional compiler optimizations like constant propagation, copy propagation, common sub-expression elimination, loop-invariant code motion, redundant load elimination, store forwarding, dead code elimination, software pipelining, etc. have been successfully used to achieve significant performance improvements. Last decade has also seen the emergence of dynamic optimizations that are performed at runtime. These optimizations benefit from the availability of runtime information that is not available at compile time.

Performance has traditionally been the main focus of computer architects. However, power consumption of the microprocessors and battery life requirements of portable devices have made power/energy consumption an equally important factor. Therefore, computer architects now have to achieve a balance between performance and energy consumption. To address the problem, computer architects have turned their focus on designing simple cores by eliminating power hungry components of a complex core. Since simple cores provide lower performance than complex cores, several proposals have been made to improve overall throughput. For instance, Hardware/Software (HW/SW) Co-designed processors employ dynamic optimizations and speculative execution to improve the overall performance. Chip multiprocessors (CMP) have multiple simple cores on the same die and by simultaneously executing several threads on different cores, the overall throughput is increased.

Since HW/SW co-designed processors provide an opportunity to optimize the applications dynamically using the information about runtime behavior of the application, this thesis focuses on efficient code generation for SIMD accelerators in a HW/SW co-designed environment.

1.1 SIMD Execution Model

SIMD execution, as the name suggests, consists of operating on multiple data elements in parallel using a single instruction. Therefore, the first step to enable SIMD execution is to encode multiple data elements that can be operated on in parallel, in a single instruction. This step is called vectorization. In general, vectorization is done at compile time by a vectorizing complier. Compilers do program analysis to find independent scalar instructions performing the same operation. Several of these instructions, depending on the data type of instructions and the SIMD accelerator width, are packed together in a single instruction, called a vector/SIMD instruction. Vectorized instructions are then executed on the SIMD accelerator. SIMD execution has several advantages over scalar execution. Some of them are described below:

1) Since SIMD accelerators perform multiple scalar operations encoded in a single vector instruction in parallel, they improve the overall performance.
2) Since a single vector instruction encode multiple operations, the total number of instructions to execute an application reduces. This results in lower instruction cache size requirements. Alternatively, it results in improved instruction cache hit rate and improved performance.
3) Having fewer instructions also reduces the amount of work the processor front-end needs to do. It needs to fetch, decode, and schedule less instructions. Also, the back-end needs to retire fewer instructions. This translates to better energy efficiency.
4) Having multiple operations encoded in an instruction allows a high number of effective operations in the instruction window. This leads to better instructions scheduling.
5) Vector memory instructions lead to fewer memory accesses and better utilization of available memory bandwidth.

1.2 Challenges in SIMD Execution

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. 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.

1.2.1 Static Vectorization Limitations

The Instruction Set Architecture (ISA) of microprocessors contains special instructions that are executed over SIMD accelerators. However, in the early days, compilers were not smart enough to generate these instructions automatically. Therefore, programmers used to target these extensions mainly using in-line assembly or specialized library calls. Later, automatic generation of SIMD instructions (auto-vectorization) was introduced in compilers, which borrowed their methodology from vector compilers.

A recent evaluation of vectorizing compilers by S. Maleki et al. [74] 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. [45] 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 UTDSP benchmark suit [13]. 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.

1.2.2 Wider Vector Units

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. For example, Intel´s MMX [4] had vector length of 64-bits, which was increased to 128-bits in SSE [4] extensions. Intel´s recent SIMD extensions AVX [4] and AVX2 [4] support 256-bit wide vectors. Furthermore, Intel Xeon Phi [12] and its visual computing architecture Larrabee [93] perform 512-bit wide vector operations.

Although SIMD accelerators are amenable to scaling from the hardware point of view, generating efficient code for higher vector lengths 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.

1.2.3 Leakage in SIMD accelerators

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 DLP, 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.

Many leakage control techniques have been studied [46][96][116], power gating [46] being one of the most prominent ones. Power gating cuts the supply voltage to the idle functional units, resulting in leakage energy savings. However, power gating has an energy and performance overhead associated with it. The energy and performance penalty 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.

1.3 Contributions

This thesis focuses on optimizing SIMD execution in HW/SW co-designed processors including efficient code generation and reducing leakage energy of SIMD accelerators.

1.3.1 Speculative Dynamic Vectorization

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. 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.

This work has been published in the Proceedings of 20th International Conference on High Performance Computing (HiPC 2013) [61] as a full length technical paper and in the Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT 2012) [65] as a short paper. Furthermore, we also presented this work at Hipeac Compiler, Architecture and Tools Conference at Haifa, Israel in November 2013 [62].

1.3.2 Vectorizing for Wider Vector Units

We discovered two major problems in generating code for wider vector units: 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, on the other hand, packs maximum number of scalar instructions together, even if the number is less than the number of vector lanes available. Therefore, the dynamic instruction stream coverage for vectorization increases and the dynamic instruction count decreases.

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 consists of two techniques. The first technique eliminates permutation instructions altogether if the result of a scalar instruction is read only by one scalar instruction. The other technique reduces the number of instructions required to pack N values from N-1 to N/2.

This work resulted in a publication in the Proceedings of 15th International Conference on High Performance Computing and Communications (HPCC 2013) [63].

1.3.3 Dynamic Selective Devectorization

Dynamic Selective Devectorization (DSD) is a technique to efficiently power gate higher SIMD lanes. It helps reducing the leakage energy of higher vector lanes and, as a result, of whole SIMD accelerator and the core.

Power gating [46] is a widely used technique to reduce leakage energy consumption of functional units. Power gating cuts the supply voltage to the idle functional units, sending it to sleep state, resulting in leakage energy savings. 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. As a result, the higher lanes can be power gated for longer time intervals without intermittent awakening. Therefore, DSD enables power gating to save more leakage energy.

This work has been published in the Proceedings of the 25th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2013) [64].

1.4 HW/SW Co-designed Processors

HW/SW Co-designed processors [36][39][91] have enticed researchers for more than a decade. Moreover, there is a renewed interest in them in both industry and academia [5][28][29][73][79][87][113]. These processors employ a software layer that resides between the hardware and the operating system. This software layer allows host and guest ISAs to be completely different, by translating the guest ISA instructions to the host ISA dynamically. The host ISA is the ISA which is implemented in the hardware, whereas, guest ISA is the one for which applications are compiled. The basic idea behind these processors is to have a simple host ISA to reduce power consumption and complexity.

The software layer translates the guest ISA instructions to the host ISA in multiple phases. Generally, in the first phase, guest ISA instructions are interpreted. In the rest of the phases, guest code in translated and stored in a code cache, after applying several dynamic optimizations, for faster execution. The number of translation phases and optimizations in each phase are implementation dependent.

1.5 Why HW/SW Co-designed Environment

HW/SW co-designed processors provide some unique features that enable optimized vector code generation. Some of these features include**:**

Support for Speculation and Recovery: HW/SW co-designed processors provide efficient speculation and recovery support. It enables them to apply aggressive and speculative runtime optimizations. We speculatively vectorize the code when dependence between the two memory references is unknown. The hardware support is then leveraged to recover from any speculation failure caused due to speculative vectorization.

Bigger Optimization Regions: Compilers, in general, vectorize at basic block level in the absence loops. On the other hand, runtime optimizations in HW/SW co-designed processors are applied on bigger optimization regions called superblocks. Superblocks include multiple basic blocks following the biased direction of branches. Applying vectorization at superblocks increases the scope of vectorization and hence, available opportunities.

Decoupled ISA and Hardware: HW/SW Co-designed processors decouple the ISA from hardware implementation by means of a software layer. This decoupling allows to modify the hardware transparently to the software stack. As a results, different SIMD accelerators can be targeted without recompiling the code.

Dynamic Optimizations: HW/SW co-designed processors dynamically optimize the program binary at runtime. Therefore, vectorizing the program binary, instead of source code, allows legacy code vectorization.

Online Profiling: HW/SW co-designed processors employ online profiling to discover runtime application behavior. This information about runtime behavior of applications allows more specialized optimizations. For example, runtime application behavior can be used to decide which portions of application can be devectorized for efficient power gating of higher vector lanes without affecting performance.

1.6 Thesis Organization

The rest of the thesis is organized as follows:

Chapter 2 provides the necessary background for the work presented. First we present an historical evolution of microprocessors and SIMD accelerators, followed by a description of different vectorization techniques. Then, we introduce dynamic optimization and different environments where they are applied. Finally, we present an overview of HW/SW co-designed processors and the features they offer.

Chapter 3 provides the details of our simulation environment. First, we present our simulation infrastructure, DARCO that is an in house development in collaboration with other members of the research group. Then, we describe benchmarks used for the evaluation of our proposals. Afterwards, we present the vector instruction set of the host ISA.

The next three chapters illustrate our proposals in detail. Each chapter starts with an introduction and motivation for the work presented. It is then followed by a description of the proposed algorithms or techniques. Thereafter, we present the experimental evaluation results. Finally, we compare out proposals with the related work in the corresponding area.

Chapter 4 explains our proposal of Speculative Dynamic Vectorization to vectorize application at runtime and extract vectorization opportunities missed by compilers due to conservative memory disambiguation analysis. Chapter 5 discusses problems in vectorization at higher vector lengths and provides details of our proposals: Variable Length Vectorization and Selective Writing. Chapter 6 presents the proposed Dynamic Selective Devectorization technique and how it improves power gating efficiency in saving leakage energy in SIMD accelerators. Finally, Chapter 7 concludes the thesis and outlines the future work.

Chapter 2

Background

High computational power requirements have always driven the microprocessor design. The high performance demands have been met through technology scaling and microarchitectural innovations to extract different kind of parallelism from the applications. Parallelism can broadly be divided into the following categories:

Instruction Level Parallelism:

Instruction level parallelism (ILP) takes advantage of sequences of instructions that require different functional units (such as the load unit, ALU, FP multiplier, etc.) for execution. Different architectures approach this in different ways, but the idea is to have these non-dependent instructions executing simultaneously to keep the functional units busy as often as possible.

Thread Level Parallelism:

Thread level parallelism (TLP) is extracted by running multiple flows of execution of a single application simultaneously. TLP is most often found in applications that need to run independent, unrelated tasks (such as computing, memory accesses, and IO) simultaneously. These types of applications are often found on machines that have a high workload, such as web servers. TLP is gaining importance due to the rising popularity of multi-core and multi-processor systems, which allow for different threads to truly execute in parallel.

Data Level Parallelism:

Data level parallelism (DLP) is more of a special case than instruction level parallelism. DLP appears due to the need of performing the same operation on multiple data items simultaneously. A classic example of DLP is performing an operation on an image in which processing each pixel is independent from the ones around it (such as brightening). Other types of operations that allow the exploitation of DLP are matrix, array, and vector processing.

2.1 Microarchitectural Innovations to Exploit Parallelism

A whole body of microarchitectural innovations exists to exploit different kind of parallelisms. Most prominent of these techniques are given below:

2.1.1 Extracting Instruction Level Parallelism

Arguably, the maximum number of microarchitectural innovations have been focused on extracting ILP, especially in general purpose computation domain. These include:

Instruction Pipelining: Instruction pipelining is a technique to overlap the execution of different instructions at the same time [26][31]. An instruction goes through different processor stages during its execution: fetch, decode, register read, execute, memory access, and write back are the most basic stages. The insight behind pipelining is that any given instruction is only in one of these processor stages during its execution while the rests of the stages are idle. Instead of waiting for one instruction to finish its execution and then initiating the next instruction, pipelining initiates next instruction as soon as first instruction leaves the first stage. In other words, pipelining thrives by keeping all the processor stages busy. Hence, pipelining extract parallelism among different phases of instructions by executing them in parallel and boosts overall throughput.

Dynamic Instruction Scheduling and Out-of-Order Execution: Dynamic instruction scheduling decides the order of instruction execution at runtime. It enables instructions bypassing each other during execution, i.e. out-of-order execution, instead of following the strict order in which they are stored in memory. In the absence of dynamic instruction scheduling, the whole pipeline is stalled if an instruction cannot be executed due to any reason like dependence, cache miss, etc. However, dynamic instruction scheduling allows later instructions to bypass the stalled instruction and continue their execution. Scoreboard in CDC 6600 [105] and Tomasulo´s algorithm [108] are two earliest example of dynamic instruction scheduling. Executing instructions in out-of-order fashion increases resource utilization and hence improves performance. However, complex hardware structures are maintained to allow the instructions to execute out of the original program order and track dependencies. This increases hardware complexity and increase power consumption.

Superscalar Processors: These processors consist of multiple pipelines all operating in parallel [16][20]. Having multiple pipelines allow superscalar processors to issue and commit (complete) multiple independent instructions per cycle. These processors incorporate sophisticated hardware to dynamically check dependences among instructions at runtime. This hardware sophistication results in chip complexity and power dissipation, which have become a major problem in modern superscalar processors. Even though superscalar processors incorporate multiple pipelines, the effective use of these pipelines depend on available ILP in the applications. Data dependency among consecutive instructions limits the amount of ILP that can be extracted by superscalar processors. Nonetheless, superscalar processors have been used successfully to extract ILP by executing multiple instructions in parallel.

VLIW Processors: Very Long Instruction Word (VLIW) processors [33][40][76][94] are similar to superscalar processors in the sense that they also incorporate multiple pipelines. However, VLIW processors do not have dynamic instruction scheduling, as do superscalars. Compiler is responsible for scheduling instructions to be executed in different pipelines. This reduces the complexity and high power consumption problems of superscalar processors. However, compiler needs to be aware of microarchitectural details in order to do efficient scheduling and extract ILP. The dependence of VLIW processors on the compiler support makes them a less attractive option. This is one of the reasons behind superscalars being more popular than VLIWs.

Register renaming, branch prediction [38][49][51][84][97][118][119], and speculative execution [48][50][100][101][102] are some of the other well-known microarchitectural techniques to extract more ILP. Register renaming is a technique used to avoid false (anti and output) dependence among instructions. Removing false dependences exposes more ILP. Branch predictor, as the name suggests, predicts outcome of a branch before it is executed so that the control dependent instructions can start their execution without waiting for the outcome of the branch. If the prediction is incorrect, however, there is a penalty to be paid. Speculative execution refers to performing a task before knowing whether it is to be performed or not. Brach prediction is an example of speculative execution. All these techniques focus on boosting performance by exploiting ILP.

2.1.2 Extracting Thread Level Parallelism

Exploiting ILP has driven the microprocessor design for several decades. However, diminishing returns on ILP has led architects to consider exploiting other kinds of parallelism to meet performance requirements. TLP has been targeted in the applications that perform several independent tasks. Microarchitectural supports for extracting TLP can be summarized as:

Multithreading: The basic idea behind multithreading is to execute multiple threads (workloads) on a processor simultaneously. In a multithreaded processor, the hardware resources are shared among a set of threads. However, it requires duplication of some of the resources like registers and program counter, to maintain the thread state. The three main approaches to multithreading are:

Fine-grained Multithreading: This technique issues instructions from a different thread on each clock cycle, skipping the stalled threads [44][53][67]. Since the threads are switched every clock cycle, throughput loss even for short stalls in one thread can be efficiently hidden by executed instructions from the other threads. The flip side of fine-grained multithreading is that it slows down the execution of an individual thread even if does not suffer any stalls. Nonetheless, the technique is effective for improving system throughput.

Coarse-grained Multithreading: Where fine-grained multithreading switches between threads on every clock cycle, coarse-grained multithreading switched them only for costly stalls, like L2 or L3 cache misses [14]. Therefore, slowdown in individual threads is going to be less, if they do not have costly stalls. However, thread switching only for costly stalls prevents coarse-grained multithreading to compensate for throughput loss due to short stalls.

Simultaneous Multithreading: Simultaneous multithreading technique [75][111] is a special case of fine-grained multithreading implemented in multiple issue, dynamically scheduled superscalar processors. Fine- and coarse-grained multithreading, even though switch threads, issue instructions from a single thread in one cycle. As a result, some of the issue slots may remain empty due to lack of ILP. Simultaneous multithreading solves the problem by issuing instructions from multiple threads in the same cycle. Therefore, the number of empty slots reduces and system throughput increases.

Multiprocessor: Multiprocessors [42] consist of a set processors sharing memory and peripherals. TLP is extracted by executing independent threads on different processors. Comparing to a single multithreaded processor, multiprocessors duplicate the entire processor, whereas multithreading requires duplication of only the private state such as the registers and program counter. Moreover, each processor within the multiprocessor can also support multithreading, thereby, increasing the total number of threads that can be executed on the multiprocessor. If all the processors in a multiprocessor are on the same chip, it is called chip multiprocessor or CMP. Depending upon the memory organization, multiprocessors can be categorized as either shared memory multiprocessors or distributed memory multiprocessors.

Centralized Shared-Memory Multiprocessors typically consist of small number of processors sharing a centralized memory. Since the memory is centralized, all the processors have uniform memory access latency, therefore, this configuration is called Uniform Memory Access (UMA) multiprocessors as well. UMA is suitable only for multiprocessors with few cores because it is not practical for a centralized memory to support the bandwidth requirements of large number of cores.

Distributed Shared-Memory Multiprocessors can support larger processor count, since the memory is physically distributed among the processors. The memory bandwidth increases as a result of distributed memory, however, it also means that the memory access latency is not uniform, rather depends on the location of the processor that has the requested data. Therefore, these multiprocessors are also called Non Uniform Memory Access (NUMA) Multiprocessors.

2.1.3 Extracting Data Level Parallelism

A set of architectural extensions exists to specifically extract data level parallelism from applications from different computing domains like multimedia applications, scientific computing, etc. These extensions include vector processors, graphics processing units (GPU), and SIMD accelerators. The basic idea behind all of these extensions is to perform the same operation on multiple data elements by a single instruction. In addition to the performance benefits, this also results in energy efficiency because the front-end of the microprocessor has to handle less instructions.

Vector Processors: Vector processors [43][114] are especially designed to efficiently execute data parallel applications. These processors work with long vectors, a typical example is 64 element vectors. The elements in a vector are processed in a pipelined manner, including memory load/store. These pipelined load/stores help vector processors in hiding memory latency. However, these architectures are expensive mainly because of high memory bandwidth requirements.

Graphics Processing Unit (GPU): The last decade saw the appearance of GPUs [2][6][7] as high performance graphics accelerators. However, due to their high throughput they are gaining popularity in general purpose computing domain as well. GPUs consist of highly parallel hardware that support single instruction multiple data execution model. They also support huge number of threads executing in parallel. These architectures sometimes are referred to as heterogeneous architectures because they have a system processor and system memory in addition to the GPU itself. The main challenge in GPUs is optimizing the communication between system memory and GPU.

SIMD Accelerators: In the mid-90´s SIMD accelerators [4][23][37] [68] emerged mainly to accelerate multimedia applications on general purpose processors. Like vector processors, SIMD accelerators perform the same operation on multiple pieces of data using one instruction. However, the vector length of SIMD accelerators is typically much smaller than the maximum vector length of vector processors. Furthermore, some of the fundamental features of vector processors like variable vector length, strided memory access, gather-scatter, and mask registers are generally omitted in SIMD accelerators to simplify the design. This, however, makes it difficult for compilers to generate code for them. Therefore, some of the latest SIMD accelerators have started to incorporate these features in their ISA gradually with each new generation.

Due to their simplicity, performance boosting ability, and energy efficiency SIMD accelerators form an integral part of processors from different computing domains. Even though GPUs have greater compute potential, they work as a coprocessor, whereas SIMD accelerators form part of processor´s pipeline. This fact allows them to speed up the execution of more general purpose applications. GPUs and vector processors, on the other hand, target a specific kind of applications that have significant amount of explicit DLP. Moreover, GPUs have specific programming language requirements and require programmers to explicitly expose DLP through programming language notations. Whereas, SIMD accelerators do not have any such requirements. Therefore, this thesis focuses on extracting DLP through SIMD accelerators instead of GPUs or vector processors.

2.2 SIMD ISA Extensions

SIMD accelerators [4][35][23][37][52][68][103][109] are ubiquitous in microprocessors from different domains ranging from supercomputers to smart phones. These accelerators are specially designed to target computation intensive scientific applications, multimedia applications like image processing, signal processing, graphics, audio and video encoding, etc. These applications consists of compute intensive data parallel kernels where each element can be processed in parallel. For example, in image processing, properties of individual pixels can be modified in parallel. To efficiently execute these kernels, SIMD accelerators provide multiple functional units all performing the same operation as shown in Figure 2.1.

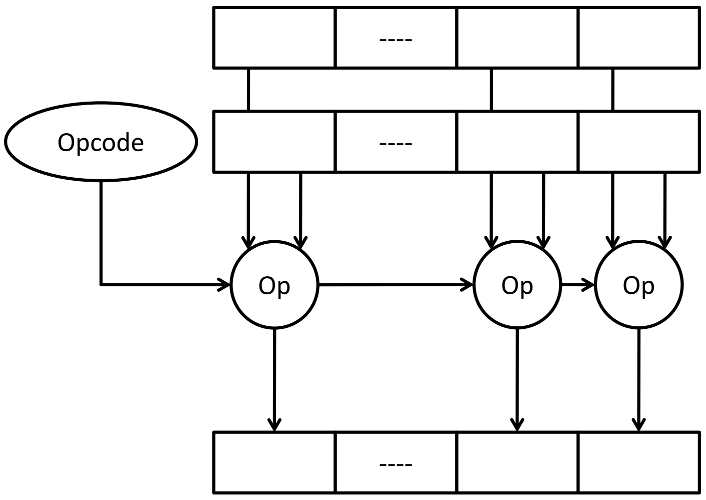


Figure 2.1: SIMD execution model.

Microprocessors have special extensions to their Instruction Set Architecture (ISA) and hardware state (e.g. new registers) to support SIMD execution. Following are some of the representative SIMD extensions in current microprocessors.

2.2.1 Intel´s SIMD Extensions

Intel´s SIMD extensions [4] have gone through major transformations since they were first introduced in 1996. Intel introduced MMX as 64-bit wide multimedia extensions for their Pentium processors. MMX supports 8-bit, 16-bit, and 32-bit vectors. This means that the 64-bit wide vector path can perform eight 8-bit, four 16-bit or two 32-bit operations in parallel. Following the IA-32 semantics, MMX implements two-operand destructive instruction encoding, where one of the source registers is also the destination register. However, MMX does not include floating-point vector instructions.

MMX defines eight 64-bit vector registers named MM0-MM7. However, these registers are mapped to x87 floating-point registers to avoid saving extra state on context switches. Therefore, this register aliasing prohibits MMX and x87 code intermixing. The register values need to be saved and retrieved before switching between MMX and x87 code.

Streaming SIMD Extensions (SSE): Intel introduced SSE [4] in 1999 in Pentium III to overcome the limitations of MMX. The vector length was increased to 128-bits from 64-bit of MMX to boost performance. SSE introduced 70 new instructions to perform single-precision floating-point arithmetic operations, memory load/store, comparison, data shuffling, and data type conversion between integer and floating-point. Moreover, SSE introduced special instructions for cacheability control, prefetch, and memory ordering. The vector single-precision floating-point instructions of SSE perform four single-precision floating-point operations in parallel, whereas the scalar floating-point instructions perform only one operation. Therefore, the scalar instructions in SSE provide an option to avoid the complex x87 Floating Point Unit (FPU) to execute floating-point code even without vectorization. The cacheability control instructions provide a fine control over when and what data to bring to caches.

SSE also introduced a separate vector register file with eight 128-bit vector registers (XMM0 – XMM7) and a 32-bit status and control register MXCSR. Having a separate register file resolves the register aliasing problem of MMX. The ability to intermix SSE and MMX code provides greater flexibility and throughput for applications operating on large arrays of floating-point and integer data.

SSE2: SSE2 [4] was introduced in 2001 with Intel Pentium IV processors. SSE2 improved on existing MMX and SSE extensions by adding 144 new instructions. SSE2 extensions featured two major enhancements: 1) Double-precision floating-point instructions and 2) 128-bit SIMD integer instructions. MMX and SSE both lacked double-precision floating-point support. Its inclusion in SSE2 enabled high precision computations in SIMD unit and as a result, better performance for scientific and engineering applications. MMX had 64-bit vector integer instructions and even though SSE supports 128-bit wider vectors, it does it only for floating-point instructions. SSE2 extended 128-bit vector support for integer instructions as well. Furthermore, SSE2 improved upon the cacheability control and prefetching support introduced in earlier SSE extensions.

SSE3/SSSE3/SSE4: SSE3 [4] was introduced in 2004 with Pentium IV processor with Hyper Threading (HT) technology. SSE3 introduced 13 new instructions in different categories. It included floating point instructions that support packed addition/subtraction and horizontal addition/subtraction. The packed addition/subtraction instructions perform addition on one element of the vector register and subtraction on the other element. The horizontal addition/subtraction instructions perform the operation on the different elements of the same source register instead of performing it on the corresponding elements of different source registers. Furthermore, two thread synchronization and one integer unaligned load instruction was also introduced in SSE3.

Supplement Streaming SIMD Extensions (SSSE3) was introduced in 2006 with Intel Core 2 processor family. It introduced 32 new instructions mainly to accelerate multimedia and signal processing applications with integer data. It extended the horizontal addition/subtraction support of SSE3 to integers as well. Further, SSSE3 introduced new instructions like absolute value evaluation, multiply and add for dot products, negating packed integers, etc.

SSE4 consists of two extensions: SSE4.1 and SSE4.2. The 47 new instructions of SSE4.1 are targeted at improving the performance of media, imaging, and 3D applications whereas SSE4.2 focuses on string and text processing.

Advanced Vector Extensions (AVX): AVX [4] is Intel´s 256-bit SIMD extension supported by Sandy Bridge architecture. A major difference between AVX and earlier SIMD extensions, apart from the vector length, is the instruction encoding. AVX uses a three operand non-destructive instruction encoding. It helps in reducing the number of register-register copy operations.

Further, AVX enhances the 128-bit floating point instructions to operate on 256-bits. However, vector integer instructions do not have a 256-bit version. Apart from enhancing existing instructions to 256-bits, AVX offers several new instructions for non-unit-stride fetching of data, intra-register manipulation of data, etc. AVX also introduces a new register file consisting of eight 256-bit registers YMM0 –YMM7. However, the lower 128-bits of the YMM registers are mapped on to XMM registers.

FMA and AVX2 Extensions: Fused-Multiply-Add (FMA) [4] extensions include various combinations of fused instructions to further improve the performance. The instructions in this extension includes fused multiply-add, fused multiply-subtract, fused multiply add/subtract interleave, and signed-reversed multiply on fused multiply-add and multiply-subtract. All these instructions can operate on single-precision and double-precision floating-point data. Moreover, scalar and vector (packed) version of all these fused operations are available except for fused multiply add/subtract interleave that has only the vector version.

Advanced Vector Extensions 2 (AVX2) [4] promotes the majority of integer instructions to support 256-bit vector operations, whereas AVX supports 256-bit vector operations only on floating-point data. Moreover, AVX2 provides gather support to fetch data from non-contiguous memory locations. It also includes vector shift instructions with per element shift count. Furthermore, it improves the broadcast and permutation functionality of AVX instructions.

Intel´s 512-bit Vector Extensions: Intel´s Many Integrated Core (MIC) architecture and products based on it, like Xeon Phi [12], support 512-bit wide vector instructions. Apart from wider vectors, this architecture supports masked vector execution that enables vectorizing short conditional branches. This improves overall efficiency of vector processing unit. Furthermore, MIC supports gather and scatter instructions to access non-unit stride memory accesses. Vectorizing irregular memory access patterns helps extracting additional vectorization opportunities and improves overall performance.

2.2.2 PowerPC Altivec

Altivec [37] SIMD extensions were the result of a joint effort from Apple, IBM, and Motorola in late 90´s. The name Altivec, however, comes from Motorola. IBM called these extensions VMX, whereas Apple referred to them as Velocity Engine. Like SSE, Altivec supports a vector length of 128-bits, thought it was launched before SSE. It included a rich set of both integer and floating-point vector instructions; however, support for double-precision floating-point is absent. Furthermore, Altivec implements non-destructive instruction encoding.

Altivec has a vector register file with thirty-two 128-bit registers compared to eight 128-bit registers of SSE (though x86-64 has sixteen 128-bit registers). A notable difference between Altivec and SSE is that Altivec does not have scalar floating-point instructions like SSE. Scalar code has to be executed on a separate floating-point unit (FPU). Furthermore, there is no instruction for moving data between vector and floating-point register files.

Altivec provides some very flexible data permutation instructions like *vperm*. It allows each byte of a resulting vector to be taken from any byte of either of two other vectors, parameterized by yet another vector. *vsel* is another permutation instruction that operates at bit level. In addition, Altivec offers vector compare instructions and select mechanism to allow control flow in the vectorized code. Like SSE, Altivec also provides instructions for cacheability control to avoid polluting the cache with non-temporal data.

2.2.3 ARM Neon

Neon is 128-bit (or alternatively 64-bit) wide SIMD extension for ARM processors. Neon provides a rich set of integer and floating-point vector instructions. It supports scalar single-precision and double-precision computations. However, only single-precision floating-point instructions have vector equivalents. Neon also provides scalar and vector half-precision conversions instructions. Like Altivec, it uses a non-destructive instruction encoding for most of the instructions.

Neon has a vector register file that can be viewed as: 1) Thirty-two 64-bit registers D0-D31 or 2) Sixteen 128-bit register Q0-Q15. In other words, each D register is one half, either lower or higher, of a Q register. Alternatively, a Q register is composed to two D registers. For example Q0 is the same register as D0 and D1 combined. This dual view of registers allows to promote/demote the elements within a single operation. A "promotion" doubles the precision of the elements, for example instructions like VMULL, VADDL, etc. promotes elements from 16-bits to 32-bits. These instructions read the source operands from D registers and place the results in the destination Q register. Similarly, a "demotion" reduces the precision by reading input operands from Q registers and placing the result in D register, e.g. VADDHN. Furthermore, there are instructions that promote only the second operand like VADD, VSUB, etc.

In addition, Neon instruction set allows vector operations with scalar values. In this case, the scalar is used for all the operations instead of having to operate on corresponding elements of the vector registers. It has the same effect as first duplicating the scalar to all the elements of a vector register and then performing a vector-vector operation. Neon also provides a variety of memory load/store instructions like broadcasting, interlaced load/store, inserting/extracting to/from a particular element of a vector register, etc.

2.3 Code Generation for SIMD Accelerators

Despite their hardware simplicity, the code generation for SIMD accelerators is a challenging task. Initially, SIMD extensions were targeted using inline assembly or specialized library routines. Even though these solutions are capable of utilizing the SIMD accelerators, they have their own limitations. For example, they are time consuming, error prone, and require the programmers to have in-depth knowledge of the underlying SIMD architecture. Moreover, the code is not portable between different SIMD accelerators. These limitations led the compiler technology to target these extensions automatically. Most of the compiler vectorization techniques operate at source code level. However, S. Larsen et al. [66] introduced a technique that operates at low-level intermediate code. Since then, there have been other proposals that operate at lower level. This section reviews the existing compiler approaches to vectorization.

2.3.1 Traditional Compiler Vectorization

Compiler vectorization traditionally targets loops for vector code generation. The vectorizer, first of all, strip-mines the loop iteration space by vector length *vl*. For example, on a 128-bit wide SIMD accelerator, for a loop operating on 32-bit variables, the iterations space is strip-mined by 128/32 = 4. Then a vectorized version of the loop is generated along with some pre- and post-vectorization steps. The following scalar loop

```
            for (i = 0; i < n; i++)  { //scalar statements }
```

will be vectorized by the compiler as shown below:

```
            i = 0;
prelude:    if (runtime-test)   goto cleanup;
            m = n - (n % vl)
            …….

vector loop: for ( ; i < m; i += vl) { //vector statements }

postlude:   ……

cleanup:    for ( ; i < n; i++) { //scalar statements }

exit:
```

As shown above, the vector version of the loop has several parts: prelude, vectorized code, postlude, cleanup, and exit. Prelude does some pre-processing steps for vectorization. For example, it might include a runtime test to check for number of iterations if loop trip count is not known statically or a runtime test to check multiple arrays for aliasing. In both of these cases, the vectorized loop will not be executed if the test fails. "Vectorized code" section includes the vectorized version of the loop which is executed if runtime tests in prelude succeed. Postlude is used for post-processing after executing vectorized code. One example of this post-processing is generating the final result of a *reduction* after executing vectorized code. The cleanup section consists of scalar version of the loop. Cleanup is necessary when the loop trip count is not evenly divided by vector length. Therefore, cleanup executes iterations that remain after executing vector loop. Finally the loop exits and the rest of the application is executed. It is important to note that not all of these sections are needed for all the vectorized loops. For example, if number of iterations is evenly divided by the vector length, we might not need cleanup.

2.3.2 Superword Level Parallelism

Superword Level Parallelism (SLP) is defined as short SIMD parallelism in which the source and result operands of a SIMD operation are packed in a storage location [66]. The technique works at low-level intermediate code level rather than operating at source code level like conventional vectorization approaches. Traditional vectorization schemes target loops for generating vectorized code, whereas SLP targets parallelism within a basic block, possibly after loop unrolling. Therefore, SLP can vectorize parts of a loop, if the loop is not completely vectorizable. Moreover, SLP avoids complex loop transformations like loop fission and scalar expansion, while it is still able to extract equal or even more parallelism.

The technique first unrolls a loop to convert vector parallelism into superword level parallelism. Then, SLP starts vectorization by locating adjacent memory references and packing them together in groups of two instructions. Then they follow def-use and use-def chains of these initial packs and newly created packs, if any. These groups are then merged together to form bigger groups depending on the SIMD accelerator width. The vectorized code is then scheduled and groups involved in acyclic dependences, if any, are eliminated. The technique also includes a cost model to estimate the profitability of vectorization based on the number of packs created and the cost of permutation instructions generated.

2.3.3 SLP in Presence of Control Flow

Since the technique presented by S. Larsen works at basic block level, it fails to vectorize loop with control flow. To overcome this limitation, J. Shin et al. [95] extended SLP in the presence of control flow. The basic idea behind their technique is to execute both if and else parts of an “if statement” in vector form and then choosing the correct result based on the outcome of the control instruction.

Their mechanism, like SLP, starts by unrolling a loop. Then, if-conversion is applied to convert control dependences into data dependencies. However, each instruction, that was control dependent earlier, now has a predicate associated with it. The predicate might be *true* or *false* depending upon the outcome of the control instruction. Therefore, if-conversion removes control dependences and creates a bigger basic block with predicated instructions.

The modified SLP algorithm takes this basic block as input for vectorization and produces an output that is a mix of predicated scalar and superword (packed) instructions. If the underlying architecture supports predicated instructions, nothing more needs to be done. However, if it does not, then the predications need to be removed either by restoring the control flow or by mapping them to the features provided by the underlying architecture. For example, predicated superword instructions can be mapped to the vector select instruction *vsel* of Altivec.

2.3.4 Speculative Dynamic Vectorization

Both of the last two vectorization schemes that we have seen, work on low-level intermediate format. A. Pajuelo et al. [83] proposed yet another vectorization scheme that vectorizes the applications at binary level. Instead of compiler vectorization, they proposed a microarchitectural extension for superscalar processors that speculatively vectorizes the program binary. The basic idea is to predict when certain operations are vectorizable and generate vector code that speculatively fetch and precompute data using the SIMD accelerator.

The vectorization begins when a scalar strided load instruction is detected. Next, a vectorized version of this instruction is generated/executed that speculatively loads into a vector register assuming that the future instance of this instruction will be executed with same stride and without any intermediate store to the corresponding locations. Next occurrences of the given load instruction are used to validate the assumptions that were made speculatively. Arithmetic instructions are vectorized when any of the source operands is already vectorized. As with load instructions, the rest of the instances of the scalar arithmetic instructions are used to check that the corresponding source operands are still valid for vector operation. Speculation failure triggers a recovery mechanism and execution in non-speculative scalar mode.

