Optimizing Static Power Dissipation by Functional Units in Superscalar Processors *

Siddharth Rele[1], Santosh Pande[2], Soner Onder[3], and Rajiv Gupta[4]

[1] Dept of ECECS, University of Cincinnati, Cincinnati, OH-45219
[2] College of Computing, Georgia Tech, Atlanta, GA-30318
[3] Dept. of Computer Science, Michigan Tech. Univ., Houghton, MI 49931
[4] Dept. of Computer Science, The Univ.of Arizona, Tucson, Arizona 85721

Abstract. We present a novel approach which combines compiler, instruction set, and microarchitecture support to turn `off` functional units that are idle for long periods of time for reducing static power dissipation by idle functional units using power gating [2, 9]. The compiler identifies program regions in which functional units are expected to be idle and communicates this information to the hardware by issuing directives for turning units `off` at entry points of idle regions and directives for turning them back `on` at exits from such regions. The microarchitecture is designed to treat the compiler directives as hints ignoring a pair of `off` and `on` directives if they are too close together. The results of experiments show that some of the functional units can be kept `off` for over 90% of the time at the cost of minimal performance degradation of under 1%.

1 Introduction

To cater to the demands for high performance by a variety of applications, faster and more powerful processors are being produced. With increased performance there is also an increase in the power dissipated by the processors. High performance superscalar processors achieve their performance by exploiting instruction level parallelism (ILP). ILP is detected dynamically and instructions are executed in parallel on multiple functional units. Therefore one source of power dissipation is due to the functional units. It is well known that ILP is often distributed nonuniformly throughout a program. As a result many of the functional units are idle for prolonged periods of time during program execution and therefore the power dissipation by them during these periods is wasted.

The goal of this work is to minimize power dissipated by functional units by exploiting long periods of time over which some functional units are idle. The power consumed by functional units falls into two categories: dynamic and static. With current technology the dynamic power is the dominating component of overall power consumption and by using *clock gating* techniques the dynamic power dissipated by functional units during idle periods can be reduced [4, 12, 13]. However, it is projected that in a few generations the static power dissipation will equal dynamic power dissipation [11]. Specifically for different kinds of

* Supported by DARPA award no. F29601-00-1-0183.

adders and multiplers, the increase in static power with changing technology is shown in Table 1 [5]. Therefore it is important to also minimize the static power consumption when the functional units are idle.

Table 1. Static power dissipation by functional units.

Functional unit type	Technology (μm)				
	0.354	0.18	0.13	0.10	0.07
Adders	Static power dissipation (mW)				
Ripple Carry	0.07	0.08	0.12	0.14	0.15
Carry Lookahead	0.09	0.11	0.19	0.20	0.19
Manchester Carry	0.10	0.16	0.23	0.25	0.28
Multipliers	Static power dissipation (mW)				
Serial	0.29	0.32	0.43	0.51	0.50
Serial/Parallel	0.35	0.41	0.48	0.55	0.58
Parallel	0.37	0.46	0.50	0.60	0.62

There are two known techniques that are suitable for reducing static power dissipation by functional units during long periods of idleness. The first technique is *power gating* [10, 9] which turns off devices by cutting off their supply voltage. The second technique uses the *dual threshold voltage* technology – by raising the threshold voltage during idle periods of time the static power dissipation is reduced [14]. In both of the above approaches there is a *turning on latency* involved, that is, when the unit is turned back on (either by providing the supply voltage or lowering the threshold voltage) it cannot be used immediately because some time is needed before the circuitry returns to its normal operating condition. While the latency for power gating is typically few (5-10) cycles [2], the latency for dual threshold voltage technology is much higher. In this work we assume that power gating is being employed to turn off functional units and we assume a latency of ten cycles for turning a functional unit on in all our experiments.

The shutting down of functional units is most effectively accomplished by employing a combination of compiler and hardware techniques. To understand the reasons for this claim lets examine the problems that we must address in designing an algorithm for turning functional units off and then back on, and then evaluate the suitability of the means for solving the problem, that is, whether to use compiler support or hardware support in addressing the problem. We also describe the approach that we take in addressing each of the problems.

