

# Dynamic Selective Devectorization for Efficient Power Gating of SIMD units in a HW/SW Co-designed Environment

**Abstract**—Leakage power is a growing concern in current and future microprocessors. Functional units of microprocessors are responsible for a major fraction of this power. 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 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 paper focuses on increasing the idle interval for the higher SIMD lanes. 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. Our experimental results show average energy savings of 12% and 24% over power gating, for SPECFP2006 and Physicsbench. Moreover, the slowdown caused due to devectorization is less than 1%.

**Keywords**— *HW/SW Co-designed processor, Devectorization, Power Gating, Leakage*

## I. 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 [2][5][6][8][10][12][18]. 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 [9][11][19][20], 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 [15][22]. 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. This paper 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 devectorization might also result in significant slowdown.

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.

The paper proposes to extract maximum vectorization opportunities at compile time. Then, at run-time, 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. The main contributions of the paper can be summarized as:

- 1) Proposes a mechanism to increase power gating efficiency by increasing the idle interval duration.
- 2) Proposes a dynamic selective devectorization algorithm to keep the higher SIMD lanes idle for long time duration without significant effect on performance.
- 3) A dynamic profiling technique to discover devectorizable portions of the code.
- 4) Evaluation of the proposals and comparison with power gating. The proposed technique achieves 12% and 24% more energy savings than power gating for SPECFP2006 and Physicsbench.

The rest of the paper is organized as follows: Section II provides a background and related work on HW/SW co-designed processors and power gating. Section III briefly provides the motivation for the work presented in this paper. Section IV describes the proposals. Evaluation of the proposals using SPECFP2006 and Physicsbench applications is presented in Section V. Section VI conclude the paper.

## II. BACKGROUND AND RELATED WORK

HW/SW Co-designed processors [7][16] 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 is 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.

As leakage is becoming a growing concern in the current microprocessor designs, several leakage control mechanisms have been studied [9][11][19][20]. All these mechanisms try to reduce leakage when the circuit is in idle state. Power gating [9] consists of shutting down parts of the circuit by cutting their power supply by means of high threshold header or footer transistors. SSGC [11] 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 [20] 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 [4][19] 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 most commonly used leakage control technique. There have been several proposals to increase the efficiency of power gating. Hu et al. [9] 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 have been proposed to minimize this energy loss [3][15][22]. A. Youssef et al. [22] 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. [15] 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. [3] 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, this paper focuses on how to keep a unit shut down for longer time intervals once it is already power gated. Even though the paper targets 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 devectored and executed on the lowest SIMD lane.

### III. 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. 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.

We profiled SPECFP2006 to discover the higher SIMD lanes usage pattern. Figure 1 shows the percentage of vector instructions (higher lanes usage profile) in the dynamic instruction stream, over the execution time for 434.zeusmp<sup>1</sup>. 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.

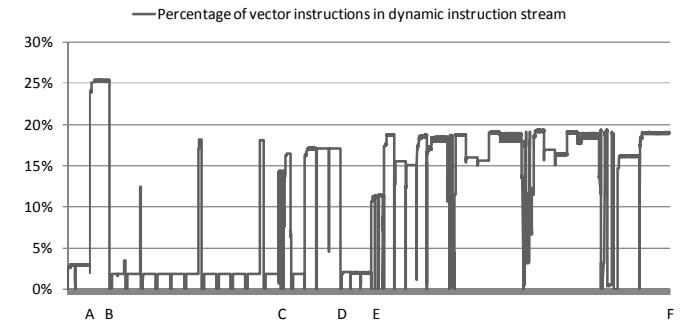


Figure 1 Percentage of vector instruction (excluding memory instructions) in the dynamic instruction stream over the time (4 billion instructions).

We propose to devectored 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. Devectoredizing 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.

### IV. PROFILING AND DEVECTORIZATION

This section provides the details of the dynamic profiling and devectoredization schemes. The software layer of our co-designed processor is called Translation Optimization Layer (TOL). TOL 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). Devectoredization is done in SBM, which is the most aggressive translation/optimization level, after applying several standard compiler optimizations.

#### A. Profiling and Superblock Creation

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. A superblock generally includes multiple basic blocks following the biased direction of branches.