Speculative Dynamic Vectorization is able to get away with the complexities encountered by compiler. Moreover, legacy code can also be vectorized using it. However, the cost of speculation failure can be high. A. Pajuelo et al. [83] reported that more than half of speculative work was useless due to misspeculations. This is also important from energy efficiency point of view, which is already a big issue in superscalar processors.

2.3.5 Liquid SIMD

Liquid SIMD [34] targets the problems of binary compatibility, software migration cost, and redesigning of SIMD ISA that arise due to increased functionality and larger vector width with each new generation of SIMD accelerators. Liquid SIMD decouples the SIMD ISA from the SIMD accelerator hardware by means of delayed binding. The delayed binding is achieved by a combination of compiler support and translation system.

The compiler support is used to translate vectorized SIMD instructions to a virtualized representation using the processor´s baseline instruction set. Moreover, compiler adds vectorization hints in the translated code, so that the translation system can easily discover the vectorizable code. The conversion can be done at compile time or as a post conversion step. Moreover, the conversion applies to both complier vectorized code and hand coded in-line assembly.

The principle task of the translation system is to convert the virtualized code to equivalent SIMD representation targeted at the specific implementation of the SIMD accelerator. The translation system could be a binary translator, just-in-time compiler, or a hardware extension; each with its own pros and cons.

2.3.6 Vapor SIMD

Vapor SIMD targets portable vectorization across disparate SIMD accelerators with different vector lengths [82]. It opts for a split-compilation approach where the final machine code is generated by a combination of two separate yet synergistic compilation phases. The first phase is an aggressive and generic offline compilation stage that generates an optimized target independent intermediate code. The second phase is an online just-in-time compilation stage that reads the optimized target independent intermediate code and generates target specific vectorized SIMD code.

2.4 Code Optimizations

Section 2.1 presented a summary of most commonly used microarchitectural techniques to improve the performance. Apart from these microarchitectural innovations, a whole body of optimizations exists that plays an important role in meeting the high performance requirements.

Optimizations transform a piece of code to make it more efficient. The efficiency may come in the form of better performance, smaller memory/cache size requirements, energy efficiency, etc. Moreover, optimizations must not change the output of a program, or in other words, the output remains the same with and without optimizations. However, this may not be true for incorrectly written programs e.g. uninitialized variables. The only effect of optimizations is that the optimized program runs faster and/or consumes less energy and/or consumes less memory, etc. Code optimizations can be broadly categorized as:

1) Static Compiler optimizations.
2) Dynamic Binary optimization.

2.4.1 Static Compiler Optimizations

The primary task of compilers is to generate machine code from higher level source code. While generating the machine code, the prime responsibility of the compiler is to ensure correctness and then whatever extra benefit that may come from the optimizations. Compiler optimizations can be divided into two categories: 1) Local Optimizations and 2) Global Optimizations.

Local Optimizations: Local optimizations are performed within the scope of a basic block. These are relatively easier to perform since there is no control flow to be worried about. Some of the most common local optimizations are:

1) Constant Folding.
2) Constant Propagation.
3) Copy Propagation.
4) Common Sub-expression Elimination.
5) Algebraic Simplification.
6) Operator Strength Reduction.
7) Dead Code Elimination.

Global Optimizations: The scope of Local Optimizations is limited to a single basic block. By widening this scope, more optimization opportunities can be discovered. This is exactly what Global Optimizations do. By doing some additional program analysis, they expand the scope and apply local optimizations across multiple basic blocks. Global optimizations typically optimize a whole function at a time. In addition to increasing the scope of local optimizations, following global optimizations further improve the generated code quality:

1) Loop Invariant Code Motion.
2) Register Allocation.
3) Instruction Scheduling.
4) Peephole Optimizations.
5) Machine Code Optimizations.

2.4.2 Dynamic Binary Optimizations

As the prime responsibility of compilers is to ensure correctness, they optimize the code conservatively to ensure correctness. Furthermore, the lack of knowledge of the runtime program behavior also restricts compilers ability to apply aggressive optimizations. Dynamic binary optimizers (DBO), on the other hand, optimize the program binary at runtime. This allows them to profile the code to understand the dynamic program behavior and, accordingly, apply aggressive, even speculative, optimizations. However, certain support is required either in hardware or software to recover from speculation failures, if any.

Dynamic binary optimizers profile the program binary on the fly to determine what and when to optimize. This dynamic profiling leads to optimizations that are not coupled to one or more particular program inputs, as would be the case for offline profiling. The profiling provides, among others, two important characteristics of execution: 1) the most frequently executed portion of the code and 2) the most frequently followed paths in the execution i.e. the sequence of execution of basic blocks. In order to keep the optimization overhead minimum, as it is done on the fly, only the most frequently executed basic blocks are optimized. Moreover, the knowledge of most frequently followed paths helps in applying local and global optimizations described above only across the basic blocks that follow each other during execution.

Dynamic binary optimizers can be divided into two categories depending on whether they are implemented in software or hardware:

1) Software Dynamic Binary Optimizers.
2) Hardware Dynamic Binary Optimizers.

Software DBOs are a software layer that intercepts the program binary, before it starts execution, to profile and later optimize it. However, they might need some hardware support in order to apply some aggressive optimizations like speculative instruction reordering. Software DBOs introduce a certain performance overhead, since they share the execution time with applications. However, they thrive by compensating this overhead by performing aggressive code optimizations to improve overall performance. The examples of Software DBOs include Dynamo [21], IA-32 EL [22], Strata [92], DynamoRIO [30], etc.

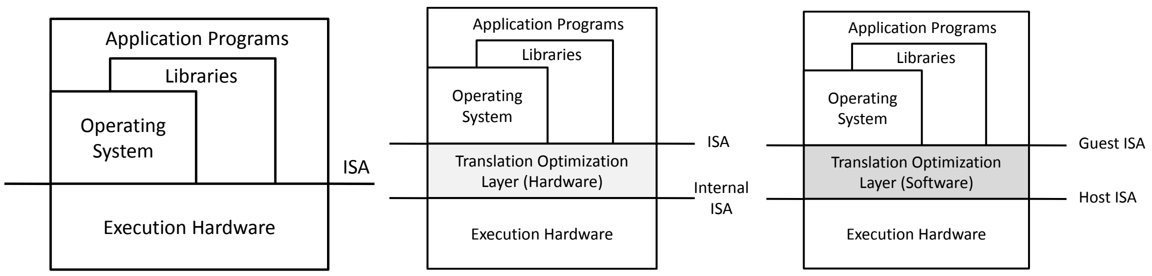
Hardware DBOs are completely implemented in hardware; though their presence is not reflected in the ISA. Due to their hardware implementation, they do not introduce performance overhead like Software DBOs. However, the down side of Hardware DBOs is increased hardware complexity and more energy consumption. rePLay [85] and PARROT [90] are the most representative Hardware Dynamic Binary Optimizers.

Apart from Software and Hardware DBOs, dynamic optimizations are also employed by Hardware/Software Co-designed Processors as explained in the next section.

2.4 Hardware/Software Co-designed Processors

HW/SW Co-designed processors [36][39][91] have enticed researchers for more than a decade. Moreover, there is a renewed interest in them in both industry and academia [5][28][29][73][79][87][113]. These processors are specially designed to achieve energy efficiency, design simplicity, and performance improvement. In order to achieve design simplicity, they keep the hardware simple and implement a relatively simple ISA. The simple hardware design also helps in achieving energy efficiency. Transmeta reports significant reduction in power dissipation for their HW/SW co-designed processor Crusoe compared to Intel Pentium III for a software DVD player [57]. Their data shows that Pentium III heats up to a temperature of 105º C whereas Crusoe maximum temperature goes only up to 48º C running the same software DVD player. Furthermore, to achieve the performance goal, HW/SW co-designed processors employ dynamic binary optimizations.

In general, HW/SW co-designed processors implement a proprietary ISA in order to achieve design simplicity and power efficiency. Therefore, they need to apply binary translation to map the guest ISA on to the host ISA. We define host ISA as the ISA which is implemented in the hardware, whereas, guest ISA is the one for which applications are compiled. The binary translation can be implemented in either hardware or software. Modern processors implementing CISC ISA, like x86, implement binary translation in hardware [99]. The hardware binary translator translates CISC instructions to RISC like instructions dynamically to simplify the execution pipeline implementation. However, the hardware implementation leads to significant complexity and power consumption. HW/SW co-designed processors, on the other hand, implements binary translation in software which leads to power efficiency.



a) Conventional RISC processor. b) Conventional CISC processor. c) HW/SW co-designed processor.

Figure 2.2: HW/SW interface in processors.

Figure 2.2a shows the hardware/software interface in a conventional RISC processor where the software stack directly interacts with the hardware. Conventional

CISC processors implements a RISC like ISA in hardware. As shown in Figure 2.2b, they employ a hardware dynamic binary translator to translate CISC instructions to the internal ISA instructions. The binary translation in HW/SW co-designed processors is performed by a software layer as shows Figure 2.2c. We call this software layer as Translation Optimization Layer (TOL) in this thesis. Doing the binary translation/optimization in software layer provides several benefits over the hardware implementation. For example, the software implementation significantly reduces hardware complexity and power consumption. Furthermore, it allows to upgrade a processor in the field by introducing new optimizations in the software layer. On the contrary, if TOL is implemented in hardware, adding new optimizations in the existing processor is not feasible. Additionally, software implementation of TOL significantly reduces hardware validation and verification cost and time.

In HW/SW co-designed processors, TOL resides in a ROM and is the first program to start execution when system boots up. Since TOL acts as in insulation layer between the conventional software stack and the hardware, the host ISA can be changed arbitrarily without having to make changes in the conventional software stack. The only modification needed in this case would be having a new version of TOL that translates guest ISA code to the new hardware. Since the execution of TOL itself requires some processor time, it might affect the overall performance. However, in addition to binary translation, TOL is also responsible for optimizing the translated binary to boost the performance and compensate for its own execution overhead.

2.4.1 Memory System in HW/SW Co-designed Processors

As shown in Figure 2.3, the system memory in HW/SW co-designed processors is divided into two parts:

1) Conventional memory.
2) Concealed memory.

The conventional memory part works as a normal memory system of a hardware only processor. All the guest ISA application code, operating system, and data reside in this part. Moreover, this is the only part of the memory that is visible to the conventional software stack including operating system.

The concealed memory contains TOL binary, TOL data, and code cache. Code cache is the memory where TOL stores translated and optimized user/system code. Since TOL takes control as soon as the system boots up, it hides the concealed memory from

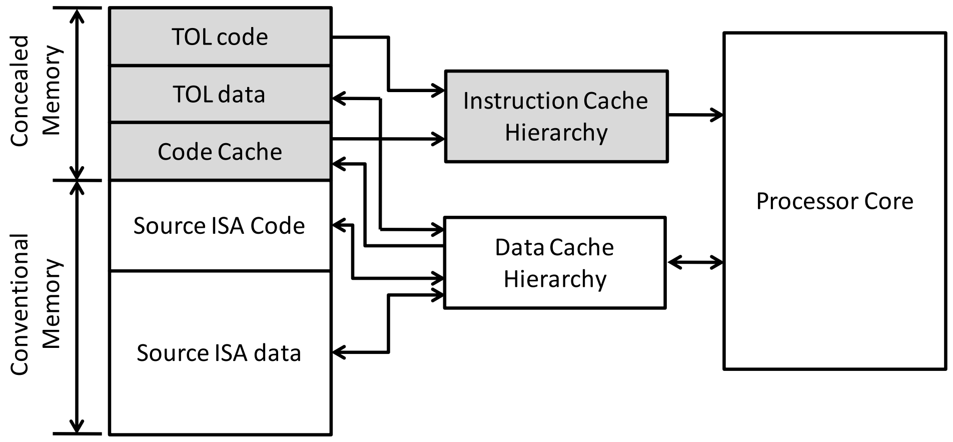


Figure 2.3: Memory system in a HW/SW co-designed processor.

conventional software stack. Any attempt to access the concealed memory by the conventional software stack results in an invalid memory access.

It is also important to note that only the TOL code and translated/optimized code from the code cache enter the instruction cache hierarchy. The source ISA code goes to data cache hierarchy, since it acts as data for TOL translator. Thereafter, TOL translates and stores this code in code cache. Then, it is read from the code cache, through the instruction cache hierarchy for execution.

2.4.2 TOL translator/optimizer

As said before translating guest ISA code to host ISA is the prime responsibility of TOL. The translation is done dynamically and generally, in multiple phases. Usually, in the first phase, an interpreter decodes and executes guest ISA instructions sequentially. In the rest of the phases, guest code in translated to host ISA code and stored in the code cache, after applying several dynamic optimizations, for faster execution. The number of translation phases and optimizations in each phase are implementation dependent.

Figure 2.4 shows a typical two stage translation/optimizations flow in a TOL. It starts by interpreting guest ISA instruction stream sequentially. While interpreting, TOL also profiles the guest code to collect information about most frequently executed code and biased branch directions. The execution frequency guides TOL to decide which guest code basic blocks to translate. When a basic block has been executed more than a predetermined number of times, TOL invokes the translator. The translator takes the guest ISA basic blocks as input, translates them to host ISA code and saves the translated code into the code cache for fast native execution. Instead of translating and optimizing each basic block in

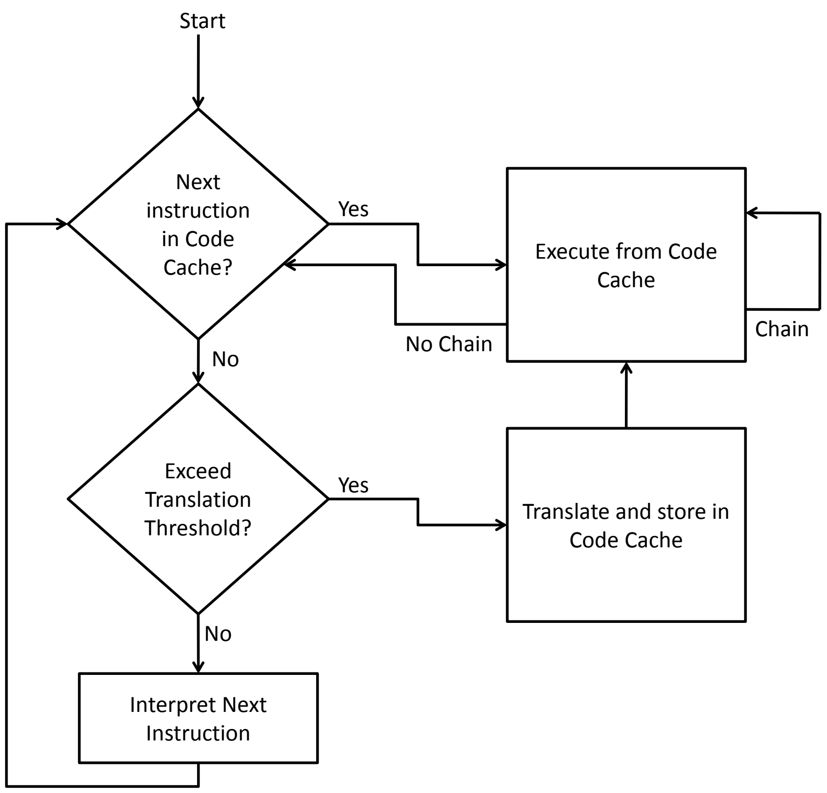

Start
Next instruction in Code Cache?
Yes
No
Execute from Code Cache
No Chain
Chain
Exceed Translation Threshold?
Yes
No
Translate and store in Code Cache
Interpret Next Instruction

Figure 2.4: Typical two stage TOL control flow.

isolation, the translator uses biased branch direction information, collected during interpretation, to create bigger optimization regions, called superblocks. A superblock, generally, consists of multiple basic blocks following the biased direction of branches. Therefore, superblocks increase the scope of optimizations to multiple basic blocks and allow more aggressive optimizations. Superblocks have a single entry point that is the first instruction of the first basic block included in the superblock. However, depending on the implementation they might have multiple or a single exit point, making them a single-entry multiple-exit or single-entry single-exit structure.

Initially, the control is transferred back to TOL after executing a superblock from the code cache. Then, TOL searches the next instructions to be executed. If the next instruction is not already translated, it has to be interpreted. However, if it is already translated, TOL patches the last branch of the first superblock (the one that transferred the control back to TOL) to the beginning of the second superblock. This process is called chaining [36] or linking [21]. Chaining enables the control to be transferred directly from one superblock to the other without having to come back to TOL. This reduces TOL overhead of looking up a translation in the code cache.

The example binary translation/optimization mechanism that we just saw has two stages: interpretation and one translation phase. However, there are systems with more translation stages, with each translation stage applying progressively more complex optimizations. Moreover, there are some systems that skip interpretation stage and directly go to translation like IA-32 EL [22] and DynamoRIO [30]. The different interpretation/translation stages provide a tradeoff between startup and steady state performance. For example, applying aggressive optimizations is costly in terms of overhead; however, they generate a highly optimized code that runs faster than un-optimized code. Hence, a system that starts with aggressive optimizations, skipping interpretation and simple translation, would have unacceptably poor startup performance and excellent steady state performance. However, the overall performance of such a system would depend on how much of the startup delay or translation/optimization overhead could be offset by the optimized code execution. They might end up having poor overall performance if the translation/optimization overhead in not compensated by the optimized code execution. Therefore, most systems start with interpretation or lightweight translations to improve startup performance, whereas aggressive optimizations are applied only to hot code that dictates the steady state performance.

2.4.3 Hardware Support in HW/SW Co-designed Processors

Hardware support is needed for efficient and correct execution of the guest ISA instructions on the host architecture. Memory speculation is the key to several optimizations performed by HW/SW co-designed processors. For example, Transmeta Crusoe [36] reports that, on average, suppressing memory reordering causes 10% and 33% performance loss in operating system boots and user applications respectively. To ensure the correctness of memory speculation, hardware support is provided to detect speculation failure and recover from it. Transmeta shadows all the registers holding x86 state i.e. they keep a working and a shadow copy of each register [57]. During the execution, only the working copy is updated. If the execution reaches the end of a translation without speculation failures or exceptions, a special commit operation copies the working register into the corresponding shadow registers. To ensure the correctness of memory state, Transmeta employs a “gated store buffer”. The memory store instructions write their results to this buffer which are forwarded to memory only if a translation finishes without exceptions and speculation failures. BOA [91] also incorporates similar techniques by having two sets of registers. However, they schedule stores in the original program order.

Furthermore, hardware support is necessary for providing precise exceptions and detecting self-modifying code. Moreover, overhead of indirect branches and function returns can be reduced by having some hardware support [54][55].

2.4.4 Salient features of HW/SW Co-designed Processors

HW/SW co-designed processors provide certain features that set them apart from traditional hardware only processors. These features include:

Hardware Simplicity: These processors employ simple hardware to cut down the complexity. To simplify the hardware they implement a simple RISC ISA. Furthermore, TOL is implemented as a software layer whereas, the hardware implementation of TOL in conventional CISC processors contributes significantly to the hardware complexity.

Power Consumption: Having a simple hardware allows HW/SW co-designed processors to keep power consumption within limits. The simple RISC ISA allows to have a simple front-end and avoid power hungry components.

Flexibility: The software implementation of TOL makes it relatively simple to upgrade a processors by introducing new features in the software layer, in the field. On the other hand, due to the hardware implementation, the conventional CISC processors cannot introduce new features in TOL or fix a bug once the processor is rolled out.

Multiple Guest ISA Support: HW/SW co-designed processors also provide an excellent opportunity to run multiple guest ISA on a single host ISA. In this case, TOL needs to support multiple front-ends where each front-end corresponds to a different guest ISA. Once a front-end has done guest ISA to intermediate representation translation, the common back-end can be used to generate the host ISA code. This feature allows to execute codes complied for different architectures to be executed on the same hardware. This is going to be especially important in the future architectures as one would like be able to run any application on any computing device.

Binary Compatibility: TOL also allows HW/SW co-designed processors to maintain forward and backward binary compatibility without any additional hardware complexity. TOL can translate binaries targeted for old architectures to run them on a latest one and vice-versa.

Chapter 3

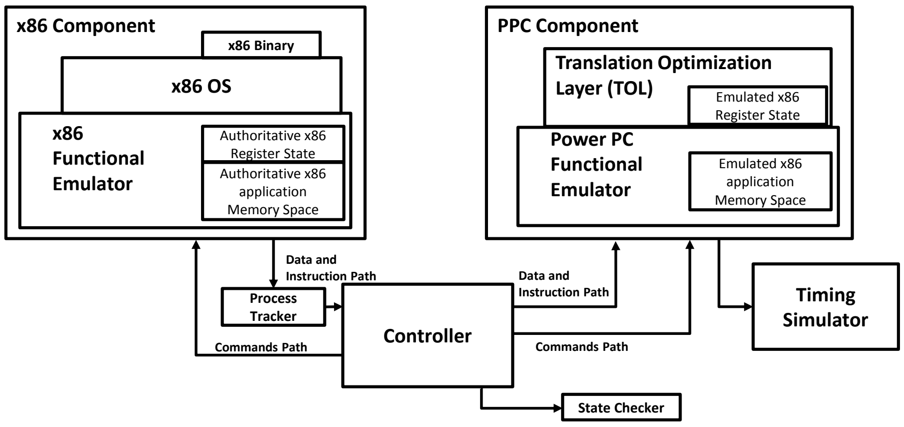
Experimental Framework

This chapter presents DARCO [86], the tool we used to evaluate our proposals. DARCO is an infrastructure for research in the field of HW/SW co-designed virtual machines. Due to lack of appropriate tools in this research area, we developed DARCO in collaboration with other members of our research group.

An infrastructure for modeling HW/SW co-designed processors needs to provide functional emulators for the host and guest ISAs, cycle-accurate timing simulation for the host processor, and a software layer that is able to interpret, translate, and dynamically optimize the guest binaries. Developing from scratch and debugging all these components has a multiple man-years cost. Using existing components to build such an infrastructure is a better alternative. Therefore, we used QEMU [9] to build some of the components of DARCO. However, it still required a significant effort with multiple people contributing to it for approximately two years. Currently, DARCO is being used by several member of our research group and has led to a number of research publications in various conferences [27][28][29][32][61][62][63][64][65][86].

DARCO models a HW/SW co-designed processor with guest x86 ISA [4] and a PowerPC (PPC) [8] like RISC host ISA. QEMU is used for emulating both the guest as well as the host ISA. The software layer of DARCO, called Translation Optimization Layer (TOL), translates and optimizes guest binaries to run on the host architecture. TOL provides staged compilation through an interpreter, a translator, and an optimizer. In addition to the functional emulators for guest and host ISAs, DARCO provides a cycle-accurate timing simulator and a debugging tool chain. DARCO has a clean interface for including new optimizations in TOL and allows easy implementation of new hardware features.

Except for the x86 and PowerPC functional emulators, for which we used modified versions of QEMU, all other components are in-house developments. The current version of DARCO supports only user-level x86 instructions.



x86 Component
x86 Binary
x86 OS
x86 Functional Emulator
Authoritative x86 Register State
Authoritative x86 application Memory Space
PPC Component
Translation Optimization Layer (TOL)
Emulated x86 Register State
Power PC Functional Emulator
Emulated x86 application Memory Space
Data and Instruction Path
Process Tracker
Controller
Commands Path
Data and Instruction Path
Commands Path
Timing Simulator
State Checker

Figure 3.1: DARCO Main components.

3.1 Main Components

DARCO is composed of four main components as shown in Figure 3.1.

1) x86 component.
2) PPC component.
3) Timing Simulator.
4) Controller.

The x86 component provides a full-system functional emulator for the guest x86 ISA. It runs an unmodified operating system and is the only component that interacts with the operating system. The operating systems is completely unaware of the existence of the software layer, which follows the concept of HW/SW co-designed processors like Crusoe [57] and Efficeon [59]. The authoritative register and memory state is also kept by this component because it emulates x86 code and not the translated/optimized code. The main role of this component is to permit co-simulation [58], a technique that is very useful for debugging, that checks if the co-designed execution state (kept by PPC component) after translations/optimizations done by TOL is consistent with authoritative x86 state or not.

The PPC component provides the co-designed processor functional model for DARCO. This component consists of a modified version of PowerPC emulator provided by QEMU. The name “PPC component” comes from the fact that PowerPC is the baseline RISC host architecture. However, we modified the baseline PowerPC ISA to meet our needs as will be explained in Section 3.3.2 and Section 3.5. The PPC functional emulator is executing TOL, which translates and optimizes the x86 instruction stream to host instructions. This component keeps the emulated x86 register and memory state, which is updated as the application execution proceeds. Consistency among authoritative and emulated x86 states validates the correctness of translation/optimizations done by TOL.

The timing simulator models a parameterized in-order core. It receives the dynamic instruction stream from the PPC component and provides detailed execution statistics. Moreover, it is able to distinguish the instructions corresponding to the emulation of the x86 application from those corresponding to TOL and its various modules. The use of the timing simulator is optional and doesn't affect the functionality of the rest of the infrastructure.

The controller is the main interface of DARCO with the user. It provides full control over the execution of the application as well as debugging utilities. The main task of the controller is to provide synchronization among different components and the resolution of the various requests from the PPC component as will be explained in Section 3.2. The controller is also responsible for comparing authoritative and emulated x86 states to ensure the correctness of translation/optimization process of TOL.

3.2 Execution Flow

The execution flow of an application passes through three distinct phases: 1) Initialization, 2) Execution, and 3) Synchronization. During the Initialization phase, the controller first starts the PPC component, which in turn, initiates the execution of TOL. The PPC component then remains idle until the controller sends the initial x86 register state of the application to be executed to it.

As for the x86 component, when launched by the controller, it initiates the execution of the application defined by the user. When it reaches the system call EXECVE (which always takes place at the beginning of an application) the execution pauses. A process tracker is initialized with the application's Control Register 3 (CR3) value, which can be used to distinguish the specific process from the rest of the applications running on top of the operating system. The process tracker is used throughout the execution of the application in the x86 component in order to ease synchronization and tracking of the changes made to the x86 state (register and memory) of the application. After the process tracker is initialized, the x86 component sends the initial x86 register state of the application to the controller. The Initialization phase is completed when the controller sends this state to the PPC component. At this point, the x86 register state is the same in both components.

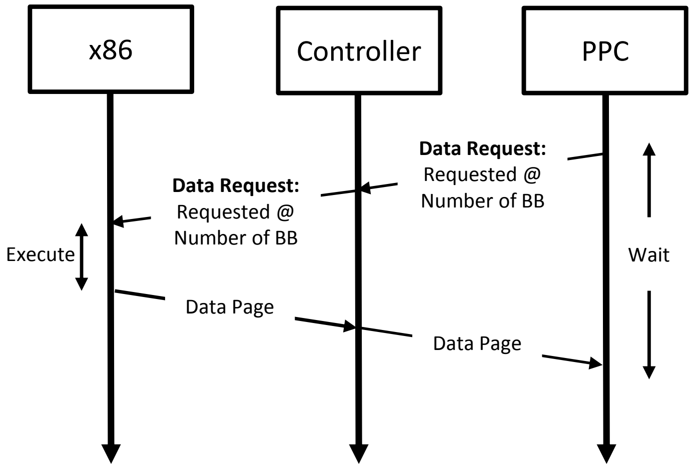


Figure 3.2: Data page request from the PPC component, enforcing the synchronization phase.

During the Execution phase, TOL begins by executing code from the initial Program Counter value it received during the Initialization phase. All changes made to the x86 register state from the emulation of the x86 instructions are stored in the "Emulated x86 register state", which resides in the memory space of TOL. Changes made to the memory space of the x86 application are stored in the "Emulated application x86 memory space", which also resides in the memory space of TOL. While the x86 application is making forward progress in PPC component, the x86 component remains idle.

The Synchronization phase is initiated by the PPC component when any of the following three events occurs during the execution phase: 1) data request, 2) system call, or 3) end of application. The data request event is raised when the PPC component encounters a load or store instruction that accesses an x86 memory page for the first time. The subsequent actions from the different components are depicted in Figure 3.2. The PPC component sends a request to the controller for the particular data page along with the total number of dynamic x86 basic blocks that has been executed until this point. Then, it remains idle until the request is satisfied. The controller forwards the request to the x86 component, which in turn continues the execution of the application until it reaches the same execution point as the PPC component (remember that the x86 component remained idle after the initial launch of the application). When the correct execution point is reached, the data page is sent to the controller and forwarded to the PPC component. This process guarantees that after every Synchronization phase, the x86 application state, register and memory, is identical between the x86 component and PPC component. Otherwise, the system complains and execution is aborted. This is also a useful technique to debug

DARCO. The exact same process is followed for the other two events: system calls and end of application.

System calls raise the synchronization event because TOL models only user-level code. Therefore, the system calls are executed only in the x86 component. Any changes made to the x86 state during the execution of system calls are passed to the PPC component after the completion of the system call. As for the end-of-application, the synchronization phase is necessary in order to ensure that the execution of the application on the PPC component was correct.

3.3 Translation Optimization Layer

Translation Optimization Layer (TOL) is the software layer that executes on-top of the host RISC processor. It is responsible for translating the target x86 code to the host ISA. It does that in three different execution modes: 1) interpretation mode (IM), 2) basic block translation mode (BBM), and 3) superblock translation and optimization mode (SBM).

TOL starts by interpreting guest x86 instruction stream in IM. When a basic block is executed more than a predetermined number of times, TOL switches to BBM. In this mode, the whole basic block is translated and stored in the code cache and the rest of the executions of this basic block are done from the code cache. Moreover, branch profiling information for direction and target of branches is also collected. Once the execution of a basic block exceeds another predetermined threshold, TOL creates a bigger optimization region, called superblock, using the branch profiling information collected during BBM. The superblock goes through several optimizations and is stored in the code cache. The high level view of the execution flow of TOL is shown in Figure 3.3.

3.3.1 Interpretation

TOL begins the execution of the application in IM. While in IM mode, x86 instructions are interpreted one by one and the x86 state is updated accordingly. The IM guarantees forward progress of the application and also is used as a safety-net in case instructions cannot be included in basic block translations and superblocks. Moreover, interpretation is necessary to make forward progress, in case of speculation failures in superblock due to aggressive optimizations.

There is one caveat concerning the interpretation method employed in DARCO. Due to the complex and time consuming nature of building an interpreter, we decided to

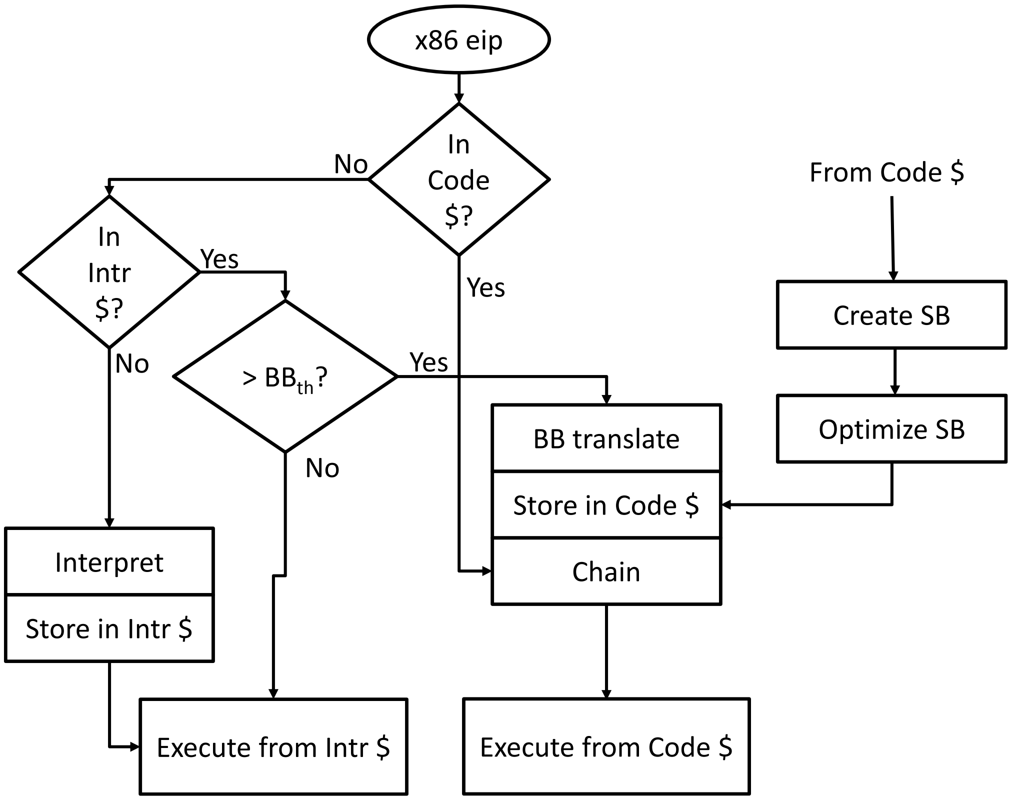


Figure 3.3: Translation Optimization Layer (TOL) execution flow. The left path is followed in IM, the middle in BBM and the right in SBM.

use the translator provided by QEMU but instead of translating one basic block at a time, it was modified to translate one instruction at a time. Since QEMU's translator was designed for portability (it supports translation from various guest to host ISAs), using it to translate just one instruction introduces high overhead. In order to accommodate the high cost of such interpretation method, an interpretation cache is used to store the interpretations. Interpretation cache is a typical code cache used in HW/SW co-designed processors; the only difference being instead of storing whole basic block translation or superblocks, it stores translation for individual instructions. Once the translation of an x86 instructions has been stored in the interpretation cache, its subsequent executions are done from this cache. This modification significantly reduced the cost of interpreting an x86 instruction. Also note that no chaining is done between interpretations.

3.3.2 Basic Block Translation

During IM, profiling information is collected for execution frequency of the basic blocks using software repetition counters. When the repetition counter of a basic block

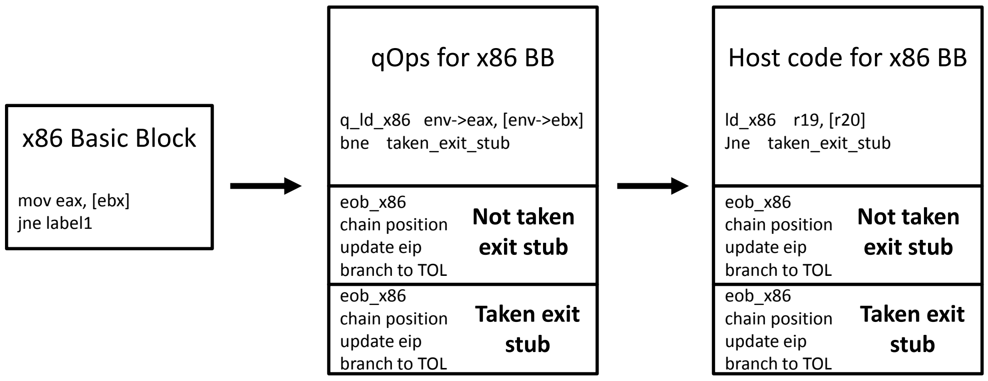


Figure 3.4: Abstract translation of an x86 basic block to host ISA. The eob_x86 instruction is used by DARCO for execution synchronization and special ld_x86 instructions to point out accesses to x86 memory space.

reaches the *BB_translation_threshold*, TOL switches to BBM in order to translate the corresponding basic block.

Note that since we use a modified version of the QEMU translator and code generator, we also inherit some of the nomenclature. The intermediate representation of the instructions in DARCO is called *qOps*.

Figure 3.4 shows an abstract version of a typical translation of an x86 basic block. The original code is translated into an equivalent set of qOps. TOL translates all x86 memory operations in a special way. We introduced new qOps and host instructions for all load and store instructions in order to be able to distinguish during the execution whether a memory access corresponds to the application itself or TOL. There are two reasons for doing this. The first regards to functionality. The PPC component needs to know if there is an access to the x86 memory space and in the uncommon case that the data page was not communicated before, requests the page to the controller as explained before. The second reason regards to evaluation, since we would like to be aware of the performance characteristics of each translation.