Identifying idle regions. In order to turn off a functional unit we first must identify regions of code in the program over which the functional unit is expected to be idle. The use of hardware for predicting or detecting idle regions has the following problems. First the additional hardware for predicting idle regions will also consume additional power throughout the execution as it must remain active all along. Second we will not be able to exploit the idle regions during the warm up period of the prediction mechanism – only after enough history has been acquired by the prediction hardware will the predictions be effective.

Our solution to the above problems is to rely on the compiler to identify program regions with low ILP and thus low functional unit demands. The compiler can examine all of the code off-line and therefore identify suitable regions for turning the functional units off. Furthermore it can also identify the type of functional units and determine the number of functional units that should be turned off without degrading performance. This information is then communicated to the hardware by generating special `off` and `on` directives.

Tolerating the latency of turning a functional unit off. The functional unit must be turned off sufficiently prior to entering the program region in which it can be kept idle. This is because there is a latency for turning the unit off and we must account for this latency to *maximize the power savings*. The latency arises because time is needed to drain the functional unit by allowing it to execute the instructions already assigned to it. Let us assume that we have two functional units of a given type and we would like to turn one of them off. When the `off` directive is encountered, the functional units may already have instructions assigned to them. One of the unit must be selected and drained before it is turned off.

This problem is also not suitable for handling by hardware because even if we were to overcome the problems described earlier and develop a mechanism for efficiently detecting idle regions in hardware, now we would have to predict them even earlier. Therefore our solution is to allow the compiler to place the `off` directive sufficiently in advance of reaching the idle region whenever possible.

Tolerating the latency of turning a function unit on. The functional unit must also be turned on prior to exiting the idle region. This is because there is a several cycle latency before which the functional unit comes on-line and is ready to execute operations [2]. By tolerating this latency we can *minimize the performance degradation* while executing instructions from the region following the idle region. Again our solution to this problem is to place the `on` directive sufficiently in advance of exiting the idle region whenever possible.

Dealing with variable length idle regions. Sometimes the duration of an idle region may vary from being very small in one execution of the region to very long in the next execution of the same region. For example, the idle region may contain a while loop or conditionals which may lead to this variation. Introduction of an `off` directive in such a situation can be based upon a conservative policy or an aggressive policy. A compiler based upon a conservative policy will introduce the `off` and `on` directives only if it is certain that the duration of the idle region is long. The problem with this approach is that the reductions in power dissipation that could be obtained by turning a unit off are sacrificed.

We propose to use an aggressive policy in which the compiler introduces the `off` and `on` directives to maximize savings. If the duration of the idle region is long, power savings result. On the other hand if the duration is very small, the `on` directive is issued on the heals of issuing an `off` directive. If the latter situation arises frequently, while little or no savings in power result, some amount of dynamic power is dissipated during switching of the functional unit state. Moreover the performance is hurt as the functional unit goes off-line for several cycles each time such a spurious pair of `off` and `on` directives are encountered. We address this issue by providing adequate microarchitecture support for nullifying spurious `off` and `on` pairs. The microarchitecture is designed to treat the compiler directives as hints ignoring a pair of `off` and `on` directives if they are too close together. In this way the state of the unit is not actually switched, the unit stays on-line, and dynamic power for switching the unit off and on as well as the degradation in performance are minimized.

We have incorporated the power-aware instructions into the MIPS-I instruction set and simulated a superscalar architecture which implements these instructions using our FAST simulation system [8]. The compiler algorithms have been incorporated into the *lcc* compiler. The results of experiments show that some of the functional units can be kept `off` for over 90% of the time resulting in a corresponding reduction in static power dissipation by these units. Moreover the power reductions are achieved at the cost of very minimal performance degradation – well under 1% in all cases.

The remainder of the paper is organized as follows. In section 2 we discuss instruction set extensions and microarchitecture modifications required to implement the new instructions. In section 3 we discuss in detail the compiler algorithms for introducing `on` and `off` instructions. In section 4 we describe our implementation and in section 5 we present results of experiments. Conclusions are given in section 6.

2 Architectural Support

Power aware instruction set. As mentioned earlier, we support instructions that will allow us to turn functional units on or off. Such instructions must also indicate the type of functional unit that is to be turned on or off. The solution we developed adds an `on` or an `off` directive as a suffix to existing instructions. The type of functional unit that is to be turned on or off is the same type as that is used to execute the instruction to which the directive is added. In case multiple functional units of a particular type are present, the decision as to which specific unit will be turned off is left up to the hardware. In some architectures certain operations can be executed by functional units of more than one type (e.g., integer and floating point). However, we assume that in such cases the `off` and `on` directives are attached to instructions that must execute on a functional unit of specific kind.

