Bit-Sliced Datapath for Energy-Efficient High Performance Microprocessors

Sumeet Kumar, Prateek Pujara, and Aneesh Aggarwal
ECE Department
Binghamton University
Binghamton, NY 13902
{skumar1,ppujara1,aneesh}@binghamton.edu

Abstract

In the recent years, both power and performance have become important in the design of microprocessors. In this paper, we investigate exploiting the small-sized data values for energy-efficient high performance microprocessors. For this purpose, we bit-slice the execution core (which includes the functional units, register files, data caches, and forwarding logic), so that small portions of the data are operated upon in different bit-slices. The bit-slices operating upon the higher order bits are activated only if required, saving significant energy consumption. We also investigate further advantages facilitated by bit-slicing such as energy savings obtained by reducing the number of ports provided in the higher order bit-slices and by "shutting off" bit-slices to reduce leakage energy consumption. We found that a significant energy saving can be obtained in the register file (about 20%) and the Level-1 Data Cache (about 30%) with a negligible loss of only about 2% in the instruction throughput. Our studies also showed almost 20% savings in the register file leakage energy consumption, when the unwanted higher order bit-slices are "turned off". Bit-slicing also helps in reducing the latency of the different hardware structures, which can facilitate clock speed improvements.

Keywords : Bit-sliced Datapath, Dynamic Energy Consumption, Register File, Data Cache, Leakage Energy Consumption

1 Introduction

Energy consumption has emerged as an important criteria in the design of microprocessors [9]. The major contributor to the overall energy consumption in a chip is the dynamic energy consumption. However, the leakage energy consumption is also on the rise [7], and has started to become a concern in the microprocessor designs. Dynamic energy consumption results from the activity in a processor and is caused by the charging and discharging of the capacitive loads in the processor. Leakage energy consumption, on the other hand, is the result of shrinking transistor sizes (used to facilitate faster switching) which leads to increased sub-threshold current. Many architectural techniques have been recently proposed to reduce both the dynamic and the leakage energy consumption in the processor.

One important approach towards reducing the energy consumption in a processor, while not hurting the performance, is to limit the amount of unnecessary work performed by the processor. Clock gating [2] is a good example of this approach, where the hardware that does not need to be activated during an operation is not provided with the clock signal. The architecture presented in this paper is in the same spirit, *i. e.* limit the amount of unnecessary work performed by the processor. For this, we exploit the small-sized data values. Studies [1, 2, 12] have shown that a significant number of data values being operated upon in the processor core are of small size and have a large number of leading zeros (or leading ones for small negative values). In this paper, we bit-slice the processor datapath. In the bit-sliced architecture, a particular bit-slice operates only on certain data bits, and other bit-slices operate on other data bits. In this architecture, operations in the higher end bit-slices are performed only if required, thus reducing the unnecessary work and saving energy. Even though there are many techniques in the literature that exploit the narrow-width data property to various effects, bit-sliced datapath has only been proposed to a limited extent in [4], and not to the extent to which we bit-slice the execution core datapath. Bit-slicing the datapath also has the potential of improving the clock speed by reducing the access latencies of each of the bit-sliced hardwares. We experiment with a 32-bit RISC architecture, however, the benefits of bit-slicing are expected to increase as the processors use wider data sizes (64-bit processors and beyond). To motivate the approach, we present the sizes of the data values in the processor.

1.1 Data-Sizes

We measure the operand sizes for the integer instructions operating on the integer data values. For the measurements, we separate the integer instructions into simple integer instructions (such as Add, Subtract, And, Or, etc.), complex integer instructions (such as Multiply, Divide, and Shifts), load instructions and store instructions. Figures

1(i), 1(ii), and 1(iii) give the percentage of simple, complex, and load/store instructions, respectively, that have either one or both operands of size greater than 8, 16, and 24 bits for a 32-bit RISC architecture. The legend for these graphs is given in Figure 1(iii). Operands of sizes greater than 24 bits also include the negative values. Note that, in the figures, the instructions that have at least 1 operand less than or equal to 8 bits (in the second bar) can have the other operand of a larger size. For the load and store instructions, we also measure the size of the values loaded from the data cache and stored into the data cache, respectively, and is shown in Figure 1(iv).

Figure 1(i) shows that there are about 60% of simple instructions that have at least 1 operand that is greater than 16 bits (*i.e.* about 40% of the simple instructions have both the operands less than or equal to 16 bits). When both the operands are considered, there are only about 10% of the simple instructions that have both the operands greater than 16 bits. This means that, even if an instruction has an operand that is greater than 16 bits in size, it rarely is performing any operations on the higher order bits because the other operand is less than 16 bits in size (there may be a few cases with a carry generated by the lower bits). When considering the complex instructions, in Figure 1(ii), there are considerably fewer instructions (about 30% on an average) that have at least one operand greater than 16 bits in size. For the complex instructions as well, there are only about 10% of instructions that have both their operands of size greater than 16 bits. The load and store instructions are different and they always have one operand (among the operands used for effective address calculation) that requires almost 32 bits for representation. Considering the size of the values that are loaded from the data cache and stored into the data cache, there are only about 50% of the values that have a size that is greater than 16 bits (these include the negative values as well). The percentage of wider values loaded from the cache and stored into the cache is relatively higher for the FP benchmarks (`applu`, `art`, `ammp`, `mesa`, `mgrid`, `swim`, and `wupwise`), because they load and store floating point values that typically use the entire 32 (64 bits for double precision) for representation.

Figure 1 suggests that significantly small number of operations are performed on the upper bits of operands, and motivate a bit-sliced architecture, where the higher order bits are operated upon only when required.