At the end of the translation, a branch instruction and two exit stubs are inserted. The outcome of the branch instruction decides which of the two exit stubs to execute. Each exit stub consists of an empty position where the chaining will be patched later during the execution, an update of the program counter and a branch to TOL where the basic block starting at the new program counter will be interpreted or translated. When the chain position is patched, the execution will not return to TOL, but instead the next basic block will be executed directly from the code cache. Finally, a new PPC instruction, *eob_x86*, is introduced. The purpose of this instruction is strictly for synchronization. In terms of timing, this instruction has no effect.

The qOps are forwarded to the code generator. There, they undergo some basic optimizations like dead code elimination and constant propagation, which contribute towards reducing the number of generated instructions. Finally, the qOps are translated to PPC instructions and stored in the code cache from where they are dispatched for execution.

3.3.3 Superblocks and Optimizations

During Basic-Block translation Mode (BBM), profiling information is gathered for all the basic blocks in BBM using software counters. This information consists of execution and edge counters. The execution counter provides the execution frequency of a basic block while the edge counters monitor the biased branch direction. Once the execution of a basic block exceeds another predetermined threshold, TOL creates a bigger optimization region, called superblock, using the branch profiling information collected during BBM.

In Superblock translation and optimization mode (SBM), TOL generates a new superblock starting from the triggering basic block. A superblock generally includes multiple basic blocks following the biased direction of branches. A superblock ends at one of the following conditions:

1) The last basic block included in the superblock ends with an indirect branch, call, or return instruction.
2) The last basic block included in the superblock ends with an unbiased branch or the probability of reaching the last basic block from the beginning of the superblock falls below a predetermined threshold.
3) The number of instructions in the superblock exceeds a predetermined threshold.
4) The number of basic blocks included in the superblock exceeds a predetermined threshold.

Moreover, the branches inside the superblocks are converted to “asserts” so that a superblock can be treated as a single-entry, single-exit sequence of instructions. This gives the freedom to reorder and optimize instructions across multiple basic blocks. “Asserts” are similar to branches in the sense that both checks a condition. Branches determine the next instruction to be executed based on the condition; however, asserts have no such effect. If the condition is true, assert does nothing. However, if the condition evaluates to false, the assert “fails” and the execution is restarted from a previously saved checkpoint in IM. Furthermore, if the number of assert failures in a superblock exceeds a predetermined limit, the superblock is recreated without converting branches to “asserts”.

As a result, this time the superblock has to be treated as a single-entry multiple-exit sequence of instructions. Having multiple exits in a superblock also reduces available optimization opportunities because the instructions across different exit paths cannot be reordered as freely as before.

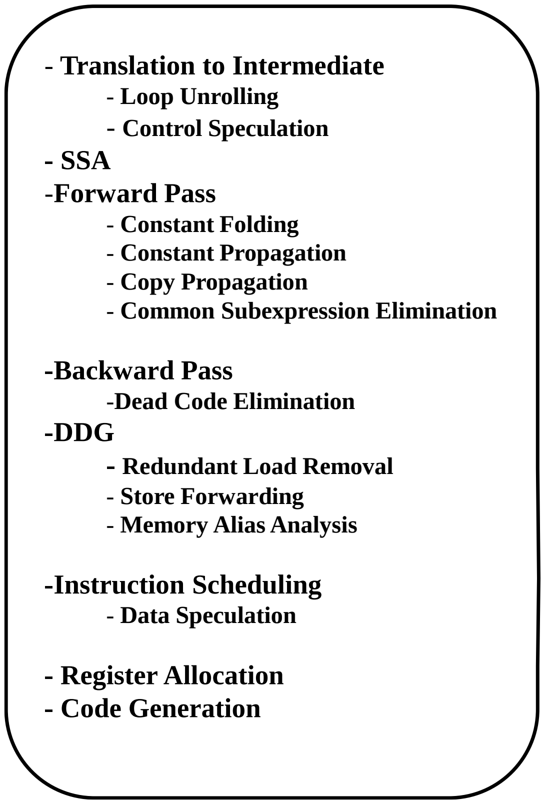
Furthermore, while creating a superblock, if a loop is detected, it is unrolled. Currently, we unroll loops consisting only a single basic block, as they are the ones which provide maximum benefit [77]. To detect and unroll the loops without control flow the following steps are followed.

1) The target address of the first branch instruction in the superblock is compared against the address of the first instruction of the superblock. In case of a loop, the addresses will match.
2) The execution and edge counters are used to determine the loop trip count.
3) Loop unroll factor is determined based upon the data types in the loop, SIMD accelerator width, and the loop trip count determined in the last step. For example, if a loop contains only single-precision floating-point data types, then for a 128-bit wide SIMD accelerator the loop is unrolled 4 times if the loop trip count is more than or equals to 4.

Moreover, the unrolled version of the loop is followed by the original loop (without unrolling). During execution, a runtime check is performed to determine whether to execute the unrolled version or the original loop. If the number of iterations left for execution are less than the loop unroll factor, then the original loop is executed instead of the unrolled loop.

The optimizer applies several transformations on the superblock. Figure 3.5 shows different optimizations performed by the optimizer. First, the qOps are transformed into a Static Single Assignment format. This transformation removes anti & output dependences and significantly reduces the complexity of subsequent optimizations. Second, a forward pass applies a set of conventional single pass optimizations: constant folding, constant propagation, copy propagation, and common subexpression elimination. Third, a backward pass applies dead code elimination.

After the basic optimizations, the Data Dependence Graph (DDG) is prepared. To create DDG, the input and output registers of the instructions are inspected and the corresponding dependences are added. During DDG creation, we perform memory disambiguation analysis. If the analysis cannot prove that a pair of memory operations will never/always alias, it is marked as “may alias”. In case of reordering, the original memory



- Translation to Intermediate
    - Loop Unrolling
    - Control Speculation
- SSA
- Forward Pass
    - Constant Folding
    - Constant Propagation
    - Copy Propagation
    - Common Subexpression Elimination
- Backward Pass
    - Dead Code Elimination
- DDG
    - Redundant Load Removal
    - Store Forwarding
    - Memory Alias Analysis
- Instruction Scheduling
    - Data Speculation
- Register Allocation
- Code Generation

Figure 3.5: Optimizations flow in superblocks.

instructions are converted to speculative memory operations. Apart from this, Redundant Load Elimination and Store Forwarding are also applied during DDG phase so that redundant memory operations are removed. The DDG is then fed to the instruction scheduler that uses a conventional list scheduling algorithm. Afterwards, the determined schedule is used by the register allocator that implements linear scan register allocation algorithm. Finally, the qOps are translated to PPC instructions and the code is stored in the code cache. The previous entry in the code cache that corresponds to the first basic block of current superblock is invalidated and freed for use by subsequent translations.

3.4 Speculation and Recovery

Memory speculation is a key optimization to achieve performance in HW/SW co-designed systems. Considering two ambiguous memory references independent of each other provides more freedom in instruction scheduling and boosts performance. For example, Transmeta Crusoe [36] reports that, on average, suppressing memory reordering causes 10% and 33% performance loss in operating system boots and user applications respectively. This section briefly explains how the speculation and recovery mechanism works in DARCO.

A combination of software and hardware mechanisms is used to detect speculation failure and subsequent recovery. As described in the last section, if a pair of memory references cannot be proved never/always aliasing; it is marked as “may alias”. TOL labels each load/store instruction with a sequence number in the original program order. If a pair of load-store or store-store instructions that may alias is reordered, the original load/store instructions are converted to “speculative load/store” instructions.

The hardware has two sets of architectural registers: a working set and a shadow copy. Before starting the execution of speculative code, a copy of the working set is saved into the shadow registers (saving a checkpoint). During the execution, only the working copy of the registers is updated. In the case of speculation failure, the register state is restored by copying the contents of shadow registers to the working copy. Restoring the memory state is a little more complicated since it is not practical to have two copies of the whole memory state. To track the changes in the memory state, a store buffer is used. During the normal execution, store instructions write to the store buffer instead of directly writing to the memory. In the case of speculation failure, the contents of the store buffer are discarded, whereas they are forwarded to the memory if the speculated code executes successfully.

To detect a speculation failure, the hardware maintains a table to record address and size of all the memory locations accessed by “speculative load/store” instructions in the current superblock. Moreover, the sequence number of “speculative load/store” instructions is also recorded in the table. During the execution, if the hardware detects:

- that a speculative memory instruction with higher sequence number is executed before another speculative memory instruction with lower sequence number and
- they access overlapping memory locations,

an exception in raised. In this case, the contents of the store buffer are flushed; register values from the shadow registers are copied to the working set; (this has the effect of restoring the earlier saved checkpoint) and the execution is restarted in Interpretation Mode. On the other hand, in case of successful execution of speculated code, values in the store buffer are forwarded to the memory and the contents of the shadow registers are discarded.

Figure 3.6 shows an example of speculation failure detection mechanism. Figure 3.6a shows the original code sequence with two memory references where the relation between the memory addresses is unknown. The two instructions are labeled in the program order. Figure 3.6b shows the reordered code sequence. The instructions maintain their sequence number. However, they are converted to speculative instructions to inform the hardware to check them for speculation failure. Figure 3.6c shows the hardware table state just before executing the speculative load instruction. The program counter points to the current instruction and the table has entry for the executed speculated store instruction. At this point, since the instruction with higher sequence number (2) has been executed before the instruction with smaller sequence number (1), if the address of the current speculated load instruction overlaps with the address of the speculated store instruction, the hardware will generate an exception and will go to the recovery mode. If the rate of speculation failures exceeds a predetermined limit in a particular superblock, it is recreated without reordering ambiguous memory references.

```
Seq Num
   1   ld_64     v1, M[x]
   2   st_64     v2, M[y]
```

a) Original Code Sequence

```
Seq Num
   2  st_64_s    v2, M[y]
   1  ld_64_s    v1, M[x]
```

b) Reordered Code Sequence

```
PC -->      1  ld_64_s    v1, M[x]
```

Seq Num	**Address**	**Size**
2	y	8

c) Hardware Table State

Figure 3.6: Speculation Failure Detection Example.

3.5 The Timing Model

The timing simulator is attached to the PPC component and models a simple in-order processor as depicted in Figure 3.7. The simple in-order processor is chosen in congruence with the simple hardware design philosophy of the co-designed processors. The closest product to the baseline modeled processor is the PPC 440 [3].

The modeled pipeline decouples the Front-End from the Back-End using an Instruction Queue. The Front End reads (Address Calculation - AC stage / Instruction Fetch - IF stage), decodes (DEC stage), and stores the instructions in the Instruction Queue (after DEC stage). Also, it is equipped with a Gshare branch predictor and a Branch Target Buffer.

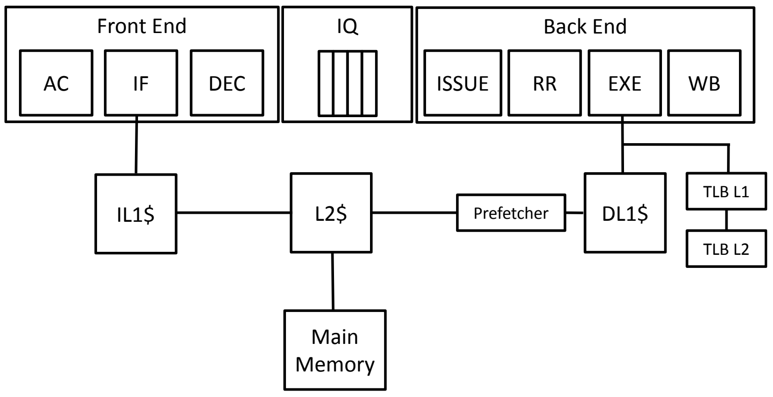


Figure 3.7: Host Processor Pipeline.

The Back-End issues and executes instructions from the Instruction Queue. Issuing is done using a scoreboard that keeps track of the availability of the source registers. During the Register Read (RR) stage, the instructions read their operands from the bypass or the register file. The register file is logically divided between TOL and application (32 registers are only accessible by TOL and 128 only by the translated application code). The EXE stage performs the required operations for instruction execution. Instructions spend different number of cycles in this stage depending on their latencies. Branch mispredictions are also detected and handled in this stage. The last stage, Write Back, writes the results computed during EXE stage to the register file.

Parameter	Value
L1 I-cache	64KB, 4-way set associative, 64-byte line, 1 cycle hit, LRU
L1 D-cache	64KB, 4-way set associative, 64-byte line, 1 cycle hit, LRU
Unified L2 cache	512KB, 8-way set associative, 64-byte line, 6 cycle hit, LRU
Scalar Functional Units (latency)	2 simple int(1), 2 int mul/div (3/10) 2 simple FP(2), 2 FP mul/div (4/20)
Vector Functional Units (latency)	1 simple int(1), 1 int mul/div (3/10) 1 simple FP(2), 1 FP mul/div (4/20)
Registers	128-Integer, 128-Vector, 32-FP
Main memory Lat	128 Cycles

Table 3.1: Host Processor Microarchitectural Parameters.

The modeled processor has two level cache hierarchy. The first level is split between instructions (L1 I$) and data (L1 D$), whereas the rest of the memory hierarchy (L2 and main memory) is shared. The TLB exists only for data and it also has a two level architecture. Furthermore, notice that the Back-End is equipped with a stride prefetcher.

Microarchitectural parameters for the modeled processor are given in Table 3.1. The issue width of the processor is 2, with both pipelines being symmetric.

We model a 128-bit wide SIMD accelerator with two 64-bit wide lanes. Even though the two pipelines are considered to be symmetric and the modeled processor has an issue width of two, the SIMD accelerator is shared by both lanes. Furthermore, all the operations except for division and reciprocal are assumed to be pipelined.

3.6 TOL Configuration

In this section, we provide a quantitative insight into TOL configuration and the characteristics of the generated code. First, we study the effect of the threshold variation for promoting basic blocks from BBM to SBM. It is followed by a discussion on dynamic x86 instruction distribution in different execution modes. Then a study of emulation cost of x86 instructions in SBM is presented. Finally, we study TOL overhead and its various components.

To configure TOL, we use applications from SPEC2006 [11] and Physicsbench [117] benchmarks suites. For SPEC2006, we instrument the benchmarks, using PIN [71], to find the most frequently executing routines. Then, we simulate four billion x86 instructions starting from these routines. The benchmarks in Physicsbench are executed till completion. The benchmarks are compiled with GNU GCC version 4.5.3, optimization flags “-O3 -ffast-math -fomit-frame-pointer”.

Parameters	Value
Basic Block Promotion Threshold	
IM to BBM	5 repetitions
Superblock creation parameters	
Maximum basic blocks in a superblock	16
Maximum instructions (intermediate representation) in a superblock	2K
Minimum probability to reach last basic block in the superblock from the entry	0.9
Minimum probability to reach a branch instruction in the superblock from the entry to convert it to "assert"	0.9
Speculation Failure parameters	
Control speculation failure threshold	500 in last 5000 executions (10%)
Memory speculation failure threshold	100 in last 10000 executions (1%)

Table 3.2: TOL configuration parameters.

Table 3.2 shows TOL configuration parameters. As the table shows, a basis block is promoted to BBM after executing it five times in IM. The maximum number of basic blocks and intermediate representation instructions allowed in a superblock is 16 and 2K respectively. Also, while creating a superblock, if the probability of reaching a basic block from the entry of the superblock falls below 0.9, no more basic blocks are included in the superblock. Similarly, for a branch instruction to be converted into “assert” the probability to reach the branch from the entry of superblock should be higher than 0.9. Alternatively, all the branches in a superblock are converted to “assert”. Moreover, if the number of assert failures in superblock exceeds 500 in the last 5,000 executions (10%), the superblocks is recreated without converting branches into asserts. Furthermore, if the memory speculation failures in a superblocks surpass a threshold of 100 in last 10,000 executions (1%), the superblock is recreated without reordering ambiguous memory references.

For the speculation and recovery, as discussed in Section 3.4, the hardware maintains a table where it stores the sequence number, direction, and size of speculative load/store instructions. We implement this table with 1K entries. Optimal duration/position to take a checkpoint is a different research problem and is out of scope of this work. For simplicity, at execution time, we take a checkpoint in the beginning of every superblock. We implement the store buffer with 1K entries. Moreover, to avoid overflow of the store buffer, we restrict the number of load/store instructions to be 1K in a superblock. Since we take checkpoint in the beginning of every superblock and a superblock cannot have more than 1K load/store, the store buffer can never overflow.

3.6.1 Optimal Promotion Threshold

The promotion threshold from BBM to SBM (BB/SB*th*) (number of times a basic block needs to be executed in BBM before it is further optimized and included in a superblock) poses a trade-off between the quality of the generated code in terms of host instruction per x86 instruction and the optimization overheads introduced by TOL for generating this code. The lower the threshold, more code is optimized thus reducing the emulation cost, but at the same time the overhead introduced by the optimizer is higher. Throughout this section, we define as overhead all the instructions that are not devoted directly to the emulation of the application. As a metric to find the optimal promotion threshold, we will use the total number of host instructions (TOL and application instructions) needed to execute a given number of x86 instructions.

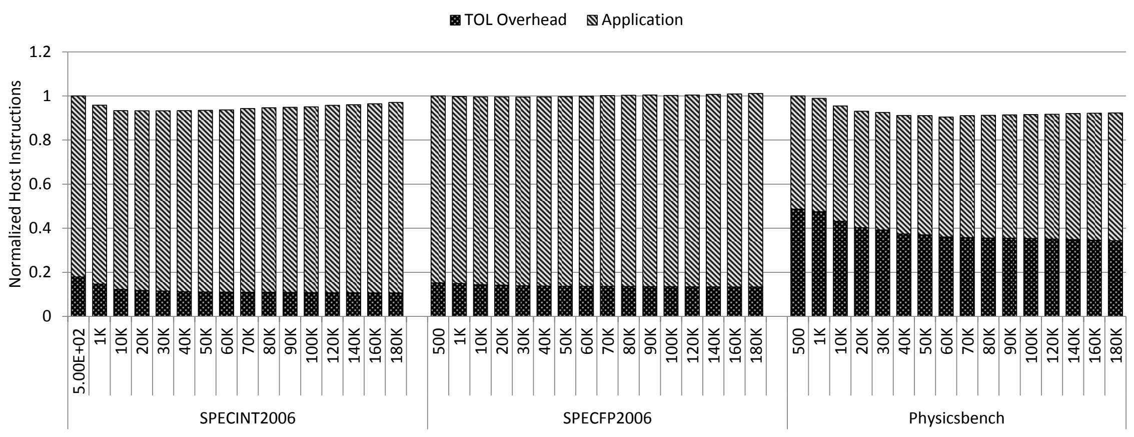


Figure 3.8: Effect of Threshold Variation on the Number of host instructions.

For this study, we considered different thresholds ranging from 500 up to 180K. The experimental results are depicted in Figure 3.8. Each bar corresponds to a different value of BB/SB*th* and it represents the number of host dynamic instructions normalized to the execution with the threshold of 500. Each bar is split into two parts: the lower part represents the overhead introduced by TOL, and the upper part represents the host instructions corresponding to the x86 code.

Two dominant trends can be observed in Figure 3.8. As the BB/SB*th* increases, TOL overhead decreases, as less static basic blocks are optimized. At the same time, the number of application instructions increases, since a smaller portion of the dynamic instruction stream has been optimized by TOL, leading to a higher ratio of host/guest instructions.

The experimental results show that the optimal BB/SB*th* is 60K for Physicsbench and SPECINT2006 whereas, SPECFP2006 is relatively insensitive to the threshold variation. It is also important to note that TOL overhead is also more in Physicsbench than in SPEC2006. The reason lies in the fact that SPEC2006 benchmarks have high dynamic to static instruction ratio than Physicsbench. Therefore, in SPEC2006, TOL overhead is amortized by repetitive executions of the translated/optimized code. Furthermore, the effect of threshold variation is more in SPECINT2006 and Physicsbench than in SPECFP2006. The high dynamic to static instruction ratio and bigger basic blocks also make SPECFP2006 less sensitive to BB/SB*th* threshold variation.

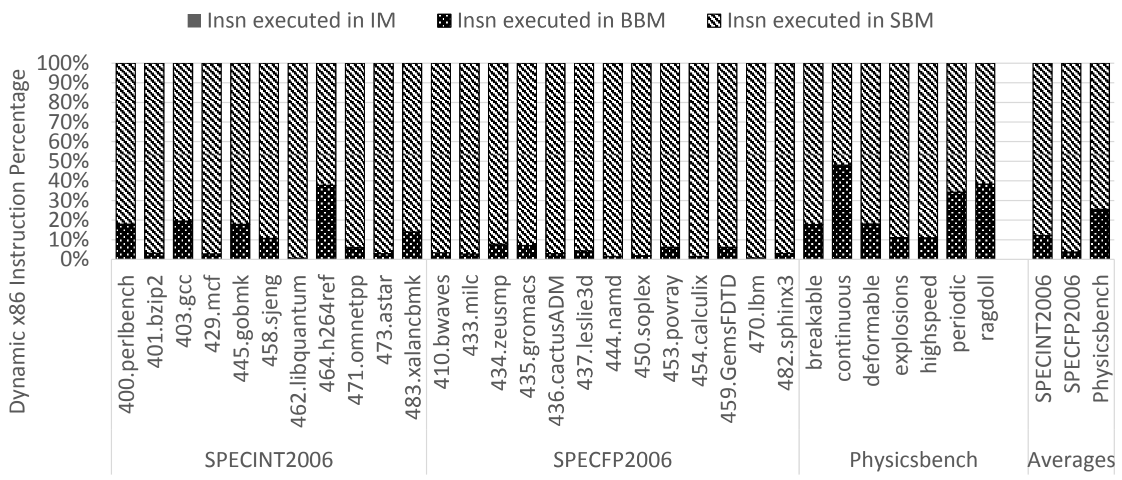


Figure 3.9: Dynamic x86 instruction distribution in IM, BBM and SBM.

It is worth mentioning that the optimal threshold varies between different systems. For example, a lightweight translator, like DynamoRIO [30], can use a lower threshold (50 repetitions for DynamoRIO) since the introduced overheads due to translation are significantly lower, mainly due to the fact that the guest and host ISA are the same. On the other hand, the overhead introduced by a different guest and host ISA translator, like TOL, is higher and as such, it requires higher threshold in order to amortize the overhead introduced by the optimizer.

3.6.2 Optimized Code Distribution

Given the optimal threshold of 60K repetitions, we studied the dynamic x86 instruction distribution in the three execution modes of TOL; the Interpreter Mode (IM), the Basic Block Mode (BM), and the Super Block Mode (SBM).

The dynamic x86 code distribution is depicted in Figure 3.9. The experimental data shows that even with a threshold as high as 60K repetitions, 88%, 96%, and 75% of the dynamic instruction stream comes from the highest level of optimization, superblocks, in SPECINT2006, SPECFP2006, and Physicsbench respectively. For three benchmarks in Physicsbench, namely continuous, periodic, and ragdoll, significant number of instructions are executed in BBM. The dynamic instruction count and dynamic to static instruction ratio in these is really small. Therefore, only a small portion of code is promoted to SBM.

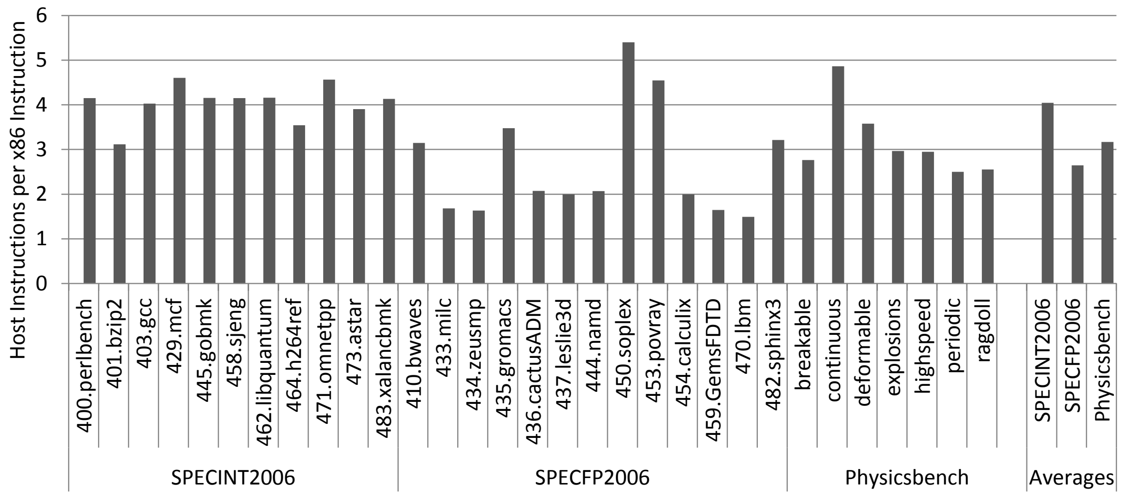


Figure 3.10: Host instructions per x86 instruction in SBM.

3.6.3 Emulation Cost

Emulation cost is the number of host instructions generated per x86 instruction. Since the dynamic instruction stream is dominated by the execution of code in SBM, we present the emulation cost only in SBM. As Figure 3.10 shows, on average TOL generates 4, 2.6, and 3.1 host instructions per x86 instructions for SPECINT2006, SPECFP2006, and Physicsbench respectively. The emulation cost of SPECINT2006 is high because of high emulation cost of branch instructions. Since the basic blocks in SPECINT2006 are smaller, the emulation cost of branch instructions dominates the overall emulation cost. Physicsbench is costlier in terms of emulation cost because it uses significant amount of trigonometric functions like sin, cos, etc. These x86 instructions are not directly mapped to the host instructions, however, they are emulated in software. Therefore, the overall emulation cost increases.

3.6.4 Dynamic Instruction and Overhead Distribution

The overall host dynamic instruction stream is composed to two components: 1) Application Instructions and 2) TOL Overhead Instructions. Application instructions are the amount of dynamic instruction stream corresponding to the emulation of x86 application. On the other hand, TOL overhead is the amount needed to translate x86 code to the host code and other housekeeping tasks performed by TOL.

Figure 3.11 shows the percentage of application instructions vs. TOL overhead in the host dynamic instruction stream. For SPECINT2006 and SPECFP2006, 16% and 13%

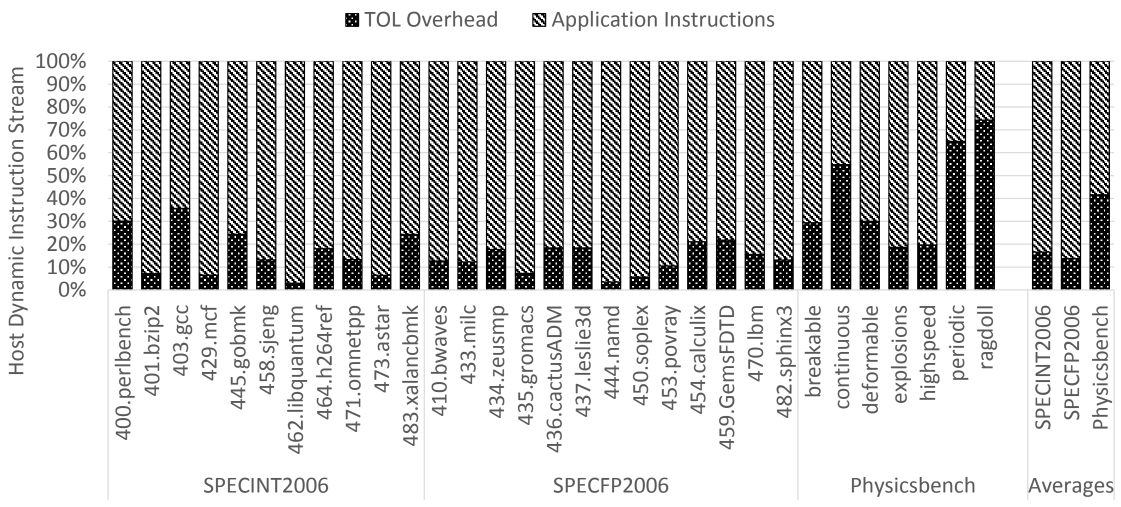


Figure 3.11: Overall Host Dynamic Instruction Distribution.

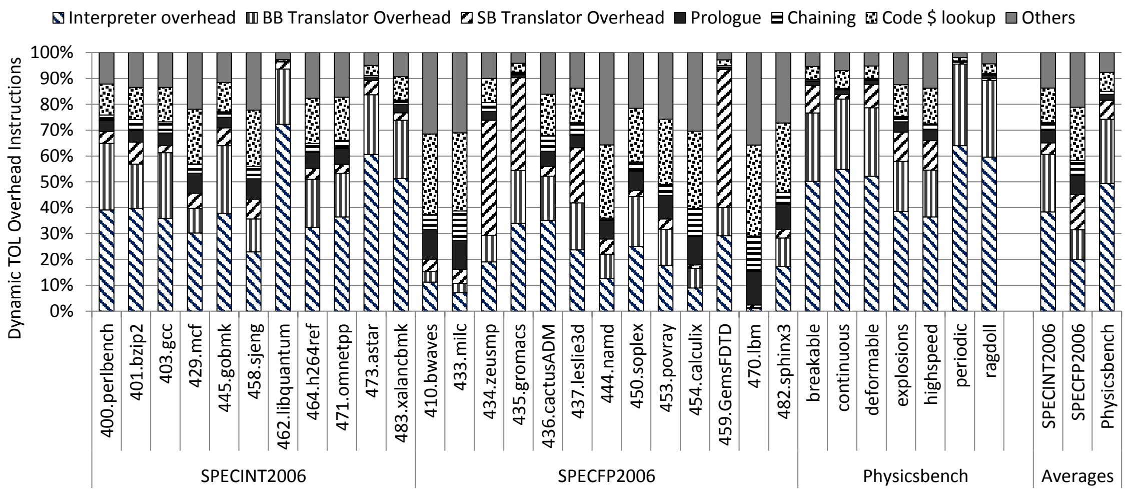


Figure 3.12: Dynamic TOL Overhead Distribution.

of the overall instruction stream corresponds to TOL overhead respectively, whereas for Physicsbench this number rises to 41%. As mentioned earlier, the high dynamic to static instruct ion ratio causes TOL overhead to be amortized in SPEC2006. On the other hand, in Physicsbench the overhead in not amortized due to fewer executions of translated code.

Figure 3.12 shows various component of TOL overhead. It is divided into seven major categories (bottom-up): 1) Interpretation Overhead: TOL overhead for interpreting the code before it is promoted to BBM. 2) BB Translator Overhead: TOL overhead for translating the basic block promoted to BBM. 3) SB Translator Overhead: TOL overhead for creating, translating, and optimizing the superblocks. 4) Prologue: Every time control is transferred between TOL and the translated code, a specific piece of code is executed to

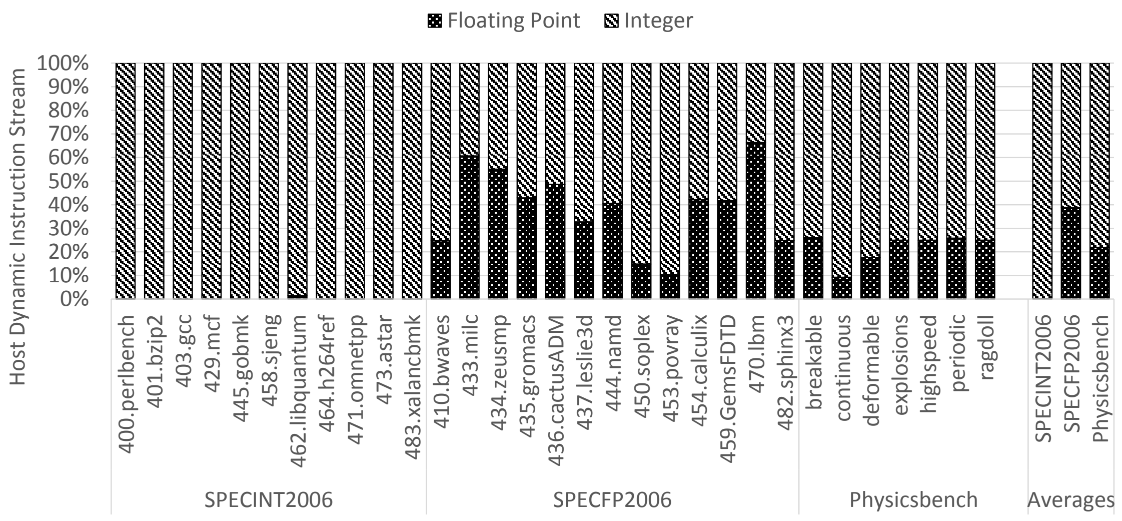


Figure 3.13: Floating Point and Integer instruction distribution in host dynamic instruction stream.

do some housekeeping stuff like stack management. Prologue overhead corresponds to executing this code. 5) Chaining: Different translated basic blocks and superblocks can be connected to each other in the code cache. To check whether chaining is possible and chaining the possible pairs constitutes "Chaining" overhead. 6) Code Cache Lookup: Every time that control is transferred to TOL, it checks whether the translation for next x86 basic block is already present in code cache or not. This lookup is termed as code cache lookup overhead. 7) Others: All other overheads like managing control flow in the main loop of TOL, collecting statistics, TOL initialization, etc. are counted under this category.

It is interesting to note that, in Physicsbench, Interpretation Overhead and BB Translator Overhead dominates the overall overhead, whereas in SPECFP2006 these overheads relatively smaller. The reason for this behavior is once again dynamic to static instruction ratio. SB Translator Overhead, where most aggressive and speculative optimizations are applied, is relatively smaller for both benchmark suites.

3.6.5 Floating Point and Integer Instruction Distribution

Figure 3.13 presents the host dynamic instruction distribution. As the figure shows 39% and 22% of the overall host dynamic instruction stream corresponds to floating point code for SPECFP2006 and Physicsbench respectively. Whereas, the amount of floating point code in SPECINT2006 is negligible. Since most of the SIMD optimizations target floating point code, we consider only SPECFP2006 and Physicsbench to evaluate our proposals in the next chapters.

3.7 Host ISA Extension

The baseline PowerPC ISA has been extended mainly to support missing features in vector execution. For example, Altivec (the SIMD extension of PowerPC) supports only packed single-precision floating-point operations. We added support for double-precision floating point operations as well. Altivec has only packed floating-point operations and does not support scalar floating-point operations. The scalar floating-point operations are executed on a different floating point unit (FPU) with its own register file. The communication between the vector register file and the scalar floating-point register file has to go through memory. We added support for scalar floating-point operation as well, that allows both scalar and vector floating point operations to execute on the same unit without the overhead of communication between the two register files. The SIMD instruction set of the host ISA consists of following instruction groups:

Data Transfer Instructions: The data transfer instruction set provides instructions for moving data among vector registers and between vector registers and memory. The memory access instructions include 32-bit, 64-bit, and 128-bit loads/stores from/to memory. Moreover, the speculative versions of these basic memory instructions are also provided. The ISA also provides instructions to move data between vector and integer register files.

Arithmetic and Logical Instructions: The arithmetic instruction set offers instructions for performing addition, subtraction, multiplication, division, maximum, minimum, and square root operations on single and double-precision floating-point data. Moreover, reciprocal operation is available only for single-precision floating-point numbers. The logical instructions provide support for AND, AND NOT, OR, and XOR operations. The ISA provides scalar and vector versions of all the instructions. The scalar version operates only on one element, whereas vector version performs multiple operations at the same time.

Conversion Instructions: The base ISA provides instructions to support conversion between:

1) Single and double-precision floating-point formats.
2) Single-precision floating-point and doubleword integer formats.
3) Double-precision floating-point and doubleword integer formats.

Scalar and vector versions of all the conversion instructions are provided.