We have incorporated the `on` and `off` directives to the MIPS-I Instruction Set Architecture (ISA) which supports MIPS 32 bit processor cores. This ISA was selected for its simplicity and the availability of encoding space to allow us to encode `on` and `off` into existing instructions. A subset of instructions we modified is shown in Fig. 1. These instructions can also be issued without any operands in which case they do not perform any operation except for switching

`add.on`	switch ON one **integer adder**
`add.off`	switch OFF one **integer adder**
`mul.on`	switch ON one **integer multiplier unit**
`mul.off`	switch OFF one **integer multiplier unit**
`add.s.on`	switch ON one **float adder**
`add.s.off`	switch OFF one **float adder**
`mul.s.on`	switch ON one **float multiplier unit**
`mul.s.off`	switch OFF one **float multiplier unit**
`mov.s.on`	move values between float regs and switch ON **float unit**
`mov.s.off`	move values between float regs and switch OFF**float unit**

Fig. 1. A subset of energy-aware instructions.

a unit of the appropriate type `on` or `off`. These are needed when `on` or `off` directives cannot be added to an existing instruction because the code does not already contain an instruction of the appropriate type around the point at which the compiler chooses to place the directive.

`On` and `off` semantics for an out-of-order superscalar processor. The `on` directive is acted upon immediately following its detection, that is, when the instruction with the `on` suffix has been decoded, a functional unit of the appropriate type is turned `on`. It takes a few cycles for the circuitry to reach normal operational state after which the unit can perform useful work. The turning `off` of a functional unit cannot be done immediately following the decode. This is because if the unit that is turned off was the last on unit of its type, then no functional unit will be available for executing the instruction carrying the suffix and the processor will deadlock. Therefore in this case, following the decode, an on unit is selected and marked as `pending-off`. When the instruction that marks the unit retires, the unit is actually turned off and its status is changed from pending-off to off. This approach works because it guarantees that all instructions requiring the unit would have executed before the unit is turned off as all instructions are retired in-order even though they may execute on the functional unit out-of-order in the superscalar processor. At the same time, introduction of an `off` directive does not constrain the out-of-order execution capability of the processor. The states of the functional units are maintained as part of the processor state. A *status table* is maintained that indicates for each functional unit whether it is currently turned on, currently turned off, or if it is in the pending-off state. No new instructions are assigned to a functional unit by the issue mechanism if the unit is in `off` or `pending-off` state.

Nullifying spurious `off-on` pairs. While savings in static energy consumption result when a functional unit is shutdown, a certain amount of performance loss may be incurred when a unit is turned off as well as a certain amount of dynamic power is expended in bringing the circuit to its normal operating state. We rely upon the compiler to identify suitable idle regions during which turning off of a functional unit is not expected to hurt performance and the dynamic power expended in turning the unit on is far smaller than the static power saved by turning it off. For this strategy to work well, it is important that the idle regions be long in duration. However, it is possible that the code representing the idle region varies greatly in duration from its one execution to another. For example, the idle region may be formed by a while loop. If very little time is needed to execute the idle region then the unit will be turned off and then immediately turned on. In this situation the savings in static power will be minimal. However, loss of performance will still be incurred while executing the code immediately following the idle region and dynamic power will still be expended in turning the unit on.

Our implementation of on and off is so designed that we are able to dynamically *nullify* spuriuous off and on pairs and thus avoid the dynamic power that would otherwise be dissipated during the transitions. When an instruction with `off` directive is encountered, a unit is selected and marked as `pending-off`. If an instruction with the `on` directive is encountered while the status of the unit is still `pending-off`, the unit state is changed to `on` from `pending-off`. When the instruction associated with the `off` directive retires, it will examine the status of the functional unit that it marked as `pending-off`. If the status is still `pending-off`, the unit is turned off; otherwise it is left on. Thus, the overall impact of the above approach is that if the `on` directive is encountered while the functional unit is in `pending-off` state, the functional unit is not actually turned off. Thus the `off-on` pair does not turn the unit off and then back on.

```
1 :  ....
2 :  mul.off − turn unit off
3 :  if (x > 0) {
4 :      wait = 0;
5 :      while(1) {
6 :          wait = wait++;
7 :          if (wait == 1000) break;
8 :      }
9 :  mul.on − turn unit on
10 : for (i = 0 ; i < 100; i++)
11 :         sum += a[i] * 10;
12 : ....
```

Fig. 2. Nullification of OFF and ON pair.