<sup>1</sup> For 4 billion instruction executed starting from the most frequently execution function/routine. Vector instructions shown do not include memory instructions since they do not use SIMD functional units.

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 in SBM

### 2) Higher SIMD lanes usage pattern:

To monitor the usage of higher SIMD lanes a N-bit shift register is employed. 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 devectoize the basic block, if it is included in a superblock. The devectoization is desirable in this case, since having less number of vector instructions indicate low usage of higher SIMD lanes. Therefore, devectoizing this code will help improving power gating efficiency without a significant impact on the overall performance. To increase the devectoization 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 than  $DV_{th}$ , the devec counter is decremented. It indicates that devectoitzion is not favored due to more utilization of higher SIMD lanes. Therefore, the final decision of whether to devectoize the basic block or not depends on the shift register values during all the executions of the basic block in BBM. This helps in devectoizing only the basic blocks which are executing during the low usage phase of higher SIMD lanes like B-C in Figure 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 devectoized. Otherwise, the superblock is kept in the vectorized form. This selective devectoization of superblocks improves leakage energy savings through power gating while maintaining the performance.

### B. Optimizations:

In this phase, several standard optimizations are applied dynamically. First of all, the superblock is converted into Static Single Assignment (SSA) form to remove anti and output dependences. Then, the optimizations Constant Propagation, Copy Propagation, Constant Folding, Common Sub-expression Elimination and Dead Code Elimination are applied. The next step is to generate the Data Dependence Graph (DDG). During DDG creation, Redundant Load Elimination and Store Forwarding are also applied to improve the quality of the generated code.

### C. Devectorization

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

```

devec(SB):
    for each instruction s in SB:
        if s is devectoizable:
            devec_len ← get_devec_len(s)
            scalar_op ← get_scalar_opcode(s)
            scalar_in_regs ← get_scalar_in_reg(s)

            scalar_out_reg ← φ
            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)

get_scalar_in_reg (s)
    scalar_in_regs ← φ

    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

```

Figure 2 Pseudo code for devectoization. “**devec**” is the top level devectoization routine while “**get\_scalar\_in\_reg**” generates Upack instruction to put vector register values to scalar register, if not already done.

Figure 2 presents the devectoization algorithm. “**devec**” is the top level routine that receives the superblock “SB” to be devectoized. 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 devectoization. The first step in devectoization is to find devectoization 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 devectoized, the corresponding input registers are already mapped to the output scalar registers of the scalar producers. However, if the producers cannot be devectoized (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*).

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 of Figure 2 for the sake of simplicity.

#### D. 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 (DP) floating point operation or two 32-bit single precision (SP) floating point operations. Devectorized code is executed on the lower lane, so that the higher lane could be switched off.

In general, a SP vector instruction would be devectorized into four SP scalar instructions. However, partial devectorization generates only two SP “half-vector” instructions. A “half-vector” instruction combines two scalar instructions that can be executed in parallel. The rational behind partial devectorization is to utilize the 64-bit wide vector lanes. Since one vector lane can execute two SP 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.

## V. PERFORMANCE EVALUATION

### A. Experimental Framework

DARCO [17], which is an infrastructure for evaluating HW/SW co-designed virtual machines, is used to evaluate the proposals. DARCO executes guest x86 binary on a PowerPC-like RISC host architecture. The proposed profiling and devectorization algorithm are implemented in TOL. Furthermore, for energy consumption analysis McPAT [13] is integrated with DARCO. 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.

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). We compare our results with both the configurations.

### B. Benchmarks

To measure the success of the proposals we use applications from SPECFP2006 [1] and Physicsbench [21] benchmarks suites. For SPECFP2006 we instrument the benchmarks, using PIN [14], to find the most frequently executing routines. Then we simulate four billion instructions starting from these routines. The benchmarks in Physicsbench are executed till completion. The benchmarks are compiled with Intel ICC version 12.1.4, optimization flags “-O3” and vectorization flag “-xSSE3”. Only floating point benchmarks in SPEC2006 are considered for evaluation since the floating point code is the main target of our proposals.

### C. Models and Parameters

To measure the success of the proposals, we refer to the energy model proposed by Hu et al. [9]. However, we changed some of model input values. Their breakeven threshold value is between 9 and 24 cycles. However, as A. Youssef [22] 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.

