Reducing Instruction Cache Energy Consumption Using a Compiler-Based Strategy

W. ZHANG
Southern Illinois University
and
J. S. HU, V. DEGALAHAL, M. KANDEMIR, N. VIJAYKRISHNAN, and M. J. IRWIN
The Pennsylvania State University

Excessive power consumption is widely considered as a major impediment to designing future microprocessors. With the continued scaling down of threshold voltages, the power consumed due to leaky memory cells in on-chip caches will constitute a significant portion of the processor's power budget. This work focuses on reducing the leakage energy consumed in the instruction cache using a compiler-directed approach.

We present and analyze two compiler-based strategies termed as conservative and optimistic. The conservative approach does not put a cache line into a low leakage mode until it is certain that the current instruction in it is dead. On the other hand, the optimistic approach places a cache line in low leakage mode if it detects that the next access to the instruction will occur only after a long gap. We evaluate different optimization alternatives by combining the compiler strategies with state-preserving and state-destroying leakage control mechanisms. We also evaluate the sensitivity of these optimizations to different high-level compiler transformations, energy parameters, and soft errors.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Cache Memories*; D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization*

General Terms: Design, Performance, Reliability

Additional Key Words and Phrases: Leakage power, cache design, compiler optimizations

1. INTRODUCTION

With the increasing number of transistors employed in current microprocessors and the continued reduction in threshold voltages of these transistors, leakage

A preliminary version of this work appears in *Proceedings of International Symposium on Microarchitecture (MICRO'02)* [Zhang et al. 2002]. This paper extends the MICRO'02 work by giving more experimental results and by presenting a soft error analysis.

Authors' addresses; W. Zhang, Department of Electrical and Computer Engineering, Southern Illinois University, Carbondale, IL 62901; email: zhang@engr.siu.edu; J. S. Hu, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, Microsystems Design Lab, The Pennsylvania State University, University Park, PA 16802; email: {jhu,degalaha,kandemir,vijay,mji}@cse.psu.edu.

energy consumption has become a major concern [Butts and Sohi 2000]. As dense cache memories constitute a significant portion of the transistor budget of current microprocessors, leakage optimization for cache memories is of particular importance. It has been estimated that leakage energy accounts for 30% of L1 cache energy and 70% of L2 cache energy for a 0.13 micron process [Powell et al. 2001].

There have been several efforts [Azizi et al. 2002; Cao et al. 2002; der Meer and Staveren 2000; Hu et al. 2002; Kawaguchi et al. 2000; Kim et al. 2002; Kuroda and Sakurai 1996; Li et al. 2002; Mutoh et al. 1995; Powell et al. 2001; Zhou et al. 2001] spanning from the circuit level to the architectural level at reducing the cache leakage energy. Circuit mechanisms include adaptive substrate biasing, dynamic supply scaling, and supply gating. Many of the circuit techniques have been exploited at the architectural level to control leakage at the cache bank and cache line granularities. The supply gating mechanism was applied at bank level granularity in Powell et al. [2001] to dynamically vary the size of the active portion of the cache. The cache miss rates were used to adapt the cache sizes in order to reduce leakage power consumption. The supply gating mechanism was employed at the finer granularity of cache line in Kaxiras et al. [2001]. This technique monitors the periods of inactivity in cache lines by associating saturating counters with them. In Zhou et al. [2001], only the data array of a cache is placed in a low power mode while the tag array is still in active mode. This helps to dynamically adjust the turn-off interval to ensure that performance closely tracks the performance of an equivalent cache without sleep mode. Another approach to leakage control at the cache line granularity involves the reduction of supply voltages to idle cache lines [Flautner et al. 2002]. Specifically, all cache lines are periodically placed in a leakage-controlled mode by scaling down their supply voltage. This implementation also chooses higher threshold voltages for the access transistors to minimize the bitline leakage. In contrast to other approaches, Heo et al. [2002] focus on reducing bitline leakage by leaving bitlines of banks that are not accessed open.

Most prior approaches have focused on utilizing hardware monitoring to manage the leakage-control modes of the caches. These techniques transition to leakage control modes after fixed periods or fixed periods of inactivity. They incur the energy penalty for decaying to the low leakage mode only after fixed periods. The approaches that dynamically change the turn-off periods attempt to address this problem. In contrast to these hardware-centric approaches, in this work, we propose a *compiler-based* leakage optimization strategy for instruction caches. This approach identifies the last use of the instructions and places the corresponding cache lines that contain them into a low leakage mode. The idea of instruction-based leakage control was suggested in Kaxiras et al. [2001] for data caches based on profiling. Their work also identified the need for compiler analysis in such an instruction-based approach. In this work, we present and analyze two compiler-based strategies termed as conservative and optimistic. The conservative approach does not put a cache line into a low leakage mode until it is certain that the current instruction in it is dead. On the other hand, the optimistic approach places a cache line in low leakage mode if it detects that the next access to the instruction will occur only after a long gap.

This paper makes the following contributions:

—We present both conservative and optimistic compiler strategies and evaluate different alternatives by combining these strategies with state-preserving and state-destroying leakage control mechanisms. We also show how state-preserving and state-destroying mechanisms can be combined by a compiler strategy to further increase energy savings over conservative and optimistic algorithms. We augment the supply voltage scaling technique proposed in Flautner et al. [2002] to dynamically support transitions between state-preserving and state-destroying modes. The state-preserving mode retains data but consumes more leakage energy as compared to the state-destroying mode.

—We compare the effectiveness of the proposed strategies with the recently proposed drowsy cache schemes [Flautner et al. 2002] using 0.07 micron technology [ber]. Our results show that compiler-based strategies are competitive with a pure hardware-based approach, and in most cases, they exhibit better cache energy and energy-delay product behaviors.

—We illustrate the impact of high-level compiler optimizations on the effectiveness of our leakage saving strategies. In particular, we point at the energy-performance trade-offs when optimizations that target data locality are applied.

—We show the trade-off between potential leakage reduction achievable and the soft-error rate due to the proposed leakage control mechanisms. While traditional metrics for comparing energy-reduction schemes have only included performance and energy-related metrics, reducing the supply voltage in the memory cells also makes them susceptible to soft errors. Our goal here is to quantify as to how long a cache line can be kept in a state-preserving mode without sacrificing reliability.

A compiler-based leakage optimization strategy such as ours makes sense in a VLIW environment (which is the focus of our work in this paper) where the compiler has control of instruction execution order. Using the Trimaran infrastructure [Tri], we demonstrate in this paper that it is possible to significantly optimize instruction cache leakage energy using compiler analysis.

The rest of this paper is organized as follows. Section 2 introduces the required circuit and compiler support for implementing our optimizations. Section 3 presents detailed evaluation of the energy and performance metrics of our approaches. A hybrid approach that combines state-preserving and state-destroying modes is explained in Section 4. The influence of compiler optimizations on the effectiveness of the leakage-control mechanisms is explored in Section 5. Section 6 analyzes the impact of using low-leakage control on the reliability of data stored in the cache. Finally, we present our conclusions in Section 7 and give future research directions.

2. OUR APPROACH

2.1 Circuit Support

We rely on the dynamic scaling of the supply voltages to reduce the leakage current in the cache memories. As supply voltage to the cache cells reduces, the leakage current reduces significantly due to short-channel effects. The choice of the supply voltage influences whether the data are retained or not. When the normal supply voltage of 1.0 V is reduced below 0.3 V (for a 0.07 micron process), we observe that the data in the cells are no longer retained. Thus, we select a 0.3 V supply voltage for the state-preserving leakage control mode. However, if state preservation is not a consideration, we switch the supply voltage to 0 V to gain more reduction in energy. Except for our hybrid scheme (discussed in Section 4) that requires *dynamic selection* between data-preserving and data-destroying modes, we use a similar circuit to that proposed in Flautner et al. [2002]. Each cache line is augmented with a power status bit that is used to control the appropriate voltage selection for the cell. A global control signal is used to set the power status and, consequently, set the voltages of all cache lines to 0.3 V (0.0 V) to place them in a state-preserving (state-destroying) leakage control mode. Whenever a cache line is accessed, its supply voltage is first switched to the normal voltage of 1.0 V before access is permitted. This is achieved by using the wordline trigger to reset the power status bit and by preventing the access until the supply voltage settles by gating the wordline. The gating must be performed as data can be corrupted when accessing the cache when the supply voltage is low. In our experiments, all cache lines are in the leakage-control mode before their first use for all strategies.

For the hybrid scheme, we augment this circuit as shown in Figure 1 to dynamically transition between active, state-preserving, and state-destroying modes. The power supply to the cache lines is set to 1.0 V, 0.3 V, or 0 V, respectively, for the three modes when using caches designed with 0.07 micron Berkeley predictive technology [ber]. Each cache line has a two-bit power status register indicating the mode (00-Active; 01- State-Preserving; 11- State-Destroying) in which it is placed. There are two global control signals (Set0, Set1) for changing the states. When a cache line in either state-preserving or state-destroying mode is accessed, the access is delayed until the supply voltage recovers to 1.0 V. When an access occurs, the status register bits are automatically set to zero. There are two special instructions that are used to place the cache lines into a state-preserving mode or state-destroying mode. The least significant bits (B0) of all the power status registers are globally set when the state-preserving transition instruction is executed. Note that this permits all cache lines in a state-destroying mode to remain in that mode even when the state-preserving instruction is executed. Similarly, the two bits (B0 and B1) of all the power status registers are set when the state-destroying transition instruction is executed.