1.2 Related Work

Exploiting the narrow-width operands for optimizations is not a new topic. There has been some past work that uses the *data widths* of the values (generated in a program) to various effects. The SIMD paradigm takes advantage of the narrow width operands to improve the performance of the multimedia applications [11, 14]. The Dynamic Zero Compression (DZC) cache [19] reduces cache energy by exploiting the small-width property of values stored in the data cache, by using a single bit to indicate that a full byte is zero.

Brooks [2] and Gabriel [12] use the small-width operand sizes to dynamically pack different small-width data values and perform simultaneous operations on them in order to improve performance, and reduce power consumption. Canal et. al. [4, 5] proposed two approaches to exploit narrow-width operands. In [4], they considered a byte-serial (8-bit) or a semi-parallel (16-bit) pipeline to exploit the narrow-width data at the architectural level. The idea is to append extension bits to data residing in the caches and registers to reflect the significant part of the data, and only load, store, or compute on the useful bytes, thus reducing switching activities. However, the limited datapath width provided can lead to significant performance loss when processing operands of a larger bitwidth (32 bits or more). In [5], the authors rely on profiling information used along with static value range propagation analysis to discover useful range of operand-width, and re-encoding operands with narrower opcodes.

There also have been some other compiler-level efforts to exploit the narrow operand widths, such as [13, 17, 6, 15].

To the best of our knowledge, no one has yet investigated the performance of a bit-sliced execution core to exploit the narrow-width properties of operands. The Pentium 4 architecture [10] uses staggered ALUs, where the operations are first performed on the lower bits and the result is immediately bypassed to the dependent instructions, to execute the simpler instructions at double the clock speed. However, the extent of bit-slicing in Pentium 4 is very limited and it still has a traditional register file and data cache.

1.3 Contributions of this Work

The main focus of this paper is to study the performance of a bit-sliced datapath in terms of its impact on the instruction throughput and the energy savings for a high performance processor. Our studies showed that about 20% and 30% dynamic energy savings can be obtained in the register file and the data cache (by bit-slicing them), respectively, for a 2-way bit-sliced datapath. However, the reduction in the instruction throughput is only about 2%, compared to a non-bit-sliced datapath. We also investigate the reasons for IPC loss in a bit-sliced datapath and recover some of the lost IPC using performance enhancement techniques such as early resolution of branches. We also investigate how bit-slicing can facilitate further reduction in energy consumption. For this, we propose reducing the number of ports in the higher order bit-slices of the storage elements such as the register file and the data cache. To reduce the leakage energy consumption, we propose "shutting off" parts of higher order bit-slices that do not store significant data. With these energy reduction techniques, the overall dynamic energy consumption can be reduced by about 25% in the register files and about 32% in the data cache, and the register file leakage energy consumption can be reduced by about 20%.

The rest of the paper is organized as follows. Section 2 presents the bit-sliced execution core architecture and its impact on performance and energy consumption. Section 3 presents and discusses both the IPC and the energy

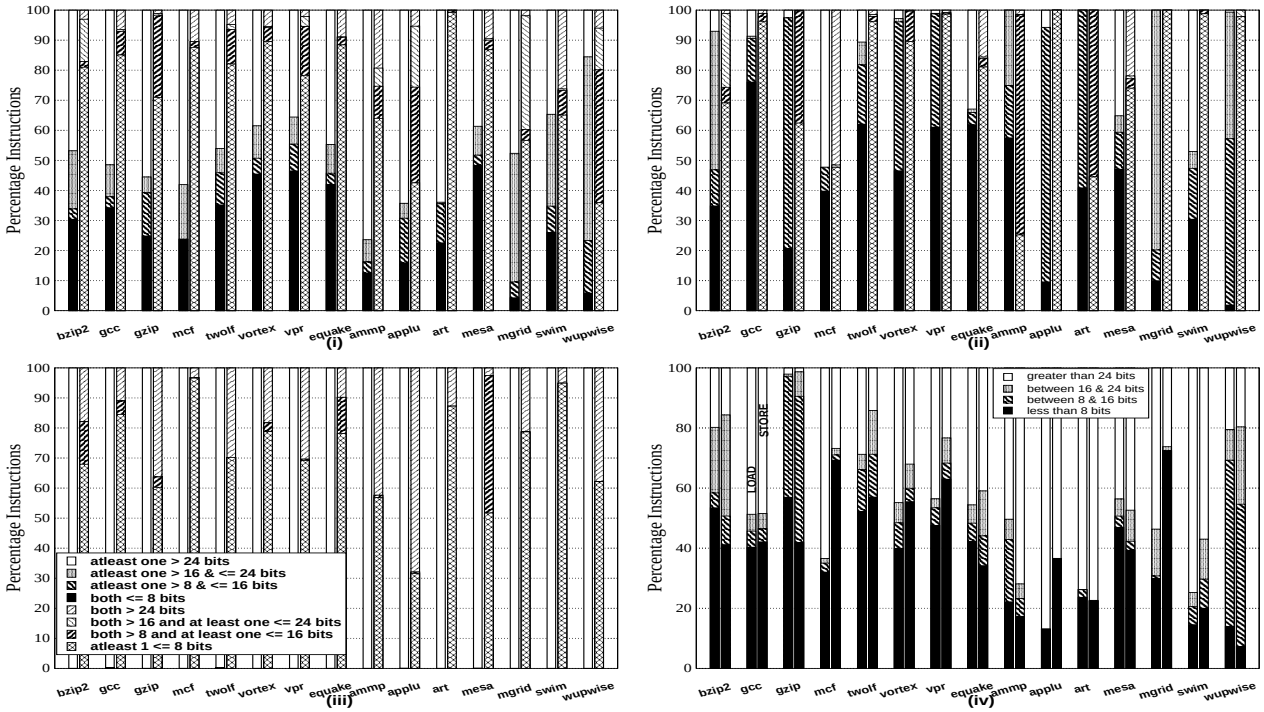

Figure 1: Operand Sizes of Integer Instructions for (i) Simple; (ii) Complex; (iii) Load/Store Instructions (1st Bar is for One Operand and 2nd Bar is for Both the Operands); and (iv) Sizes of Loaded and Stored Values

consumption results. Section 4 proposes selective delays technique to recover some of the IPC loss incurred. Section 5 presents techniques to further reduce both the dynamic and the static energy consumption with the help of a bit-sliced architecture. We conclude in Section 6.