For the example in Fig. 2, the code from line 3 to 8 takes very short time to execute when $x \leq 0$; otherwise it takes a long time to execute. During the execution of this code we would like to turn the multiplier off since it is not required. If $x > 0$ we get power savings by turning the unit off. However, if $x \leq 0$, the off and on directives are encountered in rapid succession and the unit is not turned off and then immediately turned back on. Before the instruction with the `off` directive retires, we would have already decoded the instruction with the `on` directive and changed the status of the unit from `pending-off` to on. Therefore when the instruction with off directive retires, it will find the functional unit status as on and therefore it will not turn it off. As a result the spurious off-on pair will be nullified.

3 Compiler Support

Our approach. Our compiler is designed to introduce off and on suffixed instructions in such a way that the following two goals are met. First we need to remove idleness by turning functional units off without causing an increase in program execution time (i.e., we want to reduce static power dissipation without causing performance degradation). Second the functional units that are turned off should be off for prolonged periods of time so that the dynamic power dissipated during on-off and off-on transitions is small in comparison to static power saved by keep the units off. Both the above goals are met by careful placement of on and off suffixed instructions.

In order to achieve the first goal of minimizing performance degradation we take the following approach. We classify the basic blocks in a program into two categories: *hot blocks* whose execution frequencies are greater than a certain threshold value and *cold blocks* which are all the remaining blocks in the program. We also analyze the functional unit usage in each block to identify its *requirements* and consequently identify the units that are expected to be idle in that block. We place the off and on directives in cold blocks bordering the hot blocks in which the unit is expected to be idle. This situation is illustrated by the example in Fig. 3a. In contrast the example in Fig. 3b illustrates a situation in which we forego the removal of idleness since the block neighboring the hot block in which unit is idle is another hot block where the unit is not idle. This is because the potential placement points for off and on directives are also hot and therefore such instructions will be executed with high frequency. Thus, our approach removes idleness only if such removal does not adversely effect performance.

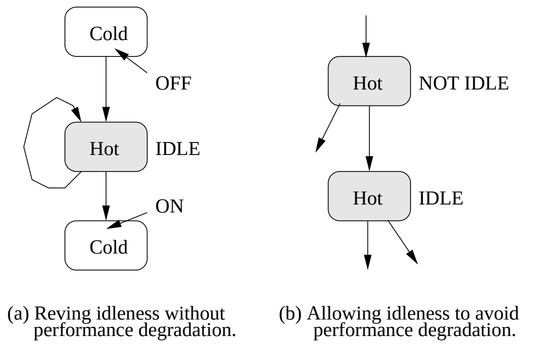


Fig. 3. Idleness removal strategy.

In order to achieve the second goal of maximizing power savings mentioned above we do not place instructions carrying off and on directives at boundaries of a region formed by a single basic block. Instead we identify larger subgraphs in the control flow graph that represent control constructs (e.g., loops) which we refer to as *power blocks*. Then we classify the power blocks as hot or cold. In addition, from the requirements of individual blocks in a power block, we identify which functional units are idle throughout the execution of the power block. When power-aware code is generated, the `off` and `on` directives are placed at boundaries of power blocks using the principles described earlier and illustrated in Fig. 3.

We have given an overview of our approach. Now we describe the three main steps of our algorithm in more detail. The first step involves construction of an *power-aware flow graph*. The second step identifies the *power blocks*. The third and final step introduces the `off` and `on` suffixed instructions.

The power-aware flow graph (PAFG). Our compiler begins by building the PAFG which is a control flow graph whose basic blocks are annotated with two types of information: the resource requirements; and the execution counts.