It must be observed that our approach relies on a specific instruction to place cache lines in state-preserving or state-destroying mode. This is in contrast to the approach used in Flautner et al. [2002] where a periodic timer is used

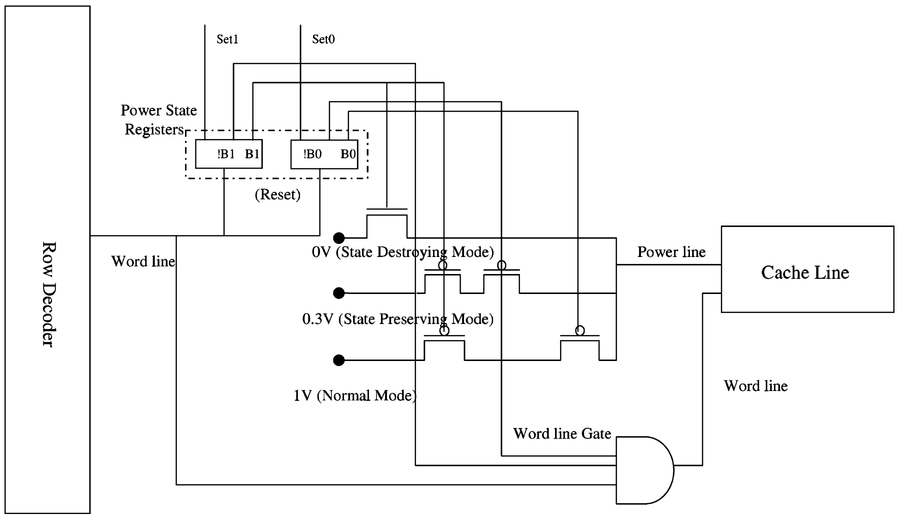


Fig. 1. Leakage control circuitry.

to place all cache lines in a state-preserving (drowsy) mode. We refer to the variants of the scheme proposed in Flautner et al. [2002] as Kill-M (where the content of the cache line is destroyed) and Drowsy-M (where the content of the cache line is preserved). Here, M is the periodic timer interval in cycles. In Section 3, we present a detailed comparison of our compiler-based strategies with Kill-M and Drowsy-M. In this paper, we refer to strategies Kill-M and Drowsy-M as the *fixed period strategies*.

The access transistors in our SRAM cells use a higher threshold voltage of 0.3 V as compared to the other SRAM transistors that use a threshold voltage of 0.2 V. This is performed to keep the contribution of the bitline leakage to a minimum. Our leakage results do not account for any gate leakage. There are efforts at designing high-k dielectrics to mitigate gate leakage. High-k dielectric films can permit a thicker insulation layer to reduce gate leakage significantly while keeping capacitance constant [Cataldo 2001]. Further, all techniques would have similar gate leakage behavior. We consider the impact of soft errors due to our voltage scaling approach later in Section 6.

2.2 Compiler Support

In order to exploit the state-destroying and state-preserving leakage control mechanisms explained above, our compiler implements two different approaches for turning off instruction cache lines. The first approach, called the *conservative strategy*, does not turn off an instruction cache line unless it knows for sure that the current instruction that resides in that line is dead (i.e., will not be referenced for the remaining part of the execution). The second approach is called the *optimistic strategy* and turns off a cache line even if the current instruction instance in it is not dead yet. This might be a viable option if there is a large gap between two successive visits to the cache line. In the

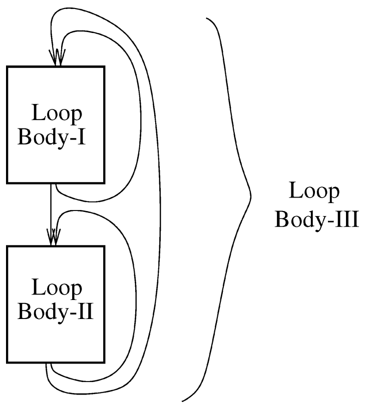


Fig. 2. A code fragment that contains three loops.

following discussion, we explain the implementations of these two strategies in detail.

The conservative strategy is based on determining the last usage of instructions. In other words, it tries to detect the use of the last instance of each instruction. Once this last use is detected, the corresponding cache line can be turned off. While it is possible to turn off the cache line immediately after the last use, such a strategy would not be very effective, because it would result in significant code expansion due to the large number of turnoff instructions inserted in the code. Also, such frequent turnoff instructions themselves would consume considerable dynamic energy. Consequently, in this work, we turnoff instructions at the *loop granularity level*. More specifically, when we exit a loop and we know for sure that this loop will *not* be visited again, we turnoff the cache lines that hold the instructions belonging to the loop (including those in the loop body as well as those used to implement the loop control code itself). While ideally we would want to issue turnoffs only for the cache lines that hold the instructions in the loop, identifying these cache lines is costly (i.e., it either requires some type of circuit support that itself would consume energy, or a software support that would be very slow). As a result, in this work, when we exit the loop, we turnoff *all* cache lines. While this has the drawback of turning off the cache lines that hold the instructions sandwiched between the inner and outer loops of a nest (and lead to reactivation costs for such lines), the impact of this in practice is not too much as typically there are not many instructions sandwiched between nested loops. Turning off all cache lines also eliminates the complexity of selectively turning off a cache line in a set-associative cache and problems associated with multiple instructions from different loops being stored in a single cache line.

The idea behind the conservative strategy is illustrated in Figure 2 for a case that contains three loops, two of which are nested within the third one. Assume that once the outer loop is exited, this fragment is not visited again during execution. Here, Loop Body-I and Loop Body-II refer to the loop bodies of the inner loops. In the conservative strategy, when we exit Loop Body-I (i.e., the loop that contains it), we *cannot* turnoff the cache lines occupied by it; because, there is an outer loop that will re-visit this loop body; in other words, the instructions in Loop Body-I are *not* dead yet. The same argument holds when we exit Loop Body-II. However, when we exit the outer loop, the conservative strategy turns off all the cache lines that hold the instructions of this code fragment (i.e., all instructions in all three loops). As mentioned above, we in fact turnoff all the cache lines for implementation efficiency. It is clear that this strategy may not perform well if there is a large outermost loop that encloses a majority of the instructions in the code. In such cases, the cache lines occupied by the said instructions will not be turned off until the outermost loop finishes its execution. And, when this occurs, it might be too late to save any leakage energy. For example, if there is an outermost loop (in the application being optimized) that encloses the entire code (e.g., a convergence test in some array applications), this strategy will not generate very good results as it will have little opportunity to turnoff cache lines.

The optimistic strategy tries to remedy this drawback of the conservative scheme by turning off the cache lines optimistically. What we mean by optimism here is that the cache lines are turned off *even if* we know that the corresponding instruction instance(s) will be visited again, but the hope is that the gap (in cycles) between successive executions of a given instruction is large enough so that significant amount of energy can be saved. Obviously, an important question here is how to make sure at compile time (i.e., statically) that there will be a large gap between successive executions of the same instruction. Here, as in the conservative case, we work on a *loop granularity.* When we exit a loop, we turnoff the instructions in the loop body if either that loop will not be visited again (as in the conservative case) *or* the loop will be re-visited but there will be execution of *another* loop between the last and the next visit. Returning to the code fragment in Figure 2, when we exit Loop Body-I, we turnoff the instructions in it. This is because before Loop Body-I is visited again, the execution should proceed with another loop (the one with Loop Body-II), and we optimistically assume that this latter loop will take long time to finish.[1] Similarly, when we exit Loop Body-II, we turnoff the corresponding instructions. Obviously, this strategy is more aggressive (in turning off the cache lines) than the conservative strategy. The downside is that in each iteration of the outer loop in Figure 2, we need to re-activate the cache lines that hold Loop Body-I and Loop Body-II. The energy overhead of such a re-activation depends on the leakage saving mode employed. Also, since each reactivation incurs a performance penalty, the overall execution time impact due to the optimistic strategy can be expected to be much higher than that due to the conservative strategy. This will be particularly felt when the state-destroying leakage control mechanism is employed (since it takes longer to come back to normal operation from state-destroying than state-preserving mode due to the L2 cache latency).

[1]While a more sophisticated approach would employ profile data to check whether that loop really takes long time, in our current implementation we do not perform such checks. Instead, a reliance is placed upon the observation that most loops (even those in nonarray applications) take a long time to execute.