2 Bit-sliced Execution Core Datapath

2.1 Basic Architecture

In a bit-sliced execution core, each wide integer ALU is partitioned into smaller width ALUs, the integer register file is partitioned into multiple smaller width register files, and even the data cache is partitioned into multiple smaller width data caches. For instance, for a 32-bit word machine, each 32-bit ALU can be partitioned into 2 16-bit ALUs, the integer register file can be partitioned into 2 banks, each of size 16 bits, and the data cache can be partitioned into 2 data caches, where each bank stores 16 bits of a word. We call each such partitioned hardware module as a *bit-slice*. In the bit-sliced datapath, the lowest bit-slice only operates on the lowest bits of the operands, and the next higher bit-slice operates on the next higher bits, and so on. A 2-way bit-sliced execution core datapath (for 2 ALUs and 1 data cache port) is shown in Figure 2, where the ALUs, the register file, the bypass network, and the L1 data cache are all bit-sliced into 2 parts.

As can be seen in Figure 2, ALU01 and ALU11 can only access RF1, and ALU02 and ALU12 can only access the RF2, and data loaded from DC1 is bypassed only to ALU01 and ALU11 and loaded only into RF1, and data loaded from DC2 is bypassed only to ALU02 and ALU12 and loaded only into RF2. The datapath for the stores is also similar. However, load and store instructions that need to load and store a single byte or a single half word, may lead to a transaction between RF1 and DC2, because the byte or the half word that is being accessed in the data cache may be present in DC2. Hence, the values from DC2 also need to be forwarded to the lower-bit ALUs and RF1, and vice versa. In our architecture, the transactions between the lower order bit slice and DC2 require an additional cycle. Since values to be stored in the cache are placed into the write buffers, the write buffers can also be bit-sliced. In our design, for simplicity, we assume that the multiplier functional unit, responsible for executing complex instructions, is not bit-sliced, because of lower frequency of complex instructions and difficulties in bit-slicing some of the complex operations. Hence, complex instructions wait till their entire operands are available, before they start execution.

In the proposed architecture, the load and store instructions cannot issue until their entire address has been computed. The advantage with the simple instructions executing on the ALUs was that the lower end bits of the operands were available for the consumer instructions even if the producer instruction had not finished execution on the upper bits. This advantage is not available for the load instructions. For simplicity, in our initial design, we assume that the AGUs wait till the entire operands are available before computing the effective address. This may delay the issue of the load and store instructions. Later in Section 4, we will see how this constraint is relaxed.

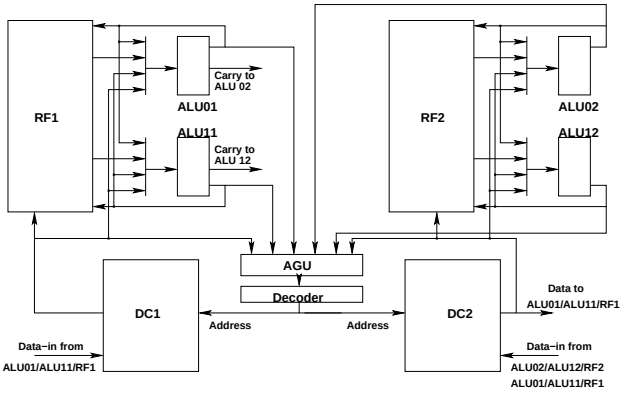

Figure 2: Schematic 2-way Bit-Sliced Execution Core Datapath

We do not bit-slice the Floating-Point (FP) subsystem[1]. However, when a value is loaded from a bit-sliced data cache, the FP instructions that are dependent on load instructions will have to wait additional cycles for their entire operands to become available. Hence, a slightly higher performance impact can be expected for the FP benchmarks. However, the simple ALU instructions dependent on the load instructions do not need to wait, because they can start executing on the lower bits when they become available and execute on the higher bits in the following cycles as and when they become available.

The execution of an integer instruction in the bit-sliced datapath takes place as follows. If an instruction is issued to the ALUs for execution, it starts execution on the lower end bits of the operands and operations on the higher end bits are performed only if required. Hence, values are read from RF2, and the result values are written into RF2 only when required. The complex instructions start execution only when all the bits of the operands are available, and these instructions write the values both in RF1 and RF2. The memory operations also compute the effective address only when the entire operands are available (delaying the issue of the load and store instructions). Once, the effective address is computed, first DC1 is accessed and then DC2 is accessed if required. When loading or storing single bytes or half-words from DC2, nothing is done in the DC1 access pipeline stage and in the next cycle DC2 is accessed.