The **requirements** of each block is calculated by first identifying the number of operations requiring each functional unit type in the block. This information by itself is enough for those functional unit types where only one functional unit of that type is present. If an operation requiring the functional unit of a certain type is present, the unit of that type is required. However, the above method is inadequate if there are multiple functional units of a given type. We must access the level of instruction level parallelism present in the operations that use the functional unit type to compute the requirements. The dependences among statements are examined to identify the parallelism and accordingly the requirements are computed. In particular, if two instructions that can execute in parallel require the same type of functional unit, then two such units are required. In other words the requirements of a basic block are computed such that they represent the number and type of units required to exploit the ILP present in the block. Another issue that must be considered during computation of requirements is that many instructions other than the integer *add* instruction may use the integer adder. For example, *base + offset* computation to compute the address of an array element requires an integer adder.

The **profile** information that annotates the basic blocks is derived from prior executions of the program. This information is used for identifying *hot blocks*. If the execution count for a particular block is more than a *threshold*, it is considered to be *hot*. The threshold value is set according to the formula given below. In this formula N is a tunable parameter that can be changed to generate higher or lower number of hot blocks and thus control how aggressively idleness is removed.

$$\mathsf{Threshold} = \frac{\mathsf{Execution\ Count\ of\ Most\ Frequently\ Executed\ Block}}{\mathsf{Some\ constant\ value\ } \mathbf{N}}.$$

An example code segment and its power-aware flow graph are shown in Fig. 4a and 4b. The requirements are annotated as a vector of values enclosed in angular brackets (the first value corresponds to integer adders and second for integer multipliers) while the profiling information is annotated as the execution count enclosed within square brackets. We set the threshold value as $MaxValue/10$ for identifying hot blocks.

Identifying power blocks. In order to identify longer periods of time over which a functional unit can be turned off, we identify subgraphs representing larger constructs such as loops, if-statements, and switch statements. These subgraphs are referred to as *power blocks*. A hierarchical graph at the power block level is created in which each power block indicates the *start* and the *end* nodes of the subgraph forming the power block. In addition, a power block holds the summary of all the information regarding the basic blocks that form the power block. The requirements of a power block are computed from the requirements of the hot blocks in the block. The reason for this will be clear when we discuss how `off` and `on` directives are generated. There is only one entry point into a *power block*, that is, the *start* node of the power block dominates all the blocks inside the power block and hence the control has to flow through that block. Therefore if the *start* node is hot, the whole power block is marked as hot even though all the basic blocks belonging to it may not be hot.

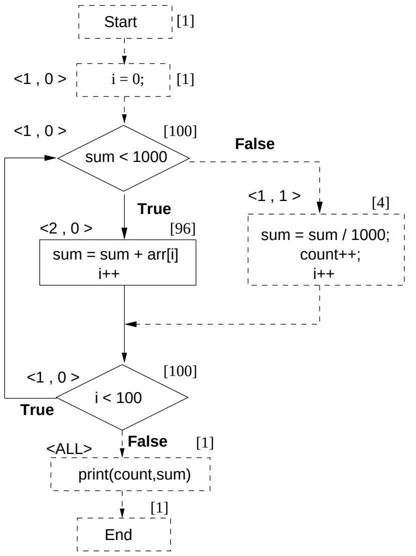
The higher level tree constructed from power blocks for our example is shown in Fig. 4c. Each leaf in this tree is a basic block. Internal nodes corresponding to higher level control constructs are the power blocks.

Inserting power-aware instructions. Once all the information regarding the requirements of each basic as well as power block is recorded in the respective blocks, we traverse the PAFG for code generation. Our basic approach for introducing the `off` and `on` instructions is as follows:

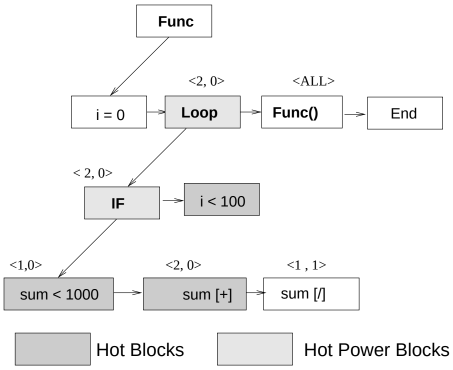
- For each *user function* we start by turning all units, except a minimal configuration of units, off. The minimal configuration is required so that execution can proceed and the processor does not deadlock. Typically this configuration will include an integer adder.
- For each call to a *library function* we assume that all units are on during the execution of the library function. This is because we do not analyze code for library functions and therefore in order to guarantee that no performance degradation occurs, we must keep all units on. Instructions to turn on units that are off are therefore introduced immediately prior to the call and upon return these units can be again turned off. The impact of this restriction can be reduced by performing our optimizations at link time.
- If a particular user function is called in a hot block such that the number of calls to the function exceed the `threshold`, then the current framework bypasses the analysis of that function, on the grounds that any switching inside this function would be too frequent and hence not beneficial (it may in fact jeopardize the execution speed).
- We compare each block with all its successors to check if there is a difference in the power requirements of the blocks. If there is a difference, then we try to generate off and on instructions at the boundaries after checking whether

```
void main() {
  for (i = 0 ; i < 100 ; i++)
    if(sum < 1000)
      sum = sum + arr[i];
    else {
      sum = sum / 1000;
      count++;
    }
  print(count,sum);
}
```

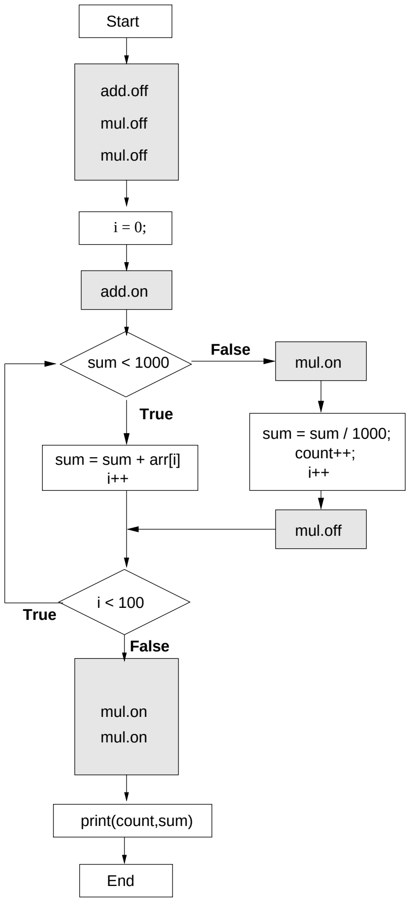
(a) Sample code segment.



(b) Power-aware flow graph.



(c) Hierarchical tree with power blocks.



(d) Final code.

Fig. 4. Introducing directives.

the blocks involved are hot or cold according to the strategy outlined earlier in this section. When a hot power block is adjacent to cold blocks, typically off instructions are generated prior to entering the power block. From the requirements of the power block we identify the units to be turned on or off. Recall that the requirements of the power block are computed from hot blocks in it. Therefore within the hot power block there may be cold blocks which require a unit that is currently off. Therefore, upon entry to such a cold block such a unit is turned on and upon exit it is turned off again. Notice that all instructions being introduced are being placed in cold blocks.

The code generated for our example is given in Fig. 4d. We assume that we have 2 integer adders and 2 integer multipliers (floating point units are omitted because we assume all operations in the code are integer operations). Note that at the beginning we turn all functional units off except the integer adder which represents the minimal configuration for this example. The loop represents a hot power block and the block preceding the loop is a cold block. Therefore we introduce instructions according to the requirements of the power block prior to entering it. Since the hot basic block in the loop containing the statements "$sum = sum + arr[i]$" and "$i{+}{+}$" requires two adders to exploit ILP, we turn on an additional adder before entering the loop. Notice that the multiplier (which we assume also performs the divide operation) is off in the loop. Therefore if we enter the cold block containing the statement "$sum = sum/1000$", a multiplier is turned on and upon exit it is turned off. Finally, prior to executing the library function call for `printf` all off units are turned on – since at this point adders are already on, only the multipliers need to be turned on.

4 Experimental Results

Implementation. We have implemented and evaluated the techniques described in this paper. We used the *lcc* [3] compiler for our work. *lburg* was used to produce code generator from compact specifications. The original code was executed on test data to generate profile information which is used by the compiler to generate `on` and `off` instructions. We use a cycle level simulator generated using the *FAST* [8] system. *FAST* generates a cycle level simulator, an assembler and a disassembler from a microarchitecture and instruction set specifications.