Table I. Four Different Implementation Choices Depending on the Leakage Control Mechanism (Mode) Used and the Compiler Strategy Employed

Strategies	Conservative	Optimistic
State-Destroying	Cons-Kill (Strategy I)	Opti-Kill (Strategy II)
State-Preserving	Cons-Drowsy (Strategy IV)	Opti-Drowsy (Strategy III)

If there are code fragments that are not enclosed by any loops, our current implementation treats each such fragment as if it is within a loop that iterates only once. It should be emphasized, however, that both our strategies turnoff cache lines considering the instruction execution patterns. Since a typical cache line can accommodate several instructions during the course of execution, turning off a cache line may later lead to re-activation of the same cache line if another instruction wants to use it. This re-activation has both performance and energy penalty which should also be accounted for. Note that as the instruction cache gets bigger this problem will be of less importance. This is because in a larger cache we can expect fewer cache line sharing among instructions.

2.3 Alternative Strategies

Since we have two different compiler strategies (conservative and optimistic) and two different leakage-saving mechanisms (state-preserving and state-destroying), clearly, we have four different implementation choices. These choices are summarized in Table I. Among the choices we have, Strategy *Cons-Drowsy* (IV) does not make much sense since being conservative means that we do not turnoff cache lines unless we are sure that the instructions are dead. Therefore, there is not much point in employing a state-preserving leakage control mechanism. Consequently, in the rest of this paper, we focus only on the remaining three strategies: *Cons-Kill* (I), *Opti-Kill* (II), and *Opti-Drowsy* (III), and compare them with fixed period strategies *Kill-M* and *Drowsy-M* [Flautner et al. 2002]. Note that while one can select the best M value for a given application, it is possible that each application (and even different parts of the same application) demands a different M value. In contrast to fixed period strategies, our compiler strategies can automatically tune the turnoff periods within different phases of the program and also based on the different characteristics of each program. Furthermore, the compiler strategies can even select the appropriate low-leakage mode if there is underlying circuit support.

3. EXPERIMENTS

3.1 Benchmarks and Simulation Platform

We target improving leakage energy consumption of the instruction cache in a state-of-the-art VLIW processor. The results reported on here are obtained using a Trimaran-based compiler/simulation infrastructure. Trimaran provides a vehicle for implementation and experimentation in state-of-the-art research in compiler techniques for instruction level parallelism (ILP) [Tri]. A program flows through IMPACT, Elcor, and the cycle-level simulator. IMPACT applies

Table II. Default Parameters Used in Our Simulations

Parameter	Value
Feature size	0.07 micron
Supply voltage	1.0 V
L1 instruction cache	16 KB direct-mapped cache
L1 instruction cache latency	1 cycle
L1 data cache	32 KB 2-way cache
L1 data cache latency	1 cycle
Unified L2 cache	512 KB 4-way cache
L2 cache latency	10 cycles
Memory latency	100 cycles
Clock speed	1 GHz
L1 cache line size	32 bytes
L2 cache line size	64 bytes
L1 cache line leakage energy	0.33 pJ/cycle
L1 state-preserving mode cache line leakage energy	0.01 pJ/cycle
L1 state-destroying mode cache line leakage energy	0.00 pJ/cycle
L1 state-transition (dynamic) energy	2.4 pJ/transition
L1 state-transition latency from state-preserving mode	1 cycle
L1 state-transition latency from state-destroying mode	1 cycle (plus miss latency)
L1 dynamic energy per access	0.11 nJ
L2 dynamic energy per access	0.58 nJ

machine-independent classical optimizations and transformations to the source program, whereas Elcor is responsible for machine-dependent optimizations and scheduling. Our conservative and optimistic algorithms are implemented in Elcor, and after all other optimizations have been performed. The increase in compilation time due to our algorithms was around 15% on average (when all benchmark codes are considered). Further, the increase in code size due to the inserted turnoff instructions is less than 5% across all benchmarks and strategies. The cycle-level simulator was augmented with a cache model and modified to recognize the power-mode control instructions for changing the supply voltages to the cache lines. The VLIW configuration used in our experiments has four IALUs (integer ALUs), two FPALUs (floating-point ALUs), one LD/ST (load/store) unit and one branch unit. Other system parameters used for our default setting are provided in Table II. The energy values reported are based on circuit simulation. In our evaluations, we performed experiments with both basic block scheduling [Muchnick 1997] and superblock scheduling [Chang et al. 1991]; since we did not observe too much difference in trends, we report here only the basic block based scheduling results.

To evaluate the effectiveness of our algorithms, we used a suite of ten programs from different benchmark sets. The salient characteristics of these codes are given in Table III. The benchmark source is indicated in the second column. The third column in this table gives the number of code lines, and the fourth column gives the input used for running the benchmark. The total execution cycles and the original instruction cache energy consumption are provided in the last two columns. In selecting these programs, we paid attention to ensure diversity. Compress and li are integer codes with mostly irregular access patterns. idea, mpeg2dec, polyphase, and rawdaudio are typical media applications. The last

Table III. Benchmark Codes Used in Our Evaluations. The Last Column Also Contains the Percentage Contribution of leakage to Overall Instruction Cache Energy. Note That No Leakage Control Mechanism is Employed in Obtaining This Data

Benchmark	Source	Lines	Input	Exec Cycles	ICache Energy (nJ)
129.compress	SpecInt95	1939	test.in	42,784,111	13,627,628 (53%)
139.li	SpecInt95	7597	train.lsp	918,252,701	230,649,411 (67%)
idea	Mediabench	1232	/	335,180	97,343 (58%)
mpeg2dec	Mediabench	9832	mei16v2.m2v	140,735,320	46,702,867 (51%)
paraffins	Trimaran	388	/	523,363	111,058 (79%)
polyphase	Mediabench	542	polyphase.IN	587,442	181,888 (54%)
rawdaudio	Mediabench	314	clinton.adpcm	7,479,483	2,870,793 (44%)
adi	Livermore	46	274.68 MB	1,490,229	435,725 (58%)
btrix	Specfp92	135	202.53 MB	270,56,699	6,337,571 (72%)
vpenta	Specfp92	114	14.42 MB	141,445,594	29,645,742 (81%)

three benchmarks (adi, btrix, and vpenta) and paraffins, on the other hand, represent array-intensive applications.

Note that our focus in this paper is on *optimizing the leakage energy consumed in the instruction cache.* In doing so, however, our strategies can also incur several energy (and performance) overheads. For example, there is a dynamic energy overhead in the instruction cache due to turning on/off a cache line placed into a leakage-control mode. Also, there is a dynamic energy overhead (in the datapath and for fetching) due to executing turnoff instructions. Since some of our strategies increase execution cycles, the extra leakage energy consumption (in the instruction cache and other components) might also be an issue. In our presentation, where significant, we quantify these overheads to illustrate the energy behavior at larger level (not just in the instruction cache). In the rest of this section, when we mention *energy* we mean the leakage energy consumed by the instruction cache plus any extra (dynamic) energy that occurs as a result of cache line turnoffs/ons and due to any additional L1 instruction cache accesses. In other words, in our results, we include the overheads associated with our optimization strategy. This extra energy might be important as some of the strategies evaluated here can incur large performance penalties and significant number of cache line turn-ons. As mentioned earlier, we also compare our optimization strategies with fixed period strategies: Kill-M and Drowsy-M; we experiment with two M values: 2K and 4K.

3.2 Cache Life-Time Analysis

We present in Figure 3 the percentage time that cache lines spend in leakage control mode for different optimization strategies. That is, each bar in Figure 3 represents the average number of cache lines that are turned off, averaged across all execution cycles. One thing to note in this graph is that, for some versions, this time is very high. Specifically, the average percentage of time that cache lines are in a leakage control mode for Cons-Kill, Opti-Kill, Opti-Drowsy, Kill-2K, Drowsy-2K, Kill-4K, and Drowsy-4K are 40.83%, 85.61%, 86.21%, 86.10%, 79.83%, 81.61%, and 77.09%, respectively. These numbers indicate that

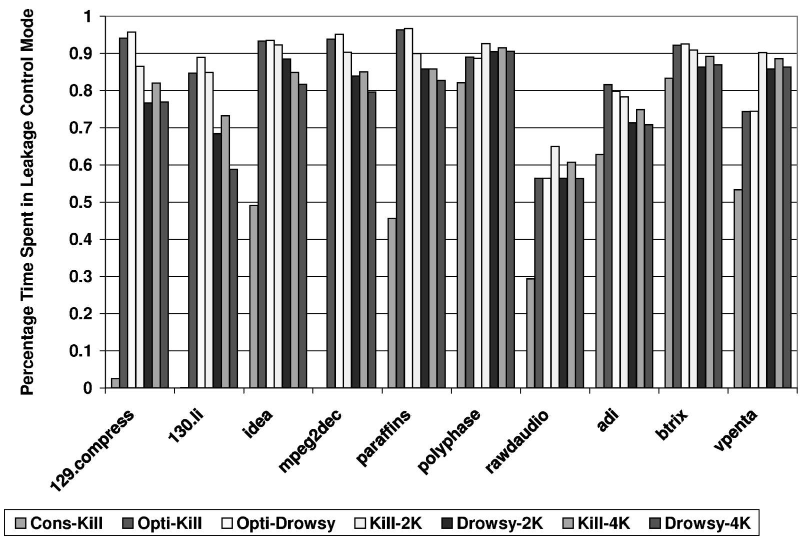


Fig. 3. Percentage of times where cache lines are in leakage control mode.

our strategies are very effective in exploiting the idleness of cache lines for potential leakage energy savings. What this means is that we may not need a finer-granularity turnoff strategy (e.g., one that works on instruction granularity) as such a strategy would increase the number of turn-ons significantly without significantly improving the leakage behavior over some of the strategies evaluated here. It should be noted, however, that to perform direct comparison between different strategies based on the results presented in Figure 3 is not conclusive. This is due to two main factors. First, for a given benchmark code, different versions might take different execution times to complete. Second, different versions sometimes use different leakage control modes and incur different energy overheads. For example, strategies Cons-Kill and Opti-Kill destroy the data in cache lines, while strategy Opti-Drowsy maintains data. Consequently, it is important to consider energy and performance profiles as well.