A. Lungu et al. [15] proposed a success monitor based improvement to the time-based power gating mechanism of [9]. 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. [15] have a fixed idle detect interval 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 [22]. Moreover, we consider power gating of only the SIMD accelerator in our experiments.

We model a simple in-order processor, in congruence with the simple hardware design philosophy of HW/SW co-designed processors, with issue width of two. Microarchitectural parameters for the modeled processor are

given in Table I. The table also shows key McPAT parameters used to get the energy consumption of the modeled processor.

TABLE I. PROCESSOR MICROARCHITECTURAL AND MCPAT PARAMETERS

Parameter	Value
<b>Processor Microarchitectural Parameters</b>	
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/SIMD 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
<b>McPAT Parameters</b>	
Technology	65nm
Clock Rate	1.5 GHz
Temperature	350 K
Device Type	High Performance

#### D. Higher SIMD lane usage profile

The dynamic selective devectorization (DSD) tries to minimize the usage of higher SIMD lane during the low utilization period. As shown in Figure 1 in Section III, 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.

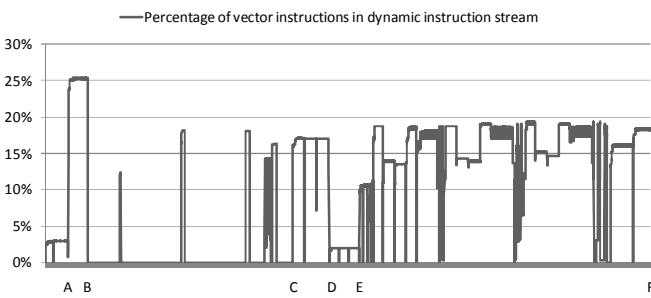


Figure 3 Percentage of vector instruction in the dynamic instruction stream after dynamic selective devectorization for 434.zesump.

Figure 3 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.

#### E. 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) Dynamic energy consumption of the lower SIMD lane increases, since it has to execute more instructions.
- 2) Dynamic energy consumption increases due to profiling and devectorization of selected superblocks.
- 3) Leakage energy of the rest of the chip (excluding SIMD accelerator) might increase due to the possible slowdown because of devectorization.

Figure 5 shows the overall energy savings of the SIMD accelerator by Combined power gating (CPG), Split power gating (SPG) and DSD (considering all the mentioned energy overhead components). As the figure shows, DSD outperforms both CPG and SPG significantly. The proposed technique has been able to save 31% and 45% energy over CPG and 12% and 24% energy over SPG 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.

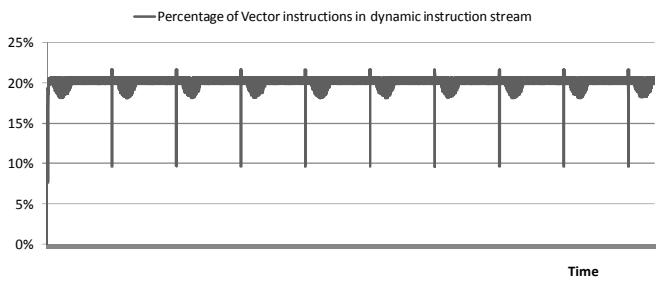


Figure 4 Percentage of vector instruction (excluding memory instructions) in the dynamic instruction stream for 470.lbm.

For Physicsbench, DSD performs better than SPG for all the benchmarks, whereas in SPECFP2006 there are few benchmarks that do not show any further energy savings. The reason for not having any additional energy savings in these benchmarks is the higher lane utilization pattern. Figure 4 shows the higher lane utilization pattern for 470.lbm. As is evident from the figure, the percentage of vector instructions (hence the utilization of the higher SIMD lane) is constant during the execution. Any attempt of devectorization would