In our experiments we simulated a superscalar that supported out-of-order execution and consisted of 2 integer adders, 2 integer multipliers, 1 floating point adder, and 1 floating point multiplier. It uses control speculation (i.e. branch prediction) and implements a precise exception model using a reorder buffer and a future file. The number of outstanding branches is not limited and branch mispredictions take a variable number of cycles to recover.

We used six benchmarks in our experiments. From Mediabench [6] we have used two programs: rawcaudio.c and rawdaudio.c. From DSPstones we have taken three programs: fir2dim.c, n-real-updates.c, and fir.c. The last benchmark, compress.c, is from SPEC95.

Removing idle time. To access the effectiveness of our idle time removal technique we measured the *utilization* of functional units before and after optimization. We define utilization as the percentage of total program execution time (in cycles) for which the unit is **on and busy executing instructions**. In Table 2 we show the utilization of the various functional unit types in the processor – for integer units the numbers represent average utilization of the two units. As we can see, except for the integer adders, the other units have very low utilization because while they are on, they are often not executing any operations. In other words there must be times when these units can be turned off. After applying our techniques we measured the utilization again. As shown in Table 3 the utilization of the integer adders shows very little change. This is because during the execution of the optimized code these units were always on. For the other three types of units the utilization has become very high because they are busy executing operations while they are on. This means that for most of the times that they were idle, we were able to turn them off. In other words these units were off for over 90% of the time for all programs except compress. Recalling the data in Table 2, we can see that turning off units for 90% of the time results in significant savings in static power dissipation.

Table 2: Utilization of functional units in original code.

Benchmark	Utilization (%)			
	Integer		Float	
	Adder	*Mult*	*Adder*	*Mult*
`rawcaudio.c`	87.71	0.0252	0	0
`rawdaudio.c`	88.76	0.00159	0	0
`fir2dim.c`	59.45	7.01	0	0
`n-real-updates.c`	61.62	2.37	0	0
`fir.c`	52.26	2.65	0	0
`compress.c`	90.08	0.045	25.70	29.06

Table 3: Utilization of functional units in optimized code.

Benchmark	Utilization (%)			
	Integer		Float	
	Adder	*Mult*	*Adder*	*Mul*
`rawcaudio.c`	87.73	99.7	99.7	99.7
`rawdaudio.c`	88.77	99.76	99.76	99.76
`fir2dim.c`	59.73	85.03	99.70	99.70
`n-real-updates.c`	61.62	94.53	99.44	99.44
`fir.c`	52.72	92.42	99.38	99.38
`compress.c`	90.31	98.90	23.99	51.15

Performance degradation. We also measured the degradation in the performance by comparing the total execution cycle counts for original and optimized code (see Table 4). The degradation is less than 1% due to the fact that we place the `on` and `off` instructions in cold blocks and units are turned *on* upon decode of instruction with the `on` suffix. The latter action reduces stalling of instructions due to unavailability of functional units.

Table 4: Performance degradation.

Benchmark	*Unoptimized*	*Optimized*	*Degradation*
`rawcaudio.c`	6,588,776	6,591,742	-0.0147
`rawdaudio.c`	5,028,710	5,049,175	-0.0041
`fir2dim.c`	4,676	4689	-0.28
`n-real-updates.c`	2,697	2,697	0
`fir.c`	2,413	2,424	-0.46
`compress.c`	453,823	454,877	-0.232

Transition activity vs off durations. For each idle period that a unit is turned off, we have a pair of transitions: *on-to-off* and then *off-to-on*. While the static power saved during the off periods depends upon the duration of the off periods, the dynamic power spent during transitions depends upon the total number of transitions actually performed.

Table 5 gives the total number of transition pairs for all the functional units types. There are no transitions for integer adders because they are always on and for integer multipliers the number given is the sum of the transitions encountered by both units of this type. These are the transitions which were actually performed during execution. Table 6 gives the average duration for which units were turned off. As we can see these durations are quite long - ranging from several hundred to several thousand cycles.

Since the durations for which functional units are off are quite long and the number of transition pairs is relatively modest, we can conclude that our approach is quite effective in saving static power wasted by idle functional units.

Table 5: Non-nullified transition pairs.