3.3 Energy and Performance Results

Figure 4 shows the energy consumptions of the strategies (given as fractions of the energy consumption of strategy Cons-Kill). Note that though not given in the graph, strategy Cons-Kill improves the energy consumption of the original code (without any leakage control but discounting the leakage of unused cache lines) by 40% on an average (ranging from 1% for mpeg2dec to 84% for btrix). One can make several observations from this figure. First, strategy Cons-Kill does not perform well as compared to other optimization strategies. In fact, it generates the worst results in most benchmarks. Second, one can see that strategies Opti-Drowsy and Opti-Kill (in some cases) generate very good energy results. In fact, in 8 of our 10 benchmarks one of these two strategies provide the best energy consumption. Third, in some benchmarks, most notably vpenta, the fixed

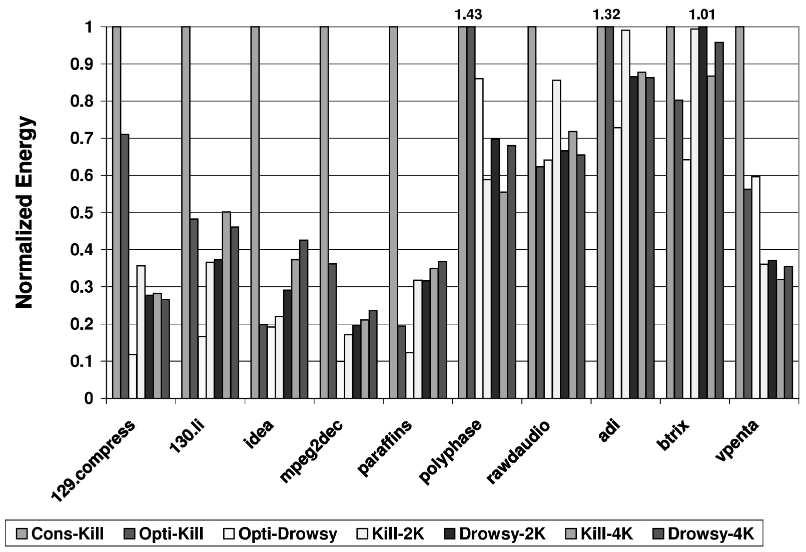


Fig. 4. Normalized (with respect to Strategy Cons-Kill (I)) energy consumption with different leakage saving strategies.

period strategies generate the best energy behavior. Now, let us try to explain the behavior of these different strategies, starting with our strategies. We can measure the magnitude of energy benefits of a given optimization strategy considering three factors: (1) how soon it turns off cache lines; (2) what leakage control mechanism it uses; (3) whether the energy overheads overwhelm the potential leakage savings. Strategy Cons-Kill does not perform well because it acts very late in turning off cache lines. Recall that it does not turnoff a cache line unless it is sure that the corresponding instruction is really dead. And, when this occurs it might be well too late to save any leakage energy. While Opti-Kill turns off cache lines quickly and employs state destroying mode to provide significant reduction in leakage energy, the energy cost of frequent extra writes to the instruction cache (when the instructions need to be fetched again from L2) nullifies this benefit in most of the benchmarks. Strategy Opti-Drowsy also turns off cache lines quickly (after each inner loop); however, it uses state-preserving leakage mode. In contrast to Opti-Kill, this scheme does not have the additional overhead of instruction cache misses. Moreover, strategy Opti-Drowsy can obtain most of the leakage savings provided by strategy Opti-Kill (due to the small difference in their leakage values).

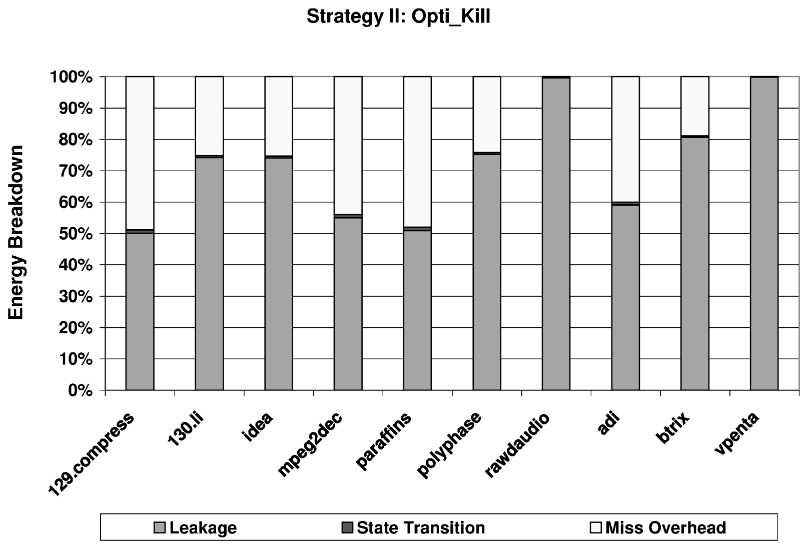
We turn our attention now to fixed period schemes. Their energy behavior compared to our strategies depends to a large extent on the execution time of the loops. For example, if a given loop takes too long to finish, even our strategies Opti-Kill and Opti-Drowsy will not achieve very good results as it will take a long time before they can turnoff the corresponding cache lines. However, the fixed period strategies can turnoff the cache lines in such a loop (maybe several times depending on the execution time). As an example, consider the following program fragment:

```
for (...)
{
   S1;
   S2;
   ...
   Sk;
   ...
   Sm;
}.
```

In this fragment, we have a loop with `m` statements in it (we assume that none of these statements themselves are loops). Suppose that statement `Sk` is experiencing several data cache misses. Consequently, it is going to take some time for the execution to move to statement `S(k+1)`. In such a case, our strategies will not be able to turnoff the cache lines that hold the instructions in statements `S1`, `S2`, ..., `S(k-1)` as they are loop based (i.e., they cannot turnoff cache lines before exiting the loop). Examples of such loops occur in vpenta. In comparison, a fixed period scheme can turnoff the said cache lines once the period (M) is reached. On the other hand, for a loop with short execution time, our approaches can turnoff cache lines as soon as it finishes its execution whereas the fixed period schemes wait for the fixed period to elapse. Returning to the loop fragment above, if we assume that none of statements `S1`, `S2`, ..., `Sm` experience significant data memory stalls, the loop might end very quickly. In this case, our strategies will turnoff the cache lines while a fixed period scheme will still wait for the duration of the period. Also, note that when the duration of a single iteration is short but the loop has a long execution time, additional turnoffs in the fixed period strategies do not benefit much as the dynamic energy overhead amortizes any leakage savings.

Figure 5 divides the energy consumption in the instruction cache into different components for strategies Opti-Kill and Opti-Drowsy. For strategy Opti-Kill, each bar is divided into three parts: the leakage energy consumed during normal operation, the dynamic energy incurred in transitioning to and from state-destroying mode, and the dynamic energy consumed in the instruction cache due to additional cache misses. We see that in most of the benchmarks, the dynamic energy overhead due to extra misses constitutes a large percentage of the energy consumption, which explains the poor behavior of this strategy. Also, note that the leakage energy consumed in state-destroying mode is zero. For strategy Opti-Drowsy, each bar is divided into three parts: the leakage energy consumed during normal operation, the dynamic energy incurred in transitioning to and from state-preserving mode, and the leakage energy consumed in the state-preserving mode. It should be observed that the state-transition overhead is small and a considerable portion of the energy is expended when cache lines are in state-preserving mode.