Comparison Instructions: The baseline ISA comparison instructions compare single-precision floating-point, double-precision floating-point, or integer values and return the result either to a destination register or to the flag register. For single and double-precision floating-point values the comparison instruction can check for “equal”, “not equal”, “less than”, “not less than”, “less than or equal”, “not less than or equal”, “ordered” (true if none of the source operands is NaN), and “unordered” conditions. For integers the only checks provided are “greater than” and “equal”. Moreover, integer comparisons are provided at byte, word, doubleword, and quadword levels.

Permutation Instructions: Permutation support in the baseline ISA is provided through several shuffle instructions. These instructions interleave the contents of two vector source registers and store the results in destination register. “Broadcast instructions” are provided to copy a value in all the elements of a vector register. The copied value may come from the memory or another register. Moreover, unpack instructions are provided to distribute the result of a packed operation to several registers.

3.8 McPAT

To measure the energy savings of power gating we integrated McPAT [70] to DARCO. McPAT is an integrated power, area, and timing modeling tool for multithreaded and multi-core architectures. In terms of power, McPAT not only models dynamic power but also the leakage power. Instead of being hardwired to a particular simulator, McPAT uses an XML-based interface to communicate with performance simulators. In addition to communicating with performance simulator, the XML interface is also used to pass static microarchitectural, circuit, and technology parameters. Running McPAT separately from the performance simulator makes it flexible and easily portable to different performance simulators.

The configuration parameters in the XML file, can be defined at three levels: architectural, circuit, and technology. The parameters defined at the architectural level are similar to the input parameters of a performance simulator. These include number of cores, core issue width, homogeneous or heterogeneous cores, in-order or out-of-order cores, number of hardware threads per core, instruction scheduling scheme, cache sizes and levels, shared or private caches, directories, network on chip, number of routers, etc. The circuit level parameters are used to specify implementation details. These include specifying whether to use flip-flops or SRAM to implement arrays, crossbar types for on-chip routers, etc. Technology level parameters include device and interconnect types. McPAT support three device types: 1) High performance, 2) Low standby power, and 3)

Low operating power. The interconnects can use either aggressive or conservative wire technology. Moreover, user can specify whether to use long channel devices or not, if there is a possibility. McPAT supports technology nodes from 22nm to 90nm. Instead of modeling power using linear scaling assumption across different technologies, McPAT uses technology projections from ITRS [10].

McPAT is composed of three main components 1) power, area, and timing models, 2) optimizer for circuit level optimizations and 3) internal chip representation. The power, area, and timing model reads the configuration parameters passed by the user in the XML file and models power, area, and timing. In general, McPAT focuses on power and area modeling, whereas the target clock rate is taken as a design constraint. The optimizer takes in the target clock frequency, power and area deviation, optimization functions, and other parameters and explores the design space to optimize each processor component. Among the configurations satisfying the power and area deviations, McPAT applies optimization function to calculate power and area results. Then the optimizer generates an internal chip representation. This chip representation along with power, area, and timing models is used to generate final chip area, timing, and peak power. The final runtime power dissipation is calculated based upon the activity factor and peak power of individual units. Most of the parameters in the internal chip representation are directly set by input parameters in the XML file like cache levels and capacity, issue width, etc.

3.8.1 Power Modeling

The main sources of power dissipation in CMOS technology includes: 1) Dynamic power, 2) Short-circuit power, and 3) Leakage power. The power consumed by the circuit while it is performing some operations (or when is it active) is called dynamic or active power. The main source of dynamic power consumption is charging and discharging of capacitive loads due to changes in the circuit states. Therefore, the dynamic power consumption of a circuit is directly proportional to the capacitive load it drives. Other factors that affect the dynamic power include the supply voltage, the voltage swing during charging/discharging, clock frequency, and activity factor. Most of these parameters are passed to McPAT in the XML configuration file like clock frequency and the supply voltage (depends upon technology used). The activity factor is provided by, or calculated by, the statistics provided from the performance simulator. Circuit decomposing techniques and analytic models are used by McPAT to find the load capacitance of a module.

The short-circuit power in CMOS circuits occurs because both the transistors (PMOS and NMOS) are momentarily in “on state” when the circuit switches its state.

During this interval, there is a path between power supply and ground that results in short-circuit power consumption. McPAT uses a model developed by Nose et al. [81] to compute the short-circuit power consumption.

In sub-nanometer technologies, the contribution of leakage power towards the overall power consumption increases significantly. The power consumed by a circuit when it is idle, is called leakage power. There are mainly two sources of leakage power: 1) Subthreshold Leakage and 2) Gate Leakage. Subthreshold leakages occurs due to current flowing between drain and source terminals of a transistor even when it is in off state. Whereas, gate leakage corresponds to the tunneling of electrons from gate terminal to the channel between drain and source of the transistor. McPAT used MASTAR [10] and data from Intel [19] to estimate the leakage power consumption.

Chapter 4

Speculative Dynamic Vectorization

The compiler based static vectorization is used widely to extract data level parallelism from computation intensive applications. Static vectorization is very effective in vectorizing traditional array based applications. However, compilers inability to reorder ambiguous memory references severely limits vectorization opportunities, especially in pointer rich applications. HW/SW co-designed processors provide an excellent opportunity to optimize the applications at runtime. The availability of dynamic application behavior at runtime will help in capturing vectorization opportunities generally missed by the compilers.

This chapter presents our proposal of complementing the static compile time vectorization with a speculative dynamic vectorizer in a HW/SW co-design processor. It also presents a speculative dynamic vectorization algorithm that speculatively reorders ambiguous memory references to uncover vectorization opportunities. The hardware checks for any memory dependence violation due to speculative vectorization and takes corrective action in case of violation.

4.1 Introduction

As we have seen in the previous chapters, Single Instruction Multiple Data (SIMD) accelerators are ubiquitous in processors from all the computing domains. They are very attractive from the hardware point of view due their simple control mechanism and replicated functional units. However, code generation for SIMD extensions has always been challenging. In the early days, programmers used to target these extensions mainly using in-line assembly or specialized library calls. Then, automatic generation of SIMD instructions (auto-vectorization) was introduced in compilers [24][78][104], which borrowed their methodology from vector compilers. These compilers target loops for generating code for SIMD accelerators. Later, S. Larsen et al. [66] introduced Superword Level Parallelism (SLP) in which they target basic blocks instead of whole loops for vectorization. These static approaches to vectorization are effective for traditional applications where memory is referenced through explicit array accesses, whereas modern applications make extensive use of pointers. Due to this, disambiguation of a pair of memory accesses becomes difficult at compile time. Since memory operations form the foundation of vectorization, current static approaches are limited in extracting SIMD parallelism.

In this chapter, we provide details of our proposal of having dynamic vectorization as a complimentary optimization to the compiler based static vectorization. It is important to note that we do not propose to eliminate static vectorization altogether because there are several complex and time consuming transformations that are not straightforward to apply at runtime and are too costly like loop distribution, loop interchange, loop peeling, memory layout change, algorithm substitution, etc. However, static vectorization alone fails to capture significant vectorization opportunities due to conservative pointer disambiguation analysis. To handle these cases we propose to have a speculative dynamic vectorizer that can speculatively reorder ambiguous memory references to uncover vectorization opportunities. Moreover, in the absence of loops, the scope of vectorization for static vectorization is a single basic block. We propose to vectorize bigger code regions, superblocks, which include multiple basic blocks and can be created at runtime following the biased direction of branches.

Furthermore, we propose a speculative dynamic vectorization algorithm that can be implemented in the software layer of HW/SW co-designed processors. The proposed algorithm speculatively reorders and vectorizes memory operations. During execution, the hardware checks for any memory dependence violations caused by 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. The hardware support required for speculative execution is already provided by co-designed processors as discussed in Chapter 2. Therefore, no additional hardware support is needed from speculative vectorization point of view.

Moreover, in the absence of static compiler vectorization, our algorithm can work as a standalone vectorizer also. Therefore, the legacy code which was not compiled for any SIMD accelerator can be vectorized using the proposed algorithm. The co-designed nature of the processor makes the vectorization portable. As a result, the algorithm can be modified to transparently target a different SIMD accelerator. It is important to note that the proposed algorithm does not require any compiler or operating system support/modification.

In the rest of the chapter, we start by briefly providing the motivation for the work presented in Section 4.2. It is followed by the details of the proposed speculative dynamic vectorization algorithm in Section 4.3. Evaluation of the algorithm using SPECFP2006, Physicsbench, and UTDSP [13] applications is presented in Section 4.4. Section 4.5 presents the related work and Section 4.6 concludes.

4.2 Motivation

Traditional compile time loop vectorization is effective for applications involving explicit array accesses since memory dependence analysis are relatively easy. Significant performance gains have been reported using compiler vectorization in the past [24][66]. However, one of the major obstacles in vectorization at compile time is memory disambiguation and dependence testing. J. Holewinski et. al. [45] showed that static vectorization fails to extract significant vectorization opportunities especially in pointer-based applications. Furthermore, S. Maleki et al. [74] showed that the modern compilers, including Intel ICC, IBM XLC, and GNU GCC, are limited in vectorizing modern applications. Extensive use of pointers and pointer arithmetic in these applications complicate memory disambiguation and dependence testing. Even though research shows that a pair of memory accesses rarely alias until and unless aliasing is obvious [41], compilers generate conservative code to ensure correctness, which limits vectorization opportunities [88]. For example, Figure 4.1a shows a function with a loop that performs pointer arithmetic. The function takes in two pointers as parameters. Since the function can be called from a number of places in the entire program, the inter-procedural analysis of compiler needs to check whether the two pointers can alias or not. If the compiler cannot prove that the two pointers always reference different memory locations, it will conservatively assume dependence between them to ensure correctness. As a result, the loop will not be vectorized.

As stated before, another approach to vectorization, SLP [66], performs vectorization at lower intermediate representation level. SLP vectorizes at basic block level instead of loop level. Therefore, SLP may vectorize portions of a loop if the whole loop is not vectorizable, whereas traditional loop vectorizers vectorize either whole loop or nothing. SLP starts by identifying adjacent memory accesses and then follows their def-use and use-def chains. A def-use chain consists of a definition of a variable and all its uses reachable from that definition without any other intervening definitions. Similarly, a use-def chain consists of a use and all the definitions of a variable that can reach that use without any other intervening definition. Figure 4.1b shows low level intermediate representation for the loop of Figure 4.1a after unrolling it once. In this case, even though *I0* and *I6* are adjacent memory references, they cannot be packed by SLP since *I4* and *I6* may alias. Similarly, *I4* and *I10* access consecutive memory locations. However, they also cannot be

```
void example(double *a, double *b)
{
    int i;
    for (i = 0; i < NUM_ITR; i++)
        a[i] += b[i] * CONST;
}
```

a) An example loop with pointers.

```
loop:     I0   ld_64     v2, M [r2 + r1 * 8]
          I1   mulsd     v3, v2, v1
          I2   ld_64     v4, M [r3 + r1 * 8]
          I3   addsd     v5, v4, v3
          I4   st_64     v5, M [r3 + r1 * 8]
          I5   add       r4, r1, 1
          I6   ld_64     v6, M [r2 + r4 * 8]
          I7   mulsd     v7, v6, v1
          I8   ld_64     v8, M [r3 + r4 * 8]
          I9   addsd     xmm0, v8, v7
          I10  st_64     xmm0, M [r3 + r4 * 8]
          I11  add       r1, r4, 1
          I12  cmp       r1, r0
          I13  jne       loop
```

b) Unrolled lower level representation.

```
          V0   Pack1          v1, v1, v1
loop:     V1   ld_128_spec    v2, M [r2 + r1 * 8]
          V2   mulpd          v3, v2, v1
          V3   ld_128         v4, M [r3 + r1 * 8]
          V4   addpd          v5, v4, v3
          V5   st_128_spec    v5, M [r3 + r1 * 8]
          V6   add            r1, r1, 2
          V7   cmp            r1, r0
          V8   jne            loop
          V9   Unpack         xmm0, v5
```

c) Speculatively vectorized version.

Figure 4.1: An example loop with pointer arithmetic.

vectorized because *I6* might alias with them. Thus, memory dependences affect both traditional loop vectorizers as well as modern SLP.

[1] Pack/Unpack instructions are explained in Section 4.3.1.

One possible solution that compilers may provide is to generate two versions of the loop: one without vectorization and another vectorized with a runtime test to check for aliasing. However, this solution is not optimal because:

1) runtime test has to be executed every time before executing the loop, thus resulting in performance loss. Moreover, as the number of arrays to be checked for aliasing increases the number of checks to be performed also increases.

2) Having multiple versions of the loop increases the static code footprint of the application, which results in higher instruction cache size requirements.

Another way of vectorizing the example loop is through "__restrict" keyword. It can be used to indicate that a symbol is not aliased in the current scope. If the programmer knows that the two pointers to the function will not alias in any case, he can pass this information to compiler using the "__restrict" keyword. Once the compiler is sure that the two pointer always access non-overlapping memory locations, it can vectorize the loop. However, it requires source code modification which is not always possible e.g. unavailability of the source code or any other reason. In contrast, the proposed mechanism does not require any source code modification.

HW/SW Co-designed processors provide an excellent opportunity to handle these cases: instead of generating multiple versions, a single speculatively vectorized version can be generated by the software layer and the hardware can be tailored to execute the vectorized code efficiently and safely. The proposed algorithm speculatively reorders memory operations to expose vectorization opportunities. For example, in the code of Figure 4.1b, our algorithm speculatively assumes that *I4* and *I6* will never alias and reorders them to pack *I0* and *I6* together, as shown in Figure 4.1c. Moreover, due to the speculative reordering, *V1* is converted to a speculative load and *V5* to a speculative store. If during the execution it turns out that *V1* and *V5* access overlapping memory locations, the hardware will detect this condition and will take corrective measures. In this example, by vectorizing speculatively, we are able to vectorize the whole loop, whereas loop vectorization and SLP could not find vectorization opportunities.

Therefore, having two complementary vectorizing schemes helps to get the best of both the worlds. First, static vectorization applies more complex and time consuming loop transformations, even though vectorizes conservatively. Later at runtime, a dynamic vectorization phase catches the opportunities missed by static vectorization and speculatively vectorizes ambiguous memory references and their dependent operations.

- Translation to Intermediate
  - **Loop Unrolling**
  - **Control Speculation**

- SSA

-Forward Pass
  - **Constant Folding**
  - **Constant Propagation**
  - **Copy Propagation**
  - **Common Subexpression Elimination**

-Backward Pass
  - **Dead Code Elimination**

-DDG
  - **Redundant Load Removal**
  - **Store Forwarding**
  - **Memory Alias Analysis**
  - ***Consecutiveness Analysis***

- *Vectorization*

- *Dead Code Elimination*

- Instruction Scheduling
  - **Data Speculation**

- Register Allocation

- Code Generation

Figure 4.2: Optimization Sequence in Superblocks for Vectorization Support.

4.3 Dynamic Vectorization Algorithm

This section provides the details of the proposed speculative dynamic vectorization scheme. Vectorization is applied only on superblocks in SBM, as they represent the most frequently executed portion of the code. Moreover, vectorization is applied after applying all other dynamic optimizations. This ensures that all the dead code is eliminated before vectorization and only the code contributing to the final result is vectorized.

Figure 4.2 shows the modified optimization sequence in superblocks. The changes, with respect to the baseline, are shown in italic. First of all, we do additional analysis during DDG to discover consecutive memory accesses. We call it consecutiveness analysis. This analysis discovers consecutive memory accesses so that they can be packed together by the vectorizer if possible. After consecutiveness analysis, vectorization itself is performed. A backward pass follows the vectorization phase and converts the scalar version of vectorized instructions to “nops”.

4.3.1 The Vectorizer

This section explains the vectorization algorithm with pseudo-code and using a practical example. The pseudo code for the vectorizer is listed in Algorithm 4.1. The vectorizer packs together a number of independent scalar instructions that perform the same operation, and replaces them with one vector instruction. The number of scalar instructions packed depends on two factors:

- data-types of scalar instructions
- host vector length

For example, for a host vector length of 128-bit, four 32-bit single-precision floating-point instructions can be packed together in a single vector instruction. Therefore, vectorization reduces dynamic instruction count and improves performance. Before describing the algorithm itself, we define a set of conditions that a pair of instructions must satisfy to be included in the same pack:

- The instructions must perform the same operation.
- The instructions must be independent.
- The instructions must not be in another pack.
- If the instructions are load/store, they must be accessing consecutive memory locations.

Vectorization starts by marking all the instructions which are candidates for vectorization. Moreover, we mark *First Load* and *First Store* instructions. *First Load/Store* instructions are those for which there are no other loads/stores from/to adjacently previous memory locations. For example, if there is a 64-bit load instruction I_L that loads from a memory location [M] and there is no 64-bit load instruction that loads from address [M – 8], we call I_L *First Load*.

Vectorization begins by packing consecutive stores, starting from a *First Store*. The decision of starting with stores instead of loads is based on the observation that a given kind of operation always has the same number of predecessors, e.g. all the additions always have two predecessors, whereas the number of successors may vary depending on how many instructions consume the result. Consequently, following a bottom-up approach results in a more structured tree traversal than a top-down approach.

Once a pack of stores is created, their predecessors are packed (*Pack_pred_succ* rountine), before packing other stores, if they satisfy the packing conditions. Moreover if

Algorithm 4.1a. Top Level Vectorization Function

```
Vectorize (SB):
    Set_packable(SB,Available_for_pack, First_St,First_Ld)
    Pack_ldst(SB, Available_for_pack, First_St, packs)
    Pack_ldst(SB, Available_for_pack, First_Ld, packs)
    Set_Arith(SB, Available_for_pack, Arith)
    Pack_Arith(SB, Available_for_pack, Arith, packs)
```

Algorithm 4.1b. Load-Store Vectorization

```
Pack_ldst(SB, Available_for_pack, First_LdSt, packs):
    for inst in First_LdSt:
        vec_length = get_vector_length(inst)
        P = [inst]
        for i in range(1, vec_length):
            if inst has next_ldst:
                if inst_can_pack(P,next_ldst, Available_for_pack):
                    P.extend(next_ldst)
                    inst = inst.next_ldst
                else:
                    break

        if len(P) == vec_length:
            packs.extend(P)
            Make_unavilable(P, Available_for_pack)
            First_LdSt.extend(inst.next_ldst)
            Traverse_pred_succ (SB, Available_for_pack, packs)
```

Algorithm 4.1c. Traverse Predecessors/Successors

```
Traverse_pred_succ(SB, Available_for_pack, packs):
    need_Pack = Pack_pred_succ(SB, Available_for_pack, packs[latest].preds, packs)
    if need_Pack:
        generate_Pack_inst
    need_Unpack = Pack_pred_succ(SB, Available_for_pack, packs[latest].succs, packs)
    if need_Unpack:
        generate_Unpack_inst
```

Algorithm 4.1d. Vectorize Predecessors/Successors

```
Pack_pred_succ(SB, Available_for_pack, pred_succ, packs):
    for inst in pred_succ:
        if inst in Available_for_pack:
            vec_length = get_vector_length(inst)
            P = [inst]
            for i in range(1, vec_length):
                for inst1 in pred_succ[i]:
                    if inst_can_pack(P, inst1, Available_for_pack):
                        P.extend(inst1)
                        break
            if len(P) == vec_length:
                packs.extend(P)
                Make_unavilable(P, Available_for_pack)
                Traverse_pred_succ (SB, Available_for_pack, packs)
    if All_pred_succ_packed(pred_succ):
        return NO
    else
```

```
                return YES
```

Algorithm 4.1e. Vectorize Remaining Arithmetic Operations

```
Pack_Arith(SB, Available_for_pack, Arith, packs):
    for inst in Arith:
        if inst in Available_for_pack:
            vec_length = get_vector_length(inst)
            P = [inst]
            for inst1 in Arith[pos(inst):len(Arith)]:
                if inst_can_pack(P, inst1, Available_for_pack):
                    P.extend(inst1)
                    if len(P) == vec_length:
                        packs.extend(P)
                        Make_unavilable(P, Available_for_pack)
                        Traverse_pred_succ (SB, Available_for_pack, packs)
                        break
```

Algorithm 4.1: Pseudo code for the algorithm. *Vectorizer* is the top level function that starts vectorization. First of all, vectorizable instructions are marked and the staring points *First Stores/Loads* are identified. *Pack_ldst* function creates packs from consecutive memory operations and then calls *Traverse_pred_succ* function to pack predecessors/successors. After packing all packable predecessors/successors with *Pack_pred_succ* function, Pack and Unpack instructions are generated if necessary. Finally, all the remaining arithmetic operations are packed by *Pack_Arith* function. "packs" contains the packs created during vectorization. *Set_Arith* function marks remaining arithmetic operations for vectorization. *inst_can_pack* checks if instructions can be packed together. *Make_unavilable* marks instructions as unavailable for vectorization, since they are already included in other pack.
the last store in the pack has a next adjacent store, it is marked as *First Store* so that a new pack can start from it.

Once all the stores are packed and their predecessor/successors chains have been followed, we check for remaining load instructions that satisfy the packing conditions and pack them in the same way as stores. *Pack_ldst* routine provides the functionality for packing loads and stores.

Vectorization starting from adjacent loads/stores has an obvious limitation: if a superblock does not have any consecutive loads/stores, nothing can be vectorized. To tackle this problem, after packing all loads/stores and their predecessors/successors, we check if still there are some arithmetic instructions which can be packed together. If yes, we vectorize them and follow their predecessor/successor trees *(Pack_Arith)*. This allows us to partially vectorize loops with interleaved memory accesses.

While traversing the predecessor/successor chains, if we find out that the predecessors of a pack cannot be vectorized, a *Pack* instruction is generated. This *Pack* instruction collects the results of all the predecessors into a single vector register and feeds the current pack. Similarly, if all the successors of a pack cannot be vectorized, an *Unpack* instruction is generated. This *Unpack* instruction distributes the result of the pack to the scalar successor instructions. *Traverse_pred_succ* routine provides this functionality. For example, in the case of loops with interleaved memory access, when we reach several load instructions while traversing the tree, we find out that they cannot be packed since they are not consecutive. Therefore, we leave them in scalar form and assemble their results using a *Pack* instruction.

Moreover, *Pack* instructions are needed if a pack contains an instruction whose input is live-in of the superblock. Similarly, *Unpack* instructions are needed to put the results from a pack to the architectural registers that are live-outs of the superblock.

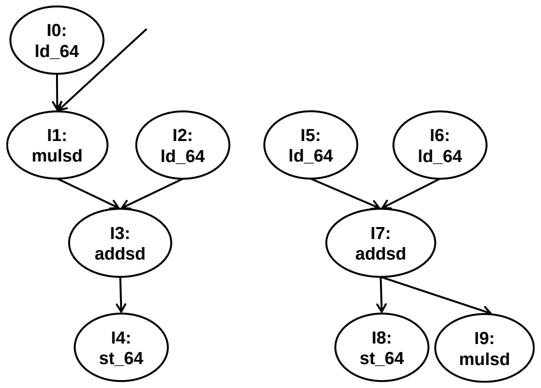
4.3.2 Avoiding Cyclic Dependences

One of the important points that should be taken care of during vectorization is that, after creation of a pack, two instructions that were earlier independent may become dependent. If we pack these instructions in a new pack, there will be a cyclic dependence in the DDG. Figure 4.3 shows an example of this scenario. Figure 4.3a shows the unvectorized code. We start vectorization by packing two consecutive and independent store instructions (*I4* and *I8*). Then following the predecessor chains we pack *I3* and *I7* also. After this step *I9* becomes dependent on *I1* as shown in Figure 4.3b, however these two instructions were independent in the original scalar code of Figure 4.3a. Therefore, we cannot select them to be packed together because it would produce a cyclic dependence.

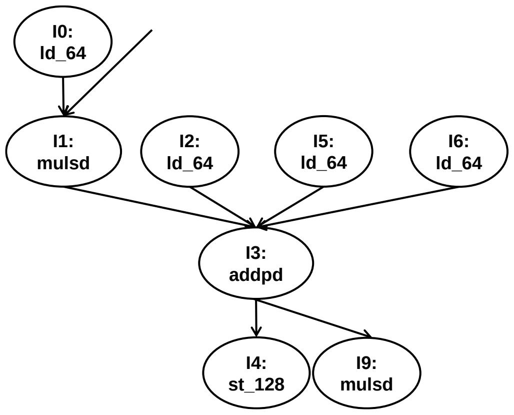
One way to solve the problem of inadvertently packing dependent instructions together is to address it during instruction scheduling and undo one of the packs involved in the cyclic dependence. However, it is not an optimal solution since dependence violation may have gotten propagated while traversing predecessor/successor chains. Therefore, we decided to update the DDG every time we create a new pack. As a result, cyclic dependences never appear in the DDG. This also allows us to check for alternative packing possibilities whereas, if we remove cyclic dependence during instruction scheduling, we cannot pack instructions of dissolved packs with other instructions.

4.3.3 Static vs Dynamic Vectorization

Loops are the basic program structures that the vectorizers target for extracting parallelism through vectorization. Several loop transformations are sometimes needed to make a loop vectorizable. The transformation like loop distribution, loop interchange, loop



a) DDG for unvectorized code.



b) DDG after vectorizing I4-I8 and I3-I7.

Figure 4.3: Additional dependence after vectorization.

peeling, node splitting, memory layout change, algorithm substitution, etc are generally applied to make a loop vectorizable. These time consuming transformations are better suited at compile time than at runtime. However, compile time vectorization suffers from several limitation like: 1) limited vectorization opportunities due to conservative memory disambiguation analysis, 2) scope of vectorization is limited to basic blocks if the loops cannot be unrolled e.g. due to complex control flow, and 3) legacy code cannot be vectorized.

The proposed speculative dynamic vectorization gets rid of all these limitations. 1) The proposed algorithm relaxes the restrictions on memory disambiguation by speculatively reordering/vectorizing the ambiguous memory references, 2) Since the scope of vectorization for the proposed algorithm is a superblock, it crosses the basic block boundaries to vectorize instructions from multiple basic blocks, and 3) Since the dynamic vectorization is applied at runtime on the program binary and not at the source code level, the legacy code can also be vectorized.

Moreover, dynamic vectorization provides some additional benefits. For example, for the loops where the number of iterations are not known statically, it is difficult to decide the unroll factor at compile time. The availability of dynamic application behavior, at runtime, allows to detect the loop unroll factor dynamically. Unrolling the loops correspondingly helps dynamic vectorizer to extract significant vectorization opportunities. Furthermore, since the dynamic vectorization is done at runtime by TOL, the vectorization algorithm can be modified to transparently target a different SIMD accelerator.

4.3.4 Working through an Example

Figure 4.4a shows the DDG for the example code of Figure 4.1b. Since the loop is unrolled once and there is no loop carried dependences, assumed speculatively, the two trees are completely separated from each other. For the sake of simplicity, we do not show loop control code in this figure. Also, pairs of ambiguous memory reference instructions like *I4* and *I6* are considered independent speculatively. As our algorithm begins with consecutive stores, the stores *I4* and *I10* are packed together as shown in Figure 4.4b. Moreover, the new store instruction is speculative one and *I6* is also converted to speculative load. Following the predecessor tree, we see that *I3* and *I9* satisfy the packing conditions and vectorize them. Notice here that *I9* writes to a live-out architectural register. As a result, we have to generate an *Unpack* instruction to write the result to the live-out register. This is shown in Figure 4.4c.

Traversing up the tree, we vectorize multiplication instructions *I1* and *I7*. One of the inputs of the multiplication instructions is a live-in to the superblock. Hence, we generate a *Pack* instruction to put the live-in values in a vector register as shown in Figure 4.4d. As explained earlier, before packing the other predecessors of additions (*I3* and *I9*), we traverse the tree up for the predecessors of *I1* and *I7*. We discover that the loads *I0* and *I6* are independent and consecutive, thus, they are packed next. Also, the new vector load instruction is speculative since *I6* was speculative, Figure 4.4e. Finally, Figure 4.4f shows the second inputs of additions (*I3* and *I9*): the two load instructions (*I2* and *I8*) are also vectorized. *Pack* and *Unpack* instructions generated to read and write architectural registers in this example can be moved outside the loop as loop invariant code during instruction scheduling, as shown in Figure 4.1c. This way, we are able to vectorize the whole loop.

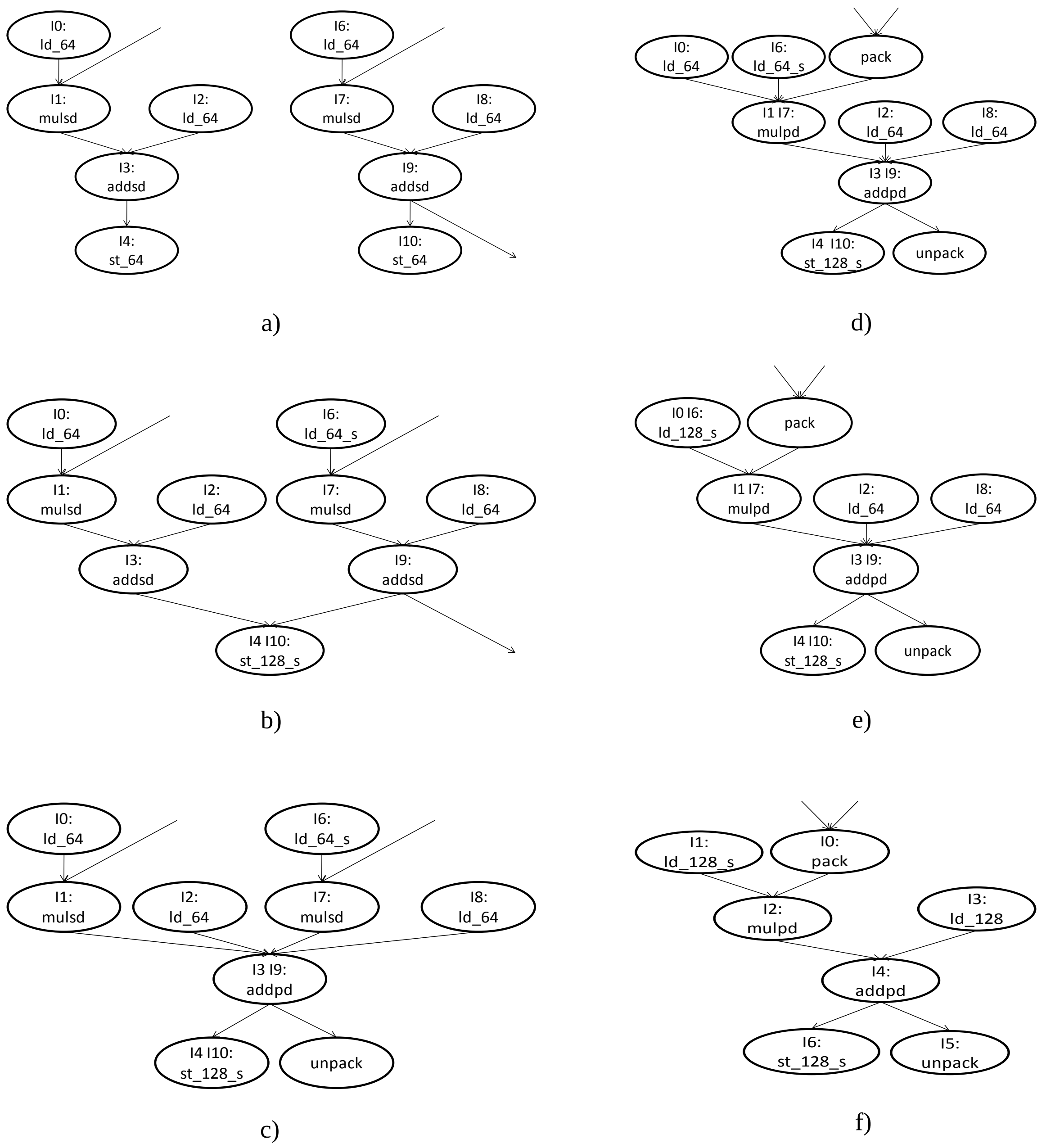


Figure 4.4: Example for vectorization of the code of Figure 4.1b. a). Shows the DDG for the loop which is unrolled once. We don´t show loop control code for the sake of simplicity. Since two iterations are completely independent we have two completely separated trees. Two arrows coming in to I1 and I7 represents live-in and arrow going out of I9 represents live-out of the superblock. Also, speculatively, we assume there is no dependence between the memory instructions until and unless its obvious b) Shows the state of DDG after vectorizing consecutive stores, also, the new store instruction is speculative one. c) Then, we follow the predecessor chains and pack addsd instructions. Since I9 writes to an architectural register, we need to unpack the results and write to the architectural register. d) Packs two mulsd instructions and since one of the inputs to both of these instructions is a live-in, a Pack instruction is also generated to pack the inputs in a single vector register. e) and f) pack remaining load instructions and f) Shows the final state.

4.4 Performance Evaluation

As explained in Chapter 3, we use DARCO [86], which is an infrastructure for evaluating HW/SW co-designed virtual machines, to evaluate our proposals. In our experiments, we assume that the host architecture supports a vector width of 128-bits. Moreover, we consider only floating point operations for vectorization (because most SIMD optimizations tend to focus on them) and no integer operation is vectorized. For this reason, we show only floating-point instructions in the results presented in this section. However, for performance study we included both floating-point as well as integer code.

In addition to SPECFP2006 and Physicsbench, we use UTDSP benchmark suite [13] as well. This benchmark suite consists of array and pointer based version of several signal processing kernels. Both versions provide identical functionality, the only difference being the use of arrays or pointers to traverse the data structures. Vectorization results for UTDSP kernels show TOL´s effectiveness in vectorizing array and pointer based code. All the benchmarks are executed till completion. Moreover, SPECFP2006 benchmarks are executed using "train" input. Furthermore, we choose only the benchmarks which have less than 150 billion dynamic instructions to keep the execution time manageable.

4.4.1 FP Dynamic Instruction Elimination

This section presents the percentage of dynamic instructions eliminated by 1) only GCC vectorizer , 2) only TOL, and 3) GCC+TOL vectorizations, first for SPECFP2006 and Physicsbench benchmarks suites and then for UTDSP Kernels. GCC and TOL represent static and dynamic vectorization respectively. For TOL vectorization the input binary is compiled by GCC but not vectorized. Also, TOL vectorization results show its effectiveness in vectorizing legacy code, since input binary is not vectorized for any SIMD accelerator. For GCC+TOL case, the input binary to TOL is already vectorized by GCC. The results of this case show the vectorization opportunities missed by GCC but captured by TOL.

Benchmarks: For SPECFP2006, on average, the combined GCC+TOL approach eliminates approximately twice the number of instructions than only the static GCC vectorization as shows in Figure 4.5. GCC+TOL vectorization outperforms GCC for all the SPECFP2006 benchmarks except for 436.cactusADM and 459.GemsFDTD. GCC completely vectorizes these benchmarks and hence TOL does not get any further vectorization opportunities. Therefore, instruction elimination is same for GCC and

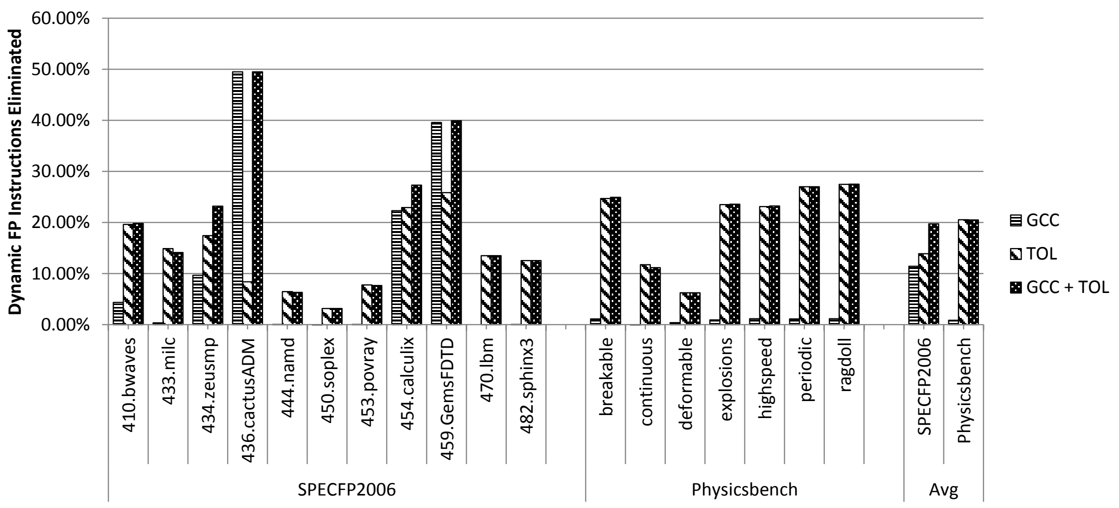


Figure 4.5: Percentage of Dynamic FP Instructions eliminated by GCC, TOL and GCC+TOL vectorizations.