2.2 Determining Requirement

In the previous section, we observed that the higher bit-slices of register file and the data cache are accessed only when required. To determine if a register file bit-slice needs to be read, we use an additional bit (called *next read* bit) for every register entry. When values are written into the register files, the corresponding bits are set. For instance, when a value is written into a register entry in RF2, the *next read* bit corresponding to that register entry is set. When a register entry in RF1 is read, the *next read* bit for that register entry is also read simultaneously, and depending on the value of the bit, the entry in RF2 is either read or not. RF1, storing the least significant bits, is always read. The *next read* bits are also used during concatenation of the bit-sliced values, read from all the bit-sliced register files, to form the entire operands for address computation and complex instructions. A similar procedure is employed for reading the Level 1 data caches, where a *next read* bit is employed for each word in the cache. However, the L1 data caches may also be written from the lower level data cache, and for the technique to work efficiently, the *next read* bit for each word needs to be set accordingly. For this purpose, when a cache block is loaded from a lower level cache into the L1 cache, the higher end bits of all the words in the cache block are checked simultaneously and the bits for the words in the cache block are set accordingly. The *next read* bits are also used to write the correct values in the lower level cache during write-back. The number of additional bits required depends on the amount of bit-slicing done. For instance, for a 2-way bit-sliced 128-entry register file, the additional number of bits required is 128, which is only about 3% additional bits, considering 32-bit registers.

To avoid additional decoding to access the bit-vector of *next read* bits and the upper bit-slices, the decoded information from the decoder used for the lower bit-slice can be used to drive all the bit-slices and the bit-vectors. This will increase the fan-out from the decoder. However, the decoder delay is significantly lower than the delay associated with reading the register file, and the additional decoder delay can be easily absorbed in a pipelined register file access without any impact on the cycle time. The slight increase in the decoder delay can also be compensated by the reduction in the register file access time due to bit-slicing, which we confirmed using a modified version of the cacti tool [16]. A similar approach also works for the data caches.

3 Performance Results

3.1 Experimental Setup

We use the SimpleScalar simulator [3], simulating a 32-bit PISA architecture. However, we modify the simulator so that it has a separate register file, issue queue and rob, instead of a single RUU structure representing all of them. The hardware features and default parameters

[1] In our processor architecture, there is separate integer subsystem and a floating-point subsystem.

that we use are given in Table 1. For benchmarks, we use a collection of 7 SPEC2000 integer programs (`gzip`, `vpr`, `mcf`, `vortex`, `bzip2`, `twolf`, and `gcc`), and 8 SPEC2000 FP benchmarks (`equake`, `applu`, `art`, `mgrid`, `mesa`, `ammp`, `apsi`, and `wupwise`), using *ref* inputs. The statistics are collected for 500M instructions after skipping the first 500M instructions for the SPEC2000 benchmarks. We use a feature size of 0.18 μm for energy and latency measurements.

3.2 IPC Results

Figure 3 shows the IPC results for a 2-way bit-sliced configuration, compared to a non-bit-sliced configuration. Figure 3 shows that the IPC reduction with a 2-way bit-slicing is only about 5%, with a maximum reduction of about 9% for `gzip`. As discussed in Section 2.3, the main reasons for a performance loss with a bit-sliced architecture are (i) delays in the execution of load instructions, (ii) delays in the execution of complex instructions, and (iii) increase in the branch misprediction penalty. To find out the contribution of each of these parameters to the total performance loss, we measure the IPCs of a 2-way bit-sliced configuration with (i) only the loads not delayed (but the complex and branch instructions delayed), (ii) only the branch instructions not delayed, and (iii) only the complex instructions not delayed.

We found that delays in the load instructions has the most significant performance impact. This is expected, because only the mispredicted branches impact performance, and branch misprediction rates are very small for almost all the benchmarks. The percentage of complex instructions is also usually small in the benchmarks, and hence the performance impact of a delay in their execution is not significant. We also found that the increase in IPC as the load instructions are not delayed is relatively more for the integer benchmarks as compared to the FP benchmarks. This is because, in case of FP benchmarks, the loads that load floating-point values still delay the dependent FP instructions, even if the address computation of the loads is not delayed.

We also used the cacti tool [16] to measure the access times of the 2-way bit-sliced register file and the data cache, and found that the access time of the data cache reduces by about 8% when going from a non-bit-sliced data cache to a 2-way bit-sliced data cache. The access time for the register file, on the other hand, reduced by a negligible amount, mainly because for a 32-bit 128-entry register file, the access time is controlled by the delay in driving the bit-lines, which does not reduce.

3.3 Energy Results

We use the cacti tool [16] to perform the energy consumption measurements for the register file and the data cache. In case of a bit-sliced cache, an access to the lower bit-slice (DC1 in Figure 2) accesses both the tag array and the data array simultaneously, and since by the time the higher order bit-slices are accessed, the cache block containing the data is already known, the access to the higher cache bit-slice (DC2 in Figure 2) occurs only to the cache block in which the data is present. Hence, the energy consumed in the higher bit-slices of the data cache is considerably less than that in the lowest bit-slice. Figure 4 shows the percentage savings in the dynamic energy consumption for the register file and the data cache.

Figure 4 shows that there is about 20% energy savings in the register file and about 30% savings in the data cache. From Figures 4 and 1(i) (we consider simple instructions because they form the majority of integer instructions), it can be seen that benchmarks that have a relatively higher percentage of instructions with operands of size greater than 16 bits have a relatively lower energy savings in the register file. This is because, for such benchmarks, the higher order register file bit-slice is also accessed frequently. For instance, consider the benchmarks `applu`, `art`, and `mesa`. The percentage of instructions with larger operands decreases from `applu` to `art` and from `art` to `mesa`. Correspondingly, the register file energy savings increases from `applu` to `art` and from `art` to `mesa`. In integer benchmarks as well, `bzip2` has among the largest percentage of instructions with wider operands, and among the lowest percentage savings in energy consumption. Similar results are observed for the energy savings in the data cache. However, the energy savings in the FP benchmarks was observed to be less than that in the integer benchmarks, because of the wide floating-point values loaded from and stored into the data cache, which will almost always access the higher order bit-slices (as is evident from the high percentage of wide values loaded and stored for FP benchmarks in Figure 1(iv)).

Next, we propose *selective delays*, a technique to recover some of the IPC lost due to bit-slicing.