We also need to emphasize that the strategies that kill the data in cache lines prematurely (i.e., before the corresponding instructions are dead) can also cause extra dynamic energy consumption in L2 cache and off-chip memory. Although



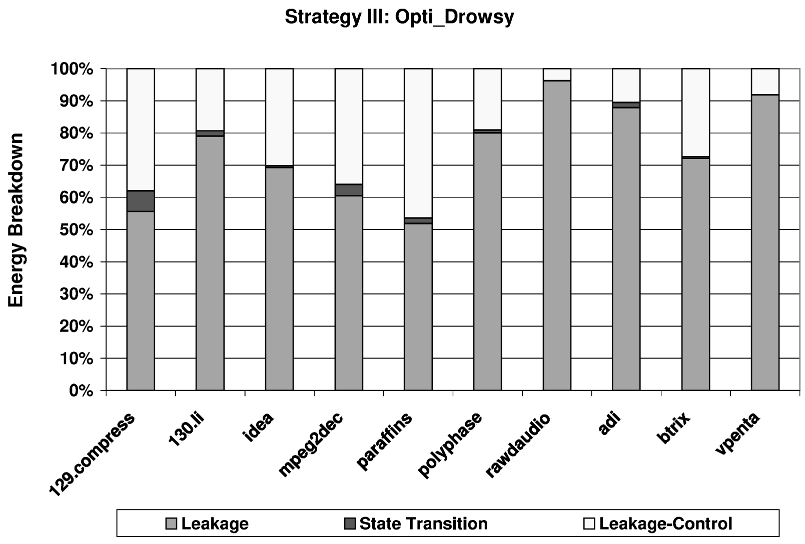


Fig. 5. Energy breakdown for Strategy Opti-Kill (II) and Strategy Opti-Drowsy (III).

in this work we focus on on-chip energy consumption, it is also important to know the magnitude of this off-chip energy. Figure 6 gives the extra dynamic energy consumption in the off-chip L2 and memory. It should be seen that among our strategies only Opti-Kill can cause extra off-chip energy consumption. This is because Strategy Cons-Kill kills the contents of a cache line if and only if it is already dead and Strategy Opti-Drowsy only employs state-preserving mode. Therefore, strategy Opti-Drowsy becomes even more preferable when considering off-chip L2 and memory energy.

Obviously, energy behavior is only a part of the picture. To have a fair evaluation of all strategies considered, we need to look at their performance

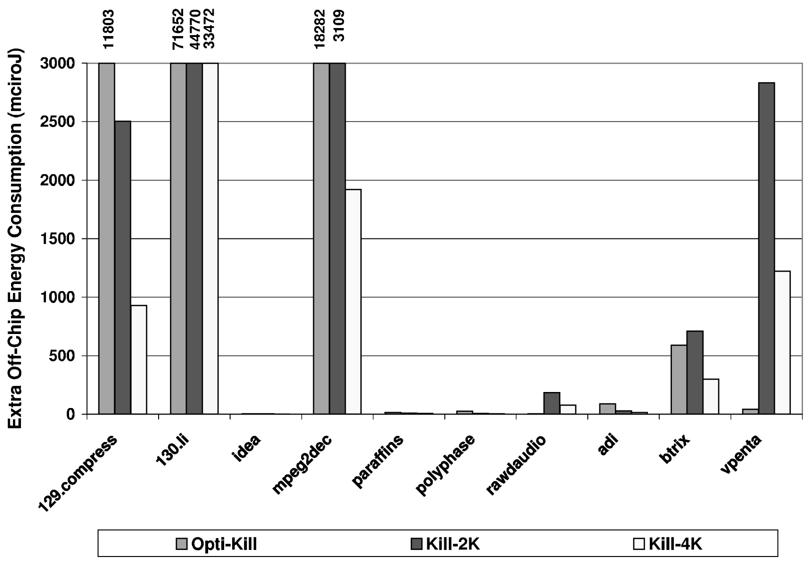


Fig. 6. Extra dynamic energy consumption in off-chip L2 and memory. Only the strategies that employ state-destroying mode incur this penalty.

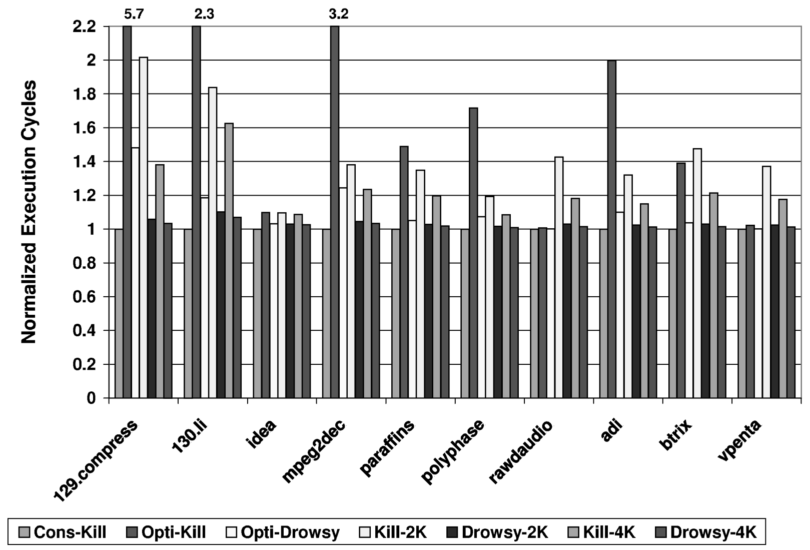


Fig. 7. Normalized (with respect to Strategy Cons-Kill) number of execution cycles with different leakage saving strategies.

behavior as well. The normalized execution cycles with our base configuration are presented in Figure 7. All values are normalized with respect to that of strategy Cons-Kill (I). Two factors influencing the performance penalty are the number of cache lines turned on and the number of cycles spent per turn-on. The second factor is dependent on whether the cache line was in state-preserving or state-destroying mode before turn on. On the average, the number

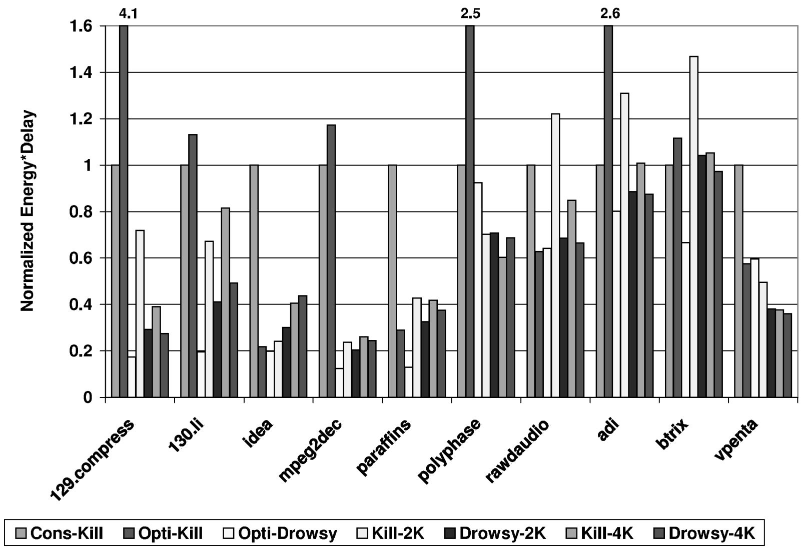


Fig. 8. Energy-delay (with respect to Strategy Cons-Kill) products with different leakage saving strategies.

of turn-ons for strategies Cons-Kill, Opti-Kill, Opti-Drowsy, Kill-2K, Drowsy-2K, Kill-4K and Drowsy-4K are 1448, 22,77,7521, 22,777,521, 14,096,265, 10,968,055, 10,086,726, and 7,428,348, respectively. The number of turn-ons in our schemes is typically larger than that of the fixed period schemes (except Cons-Kill). Note that the performance penalty for the same number of turn-ons for Opti-Kill is much larger than that of Opti-Drowsy due to L2 access latencies. This clearly shows the trade-off between energy savings and performance overhead.

The energy-delay product helps to balance the benefits in energy savings with any potential degradation in performance. Figure 8 shows the normalized energy-delay products for our applications. We see that strategy Opti-Drowsy is very successful. This is because in many cases its percentage energy benefits are higher than its performance losses, and also it strikes a good balance between performance and energy. We observe that the average normalized energy-delay product for Opti-Drowsy is 0.47 that is 13% better than that of the best fixed-period scheme (Drowsy-2K—0.54) for the considered applications.

Our use of energy-delay product in this paper should be interpreted with care. The energy in our energy-delay product considers only the leakage energy in the instruction cache and extra dynamic energy consumed within the chip in applying our leakage-control mechanisms. While the overall system energy would be a more useful metric, it is difficult to estimate the leakage of other parts of the chip using our current infrastructure. Further, leakage optimizations can also be applied to the other components (e.g., Zhang et al. [2001]) to mitigate any adverse impact due to performance loss.