GCC+TOL. It is also important to note that on average, dynamic TOL vectorization itself outperforms static GCC vectorization. Moreover, the only benchmarks where GCC outperforms TOL are again 436.cactusADM and 459.GemsFDTD. The effectiveness of TOL vectorization, to some extent, depends on the quality of the input binary. For example, for 436.cactusADM the input binary to TOL contains GCC unrolled version of the hottest loop. This GCC unrolled loop does not fit in a single superblock due to TOL´s restriction on the maximum number of instructions in a superblock. Therefore, TOL vectorizer could not vectorize it as good as GCC. For 459.GemsFDTD GCC generates significant spill-fill code (to store/retrieve temporary values to/from memory) in the frequently executed loops. This spill-fill code affects TOL´s ability to vectorize this benchmark.

GCC could not vectorize Physicsbench mainly due to the presence of complex control flow in the most frequently executed loops. TOL also is unable to unroll these loops; however, it extracts significant vectorization opportunities through superblock vectorization. Since GCC fails to vectorize anything, GCC+TOL and TOL vectorizations both eliminate 20% of dynamic instruction stream.

Kernels: Table 4.1 shows the vectorization results for UTDSP kernels. As the table shows, GCC vectorizes the array based version of FFT, LATNRM, and Matrix Multiplication (MULT) but for the pointer based version it is able to vectorize only LATNRM. On the contrary, performance of TOL is same for the array and pointer based versions for all the kernels except for IIR. Pointer based version of IIR contains control flow inside the innermost loop and hence TOL fails to vectorize it. Furthermore, once again a combination of static and dynamic vectorization, GCC+TOL, provides the best solution.

For the array based version, TOL vectorizer outperforms GCC in vectorizing IIR. GCC is unable to resolve loop carried dependences, whereas speculative vectorization helps TOL to provide an instruction reduction of 32%. On the other hand, GCC surpasses TOL vectorization for LATNRM and Matrix Multiplication (MULT). In the current version of TOL vectorizer, reductions are not implemented. Both LATNRM and MULT have reductions, which TOL fails to vectorize. Moreover, MULT has non-unit stride memory accesses, since only one dimension of the matrix (either row or column) can be accessed in the unit-stride manner. Compliers apply optimizations like “memory layout change”, “data coping”, etc to convert non-unit stride accesses to unit-stride. However, these optimizations are not directly applicable at runtime. This adds to the loss of vectorization opportunities for TOL vectorizer.

Benchmark	Type	GCC	TOL	GCC + TOL
FFT	Array	43.28%	52.70%	43.28%
	Pointer	0.00%	49.87%	49.87%
FIR	Array	0.00%	0.00%	0.00%
	Pointer	-0.08%	0.00%	-0.08%
IIR	Array	0.00%	32.52%	32.52%
	Pointer	0.00%	0.00%	0.00%
LATNRM	Array	23.48%	7.38%	20.44%
	Pointer	19.43%	17.85%	27.76%
LMSFIR	Array	0.00%	0.00%	0.00%
	Pointer	0.00%	0.00%	0.00%
MULT	Array	64.72%	17.62%	64.72%
	Pointer	0.00%	17.62%	17.62%
Avg	Array	21.91%	18.37%	26.83%
	Pointer	3.23%	14.22%	15.86%

Table 4.1: Percentage of Dynamic Instructions eliminated by GCC, TOL and GCC+TOL vectorizations.

None of the vectorization schemes is able to extract benefit for FIR and LMSFIR, mainly because of the presence of control flow inside the innermost loop. Moreover, in these benchmarks, the number of independent instructions in the basic blocks (and even in superblocks) is not enough to enable vectorization. It is also interesting to note that TOL eliminates 53% of instructions from array version of FFT, whereas GCC+TOL eliminate only 43% (as does GCC alone). This is because the input to TOL is completely vectorized by GCC and TOL does not find any vectorization opportunities, therefore the instruction reductions stays at 43% in GCC+TOL case.

4.4.2 Dynamic FP Instruction Stream Distribution

Figure 4.6 and 4.7 present dynamic FP instruction stream distribution for SPECFP2006 and Physicsbench respectively for no vectorization, GCC vectorization,

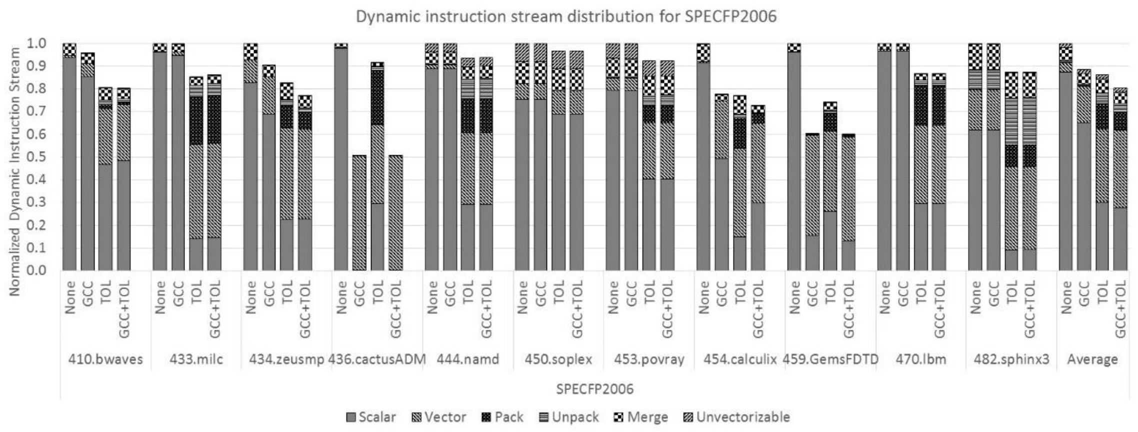


Figure 4.6: Dynamic FP instruction stream distribution for SPECFP2006: no vectorization, GCC, TOL and GCC + TOL vectorization normalized to no vectorization.

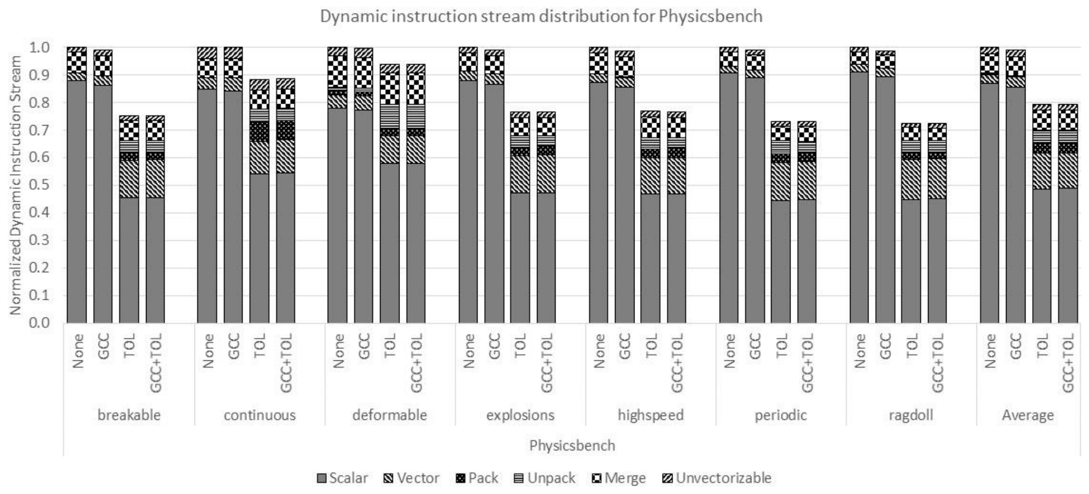


Figure 4.7: Dynamic FP instruction stream distribution for Physicsbench: no vectorization, GCC, TOL and GCC + TOL vectorization normalized to no vectorization.

TOL vectorization and GCC + TOL vectorization cases. The results shown are normalized to no vectorization case. The dynamic FP instruction stream includes: Scalar and Vector instructions, *Pack/Unpack* instructions (as described in Section 4.3.1), unvectorizable instructions (e.g. we do not vectorize conversion instructions), and Merge instructions (the instructions needed to merge correct values in live-out vector architectural registers even without vectorization).

For GCC vectorization, the majority of the dynamic instruction stream is composed of scalar instructions. However, for TOL and GCC + TOL vectorizations the percentage of scalar instructions falls to 30% and 28% for SPECFP2006 and 48% and 48% for Physicsbench respectively. Furthermore, even though scalar instructions form much smaller (30% and 28%) part of the vectorized dynamic instruction stream in SPECFP2006 than Physicsbench (48%), the overall dynamic instruction stream for both benchmarks suites is reduced by the same amount, almost 20%, by TOL and GCC + TOL vectorizations. The reason lies in the fact that SPECFP2006 benchmarks operate on 64-bit double-precision floating-point variables whereas, Physicsbench benchmarks are composed of 32-bit single-precision floating-point variables. As a result, for a vector length of 128-bits, a single vector instruction in Physicsbench replaces four scalar instructions whereas, in SPECFP2006 a vector instruction replaces only two scalar instruction. Therefore, SPECFP2006 needs more vector instructions to replace the same number of scalar instructions than Physicsbench. The fact is also evident in Figure 4.6 and 4.7 where the vector instructions form 32% and 34% of vectorized instruction stream in SPECFP2006 and only 13% in Physicsbench for TOL and GCC + TOL vectorizations.

In addition, *Pack* and *Unpack* instructions also form a moderate fraction of the vectorized dynamic instructions stream. For TOL and GCC + TOL vectorizations, they constitute 15% and 12% of vectorized dynamic instruction stream for SPECFP2006 and 8% for Physicsbench. Unvectorizable instructions on the other hand are negligible in both the benchmark suites.

4.4.3 Vectorization Overhead

Vectorization overhead is the fraction of dynamic instruction stream that corresponds to the vectorization of superblocks by TOL. A high vectorization overhead might offset the benefits of the vectorization. We calculate the vectorization overhead as:

$$= \frac{Total\ overhead_{with\ vectorization} - Total\ overhead_{without\ vectorization}}{Total\ number\ of\ dynamic\ instructions_{without\ vectorization}}$$

Our experimental results show that, on average, the vectorization overhead is less than 0.5% of the dynamic instruction stream, for all the benchmark suites. Hence, the dynamic vectorization overhead is negligible compared to its benefits. There are two main factors that make the vectorization overhead to be negligible:

1) Since vectorization is performed at superblock level, the “superblock overhead” is the only overhead component that would increase. Moreover, as shown in Figure 3.12 in Chapter 3, the “superblock creation” overhead accounts for only 14% and 9% of the overall overhead for SPECFP2006 and Physicsbench respectively. Therefore, an increase in this component has minimal effect on the overall overhead.

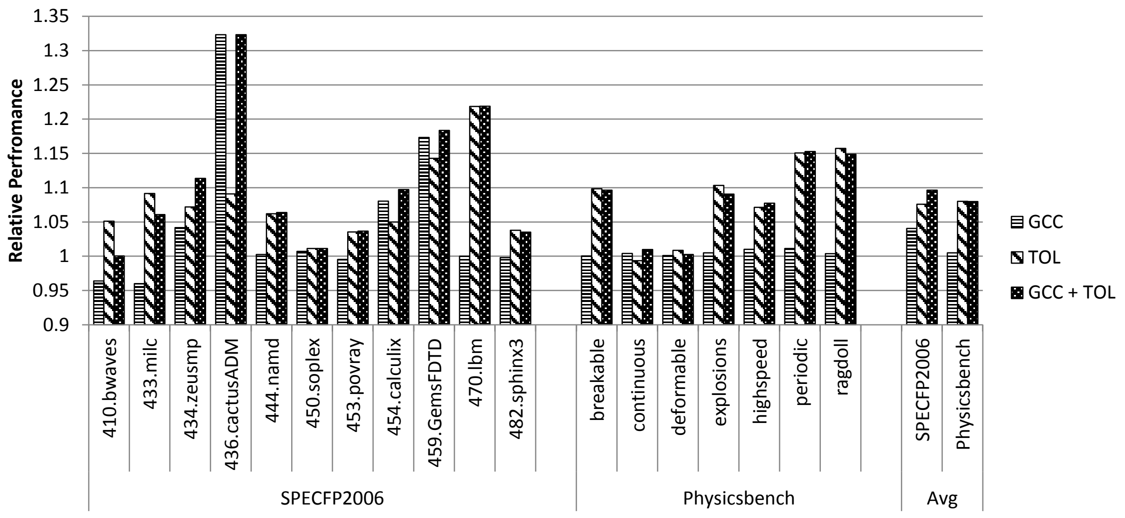


Figure 4.8: Execution speed for GCC, TOL and GCC + TOL vectorized code relative to unvectorized code. Higher is better.

2) Since vectorization reduces the total number of instructions in a superblock, the optimizations following the Vectorization pass, namely instruction scheduling, register allocation, and host code generation, now have to optimize lesser instructions. Therefore, the overhead of these optimization steps also reduces. As a result, total increase in the overall overhead is insignificant.

4.4.4 Effectiveness of Memory Speculation

One of the main factors in the success of the proposed vectorization scheme is the memory speculation. However, it might backfire if there are lots of speculation failures. A speculation failure results in executing un-optimized (and without TOL vectorization) version of the code and if the rate of speculation failure exceeds a predetermined threshold, recreating the superblock without speculation. However, our results show that, on average, we execute more than 99% of the dynamic superblocks in speculation mode. It reflects the fact that the number of speculation failures, and hence the overhead associated with it, is negligible.

4.4.5 Performance

For the performance analysis, both the floating point and integer instructions are considered, even though TOL vectorizes only the floating point code. Figure 4.8 shows the performance of the vectorized code using the different vectorization schemes relative to the unvectorized code, for SPECFP2006 and Physicsbench. The performance results in the figure conform to the results of Figure 4.5 for dynamic instruction elimination. For

SPECFP2006, GCC+TOL vectorization provides twice the performance benefit than GCC alone (10% compares to 5% of GCC alone). Also, TOL vectorization alone provides better performance than GCC alone. It is interesting to note that for 410.bwaves and 433.milc GCC vectorized code gets a slowdown even though Figure 4.5 shows dynamic FP instruction elimination. The slowdown comes because of the integer code. GCC adds more integer code than it vectorizes, hence suffers a slowdown. Moreover, for these benchmarks GCC+TOL provides worse performance than TOL alone because GCC+TOL vectorizes GCC vectorized input with extra integer code whereas TOL vectorizes unvectorized code.

As GCC fails to vectorize anything in Physicsbench it does not show any performance improvements. However, similar to the results of Figure 4.8, GCC+TOL and TOL vectorizations provide similar performance benefits for Physicsbench.

An interesting thing to note is that in Figure 4.8 GCC+TOL vectorization, on average, eliminates 20% of the dynamic instruction stream for both SPECFP2006 and Physicsbench. However, SPECFP2006 gets more speed up than Physicsbench as shown in Figure 4.8. This is because percentage of floating point code is more in SPECFP2006 than in Physicsbench as shown in Figure 3.13 in Chapter 3.

Table 4.2 shows the speedup for UTDSP kernels. These results also conform to the results of Table 4.1. For the pointer based version of the kernels GCC loses significant performance compared to the array based version. However, performance is not affected a lot for TOL vectorizer. Furthermore, the combination of static and dynamic vectorizations, GCC+TOL, is able to extract maximum performance out of the kernels.

Benchmark	Type	GCC	TOL	GCC + TOL
FFT	Array	1.26	1.50	1.26
	Pointer	1.00	1.50	1.50
FIR	Array	1.00	1.00	1.00
	Pointer	1.05	1.00	1.05
IIR	Array	1.00	1.29	1.29
	Pointer	1.00	1.00	1.00
LATNRM	Array	1.39	1.03	1.33
	Pointer	1.31	1.13	1.39
LMSFIR	Array	1.00	1.00	1.00
	Pointer	1.03	1.00	1.03
MULT	Array	2.33	1.07	2.33
	Pointer	1.17	1.23	1.16
Avg	Array	1.33	1.15	1.37
	Pointer	1.09	1.14	1.19

Table 4.2: Execution speed for GCC, TOL, and GCC + TOL vectorized code relative to unvectorized code. Higher is better.

4.5 Related Work

Speculative Dynamic Vectorization is not a much extended topic in literature. There have only been a few proposals like Speculative Dynamic Vectorization [83] and Dynamic Vectorization in Trace Processors [112]. None of them is in the context of HW/SW co-designed processors.

A. Pajuelo et al. [83] proposed to speculatively vectorize the dynamic instruction stream in the hardware for superscalar architectures. Their scheme prefetches data into the vector registers and speculatively manipulates it through arithmetic instructions. Moreover scalar instructions that are converted into vectors are not eliminated but are converted into 'check' operations to validate whether the operands used by the corresponding vector instruction were correct or not. Several hardware structures are added to support speculative dynamic vectorization, which is not a power efficient solution, especially in out-of-order superscalar processors where power consumption is already a big issue. They report, more than half of the speculative work is unless due to mispredictions, whereas the rate of speculation failure is negligible in our case. S. Vajapeyam et al. [112] builds a large logical instruction window and converts repetitive dynamic instructions from different iterations of a loop into vector form. The whole loop is vectorized if all iterations of the loop have the same control flow.

HW/SW Co-designed processors like Transmeta Crusoe [36], BOA [91], etc. apply several dynamic optimizations at runtime and evaluate their contribution in improving overall performance. Also, software dynamic binary optimizers like Dynamo [21], IA-32 [22], and hardware dynamic binary optimizers like rePLay [85] and PARROT [90] report performance improvements by applying on the fly optimizations. However, none of these systems have proposed vectorization at runtime. Y. Almog et al. [17] briefly point out that one of the optimizations applied in their system is SIMDification. Unfortunately, details of their vectorization scheme are not provided in the paper.

Liquid SIMD [34] decouples the SIMD accelerator implementation from the instruction set of the processor by compiler support and a hardware based dynamic translator. Similarly, Vapor SIMD [82] provides a just-in-time compilation solution for targeting different SIMD architectures. Thus, both solutions eliminate the problem of binary compatibility and software migration. However, both need compiler changes and recompilation. J. Li et al. [69] propose a runtime algorithm for mapping guest vector registers to host vector registers when guest ISA vectors registers support more data types than host ISA vector registers.

4.6 Conclusion

This chapter proposed to assist the static compiler vectorization with a complementary dynamic vectorization. Static vectorization applies complex and time consuming loop transformations at compile time to vectorize a loop. Subsequently at runtime, dynamic vectorization extracts vectorization opportunities missed by static vectorizer due to conservative memory disambiguation analysis and limited vectorization scope. Furthermore, the chapter proposed a vectorization algorithm that speculatively reorders ambiguous memory references to facilitate vectorization. The hardware, using the existing speculation and recovery support, checks for any memory dependence violation and takes corrective action in that case.

Our experimental results show that the combined static and dynamic vectorization improves the performance twice compared to static vectorization alone for SPECFP2006. Furthermore, we show that the proposed dynamic vectorization performs as good for pointer based applications as for the array based ones. However, GCC vectorization loses significant opportunities when source code utilizes pointers. Moreover, the overhead of runtime vectorization is only 0.5%.

Chapter 5

Vectorizing for Wider Vector Units

Due to their hardware simplicity, SIMD accelerators have evolved in terms of width from 64-bit vectors in Intel´s MMX to 512-bit wide vector units in Intel´s Xeon Phi. This chapter explores the scalability of SIMD accelerators from the code generation point of view. We explore the potential problems in vectorization at higher vector lengths. Furthermore, we propose Variable Length Vectorization and Selective Writing in a HW/SW co-designed environment to get around these problems.

5.1 Introduction

SIMD accelerators form an integral part of modern microprocessors. Since these accelerators perform the same operation on multiple pieces of data, they just require duplicated functional units and a very simple control mechanism. Due to this hardware simplicity they are relatively easy to scale to higher vector lengths. As a result, SIMD accelerators grow in size with each new generation. For example, Intel´s MMX [4] had vector length of 64-bits, which was increased to 128-bits in SSE [4] extensions. Intel´s recent SIMD extensions AVX [4] and AVX2 [4] support 256-bit wider vectors. Furthermore, Intel Xeon Phi [12] and its visual computing architecture Larrabee [93] perform 512-bit wide vector operations.

Although SIMD accelerators are amenable to scaling from the hardware point of view, generating efficient code for higher vector lengths is not straightforward. 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.

In this chapter, we explore the scalability of SIMD accelerators from the code generation point of view. We discover that there are two key factors that thwart the performance of applications at higher vector lengths: reduced dynamic instruction stream coverage for vectorization and huge number of permutation instructions. We propose Variable Length Vectorization and Selective Writing to tackle these problems. Our experimental results show average dynamic instruction elimination of 25% and speed up of 13% for SPECFP2006, for 512-bit vector length.

The rest of the chapter begins by briefly providing the motivation for the work presented and identifying key issues in efficient vector code generation for higher vector lengths in Section 5.2. Then, Section 5.3 and 5.4 provide details of the proposed Variable Length Vectorization and Selective Writing techniques, respectively. Evaluation of the proposals using a set of SPECFP2006 and Physicsbench applications is presented in Section 5.5. Section 5.6 presents related work and Section 5.7 concludes.

5.2 Motivation

The trends in the recent past have shown that vector lengths are going to increase in the future microprocessors, since it provides a simple and efficient way of achieving higher FLOPS in an energy efficient manner. Intel´s 256-bit AVX and 512-bit vector length of Xeon Phi and Larrabee are few examples of these trends. However, it is a challenge to generate efficient code to utilize these wider vector units. To demonstrate this fact, we vectorized floating point instructions in SPECFP2006 for three different vector lengths of 128, 256, and 512-bits using the speculative dynamic vectorization algorithm described in Chapter 4. Moreover, at a given vector length, all the vector instructions operate only on the maximum vector length and not on a subset of it. For example, for 512-bit vector length case, all the vector instructions operate on whole 512-bits and there is no vector instruction that operates only on 256 or 128-bits.

Our results show that there are mainly two problems in vector code generation at higher vector lengths: reduced dynamic instruction stream coverage for vectorization and huge number of permutation instructions.

5.2.1 Reduced Dynamic Instruction Stream Coverage

We define dynamic instruction stream coverage as the number of dynamic scalar instructions vectorized. Figure 5.1 shows the dynamic instruction stream coverage for vectorization at different vector lengths normalized to the 128-bit case. Best, worst and average cases are shown. We divide the applications in two categories: The first category applications have maximum dynamic instruction stream coverage at all the vector lengths, like 454.calculix. On the contrary, there are applications like 444.namd where dynamic instruction steam coverage falls by 70% at vector length of 512-bits.

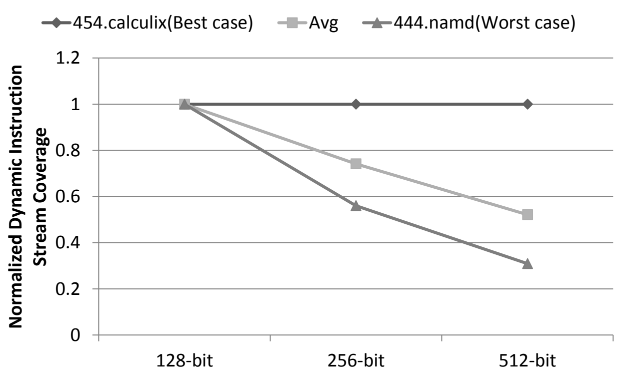


Figure 5.1: Dynamic FP Instruction Stream Coverage for vectorization at 128, 256 and 512 bit vector lengths normalized to 128-bit case.

The dynamic instruction stream coverage at different vector lengths depends upon the degree of data level parallelism available in the application, or in other words the natural vector length of the application. If an application spends most of its time in loops with higher trip counts, it will benefit from higher vector lengths, since the wider vector paths can be filled by unrolling the loops more number of times depending on the vector length. However, as shown by the average case of Figure 5.1, this is not the case for most of the applications. We see an average reduction of 25% and 48% in dynamic instruction stream coverage at 256-bit and 512-bit respectively. If this trend continues, the coverage is going to be even lesser at higher vector lengths.

5.2.2 Number of Permutation Instructions

When the input operands of a vector instruction are not available in a single vector register or are not in the same order as required by the vector instruction, permutation instructions are needed to arrange them in the correct order. Our results show that the number of permutation instructions grows significantly with increasing vector lengths.

Figure 5.2 shows the number of permutation instructions generated per vector instruction in SPECFP2006 normalized to the 128-bit case. As the figure shows, if we generate one permutation instruction for each vector instruction at 128-bit vector length, this number goes as high as 10 at 512-bit vectors in case of 444.namd. Also, there are applications for which this number does not grow that rapidly. However, the average behavior suggests that number of permutation instructions is going to be a problem at higher vector lengths.

Both of these factors become a limitation as vector paths become wider and instead of performance improvements, it starts degrading compared to the lower vector lengths. This chapter investigates both problems and proposes Variable Length Vectorization and

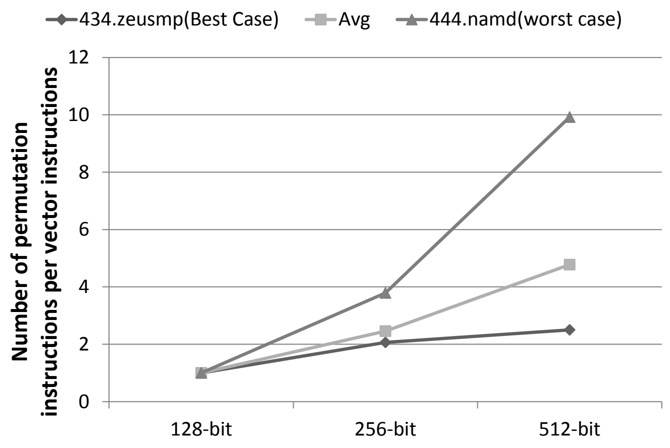


Figure 5.2: Number of permutation instructions generated per vector instruction at 128, 256, and 512 vector lengths normalized to 128-bit case.

Selective Writing to solve the problems of reduced coverage and permutation instructions, respectively.

5.3 Variable Length Vectorization

Vector instructions in the current architectures, generally, operate on all the elements of the source vector registers and write the whole destination register. Due to this reason, compilers generate a vector instruction only when there are sufficient numbers of independent operations to fill the vector path. When there are not enough instructions to fill up the vector path, all the instructions are left in scalar form. This is going to be an important issue in the future microprocessors with wider vector paths and a lot of, otherwise vectorizable, code will be left unvectorized. We propose Variable Length Vectorization (VLV) using masked vector instructions to vectorize the scalar code when it is not possible to fill the vector path entirely.

An important factor to consider here is the need of masking. Masking is used to disable unused vector lanes when a vector instruction does not use all the lanes. In general, not masking the unused lanes might work well for arithmetic instructions from the functionality point of view. However, the register file will contain invalid data because whole destination register will be written. Therefore, we would need a way to distinguish between invalid and valid data in the register file. Mixing the architectural state and temporal values is typically not a good idea. Moreover, performing unnecessary operations in the unused lanes might also generate exceptions, like divide by zero. Therefore, we would need a way to distinguish real and false exceptions. Furthermore, for memory access instructions this might result in crossing array boundaries and leading to page/segmentation faults. Also, for store instructions it would result in writing incorrect data to the memory. On the other hand, masking the unused lanes helps us get rid of all these problems.

5.3.1 Code Generation

We modify our baseline speculative dynamic vectorization algorithm of Chapter 4 to generate vector code with variable vector length. The modified algorithm starts by vectorizing for the given maximum vector length, we call it physical vector length. Once all the possible packs for the physical vector length have been created, the vectorizer reduces the logical vector length iteratively. At lower logical vector lengths, packs are created with smaller number of scalar instructions than required to fill the vector path. The left out positions in a pack are considered as no operations (NOP).

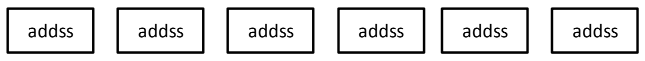
Since, the number of operations in the vector instructions varies depending on the logical vector length; we need a way to notify the hardware which vector lanes to enable and which ones not. We make use of mask registers for this purpose. Mask register has one bit per vector lane. The bits containing ones signify the corresponding vector lanes are to be enabled; 0 means otherwise. We include the mask register in instruction encoding in addition to the regular source and destination registers. The new instruction encoding for vector instructions is shown in Figure 5.3.

opcode	dest	src1	src2	mask reg	unused

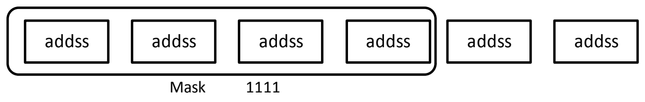
Figure 5.3: Instruction format for masked vector instructions.

Figure 5.4 shows a simple vectorization example using the proposed VLV algorithm. Figure 5.4a shows unvectorized code having six independent single-precision floating-point addition (32-bit) instructions. For a vector length of 128-bits, we can pack a maximum of four single-precision floating-point additions in a single vector addition instruction. The algorithm first packs four of the six instructions in a vector instruction and assigns a mask register with all ones to this instruction, as shown in Figure 5.4b. A mask register with all ones signifies that all the vector lanes are to be enabled.

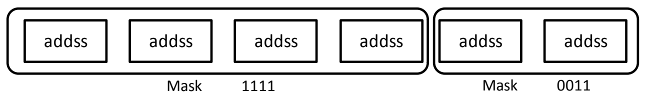
A fixed vector length vectorization algorithm will stop at this point, since there are just two ADDSS instructions left and at least four are required to generate a vector instruction. However, VLV algorithm continues and packs the remaining two addition instructions as shown in Figure 5.4c. Moreover, a mask register with ones only at lowest two positions is assigned to this instruction. It makes sure that only the two lower vector lanes are enabled during the execution of this vector instruction.



a) Unvectorized code.



b) Vectorized code for fixed vector length of 128-bits.



c) Vectorized code with variable length vectorization.

Figure 5.4: Variable Length Vectorization Example.

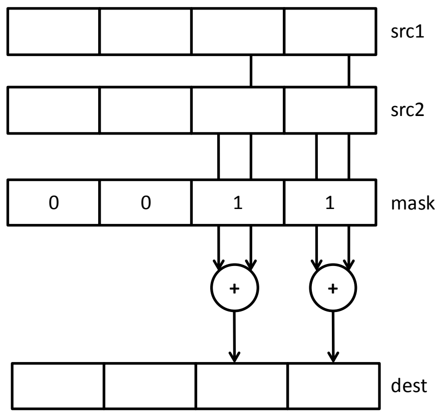


Figure 5.5: Masked Vector Instruction Execution.

5.3.2 Hardware Requirements

From the hardware perspective, we do not really need to have real mask registers in the hardware. Since we need to enable only consecutive lower order vector lanes, the number of lanes to be activated can directly be encoded in the instructions encoding. This also saves upon the extra instructions, otherwise, needed to write the mask in the registers. It is important to note that the traditional vector processors support variable vector length through a vector length register. It needs to be set to the desired vector length before executing vector instructions. However, it is not the optimal solutions for the processors targeting general purpose applications, where the vector length needs to be changed frequently. In this scenario, overhead of writing the vector length register would affect the performance severely. Therefore, instead of having a variable vector length register we propose to have Variable Length Vectorization using masked vector instructions.

For the execution of a vector instruction, the hardware now reads not only the source registers but also a mask to enable only the required vector lanes. Figure 5.5 shows the execution of second vector instruction generated in the example vectorization sequence of Figure 5.4. As shown in the figure only two of the four vector lanes are activated. This is also important from the power consumption point of view, not to activate all the vector lanes for all the vector instructions.

Variable Length Vectorization helps in vectorizing the applications which have loops with lower iterations count than required by the vector length and the straight line code with fewer independent scalar operations.

5.4 Selective Writing

This section presents the proposed Selective Writing (SWR) technique to reduce the number of permutation instructions at higher vector lengths. First, we present a technique to eliminate permutation instructions completely if the result of an instruction is read only by one instruction. Then, we present another technique to reduce the number of instructions required to pack N values from N-1 to N/2, if the values to be packed are in N different registers.

5.4.1 Eliminating Permutation using Selective Writing

If the producer instructions of a vector instruction cannot be vectorized, the results of these instructions have to be packed together before feeding the vector instruction. This is due to the fact that the scalar producer instructions write their results to the lowest element of different vector registers, whereas the vector instruction needs them to be in a single vector register and in a particular order.

Figure 5.6a shows a situation where producers of *I7* (*I0-I3*) are not vectorized and their results are packed using a permutation instruction sequence (*I4-I6*). As shown in the figure, *I0* to *I3* write their results to the lowest elements of different vector registers. Then a sequence of three instructions, *I4* to *I6*, is used to pack these results in a single vector register *xmm3*, before feeding it to the vector instruction *I7*.

```
I0   addss     xmm0, xmm6
I1   addss     xmm1, xmm6
I2   mulss     xmm2, xmm7
I3   mulss     xmm3, xmm7
I4   shufps    xmm1, xmm0, imm
I5   shufps    xmm3, xmm2, imm
I6   blendps   xmm3, xmm1, imm
I7   addps     xmm3, [M]
```

a) Traditional code sequence.

```
I0   addss     vr4, vr0, vr6, imm
I1   addss     vr4, vr1, vr6, imm
I2   mulss     vr4, vr2, vr7, imm
I3   mulss     vr4, vr3, vr7, imm
I4   addps     vr5, vr4, [M]
```

b) Proposed instruction sequence.

Figure 5.6: Packing scalar instruction results for feeding a vector instruction.

If the scalar instructions can write their results to any element of a vector register, instead of always writing to the lowest element, we can get rid of the permutation instructions. It can be done by making the scalar instructions to selectively write in the different elements of a vector register in the order they are needed by the vector instruction. This way, we can avoid putting permutation instructions altogether. This kind of selective writing capability is already available in the memory access instruction set of current architectures. For example, INSERTPS in Intel´s SSE can be used to write a 32-bit value loaded from memory to any part of the destination register. We extend this capability to the arithmetic instruction set as well. The proposed instruction format for the scalar arithmetic instructions is shown in the Figure 5.7 and the functionality in Figure 5.8.

opcode	dest	src1	src2	immd	unused

Figure 5.7: Proposed instruction format for scalar instructions.

As shown in the Figure 5.7, in addition to carry source and destination register numbers, all scalar arithmetic instructions also carry an immediate that specifies to which element of the destination vector register the scalar result is to be written.

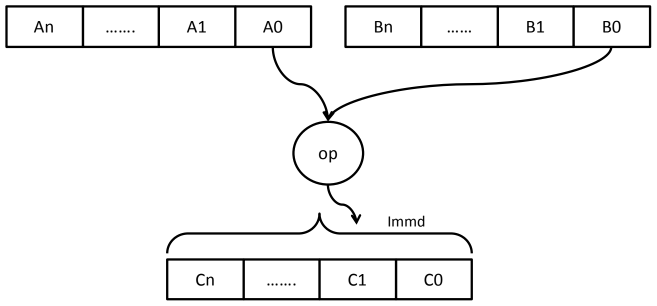


Figure 5.8: Functionality of the proposed arithmetic scalar instructions.