	Integer		Float	
Benchmark	*Adder*	*Mult*	*Adder*	*Mult*
`rawcaudio.c`	0	769	748	735
`rawdaudio.c`	0	800	919	712
`fir2dim.c`	0	2	1	1
`n-real-updates.c`	0	2	1	1
`fir.c`	0	2	1	1
`compress.c`	0	113	212	286

Table 6: Average off duration in cycles.

Benchmarks	Integer		Float	
	Adder	*Mult*	*Adder*	*Mult*
`rawcaudio.c`	-	8552	8789	8944
`rawdaudio.c`	-	5481	6296	7075
`fir2dim.c`	-	3987	4674	4674
`n-real-updates.c`	-	2550	2682	2682
`fir.c`	-	2230	2409	2409
`compress`	-	10929	496	847

Effectiveness of nullification strategy. We also measured the number of transition pairs which were nullified by our architecture design because they were too close together. The number of nullified transition pairs is given in Table 7. As we can see, this number is quite significant for some benchmarks as they contain variable length idle regions which are quite often of small duration. Therefore our approach of allowing the compiler to aggressively remove idle time and then relying on the hardware to nullify the operations if they are not useful has proven to be very successful.

Table 7: Nullified transition pairs.

Benchmarks	Integer		Float	
	Adder	*Mult*	*Adder*	*Mult*
`rawcaudio.c`	0	445	148	148
`rawdaudio.c`	1510	298	149	149
`fir2dim.c`	0	0	0	0
`n-real-updates.c`	0	0	0	0
`fir.c`	2	0	0	0
`compress`	958	0	1539	0

5 Conclusions

The static power component of power dissipation is on a rise [2,9]. We presented a technique for reducing this static power to some extent by switching *off* the idle units. Our approach uses a combination of compiler, instruction set, and microarchitecture support for maximizing power savings and minimizing performance degradation. Static power reduction of over 90% was achieved for units that were found to be mostly idle at the cost of well under 1% increase in execution times.

References

1. D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *International Symposium on Computer Architecture* (ISCA), pages 83–94, Vancouver, British Columbia, June 2000.
2. J.A. Butts and G.S. Sohi. A Static Power Model for Architects. In *IEEE/ACM International Symposium on Microarchitecture* (MICRO), pages 191–201. December 2000.
3. C. Fraser and D. Hanson. *lcc: A Retargetable C Compiler: Design and Implementation*. Adison Wesley Publishing Company, 1995.
4. M. Horowitz, T. Indermaur, and R. Gonzalez. Low-Power Digital Design. In *IEEE Symposium on Low Power Electronics*, pages 8-11, 1994.
5. K.S. Khouri and N.K. Jha. *Private Communication*. June 2001.
6. C. Lee, M. Potkonjak, and W.H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *IEEE/ACM International Symposium on Microarchitecture* (MICRO), Research Triangle Park, North Carolina, December 1997.
7. MIPS Technologies, 1225 Charleston Road, Mountain View CA-94043. *MIPS32 4k Processor Core Family, Software Users Manual*, 1.12 edition, January 2001.
8. S. Onder and R. Gupta. Automatic Generation of Microarchitecture Simulators. In *IEEE International Conference on Computer Languages* (ICCL), pages 80–89, Chicago, Illinois, May 1998.
9. M.D. Powell, S-H. Yang, B. Falsafi, K. Roy, and T.N. Vijaykumar. Gated-Vdd:a Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. In *ACM/IEEE International Symposium on Low Power Electronics and Design* (ISLPED), 2000.
10. K. Roy. Leakage Power Reduction in Low-Voltage CMOS Design. In *IEEE International Conference on Circuits and Systems*, pages 167-173, 1998.
11. S. Thompson, P. Packan, and M. Bohr. MOS Scaling: Transistor Challenges of the 21st Century. *Intel Technology Journal*, Q3, 1998.
12. V. Tiwari, R. Donnelly, S. Malik, and R. Gonzalez. Dynamic Power Management for Microprocessors: A Case Study. In *International Conference on VLSI Design*, pages 185-192, 1997.
13. V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. patel, and F. Baez. Reducing Power in High-Performance Processors. In *Design Automation Conference* (DAC), pages 732-737, 1998.
14. Q. Wang and S. Vrudhula. Static Power Optimization of Deep Submicron CMOS Circuits for Dual V_T Technology. In *International Conference on Computer-Aided Design* (ICCAD), pages 490-496, 1998.