4 Selective Delays

The main reasons for the reduction in IPC with a bit-sliced architecture include increase in branch misprediction penalty (due to a deeper pipeline) and delay in the issue of load instructions. In this section, we investigate techniques to recover the IPC loss due to these reasons. The basic idea is to prevent the delays from occurring, and the technique is called *selective delays*. Load instructions are delayed because they cannot be issued until the entire address is known. However, the saving grace here is that most of the effective address computations are typically performed on the lower end bits of the address and that the large base address is usually read from the register file (because once the base address is calculated it is repeatedly used with different offsets to load values from the cache). In this scenario, the AGU is designed such that if the operands are being read from the register files, all the bit-sliced register files are read simultaneously, and the effective address is computed. In this case, the load and store instructions do not get delayed. In case the large base address is being bypassed from the functional units or the data cache, then the address generation unit (AGU) waits till the entire operands are available. In this technique, the issue of instructions dependent on load instructions may have to be controlled according to whether the load instruction is reading the operand value from the register file or is receiving the operand value from the forwarding

Parameter	Value	Parameter	Value
Fetch/Commit Width	8 instructions	*Instr. Window Size*	96 Int/ 64 FP
ROB Size	256 instructions	*Frontend Stages*	9
Phy. Register File	96 Int/ 96 FP, 1-cycle acc. lat.	*Int. Functional units*	3 ALU, 1 Mul/Div, 2 AGU
Issue Width	5 Int/ 3 FP	*FP Functional Units*	3 ALU, 1 Mul/Div
Branch Predictor	gshare 1K entries	*BTB Size*	4096 entries, 4-way assoc.
L1 - I-cache	32K, direct-map, 2 cycle latency	*L1 - D-cache*	32K, 4-way assoc., 2 cycle latency 2 read/ write ports
Memory Latency	50 cycles first chunk 2 cycles/inter-chunk	*L2 - cache*	unified 512K, 8-way assoc., 6 cycles

Table 1: Default Parameters for the Non-bit-sliced Configuration

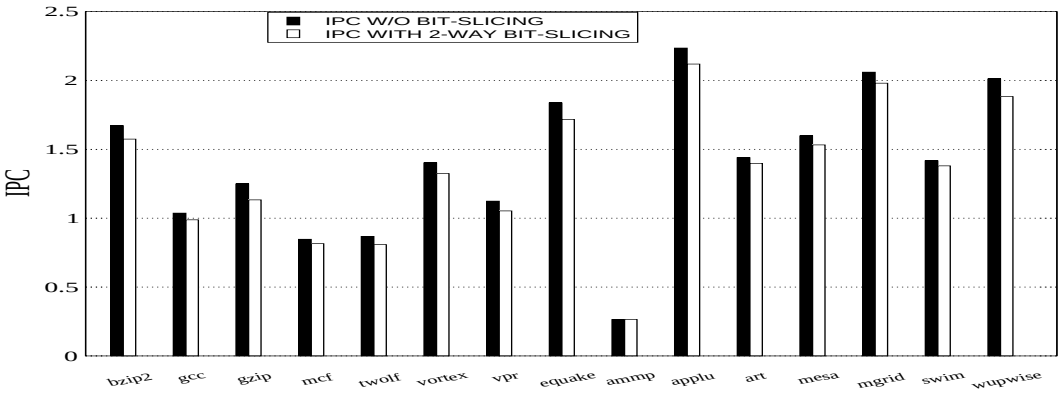

Figure 3: Performance (IPC) of a 2-way Bit-sliced Compared to a Non-bit-sliced Configuration

path.

To limit the increase in the branch misprediction penalty, we observe that the result of most of the branches is known after the computation in the first bit-sliced ALU. For instance, for the *branch if not equal*, if the lower end bits of the operands are different, then we know that that the branch evaluation is true irrespective of what the higher end bits are. Based on this observation, we propose that the results of the branch evaluations in the lower bit-slices of the ALUs be used (in parallel to the evaluations in the higher ALU bit-slices, which are activated if required) to detect branch misprediction and to start the recovery process as early as possible. This can avoid the increase in branch misprediction penalty for many of the mispredicted branches.

Figure 5 shows the IPCs when using the *selective delays* technique discussed in this section, and compares the IPC to that of a non-bit-sliced configuration and that of a bit-sliced configuration without *selective delays*. As can be seen in Figure 5, the IPC improves by about 3% with the *selective delays* technique, when compared to the baseline 2-way bit-sliced architecture.

5 Energy Reduction Techniques

In this section, we investigate techniques to further reduce the processor energy consumption with the help of a bit-sliced architecture.

5.1 Reducing Number of Ports

Limiting the number of register file ports has been proposed earlier by Tseng [18], in which the authors partition the register file into multiple banks where each bank contains certain registers and the number of ports in each bank is reduced. However, here we reduce the number of ports to the higher order register file bit-slices, because when bit-slicing the register file (RF) and the data cache (DC), the higher order bit-slices are not used as frequently as the lower order bit-slices. We propose that the number of read ports into the higher order RF bit-slice be reduced by half, while keeping the write ports intact for simplicity of the design. This results in a higher order FU bit-slice having only 1 read port into the higher order RF bit-slice. A single read port into the higher order RF bit-slice also works well for address generation units because they typically read only 1 32-bit operand (the base address) from the register file. If any FU requires to read two operands from the higher order RF bit-slice, the instruction that requires 2 operands from the higher order RF bit-slice, reads one operand in one cycle and then reads the other operand in the next cycle, using the same port. In this technique, the dependent instructions that may get issued in the immediately next cycle following the producer instruction, will also have to be stalled for one cycle. For this, we add another bit (called *phys-delayed*) to the *next read* bit-vector, that indicates whether the production of the higher bit-slice of a physical register will be delayed by a cycle or