In order to provide an insight as to how the leakage in other components would affect our results, we assume that the instruction cache, data cache, and

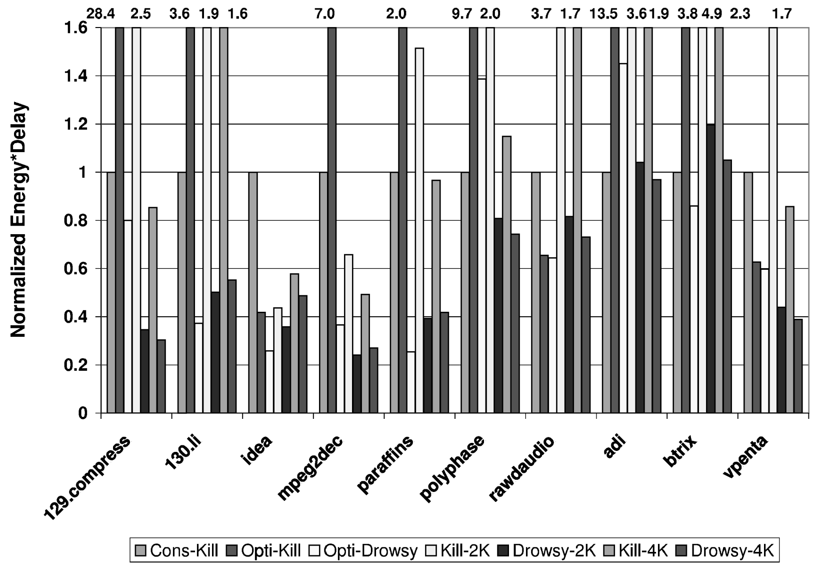


Fig. 9. Energy-delay (with respect to Strategy Cons-Kill) products for the entire processor with different leakage saving strategies.

datapath contribute equally to the total chip leakage energy before applying any optimizations. In addition, the data cache are assumed to use an aggressive leakage control mechanism [Flautner et al. 2002], and the datapath are assumed to use the leakage optimizations proposed in Zhang et al. [2001]. Our results shown in Figure 9 indicate that our compiler scheme Opti-Drowsy works best for five (instead of seven without considering the leakage in the rest of the components) out of ten benchmarks in terms of energy-delay product, considering the whole chip leakage energy consumption. This is because the increased execution cycles due to the performance loss increase the leakage energy in other components of the processor. The objective of this experiment was to illustrate the influence of modeling leakage in the other components, and the results will dependent on the accurate quantification of the leakage.

3.4 Sensitivity to Energy-Related Parameters

In this section, we study how these strategies are affected when some energy parameters are modified. In particular, we focus on four parameters: the leakage reduction factor in the state-preserving leakage control mode, the dynamic energy per access, leakage energy consumption in active mode, and the state-transition latency.

These parameters will typically be affected when the technology changes, and new techniques are used for manufacturing in the near future. As the popularity of using low V_t transistors for circuit implementation and the exponentially increasing impact of new types of leakage source such as gate leakage [Doyle et al. 2002; Guindi and Najm 2003; Lee et al. 2003], there are more challenges in controlling leakage that may reduce the effectiveness of the leakage energy reduction in drowsy mode (state-preserving). In this sensitivity analysis, we

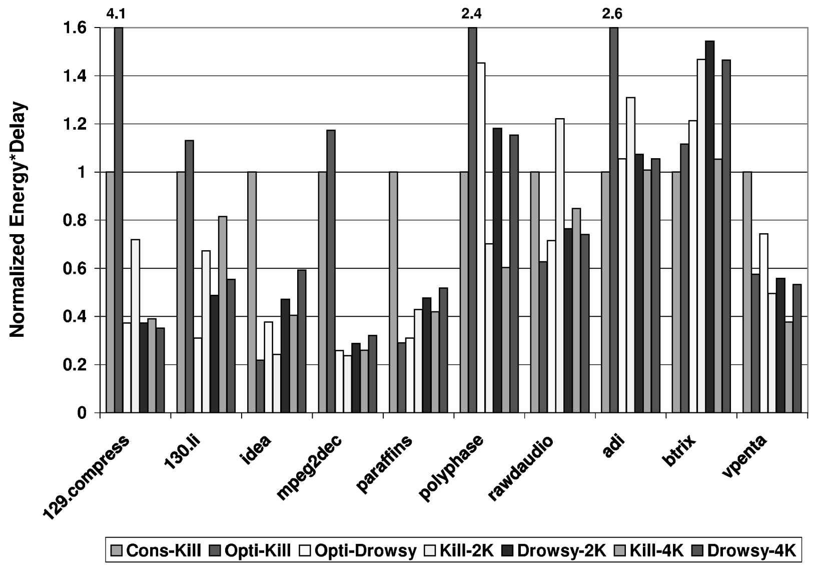


Fig. 10. Impact of larger energy consumption in the state-preserving state on energy-delay products (normalized with respect to Strategy Cons-Kill).

increase the leakage energy consumption in drowsy mode to simulate this trend. While leakage energy is projected to account for 70% of the total cache energy in 70 nm technology [Flautner et al. 2002], it is expected to increase even further as feature sizes become smaller. We capture this trend by increasing the ratio of leakage energy to that of the dynamic energy by scaling values used for dynamic and leakage energies. This ratio may be influenced differently for technologies such as SOI (silicon on insulator) where the leakage (subthreshold leakage) energy can be dramatically reduced [Das and Brown 2003]. We also explore this scenario by reducing the leakage energy.

For brevity, we focus on energy-delay product as it includes the impacts of both energy and execution time. Figure 10 shows the normalized energy-delay products when the leakage energy consumption in the state-preserving mode is increased to 0.04 pJ (from 0.01 pJ) per cycle per cache line (i.e., around 12% of the leakage energy in the normal operation mode). We perform this variation in order to model cases where there is need for less aggressive scaling of state-preserving supply voltages to provide more resilience to soft errors and also to consider speculative technologies in which new sources of leakage other than subthreshold leakage may be significant. We see that this penalizes the strategies that employ the state-preserving mode. Consequently, among our strategies, Opti-Kill starts to outperform Opti-Drowsy in energy-delay behavior for five benchmarks as opposed to only two benchmarks in the default configuration. Similarly, we observe that the Kill-M schemes become competitive with the Drowsy-M schemes, outperforming them in seven benchmarks (as opposed to one benchmark in the original case). Thus, the different schemes need to be chosen with attention to underlying circuit technology and desired reliability.

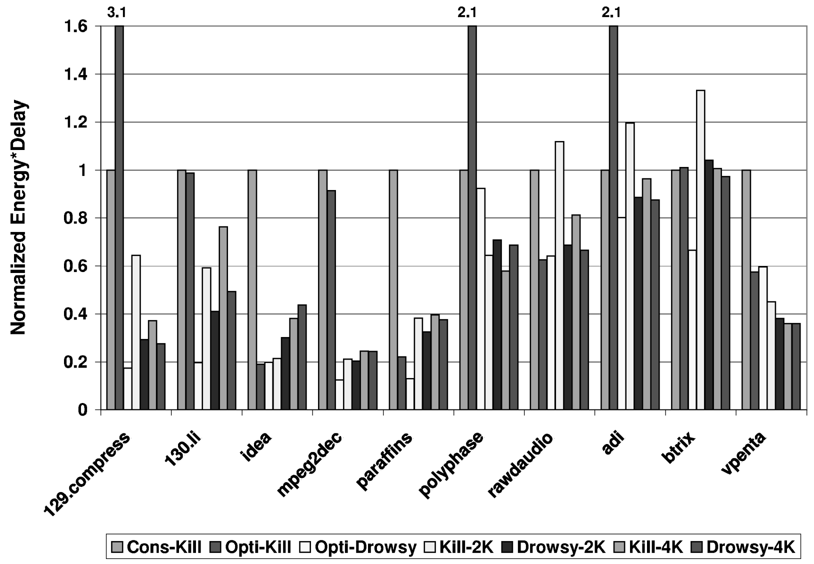


Fig. 11. Impact of more efficient dynamic energy consumption circuitry (normalized with respect to Strategy Cons-Kill).

The state-destroying schemes incur the penalty of additional L1 writes due to premature turn-offs. Thus, the dynamic energy of an instruction cache access is critical in deciding the trade-off between different schemes. We reduce the default dynamic energy per cache access by half to model variations in the relative importance of dynamic and leakage energy in different technologies and show the corresponding energy-delay products in Figure 11. We observe that Opti-Kill and Kill-M modes become more desirable as compared to the original configuration.

To investigate the impact of reduction in absolute leakage current due to technologies such as SOI, we scale the leakage energy consumption of an active cache line down to the half of the originally assumed value, 0.33 pJ / 2 = 0.165 pJ, with all other simulation parameters unchanged. Figure 12 shows the new normalized energy-delay products for all the schemes. We observe that the normalized values for strategies other than Cons-Kill are increasing, which means that the relative effectiveness of the leakage control schemes is decreasing compared to the conservative strategy Cons-Kill. Especially, for schemes using state-destroying mechanism, the product values increase much faster than those for schemes using state-preserving mechanism that indicates that the state-preserving mechanism will be much more preferable in new technologies.

We also experimented with the sensitivity to the variation in the sizing of switches used to select the power supply to the cache line. When larger switches are used, the dynamic energy expended in switching from one supply voltage to another becomes larger as compared to using smaller switches. However, smaller switches also require more time for the new supply voltage to settle and incur a larger performance penalty for getting back from a leakage control mode to normal mode.