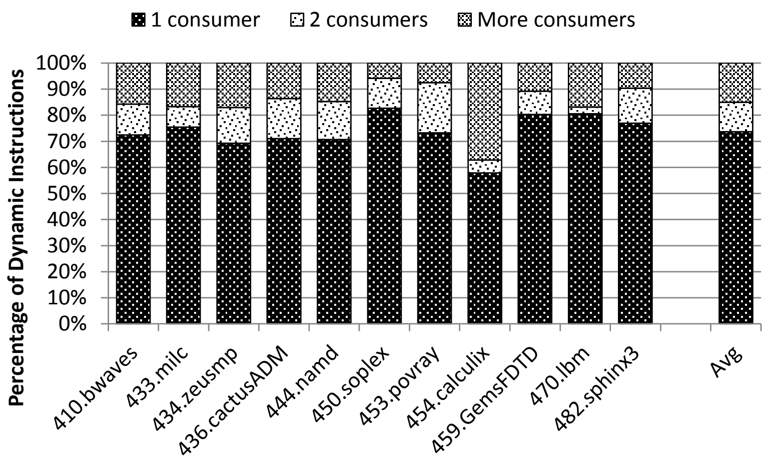


Figure 5.9: Percentage of Dynamic Instructions with one, two and more number of consumers.

If scalar instructions have written their results to a single vector register in the order in which they are needed by the vector instruction, the instruction sequence for packing these results is not needed anymore as shown in Figure 5.6b.

The limitation of SWR scheme is that it works as long as the result of a scalar instruction is consumed only by one instruction. In the case of more than one consumer, we would not get the maximum benefit out of SWR. However, our analysis of SPECFP2006 shows that more than 70% of dynamic instructions have only one consumer, as shown in Figure 5.9.

The proposed scalar instructions can be viewed as an arithmetic operation followed by a shuffle. However, this does not affect the latency of these instructions, since the results can be forwarded as soon as the arithmetic operation is finished. As Figure 5.10 shows, it requires only an additional input to the multiplexers, selecting input operands of the ALUs

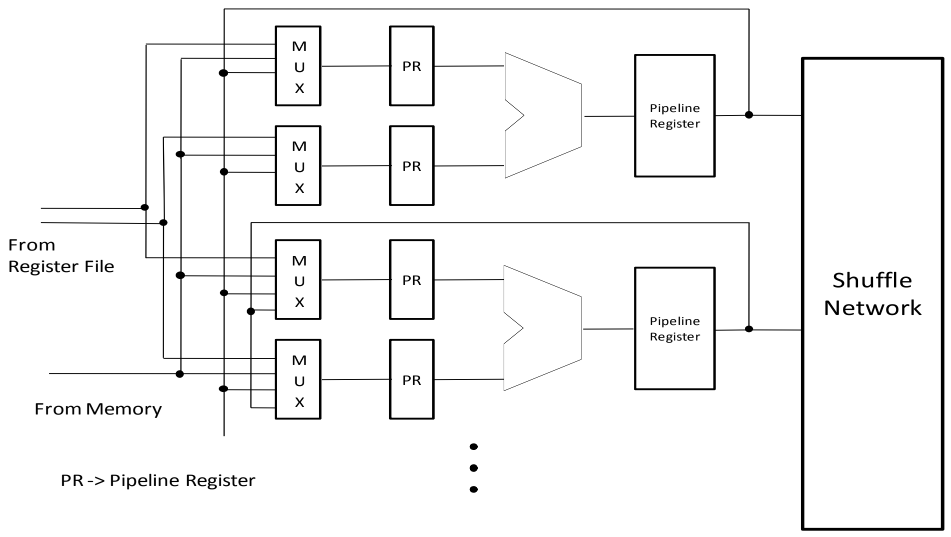


Figure 5.10: Operand forwarding before shuffle.

from the output of the first vector lane (which performs scalar operations). Consequently, forwarding the results of the first vector lane to any other vector lane provides the functionality of a shuffle operation.

5.4.2 Reducing Permutation Instruction to Pack N Values

Current architectures provide vector instruction set where N-1 instructions are required to bring N values to a register. A typical instruction sequence to bring 4 values from different vector registers to single vector register in x86 architecture is shown in Figure 5.11a. The first two shuffle instructions bring values selected by the immediate into register *xmm1* and *xmm3*, respectively. Then a BLENDPS instruction is used to combine the results from *xmm1* and *xmm3* into *xmm3*.

One of the main factors that force this instruction count to be N-1 is that, these instructions write to all the elements of the destination register. If it is possible to write only the selective elements of the destination register, then this number can be brought down. In this case, the number of instructions required will depend upon the total number of different registers to be read and the number of registers that can be read by a single permutation instruction. In a case where we need to read N registers and the permutation instruction can read only two registers, we would need N/2 instructions to collect N values in a single register. If we support more number of input registers, the number of instructions required can be brought further down. Moreover, we need a mechanism to tell which elements of the source registers are to be read and which elements of the destination register are to be written.

```
I0   shufps    xmm1, xmm0, imm
I1   shufps    xmm3, xmm2, imm
I2   blendps   xmm3, xmm1, imm
```

a) x86 instruction sequence.

```
I0   packps    vr6, vr0, vr1, imm
I1   packps    vr6, vr2, vr3, imm
```

b) Proposed instruction sequence.

Figure 5.11: Instruction sequence for packing 4 values from different registers into a single register.

We propose to have a permutation instruction with the instruction format shown in Figure 5.12 and the functionality in Figure 5.13.

The proposed instruction (PACKPS) has two input registers and a 16-bit immediate that tells which elements of the source and destination registers are to be accessed. The first four bits of the immediate [0:3] tells which element of the first source register is to be read and the next four bits [4:7] tell where it is to be written in the destination. Similarly, bits [8:11] tell which element of the second source register is to be written to the destination element selected by the bits [12:15]. Note that PACKPS is very similar to SHUFPS but with a bit more freedom in choosing source element for each destination element. Therefore, their latencies will be similar.

opcode	dest	src1	src2	16-bit imm

Figure 5.12: Instruction format for the proposed permutation instruction (PACKPS).

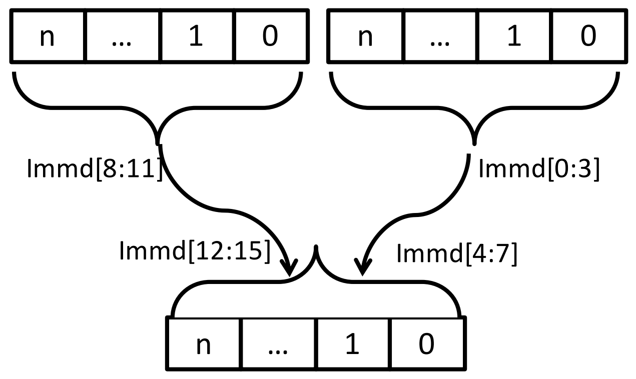


Figure 5.13: Functionality of the proposed Pack instruction.

Number of registers to be read	**256-bit**	**512-bit**
N	54%	32%
N-1	30%	18%
N-2	12%	20%

Table 5.1: Percentage of permutations requiring N, N-1 and N-2 input registers to pack N values, for 256 and 512-bit vectors.

The instruction sequence for replacing x86 instruction sequence of Figure 5.11a is shown in Figure 5.11b. In this case, we are able to reduce the number of instructions required to two. For higher vector lengths, where we need to get 8 and 16 values in a register, we need just 4 and 8 instructions, respectively, instead of 7 and 15 instructions required by the original sequence. The down side of this scheme is that it requires N/2 instructions even if the values to be collected are in less than N number of registers. However, our experiments show that in SPECFP2006, on average, about 84% and 50% of permutations, for 256-bit and 512-bit vectors respectively, need to read N or N-1 registers to pack N values as shown in Table 5.1.

Discussion

VLV and SWR are well suited to HW/SW co-designed processors than traditional microprocessors mainly due to two reasons: First, they require significant ISA changes, since all the vector instructions now carry a mask register and the scalar instructions carry an immediate. It can be achieved in HW/SW co-designed processors transparently to the user/compilers but not in the traditional microprocessors. Second, the VLV algorithm is fairly simple to extend to compilers for the static trip count loops, however for loops with unknown trip count at compile time it becomes tricky. For fixed vector length, compiler can vectorize such loops by unrolling them enough number of times to fill the vector path and putting a runtime check before the vectorized version to decide whether to execute it or not. However, for variable length vectorization, choosing a single unroll factor becomes difficult at compile time. The runtime information of the program behavior in HW/SW co-designed processors makes it straightforward to choose the correct unroll factor.

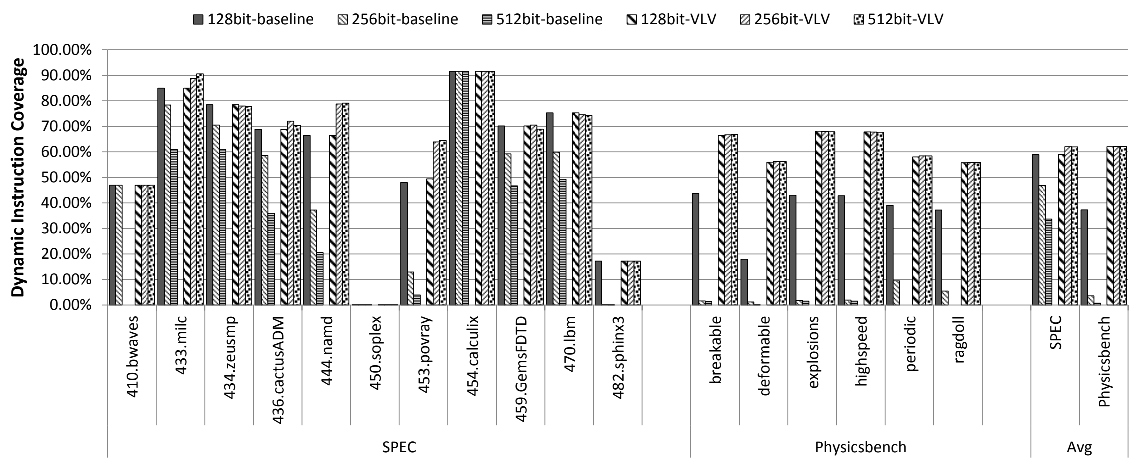


Figure 5.14: Dynamic Instructions stream coverage at three vector lengths, baseline and with VLV.

5.5 Performance Evaluation

For our experiments, we extended the host architecture to supports vector sizes of 128, 256, and 512-bits. Like in Chapter 4, we vectorize only the floating-point code and report only the number of floating-point instructions in the results shown in this section. Performance results include both integer as well as floating-point code.

5.5.1 Dynamic Instruction Stream Coverage

Figure 5.14 shows the dynamic instruction stream coverage for three vector lengths first without and then with Variable Length Vectorization (VLV). We will have maximum coverage when the number of instructions required to create a pack is minimum, i.e. two instructions. At 128-bit vector length the maximum number of 64-bit double precision operations that can be packed together is two. Therefore, 128-bit vector length provides maximum coverage, even without VLV, for double precision operations. Since all the SPECFP2006 benchmarks primarily operate on double precision floating point variables, they have maximum coverage at 128-bits as shown in Figure 5.14. For single precision floating point variables, Variable Length Vectorization helps increasing coverage even at 128-bit vector length, as is evident from the figure, for Physicsbench benchmark suite.

For the vector lengths of 256-bit and 512-bits, the benchmarks can be divided into two categories. First, the benchmarks like 454.calculix have maximum, or close to maximum, dynamic instruction stream coverage at higher vector lengths also. The hottest loops of these benchmarks have enough iterations to fill the wider vector paths. Second, the benchmarks like 436.cactusADM, 444.namd, and Physicsbench show drastic reduction

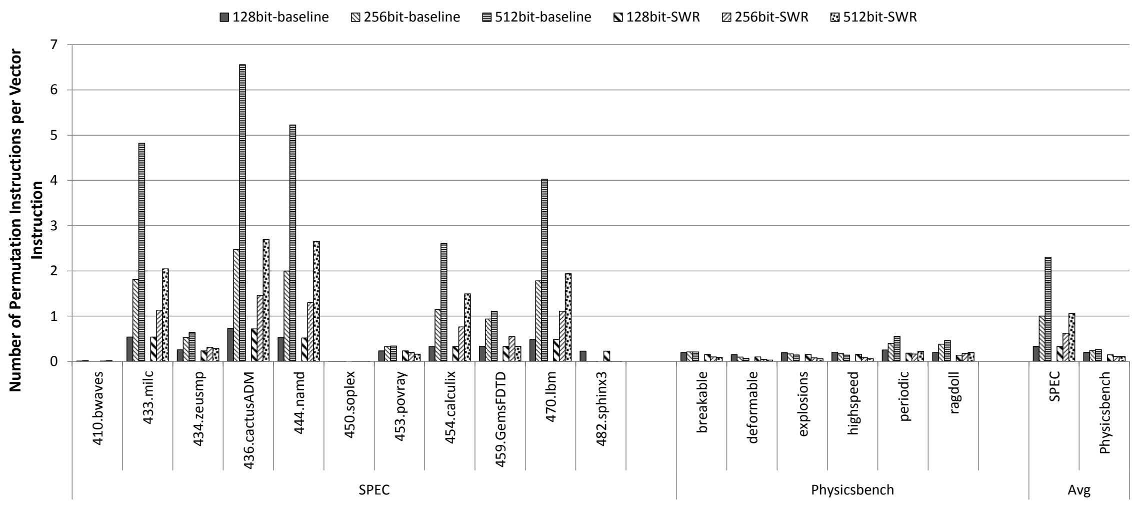


Figure 5.15: Number of Permutation Instructions per vector instruction, baseline and with SWR.

in coverage as vector length increases, due to the lack of independent instructions to fill the wider paths. These benchmarks either have loops with fewer iterations or with complex control flow. For example, the hottest loops in 410.bwave iterate four times, therefore, for 256-bit vector length it has the maximum coverage but for 512-bit, it drops down to zero. Benchmarks in Physicsbench have loops with complex control flow and cannot be unrolled. Moreover, number of independent instruction in individual superblocks is not enough to fill the vector path. Thus, the dynamic instruction stream coverage reduces severely. Using VLV, we bring the coverage for these benchmarks also to the maximum as shown in the Figure 5.14.

5.5.2 Permutation Reduction

Figure 5.15 shows the number of permutation instructions per vector instruction required at three vector lengths without and with Selective Writing (SWR). Again, we have the same two categories of benchmarks as for the dynamic instruction stream coverage. Benchmarks like 434.zeusmp, 459.GemsFDTD, and Physicsbench have, essentially, the same amount of permutation instructions across all the vector lengths. Packing the instructions from the different iterations of unrolled loops avoids generation of permutation instructions in the case of 434.zeusmp and 459.GemsFDTD. Physicsbench, however, has really less number of permutations since we fail to vectorize anything. On the contrary, 433.milc, 436.cactusADM and 444.namd show an increase in the permutation instructions at higher vector lengths. Complex control flow and lack of number of loop iterations forces us to vectorize straight line code which require higher number of permutation instructions.

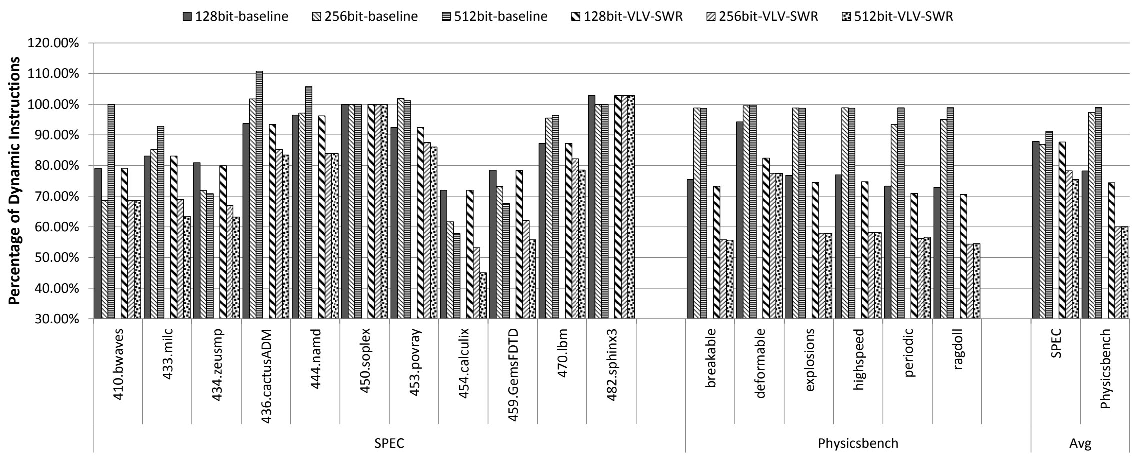


Figure 5.16: Dynamic Instruction Percentage after baseline and VLV-SWR vectorizations.

SWR helps in eliminating significant number of permutation instructions for these benchmarks.

Another point to notice in Figure 5.15 is that for 128-bit vector length there is negligible reduction in permutation instructions. This is because we need to pack two double precision values in a 128-bit register and for N=2, N/2 and N-1 are same. Therefore, we do not get much benefit. However, on average we reduce the number of permutation instruction required to half.

5.5.3 Putting Everything Together

Figure 5.16 shows the percentage of dynamic instructions after vectorization without and with VLV-SWR. As shown in this figure, after applying both the optimizations all the applications perform better as vector length is increased. Applications like 433.milc, 436.cactusADM, 470.lbm, and Physicsbench which were earlier getting worse with increase in the vector length, compared to 128-bit vector length; now perform better. On average, VLV-SWR help eliminating 9% and 16% more dynamic instructions compared to the baseline vectorization, at 256-bit and 512-bit vector lengths respectively, for SPECFP2006. Overall, vectorization with VLV-SWR reduce unvectorized dynamic instruction stream by 13%, 22%, and 25% for 128-bit, 256-bit, and 512-bit vector lengths respectively. For Physicsbench, we eliminate 40% more instructions compared to baseline vectorization and unvectorized code, at 256-bit, and 512-bit vector lengths with VLV-SWR. Baseline vectorization does not find any vectorization opportunity at higher vector lengths for Physicsbench.

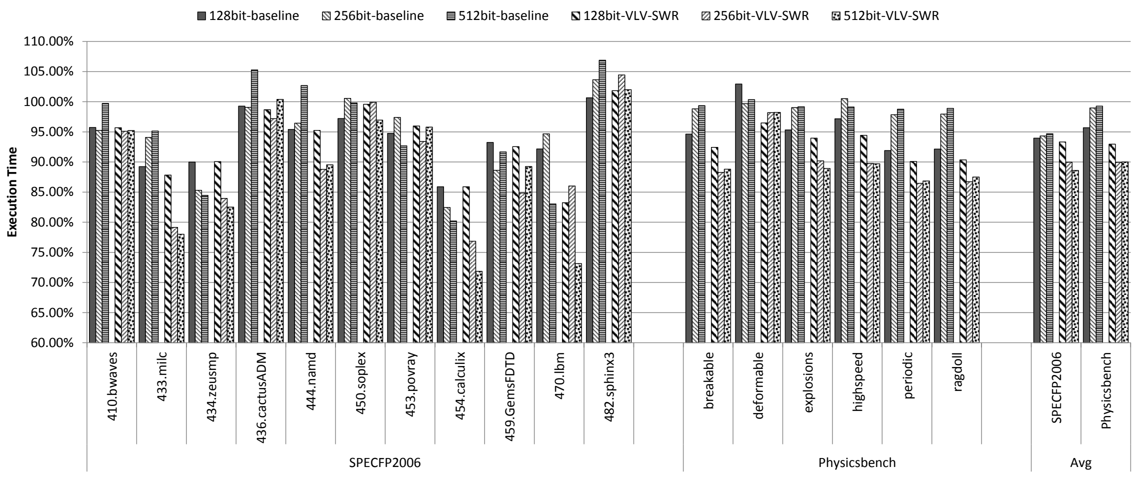


Figure 5.17: Execution time for baseline and VLV-SWR vectorizations normalized to unvectorized code execution time.

As Figure 5.16 shows, the percentage of reduced instructions is same for 256-bit and 512-bit vector lengths in case of Physicsbench and 410.bwaves. The lack of availability of independent instructions at 512-bit vector length forces VLV to vectorize the code the same way as for 256-bit vector length. However, important point to notice is that we still have more instruction reduction than 128-bit case, which was not possible without VLV.

5.5.4 Performance

This section presents the performance results. Since, we vectorize only the floating point instructions, the overall performance will depend upon the fraction of floating point instructions in the dynamic instruction stream.

Figure 5.17 shows the percentage of execution time, at three vector lengths, after vectorization without and with VLV-SWR. On average VLV-SWR provide 5% and 7% speed up over the baseline vectorization and 11% and 13% over the unvectorized code, for vector length of 256-bit and 512-bit respectively, for SPECFP2006. Similarly, for Physicsbench, we get a speed up of 10% for with VLV-SWR over unvectorized and baseline vectorization.

There are several interesting points to note in Figure 5.17. First, even though we have higher dynamic instruction elimination, e.g. 25% for SPECFP 512-bit vector length, the speed up we get is smaller, 13% for SPECFP 512-bit vector length. This is because only 40% of dynamic instructions are floating point in SPECFP, which reduces the overall performance. Second, dynamic instruction reduction is more for Physicsbench, 40% compared to 25% of SPECFP2006 for 512-bit vector length; SPECFP2006 shows more speed up, 13% compared to 10% of Physicsbench for 512-bit vector length. This is due to the fact that Physicsbench has higher percentage of integer instructions than SPECFP2006 as shows Figure 3.13 in Chapter 3.

5.6 Related Work

The proposal by M. Woh et al. [115] for supporting multiple SIMD widths is the closest to our proposal of Variable Length Vectorization. They proposed a configurable SIMD datapath that can be configured to process wide vectors or multiple narrow vectors. Unfortunately, details of their vectorization algorithm for vectorization for multiple vector lengths are not provided.

Masked operations have been used in the past for vectorization of code with control flow. However, we use them in the absence of control flow to increase dynamic instructions stream coverage. J. Smith et al. [98] proposed masked operations as a means of adding support for conditional operations in vector instruction set. J. Shin et al. [95] incorporated masked operations to vectorize loops with conditional flow in Superword Level Parallelism approach. Larrabee [93] also uses masked instructions to map scalar if-then-else control structure to the vector processing unit. All of these proposals execute both if and else clauses and select the correct results based on the values in the mask registers. Our proposal, on the other hand, uses masked operations to increase the dynamic instruction stream coverage when there not enough instruction to fill the wider vector paths.

Significant amount of work has been done on the optimal generation of permutation instructions due to their obvious effect on performance. However, previous work does not show effect of permutations at increasing vector lengths. A. Kudriavtsev et al. [60] show the relationship between operation grouping and permutation generation. They show the ordering of individual operations in SIMD instructions affect the number of permutation instructions required. G. Ren et al. [89] presented an algorithm that converts all the permutations to a generic form. Then, permutations are propagated across the statement and redundant permutations are eliminated. These solutions focus on reducing the number of permutations required, whereas our solution reduces the number of instructions for each permutation. L. Huang et al. [47] proposed a method to reduce the number of instruction for one permutation. Their system has a Permutation Vector Register File which provides implicit permutation capabilities. However, the permutation pattern is to be saved beforehand in a permutation register. Moreover, only the values from two consecutive registers can be permutated.

5.7 Conclusion

In this chapter, we showed that different applications have different natural vector lengths. Therefore, widening the SIMD accelerators do not improve the performance for all the applications. We discovered two main problems hurting the performance of natural low vector length applications for wider SIMD units: Reduced dynamic instruction stream coverage and large number of permutation instructions.

We propose Variable Length Vectorization to increase the instruction stream coverage. This technique creates packs with less number of instructions when there are not enough operations to fill the wider vector path. To reduce the number of permutation instructions, we propose Selective Writing. This enables us to write to only a particular element of vector registers and helps reducing the number of permutation instructions.

Chapter 6

Dynamic Selective Devectorization

Leakage energy is a growing concern in current and future microprocessors. Functional units of microprocessors are responsible for a major fraction of this energy. Therefore, reducing functional unit leakage has received much attention in the recent years. Power gating is one of the most widely used techniques to minimize the leakage energy. Power gating turns off the functional units during the idle periods to reduce the leakage. Therefore, the amount of leakage energy savings is directly proportional to the idle time duration.

This chapter focuses on increasing the idle interval for the higher SIMD lanes by selectively devectorizing the compiler vectorized code at runtime. The applications are profiled dynamically, in a HW/SW co-designed environment, to find the higher SIMD lanes usage pattern. If the higher lanes need to be turned-on for small time periods, the corresponding portion of the code is devectorized to keep the higher lanes off. The devectorized code is executed on the lowest SIMD lane.

6.1 Introduction

Modern microprocessors need to meet the high performance/throughput requirements of the increasingly complex applications. In addition, they have to provide such high performance under a very stringent power envelope. Moreover, the increase in leakage power at sub-nanometer technologies has put further constraints on the power budget. Therefore, it is of prime importance for computer architects to achieve a balance between the energy consumption and performance.

Single Instruction Multiple Data (SIMD) accelerators are incorporated in the processors, from different computing domains, to improve performance, especially for compute intensive data parallel applications [4][35][23][37][52][68][103]. However, due to their wider datapaths, they become main source of leakage energy for applications lacking data level parallelism. Therefore, it is crucial to control the leakage of these accelerators when they are not being utilized.

Many leakage control techniques have been studied [46][56][110][116], power gating being one of the most prominent ones. Power gating cuts the supply voltage to the idle functional units, resulting in leakage energy savings. The amount of leakage energy saved is directly proportional to the length of time interval for which the circuit remains idle. The longer the idle time interval, the more is the leakage energy saving. Therefore, it is desirable to have longer idle time intervals to save maximum leakage energy. However, power gating has an energy and performance overhead associated with it. Certain amount of energy is required to turn a functional unit off and then on again, resulting in energy overhead. Moreover, a certain number of cycles are required before the functional unit can be used after starting the turn on procedure, resulting in performance penalty. It is important to consider two special cases in power gating context:

1) Small idle intervals during periods of high utilization.
2) Small busy intervals during otherwise idle interval.

In the first case, a functional unit is awakened too early after turning it off. In this case, power gating energy overhead might not be offset by the leakage energy savings and power gating will result in net energy loss. Due to their obvious adverse effects on the net energy savings, several mechanisms have been studied to avoid such cases [72][120]. In the second case, the functional unit is awakened only for a small period of time before it is tuned off again. Power gating benefits can be increased if, somehow, the functional units can be kept off during these intervals. The gain here is twofold:

1) Since the functional unit is not turned on and then off again, there is no energy overhead.
2) Avoiding to turn on the functional unit also saves the performance overhead of power gating.

However, an alternate functional unit is required to avoid turning on the power gated (turned off) unit. The work presented in the chapter focuses on reducing these cases to improve the net energy savings.

SIMD accelerators have duplicated functional units/lanes to perform several independent operations in parallel. Lowest SIMD lane executes scalar/unvectorized code, whereas, the higher SIMD lanes comes into action when the application code is vectorized. In the cases when the higher SIMD lanes are power gated and need to be tuned on only for smaller periods of time, the corresponding portion of the code can be devectorized and executed on the lowest lane. Thus, the energy and performance overhead of power gating the higher SIMD lanes can be saved, resulting in increased net energy savings. However, the portions of the application to be devectorized should be chosen cautiously, as aggressive devectorization might also result in significant slowdown. Moreover, the slowdown might result in a net energy loss due to extra leakage energy incurred in the entire processor.

One of the ways of choosing devectorizable portions of the application is to profile the application offline and then guiding the compile time vectorizer to vectorize only the specific portions of the application. This method, however, has two major drawbacks. First, the execution profile of applications might change with the input. Thus, when an application is executed with an input other than the one with which it was profiled, the profile guided optimizations will not help. It might even result in slowdown if the frequently executed portions with the current input are not vectorized. Secondly, the existing code has to be recompiled to get benefits of the new techniques. HW/SW co-designed processors provide an excellent opportunity to profile and translate/optimize the applications at runtime. Since the profiling is done at runtime it is not coupled to any particular input.

We propose to extract maximum vectorization opportunities at compile time. Then, at runtime, profile the application dynamically to find out the candidates for devectorization. Therefore, dynamic selective devectorization discovers and devectorizes only the portions of code that help improving the power gating efficiency without having a significant effect on the performance.

For the rest of the chapter, Section 6.2 provides a background and related work on power gating. Section 6.3 provides the motivation for the work presented in this chapter. Section 6.4 describes the proposals of dynamic profiling and devectorization. Evaluation of the proposals using SPECFP2006 and Physicsbench applications is presented in Section 6.5. Section 6.6 concludes the chapter.

6.2 Background and Related Work

As leakage is becoming a growing concern in the current microprocessor designs, several leakage control mechanisms have been studied [46][56][110][116]. All these mechanisms try to reduce leakage when the circuit is in idle state. Power gating [46] consists of shutting down parts of the circuit by cutting their power supply by means of high threshold header or footer transistors, called sleep transistors. SSGC [56] is similar to power gating as this technique also cuts the power supply to the circuit. However, it is more effective than power gating in reducing leakage in data-retention circuits. Input vector activation [110] changes the input of the circuit to keep the maximum number of transistors in the off state. As the number of off transistors between power supply and ground increases the leakage reduces. Adoptive body biasing techniques [18][116] increase transistor threshold voltage by applying a reverse bias at transistor body. The increased threshold voltage reduces the sub-threshold and gate leakages.

Power gating is one of the most commonly used leakage control technique. There have been several proposals to increase the efficiency of power gating. Hu et al. [46] showed several key intervals in power gating, three of the most important being: idle detect interval, breakeven threshold, and wakeup delay. Idle detect interval is the amount of time needed to decide when to shut down a unit. At the end of idle detect interval a sleep signal is generated to shut down the functional unit. Breakeven threshold is the amount of time a unit must remain shut down to offset the power gating energy overhead. Waking up a unit before this threshold, results in net energy loss. Finally, wakeup delay is the amount of time needed before the unit can be used after turning it on. Therefore, a higher wakeup delay translates to a higher performance penalty.

Hu also proposed a branch prediction based and a counter based technique to generate sleep signal. In branch prediction based technique, the unit is shut down after a branch misprediction is detected whereas, the counter based technique generates the sleep signal after the unit has been idle for a fixed number of cycles. As noted before, if the power gated unit needs to be awakened before crossing the breakeven threshold, power gating suffers a net energy loss. Several techniques has been proposed to minimize this energy loss [15][72][120]. A. Youssef et al. [120] proposed to change idle detect interval dynamically. Their proposal increases the idle interval during the period of high utilization, when the functional units are being used frequently. Since the probability of a unit being awakened before crossing the breakeven threshold is high during these periods, increasing the idle detect interval reduces the number of power gating instances and hence the likelihood of energy loss. On the contrary, they reduce the idle detect interval during the phases of low activity to increase the number of powered off cycles and hence the energy savings. A. Lungu et al. [72] proposed to use success monitors to measure the success of power gating during a certain time interval. If power gating saves energy it is applied in the next interval as well, if possible. Otherwise, power gating would be deactivated in the next time interval even if a possibility existed. K. Agarwal et al. [15] proposed to have multiple sleep modes in power gating. Each mode has different wakeup delay and energy savings. By trading-off these two parameters during periods of different activity they achieve higher energy savings.

All of these techniques focus on improving the power gating efficiency by improving the decision of when to shut down a unit. On the other hand, our work focuses on how to keep a unit shut down for longer time intervals once it is already power gated. Even though we target SIMD accelerators to show the potential of the proposal, it can be applied to any functional units with multiple instances. To increase the length of the idle periods, the higher SIMD lanes usage is profiled dynamically. Then the portions of the code corresponding to the low utilization periods of higher lanes are located. This piece of code is then devectorized and executed on the lowest SIMD lane.

6.3 Motivation

Power gating has been used efficiently, in the recent past, to reduce the leakage energy when a functional unit is idle. Power gating cuts off the power supply to the functional unit to shut it down and hence reduce the leakage energy. However, every time a function unit is shut down and subsequently awakened, there is some energy overhead associated with it. The energy overhead comes due to the fact that the sleep signal needs to be generated and distributed to the appropriate functional units. Moreover, turning the sleep transistor on and off also requires energy. Therefore the net energy saving of power gating can be computed as:

$$\text{Net Energy Savings} = E_L * \sum_{k=0}^{n} off\_cycles[k] - (n * E_{overhead})$$

Where E_L is the leakage energy per cycle, $E_{overhead}$ is power gating energy overhead per power gating instance and *n* is the number of power gating instances. Thus, having large *off_cycles* with minimum number of power gating instances (n) results in maximum energy savings. Furthermore, a functional unit cannot be used immediately after putting the power supply back on, resulting in performance loss. Therefore, to get maximum leakage savings at minimum performance penalty, a functional unit needs to be kept shut down for longer time intervals, with minimum number of power gating instances.

Functional unit usage profile of an application changes during its execution. During the low utilization period the function unit is used scarcely. Therefore, power gating targets these periods for leakage savings. However, every time the functional unit is needed, it needs to be awakened from the power gated state and needs to be shut down afterwards. The wakeup and shutting down energy overhead reduces overall leakage energy savings. If the functional unit is kept turned off and the corresponding code is executed on some other functional unit (which are already on); the effectiveness of power gating in saving leakage energy can be increased. Specifically for SIMD accelerators, higher SIMD lanes can be switched off during sporadic usage period and the corresponding code can be executed on lowest lane after devectorization.

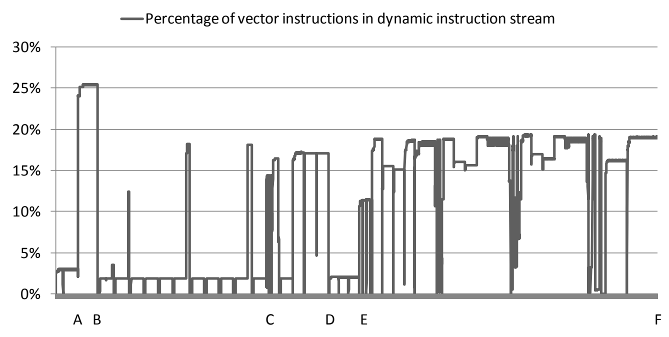


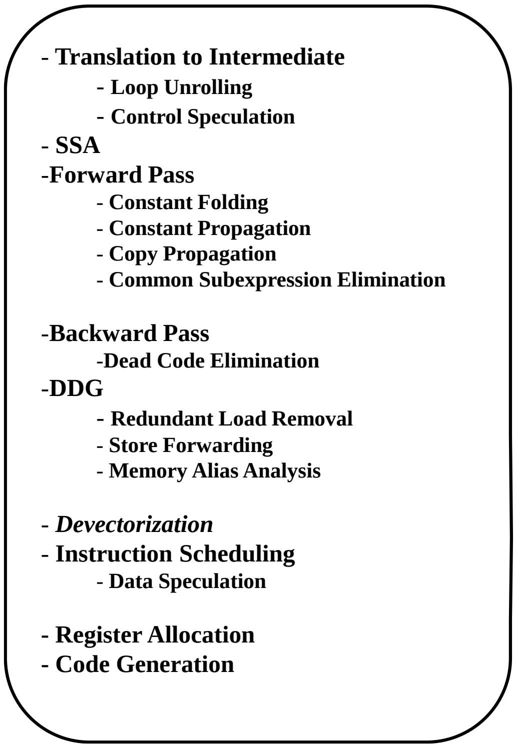
Figure 6.1: Percentage of vector instruction (excluding memory instructions) in the dynamic instruction stream over the time (4 billion instructions) for 434.zeusmp.