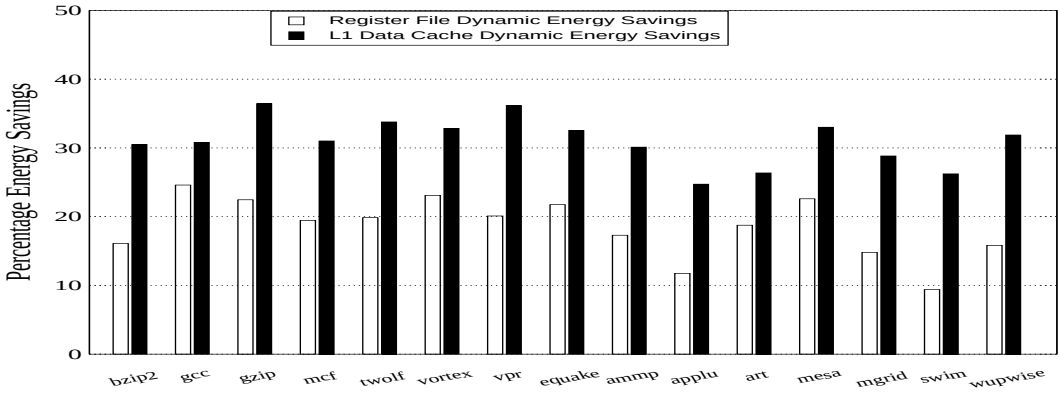

Figure 4: Percentage Dynamic Energy Consumption Savings in Register File and Data Cache wrt Non-bit-sliced Configuration

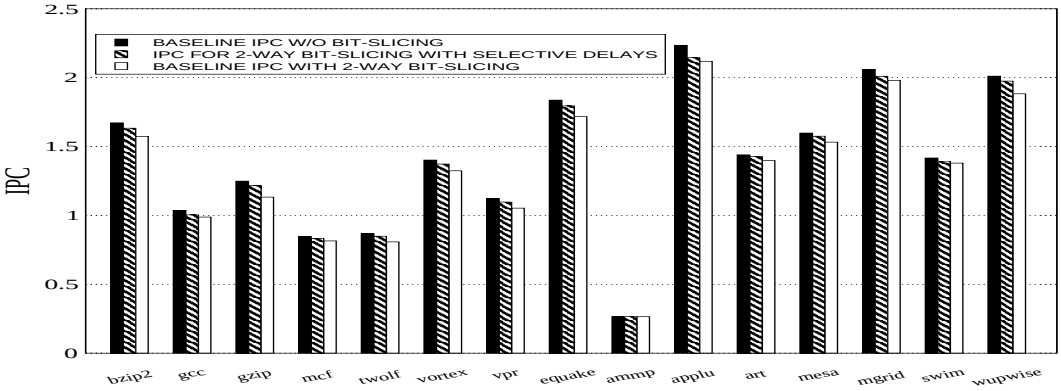

Figure 5: Performance (IPC) Using *Selective Delays* Technique

not. When an instruction issues, in parallel to reading the lower order RF bit-slice, it reads the *next read* bits and the *phys-delayed* bits for all the operands. If the *next read* bits of both of its operands is 1 or the *phys-delayed* bit of any of its operands is 1, it sets the *phys-delayed* bit for its destination and starts executing on the lower bit-slices of its operands (which are always available), and then it stalls for 1 cycle before continuing the execution on the higher order bits. The *phys-delayed* bit for each register needs to be *reset* 1 cycle after it has been *set*, to indicate that the ensuing dependent instructions need not wait for the higher order bits of their operands. Note that, if an instruction stalls in any bit-slice of an ALU, then no instructions are issued to that particular ALU, to avoid overwriting the stalled instruction with the new instruction. For the higher order data cache bit-slice, on the other hand, instead of having 1 read and 1 write port, we have only 1 read/write port. In this case, if both a load and a store need to access the higher order DC bit-slice, then the store is stalled and load is executed. Results are discussed in Section 5.3.

5.2 Reducing Leakage Energy Consumption

With reducing feature sizes, leakage energy consumption is becoming a significant fraction of the total energy consumption in the processor, and techniques have to be investigated to reduce the leakage energy consumption. To reduce leakage energy consumption, we investigate "shutting off" (by using *gated-Vdd* [7]) the higher bit-slices of the storage elements when they do not store bits significant to the representation of the value. For instance, for a 128-entry 32-bit 2-way bit-sliced register file, the top half of 64 registers could be shut down and during that time, the processor will have 64 32-bit registers and 64 16-bit registers. In this case, the rename logic can use *size prediction* (as discussed in [12]) to rename the instructions to appropriate registers. This technique could be applied to both the bit-sliced register file and the data caches. In this section, we only study techniques to reduce the leakage energy consumption in the register file. We explain the technique only for a 2-way bit-sliced register file with each register of 32 bits, however, it can be easily extended for further bit-sliced register files.