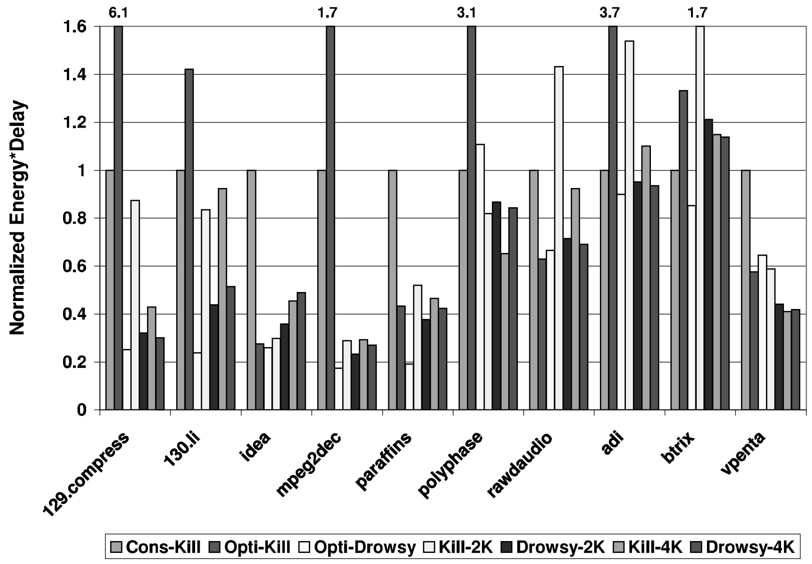


Fig. 12. Impact of smaller leakage energy consumption in the active state (normalized with respect to Strategy Cons-Kill).

3.5 Impact of Instruction Cache Capacity

The effectiveness of our leakage optimization strategies is impacted when the capacity of the instruction cache is modified. This is because our strategies turnoff cache lines by detecting when instructions are dead (or will not be accessed for long durations of time). As explained earlier, even our most conservative strategy (Cons-Kill) is optimistic in this respect. This is because there might be multiple instructions mapping into the same cache line. This, in turn, reduces the cache line interaccess times, thereby diminishing the effectiveness of the strategies. However, when the instruction cache capacity is increased, there will be fewer instructions that map into a given cache line, and consequently, we can expect better returns from leakage optimization. To illustrate this, we present in Figure 13 the leakage energy consumption using a 64 KB instruction cache as compared to our default 16 KB instruction cache results for strategies Opti-Kill and Opti-Drowsy. Each bar in this graph corresponds to the energy consumption with a 64 KB cache divided by the energy consumed with a 16 KB cache. We observe that due to the reduced conflicts Opti-Drowsy takes advantage of the larger cache for all benchmarks. However, for Opti-Kill, the overhead of additional instruction cache misses increases when moving from 16 KB cache to 64 KB cache (from 0.11 nJ to 0.16 nJ per access). Thus, in two benchmarks, it consumes larger leakage energy (including dynamic energy overheads) with the 64 KB instruction cache.

4. HYBRID STRATEGY

Our leakage optimizations evaluated so far use either state-preserving mechanism or state-destroying mechanism exclusively. It is also possible to employ

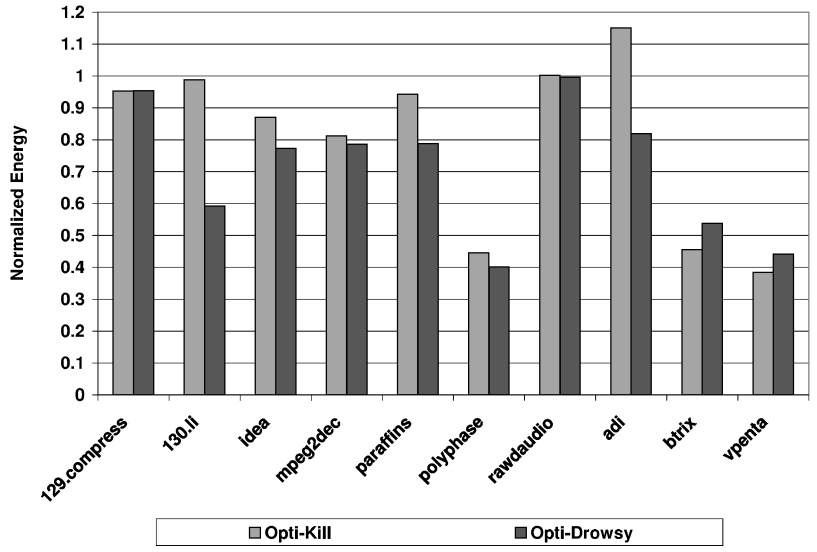


Fig. 13. Normalized energy consumption of a 64 KB instruction cache as compared to default configuration.

these two mechanisms under the same optimization strategy. That is, a leakage optimization strategy can use these mechanisms selectively. In this section, we discuss such a *hybrid strategy* and quantify its capability of saving leakage.

Our hybrid strategy proceeds as follows. When exiting a loop, if this loop is not going to be accessed again, the hybrid strategy turns off the associated cache lines using the state-destroying mechanism. On the other hand, if the loop will be visited again, it just places the cache lines into leakage control mode using the state-preserving strategy. Later in execution, when this loop finishes its last execution, the cache lines are turned off via the state-destroying mechanism. In other words, depending on whether the loop will be re-visited or not, the hybrid scheme chooses the appropriate leakage control mechanism.

The energy-delay product profile of the hybrid strategy is illustrated in Figure 14. Each bar is normalized with respect to the strategy that generated the best (excluding hybrid) energy-delay result (as far as that benchmark is concerned). One can observe from these results that in three benchmarks (compress, li, and mpeg2dec), hybrid and Opti-Drowsy generate the same energy-delay product. In polyphase and vpenta, hybrid is outperformed by Kill-4K and Drowsy-4K. However, in the remaining five benchmarks, hybrid generates the best results. Note that the hybrid scheme, however, involves extra circuit overheads for the additional supply voltage to choose dynamically between state-destroying and state-preserving modes. Strategy Opti-Drowsy performs well considering that the overhead of the additional (third) supply voltage distribution was not factored in the evaluation of the hybrid scheme. All other schemes discussed so far only use two supply voltages per cache line.

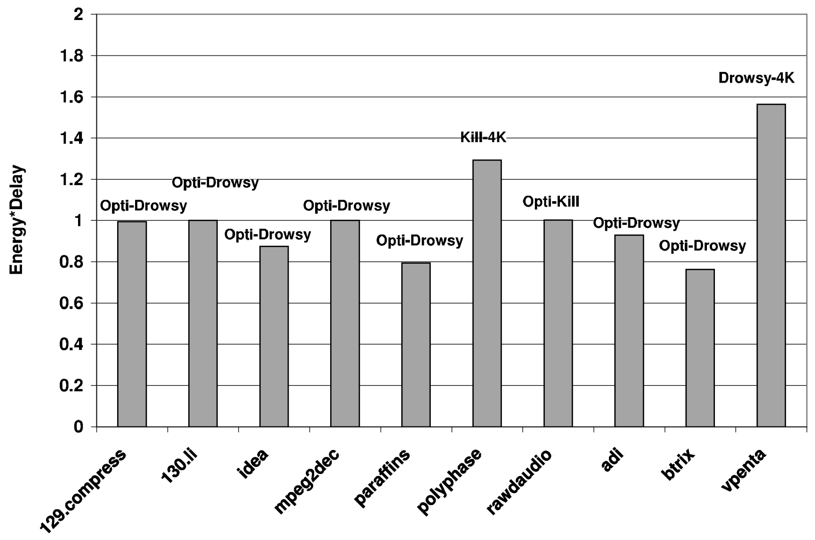


Fig. 14. Normalized energy-delay product for the hybrid strategy.

5. IMPACT OF COMPILER OPTIMIZATIONS

While evaluating a given leakage control mechanism, it is also critical to quantify its behavior under different code optimizations. This is important not only because many compiler optimizations (especially those targeting at improving data locality) can modify the instruction execution order (sequence) dramatically leading to significantly different energy picture, but also because if we can characterize the impact of such optimizations on the effectiveness of the proposed mechanism, this information can be fed-back to compiler writers, leading to better (e.g., energy-aware) compilation strategies.

In this section, we first give a qualitative assessment of two frequently used loop transformation strategies, loop fission (distribution) and loop fusion. The loop distribution transformation cuts the body of a for-loop statement in two parts [Wolfe 1996]. The first statement of the second part specifies the cut position. It is generally used for enhancing iteration-level parallelism (by placing statements with dependence sources into one loop and the ones with dependence sinks into the other), for improving instruction cache behavior (by breaking a very large loop body into smaller, manageable sub-bodies with better instruction cache locality), and even for improving data cache locality (by separating the statements that access arrays that would create conflicts in the data cache). Different optimizing compilers can employ this transformation for one or more of these reasons. A typical loop distribution algorithm that targets parallelism would proceed as follows. The dependence graph is structured into strongly connected components, each of which is recursively analyzed with an incremented dependence level. Each strongly connected component can then be checked to see whether it can be parallelized. Depending on whether the target architecture has a vector facility or not, it may be possible to replace parallel loops that have more than one assignment statement in their bodies by a set

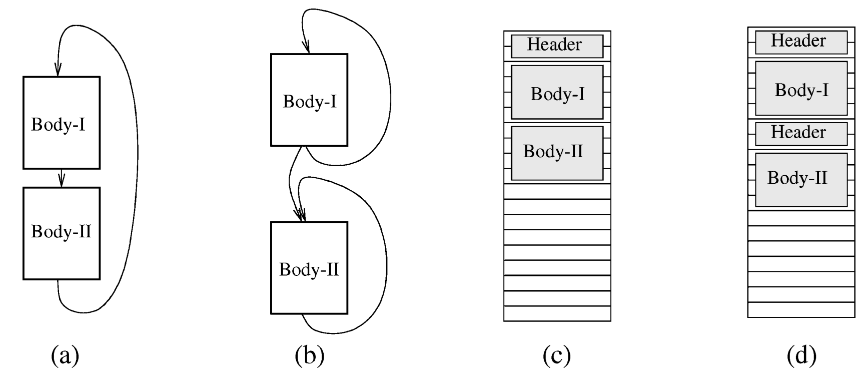


Fig. 15. (a) A code fragment with a loop. (b) The distributed version of (a). (c) The instruction cache layout for (a). (d) The instruction cache layout for (d).

of single assignment loops. These simple loops can then be replaced by vector instructions by the code generator.