We profiled SPECFP2006 to discover the higher SIMD lanes usage pattern. Figure 6.1 shows the percentage of vector instructions (higher lanes usage profile) in the dynamic instruction stream over the execution time for 434.zeusmp. The higher lanes usage profile shown in the figure is for 4 billion instruction executed starting from the most frequently executed function/routine. Moreover, the shown vector instruction profile does not include memory instructions since they do not use SIMD functional units. As can be seen in the figure, higher lane usage profile changes during the execution. During the time intervals A-B, C-D, and E-F around 20% of the dynamic instructions are vector instructions and utilize higher SIMD lanes. Therefore, higher SIMD lanes need be activated during these intervals. On the other hand, during the time intervals 0-A, B-C and D-E only less than 3% of the dynamic instructions are vector instructions. During these intervals power gating will activate SIMD lanes for short durations of time to execute these vector instructions

We propose to devectorize the portion of code corresponding to the time intervals 0-A, B-C, and D-E, if it does not affect the percentage of vectorized code in the other time intervals. Devectorizing this piece of code results in lesser number (in some cases none) of vectorized instructions during these time intervals. Therefore the number of power gating instances also reduces during these intervals. As a result, the power gating energy overhead diminishes and the net leakage savings increase. However, the dynamic energy consumption of the lowest lane increases, as it has to execute more instructions now. Nevertheless, as will be shown in the performance evaluation section, this increase is relatively small compared to the reduction in the leakage energy.

6.4 Profiling and Devectorization

This section provides the details of the dynamic profiling and devectorization schemes. Profiling is necessary to discover the code segments that can be devectorized to



- **Translation to Intermediate**
  - **Loop Unrolling**
  - **Control Speculation**
- **SSA**
- **Forward Pass**
  - **Constant Folding**
  - **Constant Propagation**
  - **Copy Propagation**
  - **Common Subexpression Elimination**
- **Backward Pass**
  - **Dead Code Elimination**
- **DDG**
  - **Redundant Load Removal**
  - **Store Forwarding**
  - **Memory Alias Analysis**
- ***Devectorization***
- **Instruction Scheduling**
  - **Data Speculation**
- **Register Allocation**
- **Code Generation**

Figure 6.2: Optimization sequence in superblocks for devectorization support.

keep the higher vector lanes power gated without affecting the performance. These code segments must not be performance critical, as devectorizing performance critical code will result in excessive slowdown. Moreover, due to the slowdown caused by devectorization overall energy consumption will increase. It is important to note that the performance critical code segments of an application might change with the input. Therefore, profiling the applications offline with a particular set of inputs might not help in deciding which code segments to devectorize. For that reason, we choose to profile the applications dynamically at runtime. Dynamic profiling discovers non-performance critical devectorization candidates and pass this information to the runtime devectorizer. The selected code segments are then devectorized, resulting in effective power gating of SIMD units.

As stated in Chapter 3, TOL (the software layer of our HW/SW co-designed processor) operates in three translation modes for generating host code from guest x86 code: Interpretation Mode (IM), Basic Block Translation Mode (BBM), and Superblock Translation Mode (SBM). We collect the profiling information for the basic blocks in BBM. This information is then used in SBM during superblock optimization phase to decide whether or not to devectorize the given superblock. Figure 6.2 shows the modified optimization flow in superblocks.

6.4.1 Profiling

In BBM, the application is profiled to get following information

1) Execution and Branch profiling information:

Software counters are used to count the number of times a basic block has been executed in BBM. Besides, software counters are also employed to get the biased direction of branches. This information is used to create bigger optimization regions (superblocks) in SBM. Furthermore, we also profile the higher SIMD lanes usage pattern that helps us in deciding which superblocks to devectorize.

2) Higher SIMD lanes usage pattern:

To decide whether to devectorize a superblock or not, we anticipate whether the higher SIMD lanes would be power gated or not when the execution reaches the particular superblock. In the case when we expect them to be power gated and if the current superblock also has few vector instruction, it is desirable to devectorize the superblock. To anticipate the status (power gated or not) of higher SIMD lanes we monitor their usage by means of an N-bit shift register. Before executing an instruction, the content of this register are shifted by one and the new position is set to 1 if the current instruction is a vector instruction, otherwise it is reset to zero. Therefore, the number of ones in the shift register gives the number of vector instructions executed in the last N instructions.

Each basic block in BBM has a software “devec” counter associated with it. Every time a basic block, having at least one vector instruction, is executed in BBM, the contents of the shift register are read. If the number read is less than a threshold (DV_{th}), it would be desirable to devectorize the basic block, if it is included in a superblock. The devectorization is desirable in this case, since having less number of vector instructions indicate low usage of higher SIMD lanes. Therefore, devectorizing this code will help improving power gating efficiency without a significant impact on the overall performance. To increase the devectorization likelihood of this basic block the devec counter is incremented. However, if during the next execution of the same basic block the number of ones in the shift register is more that DV_{th}, the devec counter is decremented. It indicates that devectorization is not favored due to more utilization of higher SIMD lanes. Therefore, the final decision of whether to devectorize the basic block or not depends on the shift register values just before all the executions of the basic block in BBM. This helps in devectorizing only the basic blocks that are executing during the low usage phase of higher SIMD lanes like B-C in Figure 6.1.

While creating a superblock, devec counters of all the basic blocks included in the superblock are examined. If all the counters are greater than a predetermined threshold, the superblock is devectorized. Otherwise, the superblock is kept in the vectorized form. This selective devectorization of superblocks improves leakage energy savings through power gating while maintaining the performance.

6.4.2 Devectorization

Once a superblock has been identified for devectorization through profiling, it goes through a devectorization phase. The devectorization pass simply replaces vector instructions by their corresponding scalar instructions and generates permutation instructions if required. Moreover, vector memory instructions are not devectorized since they do not use SIMD functional units.

Algorithm 6.1 presents the devectorization algorithm. "*devect*" is the top level routine that receives the superblock "SB" to be devectorized. The routine goes over all the instructions in the superblock in the program order. All the vector instructions (excluding memory access instructions) are candidates for devectorization. The first step in devectorization is to find devectorization length (*get_devec_len*). It is the number of scalar instructions to be generated corresponding to the vector instruction. Then the scalar opcode for the scalar instructions to be generated is obtained (*get_scalar_opcode*). Next, the "*get_scalar_in_reg*" routine checks if the input vector registers of the current instruction have already been mapped to scalar registers or not. If the producers of the current instruction have already been devectorized, the corresponding input registers are already mapped to the output scalar registers of the scalar producers. However, if the producers cannot be devectorized (producers being vector memory loads or live-in of superblock), an Unpack instruction is generated (*generate_Unpack_insn*). This Unpack instruction distributes the contents of the input vector register to set of scalar registers depending on the devect length. Once all the input vector registers have been mapped to scalar registers, new output scalar registers are allocated (*allocate_reg*) for new scalar instructions to be generated. In the next step, the scalar instructions are generated (*generate_insn*) using scalar input and output registers collected during the earlier steps. The vector output register of the current instruction is mapped to the new scalar output registers allocated (*add_to_mapped_reg*). Finally, if the output register is an architecture register or the consumers of the current instruction cannot be devectorized (vector memory stores), a Pack instruction is generated (*generate_Pack_insn*). The Pack instruction collects the values from the scalar output registers and packs them in a new vector register so that it can be used by the vectorized consumers (*generate_Pack_insn*).

Algorithm 6.1a. Top Level Dynamic Devectorization Routine

```
devect(SB):
    for each instruction s in SB:
        if s is devectorizable:
            devec_len ← get_devec_len(s)
            scalar_op ← get_scalar_opcode(s)
            scalar_in_regs ← get_scalar_in_reg (s)

            scalar_out_reg ← ø
            for i ← 0 to devec_len do:
                    scalar_out_reg ← scalar_out_reg ∪ allocate_reg()

            for i ← 0 to devec_len do:
                    generate_insn(scalar_op, scalar_in_reg, scalar_out_reg)

            add_to_mapped_reg(org_out_reg)

            if org_out_reg is architectural_reg or vectorized_consumer:
                generate_Pack_insn(scalar_out_reg)
```

Algorithm 6.1b. Vector to Scalar Register Mapping

```
get_scalar_in_reg (s)
    scalar_in_regs ← ø

    for each input_register ireg of s:
        if ireg in mapped_regs:
            scalar_in_regs ← scalar_in_regs ∪ get_mapped_reg(ireg)
        else
            generate_Unpack_insn(ireg)
            scalar_in_regs ← scalar_in_regs ∪ get_mapped_reg(ireg)

    return scalar_in_regs
```

Algorithm 6.1: Dynamic Selective Devectorization algorithm. *devect* routine devectorizes the code in a top-down manner, starting with the first instruction in the superblock. *get_scalar_in_reg* checks if a vector register is already mapped to a set of scalar register. If it is not, a new UNPACK insturction is generated to map it to scalar registers.

As the devectorization proceeds, the producer-consumer relations keep changing. Thus, it is important to update the predecessor/successors chains. However, it is not shown in the algorithm for the sake of simplicity.

6.4.3 Reducing Devectorization Slowdown

Dynamic selective devectorization serializes the parallel portions of code to save energy at small performance cost. To reduce the effect of this serialization on the performance, we do partial devectorization whenever possible. To better understand partial devectorization, consider a SIMD accelerator with two 64-bit wide lanes. Each lane can execute either one 64-bit double-precision floating-point operation or two 32-bit single-precision floating-point operations. Devectorized code is executed on the lower lane, so that the higher lane could be switched off.

In general, a single-precision floating-point vector instruction would be devectorized into four single-precision scalar instructions. However, partial devectorization generates only two single-precision "half-vector" instructions. A "half-vector" instruction combines two scalar instructions that can be executed in parallel. The rationale behind partial devectorization is to utilize the whole 64-bit wide vector lane. Since one vector lane can execute two single-precision operations, it is better to partially devectorized the code instead of full devectorization. As a result, the effect of devectorization on performance is reduced while still saving energy by power gating the higher lane. We propose to have "half-vector" instructions in the host processor ISA. However, these instructions are transparent to the compiler/user and are generated dynamically by the runtime devectorizer. The co-designed nature of the host processor allows including new instructions without any change in compiler/recompiling.

6.5 Performance Evaluation

To evaluate the proposals, we implemented the proposed profiling and devectorization algorithm in the software layer (TOL) of DARCO. Furthermore, for energy consumption analysis McPAT [70] is integrated with DARCO. The key McPAT parameters are shown in Table 6.1. Moreover, we consider only the floating point instructions for devectorization because they are the main target of SIMD accelerators. In our experiments, we assume that the host architecture consists of a 128-bit wide SIMD accelerator. Moreover, we consider that the SIMD accelerator is composed of two 64-bit wide lanes.

6.5.1 Baseline

From power gating point of view, SIMD accelerator can be viewed as a single unit or two separate lanes. In other words, both the lanes of the SIMD accelerator can be powered together or separately. If both the lanes are power gated together, we call it combined power gating (CPG). CPG, however, is not efficient, since higher lane is, generally, used lesser than the lower lane. Therefore, power gating the higher lane, even though the lower lane is functional, would result in more power savings. We call this configuration Split Power Gating (SPG). Our proposal of Dynamic Selective Devectorization also assumes that the SIMD lanes can be power gated individually. We compare our results with both configurations (CPG and SPG). Furthermore, the results presented are for the modeled host processors and include profiling and translation overheads. Only floating point benchmarks in SPEC2006 are considered for evaluation since the floating point code is the main target of our proposals.

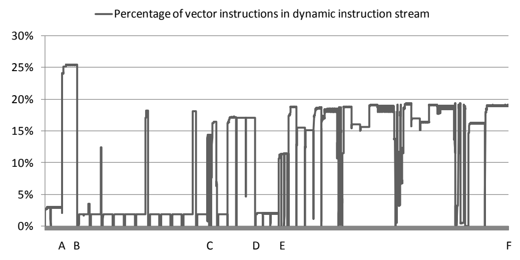
Parameter	**Value**
Technology	65nm
Clock Rate	1.5 GHz
Temperature	350 K
Device Type	High Performance

Table 6.1: McPAT Parameters.

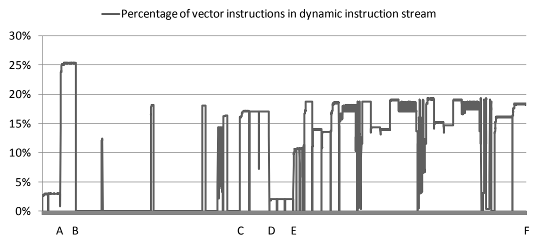
6.5.2 Models and Parameters

To measure the success of the proposals, we refer to the power gating energy model proposed by Hu et al. [46]. However, we changed some of the model input values. Their breakeven threshold value is between 9 and 24 cycles. However, as A. Youssef [120] explained, the breakeven threshold value in the real implementations can be more than 100 cycles. We use the breakeven threshold of 150 cycles. The wakeup latency of the functional units is considered to be 10 cycles. Moreover, later we show a sensitivity study for breakeven threshold and wakeup delay variations.

A. Lungu et al. [72] proposed a success monitor based improvement to the time-based power gating mechanism of [46]. They use success counters to monitor whether power gating has been successful (saved energy) or harmful (wasted energy) during a monitoring interval. Power gating in the next monitoring interval is disabled if it has been harmful in the current interval, otherwise it is enabled. This power gating scheme with success monitors serves as the baseline for our proposals. A. Lungu et al. [72] have a fixed idle detect interval of 5 cycles (or 15 cycles as an alternate) in their proposal. However, this interval is varied dynamically in our baseline, depending on the utilization of the functional units (SIMD lanes), as proposed by A. Youssef [120]. We assume a minimum idle detect interval of 5 cycles however, there is no maximum limit. Moreover, the idle detect interval is varied in step of 5 cycles. Furthermore, we consider power gating of only the SIMD accelerator in our experiments.



a) Before DSD.



b) After DSD.

Figure 6.3: Percentage of vector instruction in the dynamic instruction stream before and after dynamic selective devectorization for 434.zeusmp.

6.5.3 Higher SIMD Lane Usage Profile

The dynamic selective devectorization (DSD) technique tries to minimize the usage of the higher SIMD lane during the low utilization period. As shown in Figure 6.1, in Section 6.3 (reproduced here as Figure 6.3 a), 434.zesump has several time intervals during which the higher SIMD lane usage could be minimized. Minimizing the higher lane usages during these intervals minimizes the number of power gating instances and hence the energy overhead of power gating.

Figure 6.3b shows the vector instruction profile for the same benchmark after dynamic selective devectorization. As the figure shows, the dynamic selective devectorization has been able to reduce the higher SIMD lane usage significantly during the time interval B-C. Therefore, the energy savings by power gating during this interval will be improved. However, the vector code corresponding to the low usage periods 0-A and D-E is not devectorized. This piece of code is executed during the high usage periods also and its devectorization would result in significant performance loss. Therefore, this code is always executed in the vectorized version. Moreover, it is also important to note that the number of vector instructions during the high usage periods A-B, C-D, and E-F is the same as before devectorization. Therefore, the effect of devectorization on the performance is going to be negligible.

6.5.4 SIMD Accelerator Energy Savings

The proposed mechanism reduces the number of higher SIMD lane power gating instances to reduce power gating energy overhead and in turn, the overall leakage of the SIMD accelerator. However, dynamic selective devectorization has an energy and performance overhead associated with it. The energy overhead of DSD includes the following components:

1) Lower SIMD Lane Dynamic energy: The dynamic energy consumption of the lower SIMD lane increases, since it has to execute more instructions.

2) Rest of the core Energy: The rest of the core includes all the components of the core except for the SIMD accelerator. The dynamic and leakage energy of the rest of the core may increase due to:

a. Dynamic energy consumption increases due to profiling and devectorization of selected superblocks.
b. Leakage energy of the rest of the core might increase due to the possible slowdown because of devectorization.

Figure 6.4 and 6.5 show the SIMD accelerator energy savings for Combined power gating (CPG), Split power gating (SPG) and DSD normalized to no power gating, without and with DSD overheads respectively. There as several important points to note in these two figures. First of all, the energy overhead of DSD is minimal as most of the benchmarks show similar energy savings with and without considering DSD energy overhead. The only exception is 410.bwaves and the reason behind it is explained in Section 6.5.6 while discussing the performance results. Since the energy savings are similar with and without considering the energy overhead of DSD, the rest of this section focuses on results with overhead (Figure 6.5). As this figure shows, DSD outperforms both CPG and SPG significantly. The overall energy savings of the proposed technique are 49% and 35% greater than CPG and 15% and 12% greater than SPG for SPECFP2006 and Physicsbench respectively. In absolute energy savings terms, DSD saves 63% and 72% overall energy, SPG saves 54% and 64% overall energy whereas, CPG saves 42% and 53% overall energy for SPECFP2006 and Physicsbench respectively. CPG performs worse than SPG because it treats the whole SIMD accelerator as a single unit. Therefore, either both lanes are powered or neither of them. On the other hand, SPG can turn higher lane off even if the lower lane is in use. Therefore, SPG saves more energy than CPG. DSD goes one step ahead and keeps the higher lane powered off (because of devectorized code) for longer periods and outperforms SPG as well.

The benchmarks in Figure 6.5 can be divided into three categories depending on their energy saving pattern:

1) Moderately Vectorizable Benchmarks

The benchmarks in this category include 410.bwaves, 434.zeusmp, 435.gromacs, 454.calulix, 482.sphinx3 and most of the Physicsbench benchmarks. These are the

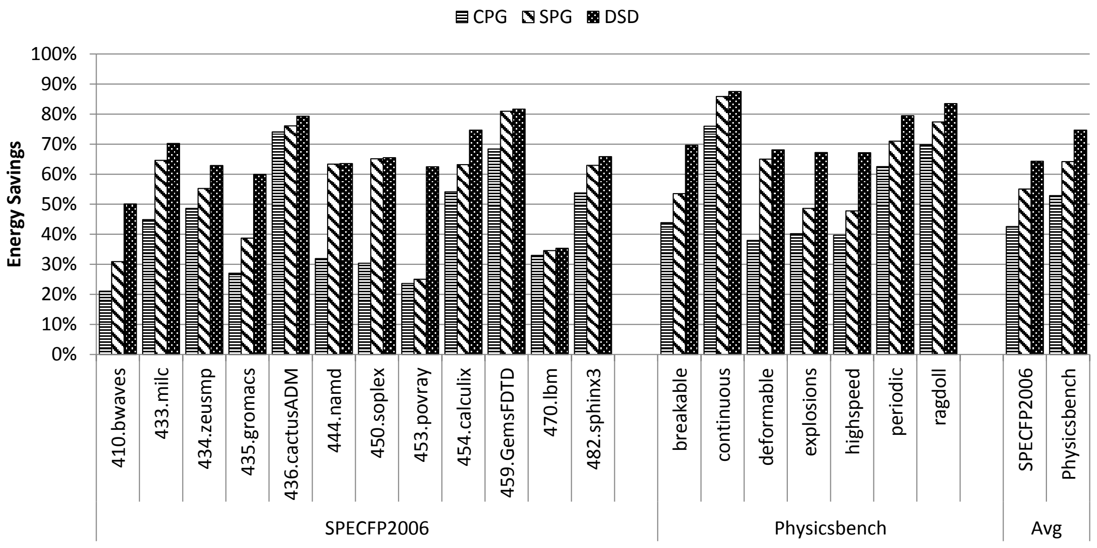


Figure 6.4: SIMD accelerator overall (dynamic + leakage) energy savings for CPG, SPG and DSD **without including** DSD energy overhead.

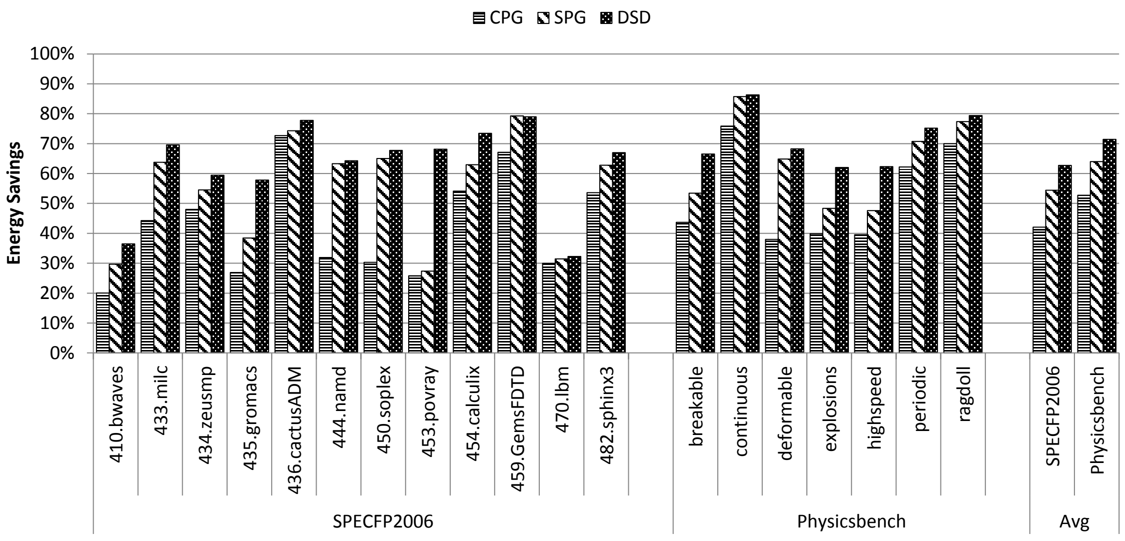


Figure 6.5: SIMD accelerator overall (dynamic + leakage) energy savings for CPG, SPG and DSD **including** DSD energy overhead.

benchmarks for which compilers are able to extract enough vector parallelism, however they are not completely vectorized. Therefore, during the periods of high lower lane usage and idle higher lane, SPG achieves energy savings over CPG by power gating only the higher lane. Moreover, these benchmarks have periods of low higher lane activity, as shown in Figure 6.1 for 434.zeusmp. The proposed mechanism devectorizes the code corresponding to these intervals and achieve even more energy savings.

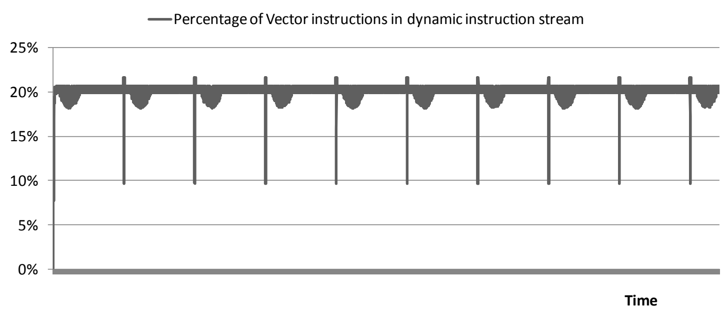


Figure 6.6: Percentage of vector instruction (excluding memory instructions) in the dynamic instruction stream for 470.lbm.

2) Highly Vectorizable Benchmarks

The benchmarks in the category are 436.cactusADM and 470.lbm. These benchmarks are completely vectorizable. In other words, the vectorized code uses either both the vector lanes or none of them. As a result SPG does not provide any additional benefits over CPG. Moreover, the higher lane utilization in these benchmarks is uniform over the execution time as shown in Figure 6.6 for 470.lbm. Any attempt of devectorization would result in significant performance loss. Therefore, these benchmarks are executed in the vectorized form and no additional leakage energy savings are achieved by DSD either. Furthermore, the energy savings for 436.cactus are much more compared to 470.lbm for all the three techniques. The energy savings depend on how long the SIMD accelerator is used during the execution time of the application. Even though both the benchmarks use both the SIMD lanes together, the overall usage of SIMD accelerator is less in 436.cactus, hence power gating provides more energy savings in this benchmark.

3) Unvectorizable Benchmarks

The benchmarks in this category include 444.namd, 450.soplex, etc. Since compilers do not find enough vectorization opportunities in these benchmarks, the higher SIMD lane is idle for most of the time. As a result, SPG is able to attain significant energy savings over CPG by power gating the higher SIMD lane alone. However, the proposed mechanism does not have enough opportunities to devectorize because compilers do not vectorize the code. Therefore, DSD does not provide much energy savings over SPG.

It is important to note that the most of the benchmarks fall in the first category “Moderately Vectorizable Benchmarks” which is targeted by DSD to achieve additional power savings over power gating. Another interesting point to note in Figure 6.5 is that in the cases where DSD is not able to reduce leakage, e.g. 470.lbm, the energy overhead of

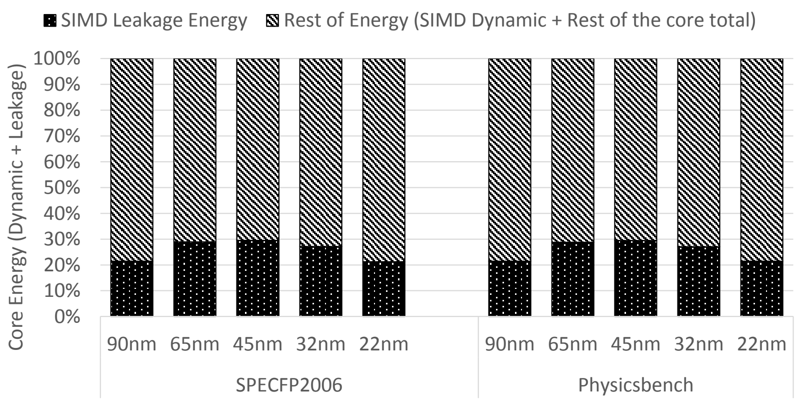


Figure 6.7: Core overall energy distribution at different technologies.

DSD is negligible. Hence, DSD has insignificant energy penalty when it fails to provide leakage benefits.

6.5.5 Overall Energy Savings

This section first presents the ratio of SIMD accelerator leakage energy to the rest of energy (SIMD accelerator dynamic energy + rest of the core overall energy) and then presents core level overall energy savings by CPG, SPG and DSD. Figure 6.7 shows the energy distribution for five different technologies: 90nm, 65nm, 45nm, 32nm and 22nm. As the figure shows SIMD accelerator leakage energy accounts for 20% to 30% of overall core energy at various technologies. It is also interesting to note that the SIMD leakage energy increases as we move from 90nm to 45nm. However, it reduces as the technology is further scaled down to 22nm. The leakage reduction comes from the enhancement in fabrication process below 45nm. Nonetheless, SIMD leakage energy still forms a significant portion of the overall core energy.

C. Bira [25] showed that according to Zedboard documentation a dual-core ARM CPU consumes a maximum of 1.25 Watts. They also reported that according to Xilinx power estimation tools the SIMD accelerator consumes 600 mW. This translates to SIMD accelerator being responsible for consuming approximately half of the CPU power. Assuming leakage being responsible for 40-50% of total power, SIMD accelerator leakage is responsible for 20%-25% of total CPU power. This estimation is in coherence with the results of Figure 6.7.

Figure 6.8 shows the overall energy savings of the whole core by CPG, SPG and DSD. Since, we consider power gating only the SIMD accelerator and no other functional unit, absolute overall energy savings are not as high as for the SIMD accelerator alone.

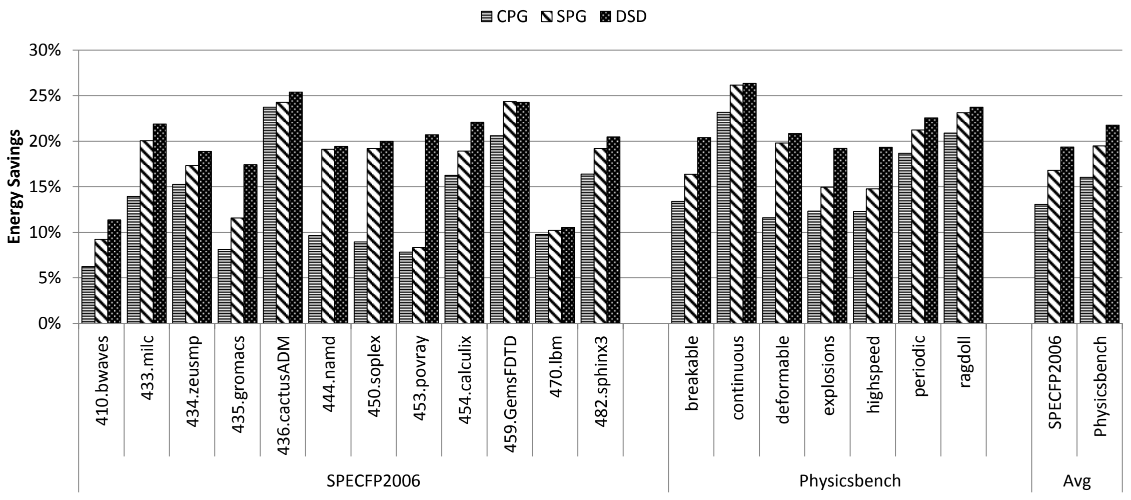


Figure 6.8: Core overall (dynamic + leakage) energy savings for CPG, SPG and DSD.

However, DSD still outperforms both SPG and CPG. DSD energy savings are 48% and 35% greater than CPG and 15% and 12% greater than SPG for SPECFP2006 and Physicsbench respectively. In absolute energy savings terms, DSD saves approximately 19% and 22% overall energy, SPG saves 16% and 19% overall energy while CPG saves 13% and 16% overall energy for SPECFP2006 and Physicsbench. As the results show, DSD is able to save comparatively more overall core energy than CPG and SPG even when SIMD accelerator is the only power gated functional unit.

6.5.6 Performance

As mentioned earlier, power gating has both energy and performance overhead associated with it. The performance overhead arises because the functional unit cannot be used immediately after sending the wakeup signal. Moreover, the performance penalty has to be paid every time the functional unit is awakened from the power gated state.

Reducing the number of power gating instances, using DSD, reduces both the energy and performance overhead of power gating. However, DSD also has its own performance overhead. This overhead arises because the lower SIMD lane has to execute more scalar instructions. Furthermore, profiling and devectorization of the selected superblocks also diminish performance.

In summary, DSD, on one hand, reduces power gating performance overhead. However, on the other hand, it adds its own overhead. Therefore the overall performance depends on the following factors:

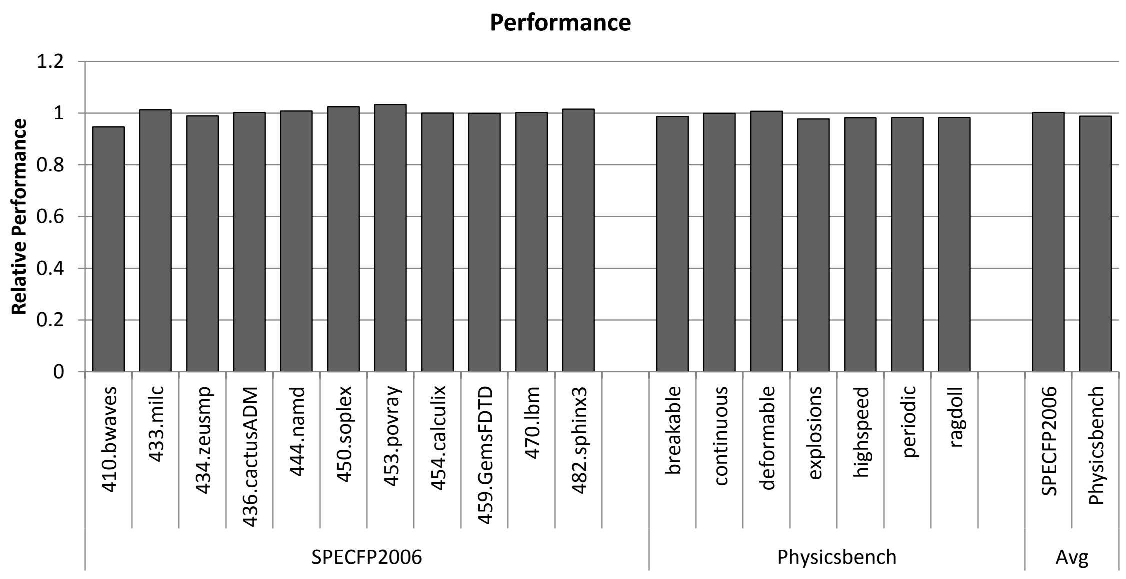
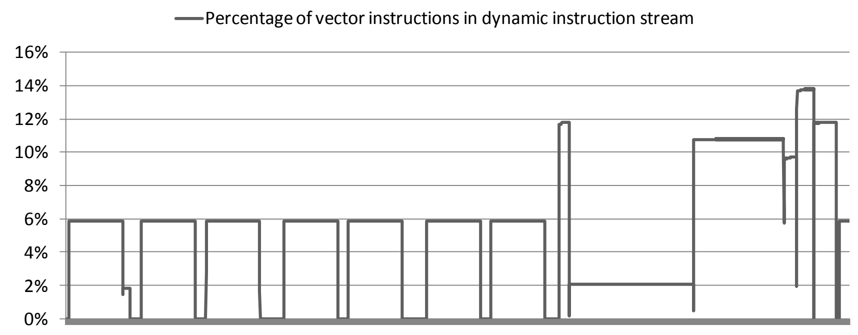


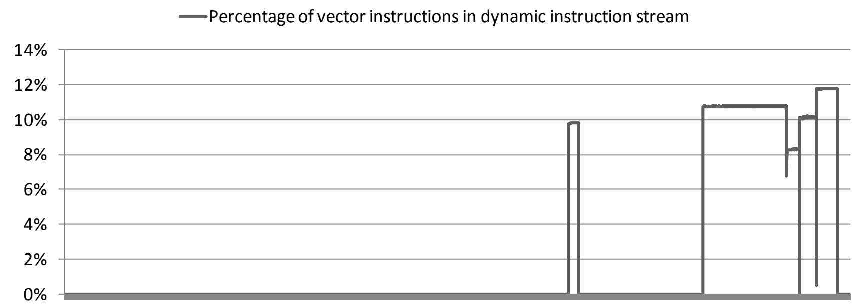
Figure 6.9: Overall Performance after DSD normalized to SPG (Higher is better).

1) Speedup, due to lesser number of power gating instances.
2) Slowdown, due to more number of scalar instructions.
3) Slowdown, due to profiling and devectorization overhead.

Figure 6.9 shows the performance results after considering all these factors. The results are normalized to SPG performance. As the figure shows, on average DSD experiences a slowdown of less than 1% for Physicsbench. Moreover, for SPECFP2006 the performance is very similar to SPG performance. It is also interesting to note that there are benchmarks like 433.milc, 450.soplex, 453.povray, 482.shpinx3, etc. that experience a small speedup. The speedup comes due to lesser power gating instances and hence lesser performance overhead of power gating. The performance increase also translates to reduction in the leakage energy in the core because it is now ON for less time. On the other hand, 410.bwaves suffers slowdown of 6% due to excessive devectorization as shown in Figure 6.10a and 6.10b. The two figures show vector instruction profiles before and after devectorization respectively. The excessive devectorization not only affects the performance but the energy savings also. Due to the slowdown, the leakage energy in the rest of the core increases, and hence net energy savings reduce. The energy savings for 410.bwaves are approximately 50% without the energy overheads of DSD as shown in Figure 6.4, however after considering the energy overheads they fall down to 38% as shown in Figure 6.5. Therefore, DSD provides a trade-off between performance and energy.



a) Before DSD.



b) After DSD.

Figure 6.10: Percentage of vector instructions (excluding memory instructions) in the dynamic instruction stream for 410.bwaves before and after DSD.

6.5.7 Sensitivity Analysis

As mentioned in Section 6.5.2, we assumed a breakeven threshold of 150 cycles and wakeup delay of 10 cycles in our experiments. These two parameters are technology dependent and to discover the effect of variations in their values we do a sensitivity study. For this study, first we vary the breakeven threshold from 20 cycles to 300 cycles while keeping the wakeup delay at 10 cycles. Next we vary the wakeup delay from 5 to 35 cycles while keeping the breakeven threshold at 150 cycles.