For a 2-way bit-sliced register file, two separate free register lists are maintained, one for the registers that have their top half "shut off" (16-bit registers) and the other for the "whole" (32-bit) registers. The size of the result produced by any result-producing instruction is predicted, and based on the prediction, an appropriate register is allocated to the instruction at rename-time. For size predictions, a *last value predictor* has been shown to work very accurately [12]. When using a *last value predictor*, the size prediction for an instruction can also be stored as an additional bit along with the instruction in the instruction cache, avoiding the use of additional tables. If no prediction can be made for an instruction, it is pre-

dicted to produce a result of size 32 bits. If a free register appropriate for the predicted size of the result is not available, then the other list is checked for free registers, and the registers are accordingly "turned on" and "turned off". "Turning on" bit-slices of registers may even take multiple cycles, and register renaming is stalled for those cycles. At the time of write-back, the size of the result is checked, and the registers are again "turned on" and "turned off" accordingly, which may again stall the pipeline for some cycles. The number of cycles the pipeline stalls depends on the number of cycles required to "turn on" the registers. As discussed earlier, if the prediction of an instruction is *more than 16 bits*, but it gets allocated a 16-bit register, then the "turn on" for the register is started at the register allocation time itself. However, by the time the instruction reaches the writeback stage, if the register is not yet completely on, the pipeline stalls for the remaining number of cycles. The stalling of the pipeline when "turning on" the registers at writeback time can have much more performance impact. The leakage energy consumption in the register file is saved by "turning off" the bits that do not store significant data. As discussed in [7], *gated-Vdd* SRAM cells (for turning off the bits) have a negligible effect on the area and the access time of the cells. Results are discussed in the next section.

5.3 Results

First, we measured the *size prediction accuracy*, and found that it hovers around the 90% mark for almost all the benchmarks. We also measured the distribution of cycles in terms of the number of 16-bit registers present in those cycles. We found that there are a considerable number of cycles that have more than 32 16-bit registers (out of a total of 96 registers). Some benchmarks (such as `mesa` and `mgrid`) even showed a considerable number of cycles with more than 64 16-bit registers.

Figure 6 shows the percentage savings in dynamic energy consumption for the register file and the data cache (with and without reduced ports) and the percentage savings in the register file leakage energy consumption. Figure 6 shows that about 5% additional energy savings are obtained in the register file when the number of read ports are reduced by half in the higher order RF bit-slice. The corresponding number for the data cache (when reducing 1 read/ 1 write port to 1 read/write port) is about 2%. The additional savings in the data cache is lower than that in the register file, because the higher order DC bit-slice as it is consumes considerably less energy than the lower order DC bit-slice (only the required cache block is accessed in the higher order bit-slice), and reduction of ports does not give significant additional energy savings. Figure 6 also shows about 20% savings in the register file leakage energy consumption. To measure the leakage energy savings, we measure the average number of register bits that are "shut off", and multiply it to the leakage energy consumption for each bit (estimated by means of Hotleakage [20]).

Figure 7 shows the IPC values with a 2-way bit-sliced configuration using reduced number of ports, and using the leakage reduction technique of "shutting off" higher RF bit-slices (with 3 different "turn on" cycle requirements of 1, 5, and 10 cycles), compared against the non-bit-sliced configuration and bit-sliced configuration with all the ports. Figure 7 shows that, when reducing the number of ports in the higher order RF and DC bit-slices (only read ports are halved for RF), the reduction in IPC is only about 1%, compared to the baseline 2-way bit-sliced configuration with all the ports. When "shutting off" the upper bit-slices of the registers, the IPC depends on the number of cycles taken to reactivate the "shut down" registers. We assume that it takes a single cycle to shut down a register[2]. Figure 7 shows only about 3% reduction in IPC, compared to the 2-way bit-sliced configuration, when the register activation time is 1 cycle. However, for activation time of 5 cycles, the reduction is about 7% and for activation time of 10 cycles, the reduction is about 15%.

6 Conclusions

Power and performance have become two very important design criteria in the design of microprocessors. However, efforts to improve either power or performance usually leads to a degradation of the other. One important approach reduce power consumption while not hurting performance is to prevent the processor from performing unnecessary work. The techniques presented in this paper are in the same spirit, where the execution core has been bit-sliced to avoid unnecessary work. Bit-slicing uses the property that a significant amount of data in the processor is of small-size. Bit-slicing has been proposed before, but never to the extent to which we bit-slice the execution core (which includes the functional units, the register file, the upper level data cache, and the data forwarding paths). In our bit-sliced execution core, each bit-slice operates on different bits of data, and the higher order bit-slices are activated only when they are required, thus reducing energy consumption. Our studies show that, on a 32-bit machine, a 2-way bit-sliced execution core reduces the energy consumption of key hardware resources such as the register file and the data cache by about 20% and 30%, respectively, whereas the instruction throughput, with the help of performance improving techniques to prevent the instructions from getting delayed, reduces by only about 2%.

Bit-slicing can also facilitates further reduction in the processor energy consumption. We use bit-slicing to reduce the leakage energy consumption in the register file. With reducing feature sizes, leakage energy consumption is a growing concern in the design of microprocessors. With a bit-sliced register file, we propose "shutting off" the higher bit-slices of the registers storing small-sized values, thus reducing the leakage energy consumption in them. Our studies showed that an average of about 40 registers (out of a total of 96) have their higher order bit-slices "shut off" every cycle, and that this technique reduces the leakage en-

[2]Note that only a few registers can be shut down and activated in each cycle, thus reducing the inductive noise that can result from mass shut downs and activations.

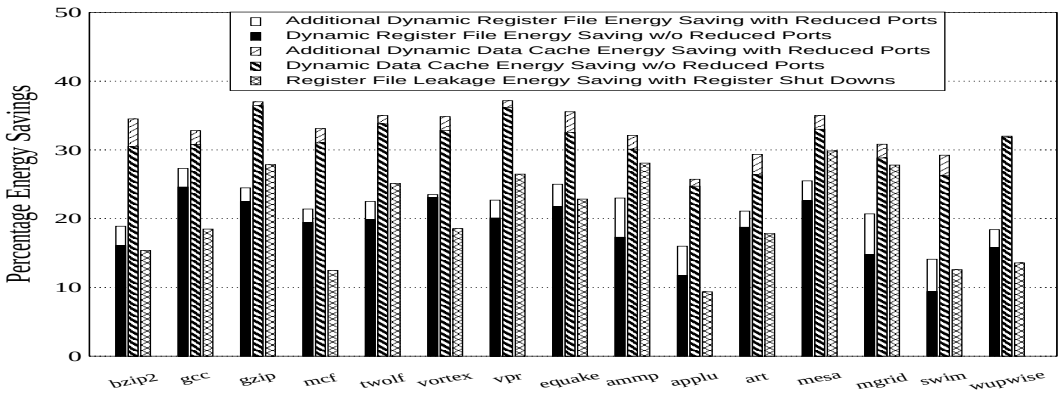

Figure 6: Percentage Dynamic Register File and Data Cache Energy Saving (with and without Reduced Ports), and Percentage Register File Leakage Energy Saving