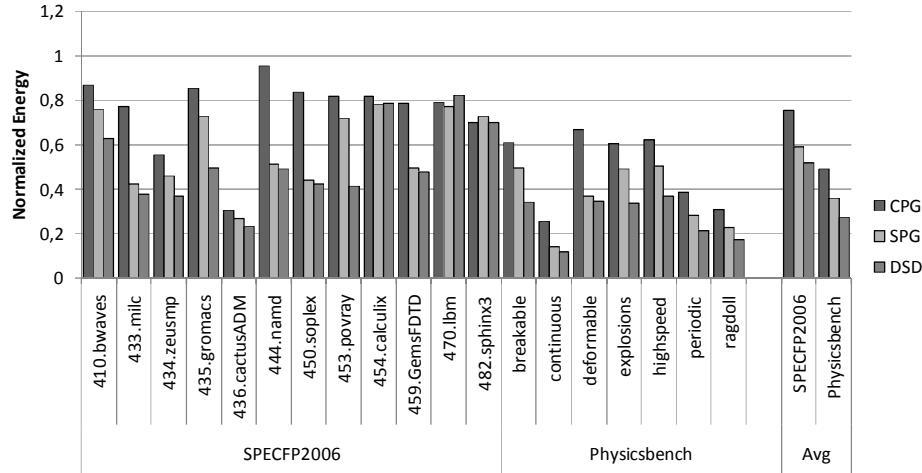


Figure 5 Energy savings in SIMD accelerator for CPG, SPG and DSD normalized to no power gating case. (Lower is better)

result in significant performance loss. Therefore, the benchmark is executed in vectorized form and no leakage energy savings are achieved. Other benchmarks with high utilization of higher lane are 454.calculix, 459.GemsFDTD and 482.sphinx3. On the contrary, there are other benchmarks like 444.namd and 450.soplex, where ICC is not able to find much vectorization opportunities. For these benchmarks SPG itself keeps the higher lane off for long intervals. Therefore, the additional benefit from DSD is negligible. However, overall results show significant energy savings through the proposed dynamic selective devectorization.

An interesting point to note in Figure 5 is that in the cases where DSD is not able to reduce leakage, e.g. 470.lbm, the energy overhead of DSD is negligible. Hence, DSD has insignificant energy penalty when it fails to increase leakage benefits.

#### F. Overall energy savings

Figure 6 shows the overall energy savings of the whole processor by CPG, SPG and DSD. Since, we consider power gating only the SIMD accelerator and no other functional unit, once again, DSD outperforms both SPG and CPG.

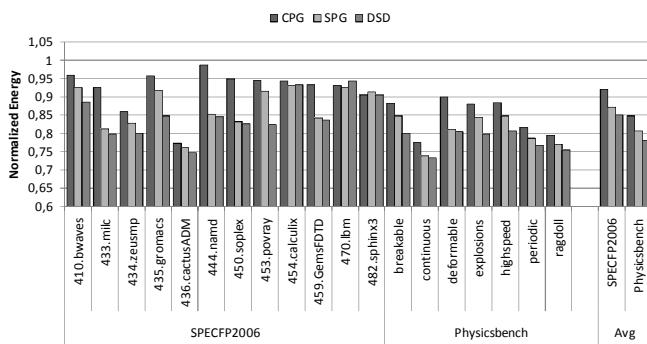


Figure 6 Processor energy savings for CPG, SPG and DSD normalized to no power gating case (Lower is better).

#### G. 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:

- 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.

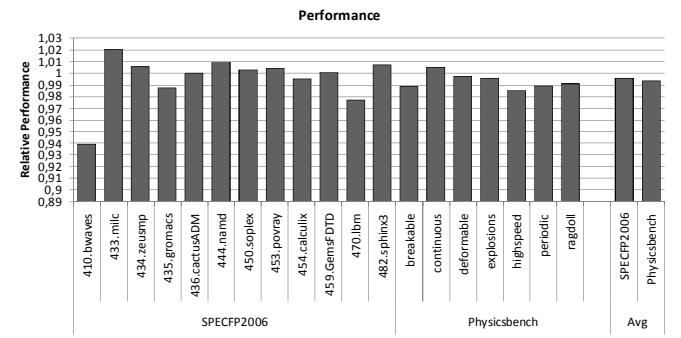


Figure 7 Overall Performance after DSD normalized to SPG (Higher is better)

Figure 7 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%. However, there are benchmarks like 433.milc, 434.zeusmp, 444.namd etc. that experience a small speedup. On the other hand, 410.bwaves suffers slowdown of 6% due to DSD. However, it also achieves energy savings of 17%. Therefore, DSD provides a

trade-off between performance and energy. Nevertheless, the energy savings of DSD are much more than the slowdown suffered.