1) Breakeven threshold sensitivity study

Figure 6.11 shows the results for the effect of breakeven threshold variations on the overall energy savings of DSD over SPG. As the figure shows, the overall energy savings of DSD increases with breakeven threshold for SPECFP2006. However, for Physicsbench the increase is less significant. As mentioned in Section 6.3, one of the components of DSD

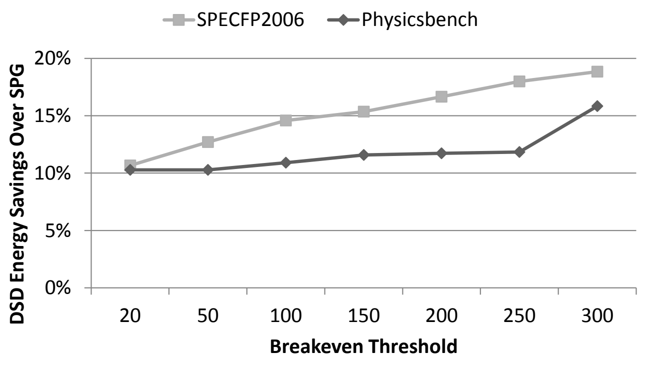


Figure 6.11: Effect of breakeven threshold variation on DSD overall (dynamic + leakage) energy savings over SPG with a fixed wakeup latency of 10 cycles.

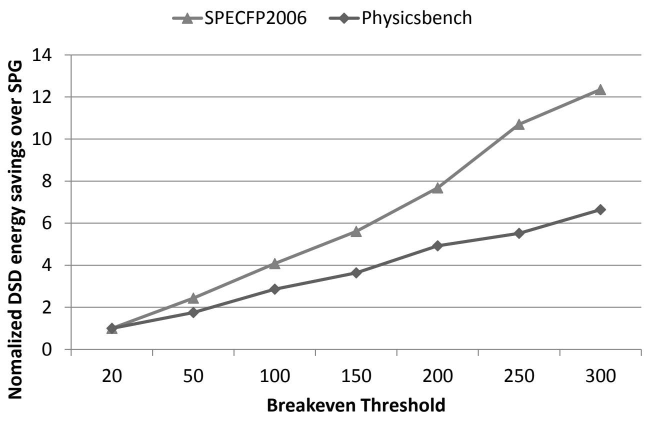


Figure 6.12: Effect of breakeven threshold variation on DSD overall (dynamic + leakage) energy savings over SPG normalized to breakeven threshold of 20 cycles, with a fixed wakeup latency of 10 cycles (no success monitors, no dynamic idle detect interval).

energy savings is directly proportional to the breakeven threshold. Therefore, one would expect more energy savings as the breakeven threshold is increased. The reason for minimal improvement in the overall energy savings is the use of success monitors and dynamic idle detect interval. If we disable these two improvements, we get more energy saving as breakeven threshold in increased, as shown in Figure 6.12. The figure shows energy benefits of DSD over SPG normalized to the savings corresponding to breakeven threshold of 20 cycles. As the figures shows the energy savings of DSD increases over SPG as the breakeven threshold increases from 20 cycles to higher values.

2) Wakeup Delay sensitivity study

Figure 6.13 shows the effect of wakeup delay variation on the overall energy savings of DSD over SPG. As with breakeven threshold variation results, these results are

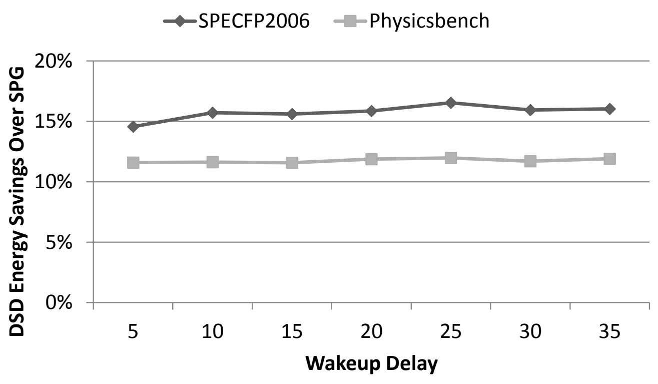


Figure 6.13: Effect of wakeup delay variation on DSD overall (dynamic + leakage) energy savings over SPG with a fixed breakeven threshold of 150 cycles.

consistent over the range of wakeup delay values. Furthermore, these results are with success monitors and dynamic idle detect interval enabled. Disabling these two features will show improvement in DSD overall energy savings as wakeup delay increases.

Discussion

As the leakage energy is becoming a growing concern in the present day microprocessors, several leakage control mechanisms have been studied. All these techniques thrive by reducing the leakage energy during the period when the functional unit is not being used. Therefore, the leakage energy savings of these techniques depend on the idle interval duration of the functional units. This chapter presented a technique called Dynamic Selective Devectorization to increase the idle durations of higher SIMD lanes. The increase in idle interval translates to an increment in the leakage energy savings.

In our experiments, we consider power gating as the leakage control mechanism implemented in the hardware. However, our proposal of dynamic selective devectorization does not restrict the choice of leakage control mechanism to power gating. DSD will work with any other leakage control mechanism equally well. The basic idea of DSD is to increase the idle intervals of the functional units independent of the leakage control mechanism.

We presented a mechanism to increase the idle period of higher SIMD lanes to save more leakage energy. DSD devectorizes certain portions of the code to reduce the higher SIMD lanes utilization during low usage periods. Even though the work in this chapter focuses on higher SIMD lanes, the basic concept can be extended to any functional unit. The only requirement is to have more than one instance of the functional unit. For example, if we have two integer units, the idle interval of the second one could be increased by executing more code on the first one. This, however, is helpful only during the low utilization period of the second unit, to reduce the performance penalty of serialization. In case of SIMD accelerator, a dynamic profiler guides the devectorizer to decide which segments of code to serialize. However, in the case of integer units, the dynamic profiler needs to guide the instruction scheduler to make serialization decisions.

6.6 Conclusion

This chapter proposed to increase the leakage energy savings by increasing the idle interval of the higher SIMD lanes. To increase the idle interval, we proposed a dynamic profiling based dynamic selective devectorization scheme. The dynamic profiler monitors higher SIMD lanes usage and discover the code corresponding to the low utilization period. A dynamic devectorizer then selectively devectorizes the code based upon the inputs from the profiler. The dynamic selective devectorization increases the idle interval during the low utilization period of the higher lanes. Increase in the idle period helps the leakage control mechanism to save more energy. The proposed mechanism can work with any leakage control mechanism like power gating, SSGC, etc. Moreover the idea of increasing idle period is general enough to be extended to other functional units as well.

Our experimental results show average SIMD accelerator energy savings of 15% and 12% relative to power gating, for SPECFP2006 and Physicsbench respectively. Moreover the slowdown caused due to devectorization is less than 1%.

Chapter 7

Conclusions

This chapter concludes the thesis by first summarizing the challenges in SIMD execution and our proposed solutions. Then it presents future directions that can be followed to make SIMD execution even more efficient.

7.1 Conclusions

SIMD accelerators are one of most proficient way of improving compute power in an energy efficient manner. Even though they are simple from hardware design perspective, code generation of them has always been challenging. In this thesis, we made several proposals for optimizing the code generation for SIMD accelerators. Furthermore, we also proposed a way of reducing leakage energy when SIMD accelerators are not being utilized to their highest potential.

Speculative Dynamic Vectorization. Chapter 4 showed that compile time vectorization loses significant vectorization opportunities due to conservative memory disambiguation analysis. We proposed to have dynamic vectorizer to assist the static compile time vectorization. In the proposed mechanism, first compiler vectorizes the code after applying complex loop transformations which are too costly at runtime. Later, during the program execution, a dynamic vectorizer catches the vectorization opportunities missed by compiler. The dynamic vectorizer speculatively reorders and vectorizes ambiguous memory references. The hardware checks for any memory order violation possibly caused due to speculative vectorization and takes corrective action.

The experimental results show that the combination of static and dynamic vectorization discovers twice the number of vectorization opportunities than the static vectorization alone. Moreover, the dynamic vectorization alone is able to outperform the static vectorization. Furthermore, the dynamic vectorization vectorizes array and pointer based applications equally well, whereas static vectorization loses significant vectorization opportunities for pointer based applications.

Vectorizing for Wider Vector Units. Even though SIMD accelerators are very amenable to scaling due to their duplicated functional unit structure, code generation for wider SIMD units is not straightforward. We discovered that two major problems in vector code generation at higher vector lengths are: 1) Reduced dynamic instruction stream coverage for vectorization and 2) Huge number of permutation instructions. We proposed Variable Length Vectorization and Selective Writing to get around these two problems.

Variable Length Vectorization starts by vectorizing for maximum vector length and then iteratively reduces logical vector length to vectorize as much as possible. Code vectorized for lower logical vector length is executed on the SIMD accelerator by masking the unused vector lanes. Selective Writing enable writing to any element of the vector register instead of always to the lowermost element by scalar instructions. Therefore, the scalar instructions write results in the vector register in the order required by subsequent vector instructions. Since the results in the vector register are already ordered, the permutation instructions are no longer required.

Dynamic Selective Devectorization. As leakage is becoming a growing concern in current microprocessors, several leakage control mechanisms have been proposed. Power gating is one of the most common leakage control technique. However, power gating has an energy and performance penalty associated with it. The penalty has to be paid every time a functional unit is sent to sleep (leakage saving) mode and later awakened. This penalty is specifically unjustified if big functional units like SIMD accelerators are to be awakened only for few cycles.

We proposed a mechanism to reduce the number of power gating instances and hence the penalty associated with it. We first profile the code, dynamically, to find the SIMD accelerator usage pattern. Then the code corresponding to the low utilization periods is discovered. Afterwards, we devectorize this code so that the SIMD accelerator can be power gated for large time intervals. This helps in reducing the power gating energy penalty and maximizing the leakage energy savings. Moreover, selectively devectorizing only the code corresponding to low utilization periods have minimal impact on the performance.

7.2 Future Work

The work presented in the thesis opens up following directions in SIMD accelerators research.

Conditional Code Vectorization. In this thesis we considered unrolling loops only with a single basic block and discovered significant vectorization opportunities. As a follow up, loops with control flow can be unrolled to further increase the vectorization opportunities. Conditional code vectorization will benefit from the availability of runtime program behavior, in particular biased branch directions. Instead of including control flow in the unrolled loop, biased branch directions can be followed to include only the most frequently executed path. The unrolled loop without any control flow will provide additional vectorization opportunities without the complexity of handling the branches. If the branches inside the loop are not biased, then the conditional code can be vectorized using masked operations.

Performance and Energy Efficient Vectorization. Our Speculative Dynamic Vectorization proposal of Chapter 4 showed that the memory speculation uncovers significant vectorization opportunities and improves performance. Then in Chapter 6 we showed that selective devectorization can provide significant energy savings. Since, these two proposals are orthogonal, their basic ideas can be combined to craft a new vectorization scheme that not only provides performance but also energy efficiency. The vectorization scheme needs to profile the code to discover both the expected performance benefits and energy efficiency before vectorizing it.

References

[1] Auto-vectorization in GCC. URL http://gcc.gnu.org/projects/tree-ssa/vectorization.html

[2] AMD Radeon. http://www.amd.com/us/products/desktop/graphics/pages/radeon.aspx

[3] IBM Microelectronics Division Research Triangle Park NC. The PowerPC 440 Core. White Paper, 1999.

[4] Intel Corporation, Intel® 64 and IA-32 Architectures Software Developer´s Manual, Volume 1-3.

[5] Intel's HW/SW co-designed processor project. http://www.eetimes.com/document.asp?doc_id=1266396

[6] Nvidia GeForce. http://www.nvidia.com/object/geforce_family.html

[7] Nvidia Tesla. http://www.nvidia.com/object/tesla-supercomputing-solutions.html

[8] PowerPC ISA. http://www.power.org/documentation/power-isa-version-2-06-revision-b/

[9] Quick EMUlation tool. http://wiki.qemu.org/Main_Page

[10] Semiconductor Industries Association, "Model for Assessment of CMOS Technologies and Roadmaps (MASTAR)," 2007, http://www.itrs.net/models.html.

[11] Standard Performance Evaluation Corporation. SPEC CPU2006 Benchmarks. URL http://www.spec.org/cpu2006/.

[12] The Intel® Xeon Phi™ Coprocessor, : http://www.intel.com/content/www/us/en/high-performance-computing/ high-performance-xeon-phi-coprocessor-brief.html

[13] UTDSP Benchmarks: www.eecg.toronto.edu/~corinna/

[14] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin. 1993. Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors. *IEEE Micro* 13, 3 (May 1993), 48-61.

[15] Kanak Agarwal, Kevin Nowka, Harmander Deogun, and Dennis Sylvester. 2006. Power Gating with Multiple Sleep Modes. In *Proceedings of the 7th International Symposium on Quality Electronic Design* (ISQED '06). IEEE Computer Society, Washington, DC, USA, 633-637.

[16] Tilak Agerwala and John Cocke [1987]. High Performance Reduced Instruction Set Processors, *Tech. Rep. RC12434*, IBM Thomas Watson Research Center, Yorktown Heights, N.Y.

[17] Yoav Almog, Roni Rosner, Naftali Schwartz, and Ari Schmorak. 2004. Specialized Dynamic Optimizations for High-Performance Energy-Efficient Microarchitecture. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* (CGO '04). IEEE Computer Society, Washington, DC, USA, 137-.

[18] Hari Ananthan, Chris H. Kim, and Kaushik Roy. 2004. Larger-than-vdd forward body bias in sub-0.5V nanoscale CMOS. In *Proceedings of the 2004 international symposium on Low power electronics and design* (ISLPED '04). ACM, New York, NY, USA, 8-13.

[19] C. Auth, *et al.*, "45nm High-k+Metal Gate Strain-Ehanced Transistors," *Intel Technology Journal*, vol. 12, 2008.

[20] H. B. Bakoglu, G. F. Grohoski, L. E. Thatcher, J.A. Kahle, C.R. Moore, D.P. Tuttle, W. E. Maule, W. R. Hardell Jr., D. A. Hicks, M. Nguyenphu, R.K. Montoye, W. T. Glover, S. Dhawan. "IBM second-generation RISC machine organization," *Computer Design: VLSI in Computers and Processors, 1989. ICCD '89. Proceedings., 1989 IEEE International Conference on* , vol., no., pp.138,142, 2-4 Oct 1989.

[21] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation* (PLDI '00). ACM, New York, NY, USA, 1-12.

[22] Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, and Yigel Zemach. 2003. IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium®-based systems. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture* (MICRO 36). IEEE Computer Society, Washington, DC, USA, 191-.

[23] M. Baron. Cortex-A8: High speed, low power. M*icroprocessor Report*,11(14):1–6, 2005.

[24] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. 2002. Automatic intra-register vectorization for the Intel architecture. *International Journal of Parallel Programming* 30, 2 (April 2002), 65-98.

[25] Calin Bira, Liviu Gugu, Radu Hobincu, Valeriu Codreanu, Lucian Petrica and Sorin Cotofana. 2013. An Energy Effective SIMD Accelerator for Visual Pattern Matching. *In Proceedings of the Fourth International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies.* Edinburgh, Scotland, 13-14 June 2013.

[26] Erich Bloch. 1959. The engineering design of the stretch computer. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference* (IRE-AIEE-ACM '59 (Eastern)).

[27] Aleksandar Branković, Kyriakos Stavrou, Enric Gibert, and Antonio González. 2014. Accurate Off-Line Phase Classification for HW/SW Co-Designed Processors. In *Proceedings of the ACM International Conference on Computing Frontiers* (CF '14). Cagliari, Italy, May 2014.

[28] Aleksandar Branković, Kyriakos Stavrou, Enric Gibert, and Antonio González. 2014. Warm-Up Simulation Methodology for HW/SW Co-Designed Processors. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (CGO '14). ACM, New York, NY, USA, Pages 284, 11 pages.

[29] Aleksandar Branković, Kyriakos Stavrou, Enric Gibert, and Antonio González. 2013. Performance analysis and predictability of the software layer in dynamic binary translators/optimizers. In *Proceedings of the ACM International Conference on Computing Frontiers* (CF '13). ACM, New York, NY, USA, , Article 15 , 10 pages.

[30] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* (CGO '03). IEEE Computer Society, Washington, DC, USA, 265-275.

[31] Werner Buchholz. 1962. *Planning a Computer System: Project Stretch*. McGraw-Hill, Inc., Hightstown, NJ, USA.

[32] José Cano, Aleksandar Branković, Rakesh Kumar, Darko Zivanovic, Demos Pavlou, Kyriakos Stavrou, Enric Gibert, Alejandro Martínez, Gem Dot, Fernando Latorre, Alex Barceló, and Antonio González. Modelling HW/SW Co-Designed Processors. In *Eighth International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems* (ACACES 2012), Fiuggi, Italy, July 2012.

[33] A. E. Charlesworth. 1981. An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family. *Computer* 14, 9 (September 1981), 18-27.

[34] Nathan Clark, Amir Hormati, Sami Yehia, Scott Mahlke, and Krisztian Flautner. 2007. Liquid SIMD: Abstracting SIMD Hardware using Lightweight Dynamic Mapping. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture* (HPCA '07). IEEE Computer Society, Washington, DC, USA, 216-227.

[35] Paul D´Arcy and Scott Beach, StarCore SC140: A New DSP Architecture for Portable Devices. In *Wireless Symposium*. Motorola, September 1999.

[36] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. 2003. The Transmeta Code Morphing™ Software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* (CGO '03). IEEE Computer Society, Washington, DC, USA, 15-24.

[37] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, and Hunter Scales. 2000. AltiVec Extension to PowerPC Accelerates Media Processing. *IEEE Micro* 20, 2 (March 2000), 85-95.

[38] D. R. Ditzel and H. R. McLellan. 1987. Branch folding in the CRISP microprocessor: reducing branch delay to zero. In *Proceedings of the 14th annual international symposium on Computer architecture* (ISCA '87), D. St. Clair (Ed.). ACM, New York, NY, USA, 2-8.

[39] Kemal Ebcioğlu and Erik R. Altman. 1997. DAISY: dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th annual international symposium on Computer architecture* (ISCA '97). ACM, New York, NY, USA, 26-37.

[40] Joseph A. Fisher. 1983. Very Long Instruction Word architectures and the ELI-512. In *Proceedings of the 10th annual international symposium on Computer architecture* (ISCA '83). ACM, New York, NY, USA, 140-150.

[41] Bolei Guo, Youfeng Wu, Cheng Wang, Matthew J. Bridges, Guilherme Ottoni, Neil Vachharajani, Jonathan Chang, and David I. August. 2006. Selective runtime memory disambiguation in a dynamic binary translator. In *Proceedings of the 15th international conference on Compiler Construction* (CC'06), Alan Mycroft and Andreas Zeller (Eds.). Springer-Verlag, Berlin, Heidelberg, 65-79

[42] John L. Hennessy and David A. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers Inc. 5 edition, 2011.

[43] R.G. Hintz and D.P. Tate, "Control Data STAR-100 processor design," In *Proceedings of IEEE Compcon*, 1972, pp. 1–4.

[44] Hiroaki Hirata, Kozo Kimura, Satoshi Nagamine, Yoshiyuki Mochizuki, Akio Nishimura, Yoshimori Nakase, and Teiji Nishizawa. 1992. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *Proceedings of the 19th annual international symposium on Computer architecture* (ISCA '92). ACM, New York, NY, USA, 136-145.

[45] Justin Holewinski, Ragavendar Ramamurthi, Mahesh Ravishankar, Naznin Fauzia, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. 2012. Dynamic trace-based analysis of vectorization potential of applications. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation* (PLDI '12). ACM, New York, NY, USA, 371-382.

[46] Zhigang Hu, Alper Buyuktosunoglu, Viji Srinivasan, Victor Zyuban, Hans Jacobson, and Pradip Bose. 2004. Microarchitectural techniques for power gating

of execution units. In *Proceedings of the 2004 international symposium on Low power electronics and design* (ISLPED '04). ACM, New York, NY, USA, 32-37

[47] Libo Huang, Li Shen, Zhiying Wang, Wei Shi, Nong Xiao, Sheng Ma SIF: Overcoming the Limitations of SIMD Devices via Implicit Permutation. IEEE 16th International Symposium on High Performance Computer Architecture (HPCA), 2010, vol., no., pp.1-12, 9-14 Jan. 2010.

[48] W. Hwu and Y. N. Patt. 1986. HPSm, a high performance restricted data flow architecture having minimal functionality. In *Proceedings of the 13th annual international symposium on Computer architecture* (ISCA '86). IEEE Computer Society Press, Los Alamitos, CA, USA, 297-306.

[49] Daniel A. Jimenez and Calvin Lin. 2002. Neural methods for dynamic branch prediction. *ACM Trans. Comput. Syst.* 20, 4 (November 2002), 369-397.

[50] M. Johnson. [1990]. *Superscalar Microprocessor Design*, Prentice Hall, Englewood Cliffs, N.J.

[51] David R. Kaeli and Philip G. Emma. 1991. Branch history table prediction of moving target branches due to subroutine returns. In *Proceedings of the 18th annual international symposium on Computer architecture* (ISCA '91). ACM, New York, NY, USA, 34-42.

[52] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. In *IBM Journal of Research and Development*, 49(4), pages 589–604, July 2005

[53] Stephem W. Keckler and William J. Dally. 1992. Processor coupling: integrating compile time and runtime scheduling for parallelism. In *Proceedings of the 19th annual international symposium on Computer architecture* (ISCA '92). ACM, New York, NY, USA, 202-213.

[54] Ho-Seop Kim and James E. Smith. 2003. Hardware Support for Control Transfers in Code Caches. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture* (MICRO 36). IEEE Computer Society, Washington, DC, USA, 253-.

[55] Hyesoon Kim, José A. Joao, Onur Mutlu, Chang Joo Lee, Yale N. Patt, and Robert Cohn. 2007. VPC prediction: reducing the cost of indirect branches via hardware-

based dynamic devirtualization. In *Proceedings of the 34th annual international symposium on Computer architecture* (ISCA '07). ACM, New York, NY, USA, 424-435.

[56] Hyung-Ock Kim; Bong Hyun Lee; Jong-Tae Kim; Jung Yun Choi; Kyu-Myung Choi; Youngsoo Shin, Supply Switching With Ground Collapse for Low-Leakage Register Files in 65-nm CMOS, *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* , vol.18, no.3, pp.505,509, March 2010.

[57] A. Klaiber. The Technology Behind the Crusoe Processors. White paper, January 2000.

[58] A. Klaiber, S. Chau. "Automatic detection of logic bugs in hardware designs," *Microprocessor Test and Verification: Common Challenges and Solutions, 2003. Proceedings. 4th International Workshop on* , vol., no., pp.47,53, 29-30 May 2003.

[59] K. Krewell. Transmeta Gets More Efficeon. *Micro-processor Report*, 17(10), 2003.

[60] Alexei Kudriavtsev and Peter Kogge. 2005. Generation of permutations for SIMD processors. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems* (LCTES '05). ACM, New York, NY, USA, 147-156.

[61] Rakesh Kumar, Aleajandro Martínez, and Antonio González. 2013. Speculative Dynamic Vectorization to Assist Static Vectorization in a HW/SW Co-designed Environment. In *the Proceedings of 20th International Conference on High Performance Computing* (HiPC 2013) Bangalore, India, December 18-21, 2013.

[62] Rakesh Kumar, Aleajandro Martínez, and Antonio González. 2013. Speculative Dynamic Vectorization to Assist Static Vectorization in a HW/SW Co-designed Environment. In *Hipeac Compiler, Architecture and Tools Conference* at Haifa, Israel, November 18-19, 2013.

[63] Rakesh Kumar, Aleajandro Martínez, and Antonio González. 2013. Vectorizing for Wider Vector Units in a HW/SW Co-designed Environment. In *the Proceedings of 15th International Conference on High Performance Computing and Communications* (HPCC 2013) Zhangjiajie, China, November 13-15, 2013.

[64] Rakesh Kumar, Aleajandro Martínez, and Antonio González. 2013. Dynamic Selective Devectorization for Efficient Power Gating of SIMD units in a HW/SW

Co-designed Environment. In *the Proceedings of the 25th International Symposium on Computer Architecture and High Performance Computing* (SBAC-PAD 2013).Porto de Galinhas, Pernambuco, Brazil, October 23-26, 2013.

[65] Rakesh Kumar, Aleajandro Martínez, and Antonio González. 2012. Speculative Dynamic Vectorization for HW/SW Co-designed Processors. In *the Proceedings of the 21st international conference on Parallel architectures and compilation techniques* (PACT '12). Minneapolis, MN, USA, September 2012.

[66] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation* (PLDI '00). ACM, New York, NY, USA, 145-156.

[67] James Laudon, Anoop Gupta, and Mark Horowitz. 1994. Interleaving: a multithreading technique targeting multiprocessors and workstations. *SIGPLAN Not.* 29, 11 (November 1994), 308-318.

[68] Ruby Lee. Subword Parallelism with MAX-2. *IEEE Micro*, 16(4):51-59, Aug 1996.

[69] Jianhui Li, Qi Zhang, Shu Xu, and Bo Huang. 2006. Optimizing Dynamic Binary Translation for SIMD Instructions. In *Proceedings of the International Symposium on Code Generation and Optimization* (CGO '06). IEEE Computer Society, Washington, DC, USA, 269-280.

[70] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO 42). ACM, New York, NY, USA, 469-480.

[71] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (PLDI '05). ACM, New York, NY, USA, 190-200

[72] Anita Lungu, Pradip Bose, Alper Buyuktosunoglu, and Daniel J. Sorin. 2009. Dynamic power gating with quality guarantees. In *Proceedings of the 14th*

ACM/IEEE international symposium on Low power electronics and design (ISLPED '09). ACM, New York, NY, USA, 377-382.

[73] Marc Lupon, Enric Gibert, Grigorios Magklis, Sridhar Samudrala, Raúl Martínez, Kyriakos Stavrou, and David R. Ditzel. 2014. Speculative hardware/software co-designed floating-point multiply-add fusion. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems* (ASPLOS '14). ACM, New York, NY, USA, 623-638.

[74] Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. 2011. An Evaluation of Vectorizing Compilers. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques* (PACT '11). IEEE Computer Society, Washington, DC, USA, 372-382.

[75] H. M. Mathis, A. E. Mericas, J. D. McCalpin, R. J. Eickemeyer, and S. R. Kunkel. 2005. Characterization of simultaneous multithreading (SMT) efficiency in POWER5. *IBM J. Research and Development* 49, 4/5 (July 2005), 555-564.

[76] Cameron McNairy and Don Soltis. 2003. Itanium 2 Processor Microarchitecture. *IEEE Micro* 23, 2 (March 2003), 44-55.

[77] Steven S. Muchnick, Advanced Complier Design & Implementation, Morgan Kaufmann, 1997.

[78] Dorit Naishlos. Autovectorization in GCC. In *The 2004 GCC Developers' Summit*, pages 105–118,2004.

[79] Naveen Neelakantam, David R. Ditzel, and Craig Zilles. 2010. A real system evaluation of hardware atomicity for software speculation. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems* (ASPLOS XV). ACM, New York, NY, USA, 29-38.

[80] A. Nicolau and J. A. Fisher. 1984. Measuring the Parallelism Available for Very Long Instruction Word Architectures. *IEEE Trans. Comput.* 33, 11 (November 1984), 968-976.

[81] K. Nose, T. Sakurai. "Analysis and future trend of short-circuit power," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* , vol.19, no.9, pp.1023,1030, Sep 2000.

[82] Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, and Ayal Zaks. 2011. Vapor SIMD: Auto-vectorize once, run everywhere. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (CGO '11). IEEE Computer Society, Washington, DC, USA, 151-160.

[83] Alex Pajuelo, Antonio González, and Mateo Valero. 2002. Speculative dynamic vectorization. In *Proceedings of the 29th annual international symposium on Computer architecture* (ISCA '02). IEEE Computer Society, Washington, DC, USA, 271-280.

[84] Shien-Tai Pan, Kimming So, and Joseph T. Rahmeh. 1992. Improving the accuracy of dynamic branch prediction using branch correlation. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems* (ASPLOS V), Richard L. Wexelblat (Ed.). ACM, New York, NY, USA, 76-84.

[85] Sanjay J. Patel and Steven S. Lumetta. 2001. rePLay: A Hardware Framework for Dynamic Optimization. *IEEE Transactions on Computers* 50, 6 (June 2001), 590-608.

[86] Demos Pavlou, Aleksandar Brankovic, Rakesh Kumar, Maria Gregori, Kyriakos Stavrou, Enric Gibert, and Antonio Gonzalez. DARCO: Infrastructure for Research on HW/SW co-designed Virtual Machines. In *In Proceedings of the 4th Workshop on Architectural and Microarchitectural Support for Binary Translation (AMAS-BT'11), held in conjunction with ISCA-38*, June 2011.

[87] Demos Pavlou, Enric Gibert, Fernando Latorre, and Antonio Gonzalez. 2012. DDGacc: boosting dynamic DDG-based binary optimizations through specialized hardware support. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments* (VEE '12). ACM, New York, NY, USA, 159-168.

[88] Gang Ren, Peng Wu, and David Padua. 2005. An Empirical Study On the Vectorization of Multimedia Applications for Multimedia Extensions. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers - Volume 01* (IPDPS '05), Vol. 1. IEEE Computer Society, Washington, DC, USA, 89.2-.

[89] Gang Ren, Peng Wu, and David Padua. 2006. Optimizing data permutations for SIMD devices. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation* (PLDI '06). ACM, New York, NY, USA, 118-131.

[90] Roni Rosner, Yoav Almog, Micha Moffie, Naftali Schwartz, and Avi Mendelson. 2004. Power Awareness through Selective Dynamically Optimized Traces. In *Proceedings of the 31st annual international symposium on Computer architecture* (ISCA '04). IEEE Computer Society, Washington, DC, USA, 162-.

[91] Sumedh Sathaye , Paul Ledak , Jay Leblanc , Stephen Kosonocky , Michael Gschwind , Jason Fritts , Arthur Bright , Erik Altman , Craig Agricola BOA: Targeting multi-gigahertz with binary translation. In *Proc. of the 1999 Workshop on Binary Translation, IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, pages 2–11, 1999.

[92] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. 2003. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* (CGO '03). IEEE Computer Society, Washington, DC, USA, 36-47.

[93] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. 2008. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.* 27, 3, Article 18 (August 2008), 15 pages.

[94] Harsh Sharangpani and Ken Arora. 2000. Itanium Processor Microarchitecture. *IEEE Micro* 20, 5 (September 2000), 24-43.

[95] Jaewook Shin, Mary Hall, and Jacqueline Chame. 2005. Superword-Level Parallelism in the Presence of Control Flow. In *Proceedings of the international symposium on Code generation and optimization* (CGO '05). IEEE Computer Society, Washington, DC, USA, 165-175.

[96] Youngsoo Shin, Sewan Heo, Hyung-Ock Kim, and Jung Yun Choi. 2007. Supply switching with ground collapse: simultaneous control of subthreshold and gate

leakage current in nanometer-scale CMOS circuits. *IEEE Trans. Very Large Scale Integr. Syst.* 15, 7 (July 2007), 758-766.

[97] James E. Smith. 1981. A study of branch prediction strategies. In *Proceedings of the 8th annual symposium on Computer Architecture* (ISCA '81). IEEE Computer Society Press, Los Alamitos, CA, USA, 135-148.

[98] J. E. Smith, Greg Faanes, and Rabin Sugumar. 2000. Vector instruction set support for conditional operations. In *Proceedings of the 27th annual international symposium on Computer architecture* (ISCA '00). ACM, New York, NY, USA, 260-269.

[99] J.E. Smith and R. Nair. Virtual Machines: A Versatile Platform for Systems and Processes. (The Morgan Kaufmann Series in Computer Architecture and Design). Elsevier 2005.

[100] James E. Smith and Andrew R. Pleszkun. 1985. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th annual international symposium on Computer architecture* (ISCA '85). IEEE Computer Society Press, Los Alamitos, CA, USA, 36-44.

[101] M. D. Smith, M. Johnson, and M. A. Horowitz. 1989. Limits on multiple instruction issue. In *Proceedings of the third international conference on Architectural support for programming languages and operating systems* (ASPLOS III). ACM, New York, NY, USA, 290-302.

[102] Gurindar S. Sohi. 1990. Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers. *IEEE Trans. Comput.* 39, 3 (March 1990), 349-359.

[103] Manu Sporny, Gray Carper, and Jonathan Turner. The Playstation 2 Linux Kit Handbook, 2002

[104] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *International Journal of Parallel Programming*, 28, 4 (August 2000), 363-400.

[105] James E. Thornton. 1964. Parallel operation in the control data 6600. In *Proceedings of the October 27-29, 1964, fall joint computer conference, part II: very high speed computer systems* (AFIPS '64 (Fall, part II))

[106] James E. Thornton, Design of a Computer: The Control Data. Scott, Foresman and Company, 1970.

[107] G. S. Tjaden and M. J. Flynn. 1970. Detection and Parallel Execution of Independent Instructions. *IEEE Trans. Comput.* 19, 10 (October 1970), 889-895.

[108] R. M. Tomasulo. 1967. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Research and Development* 11, 1 (January 1967)

[109] Marc Tremblay and Michael O'Connor and Venkatesh Narayanan and Liang He. VIS Speeds New Media Pro-cessing. *IEEE Micro*, 16(4):10-20, Aug 1996.

[110] J.W. Tschanz, S. G. Narendra, Ye Yibin, B.A. Bloechel, S. Borkar, V. De. Dynamic sleep transistor and body bias for active leakage power control of microprocessors, *Solid-State Circuits, IEEE Journal of* , vol.38, no.11, pp.1838,1845, Nov. 2003.

[111] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. 1998. Simultaneous multithreading: maximizing on-chip parallelism. *Proc. 22nd Annual Int'l. Symposium on Computer Architecture (ISCA),* June 22–24, 1995, Santa Margherita, Italy, 392–403.

[112] Sriram Vajapeyam, P. J. Joseph, and Tulika Mitra. 1999. Dynamic vectorization: a mechanism for exploiting far-flung ILP in ordinary programs. In *Proceedings of the 26th annual international symposium on Computer architecture* (ISCA '99). IEEE Computer Society, Washington, DC, USA, 16-27

[113] Cheng Wang, Marcelo Cintra, and Youfeng Wu. 2013. Acceldroid: Co-designed acceleration of Android bytecode. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (CGO '13). IEEE Computer Society, Washington, DC, USA, 1-10.

[114] W. J. Watson. 1972. The TI ASC: a highly modular and flexible super computer architecture. In *Proceedings of the December 5-7, 1972, fall joint computer conference, part I* (AFIPS '72 (Fall, part I)). ACM, New York, NY, USA, 221-228.

[115] Mark Woh, Sangwon Seo, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Krisztian Flautner. 2009. AnySP: anytime anywhere anyway signal processing. In *Proceedings of the 36th annual international symposium on Computer architecture* (ISCA '09). ACM, New York, NY, USA, 128-139

[116] Y. Ye, S. Borkar, V. De. A new technique for standby leakage reduction in high-performance circuits, *VLSI Circuits, 1998. Digest of Technical Papers. 1998 Symposium on* , vol., no., pp.40,41, 11-13 June 1998

[117] Thomas Y. Yeh, Petros Faloutsos, Sanjay J. Patel, and Glenn Reinman. 2007. ParallAX: an architecture for real-time physics. In *Proceedings of the 34th annual international symposium on Computer architecture* (ISCA '07). ACM, New York, NY, USA, 232-243.

[118] Tse-Yu Yeh and Yale N. Patt. 1992. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th annual international symposium on Computer architecture* (ISCA '92). ACM, New York, NY, USA, 124-134.

[119] Tse-Yu Yeh and Yale N. Patt. 1993. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th annual international symposium on computer architecture* (ISCA '93). ACM, New York, NY, USA, 257-266.

[120] Ahmed Youssef, Mohab Anis, and Mohamed Elmasry. 2006. Dynamic Standby Prediction for Leakage Tolerant Microprocessor Functional Units. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO 39). IEEE Computer Society, Washington, DC, USA, 371-384