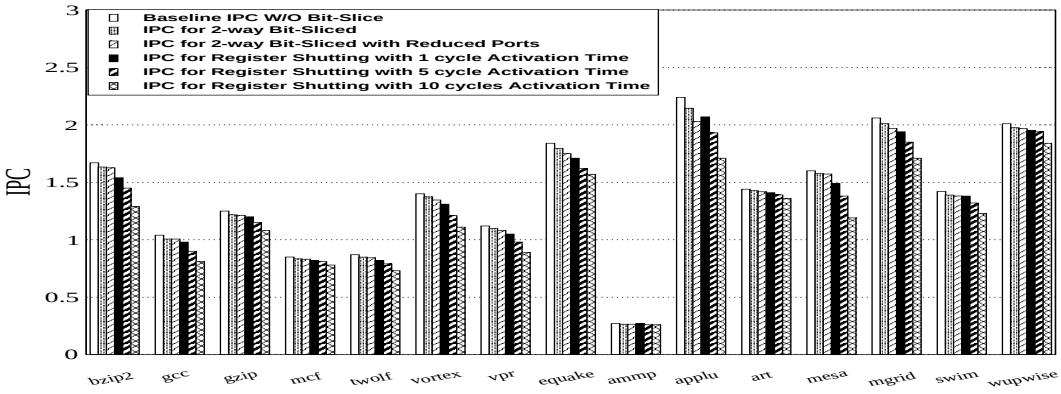

Figure 7: Performance (IPC) of a 2-way Bit-sliced Configuration With Selective Delays; With Reduced Ports; Without Reduced Ports; and With Register Shutting

ergy consumption in the register file by about 20%. We also propose reducing the number of ports in the higher order bit-slices to further reduce the energy consumption in the processor execution core.

References

[1] A. Aggarwal and M. Franklin, "Energy Efficient Asymmetrically Ported Register File," *Proc. ICCD*, 2003.

[2] D. Brooks and M. Martonosi, "Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance," *Proc. HPCA*, 1999.

[3] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," *Computer Arch. News*, June 1997.

[4] R. Canal, A. Gonzalez and J. E. Smith, "Very Low Power Pipelines using Significance Compression," *Proc. Micro*, 2000.

[5] R. Canal, A. Gonzalez and J. E. Smith, "Software-Controlled Operand-Gating," *Proc. International Symposium on Code Generation and Optimization*, 2004.

[6] Y. Cao, and H. Yasuura, "Low-Energy Design using Datapath Width Optimization for Embedded Processor-based Systems," *IPSJ Journal*, 43(5):1348-1356, May 2002.

[7] M. Powell, et. al., "Gated-Vdd: A circuit technique to reduce leakage in deep-submicron cache memories," *Proc. of ISLPED*, 2000.

[8] M. R. Guthaus, et. al., "MiBench: A Free Commercially Representative Embedded Benchmark Suite," *Proc. IEEE International Workshop on Workload Characterization*, 2001.

[9] M. K. Gowan, et. al., "Power Considerations in the Design of the Alpha 21264 Microprocessor," *Proc. DAC*, 1998.

[10] G. Hinton, et al, "A 0.18-um CMOS IA-32 Processor With a 4-GHz Integer Execution Unit," *IEEE Journal of Solid-State Circuits*, Vol. 36, No. 11, Nov. 2001.

[11] S. Larsen, and S. Amarasinghe, " Exploiting Superword Level Parallelism with Multimedia Instruction Sets," *Proc. PLDI*, 2000.

[12] G. Loh, "Exploiting data-width locality to increase superscalar execution bandwidth," *Proc. Micro-35*, 2002.

[13] S. Mahlke et. al., "Bitwidth Cognizant Architecture Synthesis of Custom Hardware Accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(11), Nov. 2001.

[14] G. Pokam, S. Bihan, J. Simonnet, and F. Bodin, "SWARP: A Retargetable Preprocessor for Multi-

media Instructions," *Concurrency and Computation: Practice and Experience*, 16(2-3):303-318, Feb. 2004.

[15] G. Pokam et. al., "Speculative Software Management of Datapath-width for Energy Optimization," *Proc. LCTES*, 2004.

[16] P. Shivakumar, and N. Jouppi, "CACTI 3.0: An Integrated Cache Timing Power, and Area Model," *Technical Report, DEC Western Research Lab*, 2002.

[17] M. Stepehenson et. al., "Bitwidth Analysis with Application to Silicon Compilation," *Proc. PLDI*, 2000.

[18] J. Tseng, and K. Asanovic, "Banked Multiported Register Files for High-Frequency Superscalar Microprocessors," *Proc. ISCA-30*, 2003.

[19] Luis Villa , Michael Zhang , and Krste Asanovic, "Dynamic zero compression for cache energy reduction," *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture,* p.214-220, December 2000.

[20] Y. Zhang, et. al., "Hotleakage: A Temperature-aware Model of Subthreshold and Gate Leakage for Architects," *Technical Report CS-2003-05*, University of Virginia, Department of CS, 2003.