#### **Discussion:**

As the leakage energy is becoming a growing concern in the present day microprocessors, several leakage control mechanisms have been studied [9][11][19][20]. 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 paper presented a technique called Dynamic Selective Devectorization to increase the idle durations of higher SIMD lanes. The increase in idle interval translates to increase 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.

The paper presented a mechanism to increase the idle period of higher SIMD lanes to save more leakage energy. DSD devectors certain portions of the code to reduce the higher SIMD lanes utilization during low usage periods. Even though the paper 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.

#### **VI. CONCLUSION**

This paper proposed to increase the leakage energy savings by increasing the idle interval of the higher SIMD lanes. To increase the idle interval, the paper 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 devectors 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 [11] 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 energy savings of 12% and 24% over power gating, for SPECFP2006 and Physicsbench respectively. Moreover the slowdown caused due to devectorization is less than 1%.

#### **REFERENCES**

- [1] Standard Performance Evaluation Corporation. SPEC CPU2006 Benchmarks. URL <http://www.spec.org/cpu2006/>.
- [2] Intel Corporation, Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1-3.
- [3] K. Agarwal et al. Power Gating with Multiple Sleep Modes. In Proceedings of (ISQED '06) 633-637
- [4] H. Ananthan et al. 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), 8-13.
- [5] PD'Arcy and S. Beach, StarCore SC140: A New DSP Architecture for Portable Devices. In *Wireless Symposium*. Motorola, Sept. 1999.
- [6] M. Baron. Cortex-A8: High speed, low power. *Microprocessor Report*, 11(14):1-6, 2005.
- [7] J. C. Dehnert et al. The transmeta code morphing™ software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In Proceedings of CGO '03, pages15-24.
- [8] K. Diefendorff, P.K. Dubey, R. Hochsprung, H. Scale. AltiVec extension to PowerPC accelerates media processing, *IEEE Micro*, , vol.20, no.2, pp.85-95, Mar/Apr 2000
- [9] Z. Hu et al. Microarchitectural techniques for power gating of execution units. In Proceedings of the 2004 international symposium on Low power electronics and design (ISLPED '04). 32-37
- [10] J. A. Kahle et al. Introduction to the Cell Multiprocessor. In *IBM Journal of Research and Development*, 49(4), pages 589–604, July 2005
- [11] H. Kim et al. Supply switching with ground collapse for low-leakage register files in 65-nm CMOS. *IEEE Trans. Very Large Scale Integr. Syst.* 18, 3 (March 2010), 505-509.
- [12] R. Lee. Subword Parallelism with MAX-2. *IEEE Micro*, 16(4):51-59, Aug 1996.
- [13] S. Li et al. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In Proceedings of the 42nd International Symposium on Microarchitecture (MICRO 42).
- [14] C. K. Luk et al. 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.
- [15] A. Lungu et al. Dynamic power gating with quality guarantees. In Proceedings of the 14th ACM/IEEE international symposium on Low power electronics and design (ISLPED '09), 377-382
- [16] S. S. Paul et al. 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.
- [17] D. Pavlou et al. DARCO: Infrastructure for Research on HW/SW co-designed Virtual Machines. In Proceedings of the 4th Workshop on Architectural and Microarchitectural Support for Binary Translation (AMAS-BT'11), held in conjunction with ISCA-38.
- [18] M. Sporny, G. Carper, and J. Turner. The Playstation 2 Linux Kit Handbook, 2002.
- [19] J.W. Tschanz et al. Dynamic sleep transistor and body bias for active leakage power control of microprocessors, In Solid-State Circuits, IEEE Journal of , vol.38, no.11, pp.1838,1845, Nov. 2003
- [20] Y. Ye et al. A new technique for standby leakage reduction in high-performance circuits, In VLSI Circuits, 1998. Digest of Technical Papers. 1998 Symposium on , vol., no., pp.40,41, 11-13 June 1998
- [21] T. Y. Yeh et al. Parallax: An architecture for real-time physics. In Proceedings of the 34th annual international symposium on Computer architecture (ISCA '07), pages 232-243.
- [22] A. Youssef et al. Dynamic Standby Prediction for Leakage Tolerant Microprocessor Functional Units. In Proceedings of the 39th International Symposium on Microarchitecture (MICRO 39), 371-384