As an example, let us consider the fragment shown in Figure 15(a). If we distribute the outermost loop over the two groups of statements (denoted Body-I and Body-II in the figure), we obtain the fragment depicted in Figure 15(b). Figures 15(c) and (d), on the other hand, illustrate how the instructions in the fragments in Figures 15(a) and (b), respectively, would map to the instruction cache. The figure is used only for illustrative purposes and masks the details. In Figures 15(c) and (d), Header is the loop control code. Note that in the distributed version, Header is duplicated. Now, let us try to understand how this optimization would influence the effectiveness of our leakage optimization strategies. First, let us focus on Figure 15(c). During execution all three blocks (Header, Body-I, and Body-II) need to be accessed very frequently, and there will be little opportunity (or energy benefit) in placing the cache lines in question into leakage control mode. If we consider the picture in Figure 15(d), on the other hand, when we are executing the first loop only the first Header and Body-I need to be activated. The second Header and Body-II can be kept in a leakage saving mode. Similarly, when we move to the second loop, during execution, only the second Header and Body-II need to be activated. Therefore, at any given time, the distributed alternative leads to the activation of fewer cache lines. However, the number of cache lines occupied by the code is one part of the big picture. Since we are focusing on the leakage energy consumption, we also need to consider the execution time. If, in this code fragment, data cache locality is a problem, then the first alternative (without distribution) might have shorter execution time if loop distribution destroys data cache locality. Consequently, although the alternative in Figure 15(d) will occupy fewer cache lines at a given time, it will keep those cache lines in the active mode for a longer duration of time. Consequently, there is a trade-off here between the number of cache lines occupied and the time duration during which they are active.

A similar trade-off exists when we consider another loop-level optimization: loop fusion. This optimization is the reverse of loop distribution. Specifically, it takes two neighboring loops and combines their loop bodies into a single

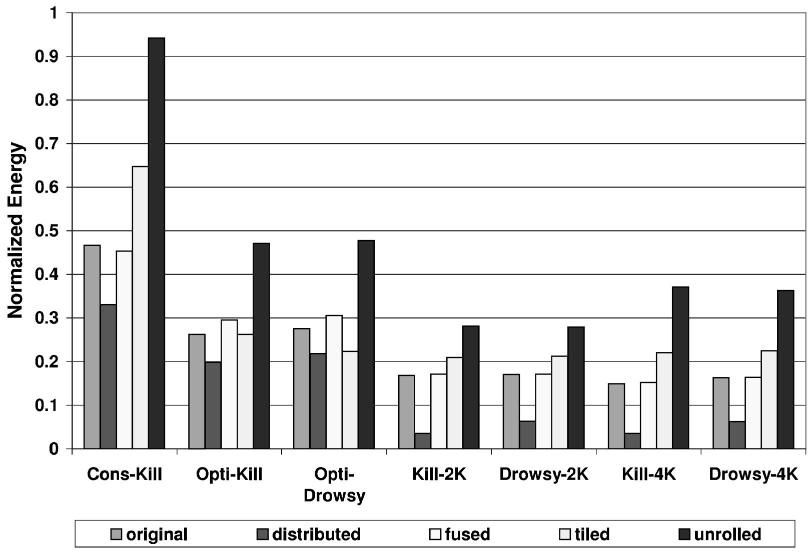


Fig. 16. Instruction cache leakage energy impact of optimizations on vpenta. All values are normalized with respect to the case without power management.

loop (e.g., going from the code in Figure 15(b) to the code in Figure 15(a)). It is generally used for enhancing data cache locality by bringing the statements that access the same set of data to the same loop [Wolfe 1996]. In our context, applying this optimization will increase the number of cache lines active at a given time. On the other hand, it might also reduce the time duration during which these cache lines are active.

In fact, the preceding discussion can be generalized to other data locality optimizations as well. Many optimizations that target at enhancing data cache performance increase code size (i.e., they reduce instruction reuse). Consequently, during the course of execution, at any given time, larger number of cache lines will be active (as a result of the optimization). However, if successful, these optimizations will also reduce the number of execution cycles (hence, the cycles in which the cache lines are active). Iteration space tiling [Lam et al. 1991; Wolfe 1996] is a typical example of that. In tiling, a loop is broken into two loops, and the order of accesses within the array is modified. In most cases, this also leads to a larger code size and reduced instruction reuse. In this section, we evaluate the impact of several data locality-oriented compiler optimizations using two of our applications.

Figures 16 and 17 show, respectively, the instruction cache leakage energy and performance behavior of different versions of vpenta. When we consider the performance behavior, we see that the tiled version generates the best code. In fact, the tiled code outperforms all the other versions in all leakage optimization strategies experimented. An interesting result here is that the distributed (loop-fissioned) version outperforms the original code. This is because placing arrays with conflicting references into separate loops reduces L2 conflict misses. When we look at the energy results, we see that loop distributed version has the best

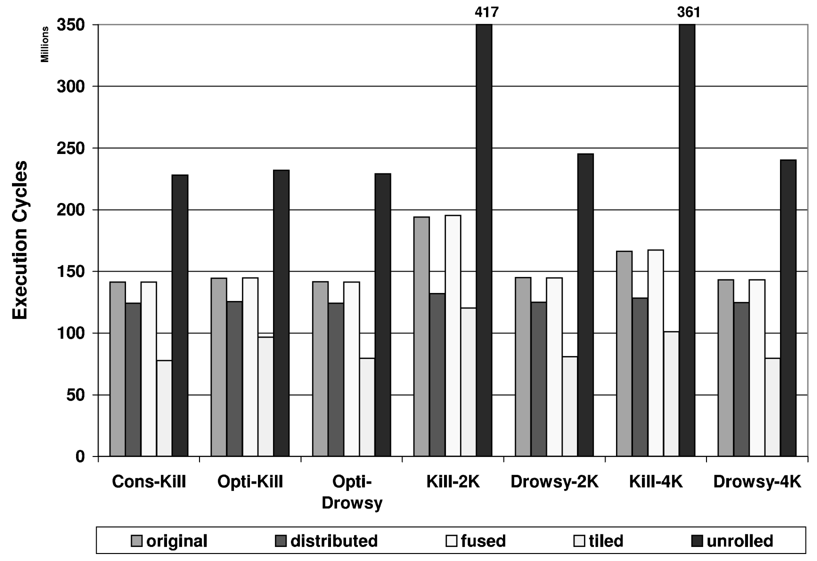


Fig. 17. Performance impact of optimizations on vpenta. Y-axis is expressed in million cycles.

energy behavior. This is because, as compared to the other optimized versions, at any given time the distributed version has fewer number of active cache lines. As a matter of fact, its energy behavior is so good that when one looks at the energy-delay product results, it was observed that in six of seven optimization strategies, it outperforms the tiled version.

Next, we focus on adi. As can be seen from the results given in Figure 19, only loop fusion is useful for adi.[2] When we look at the (instruction cache) energy results (Figure 18), however, the picture changes. As compared to the original code, the fused code incurs much larger energy consumption (except for strategy Opti-Kill). This is because at each iteration of the fused loops we need to activate more cache lines (i.e., cache lines are not held in the leakage control modes for a long enough duration of time). In contrast, the loop distributed version has a very good instruction cache energy consumption. To see the combined impact of both energy and performance, we also evaluated the energy-delay products for this benchmark. We found that as far as the fused version is concerned, the energy losses cancel out the performance benefits in most cases, and the fused code and the original code exhibit very similar energy-delay behaviors. This trade-off clearly emphasizes the importance of considering both energy and performance in deciding whether to apply a compiler optimization or not. We also observed that in four strategies the loop distributed version has the best energy-delay product (as a result of its good energy behavior).

[2]It should be mentioned that we are not trying to come up with the most appropriate use of these compiler optimizations. There might be several reasons why a compiler optimization may not perform as expected. For example, selection of tile size is very critical for the effectiveness of loop tiling [Lam et al. 1991]. A wrong tile size can lead to increased execution time. Similarly, unrolling factor is a very critical parameter in loop unrolling [Carr et al. 1996]. In this work, we have used these optimizations without trying to tune their parameters.

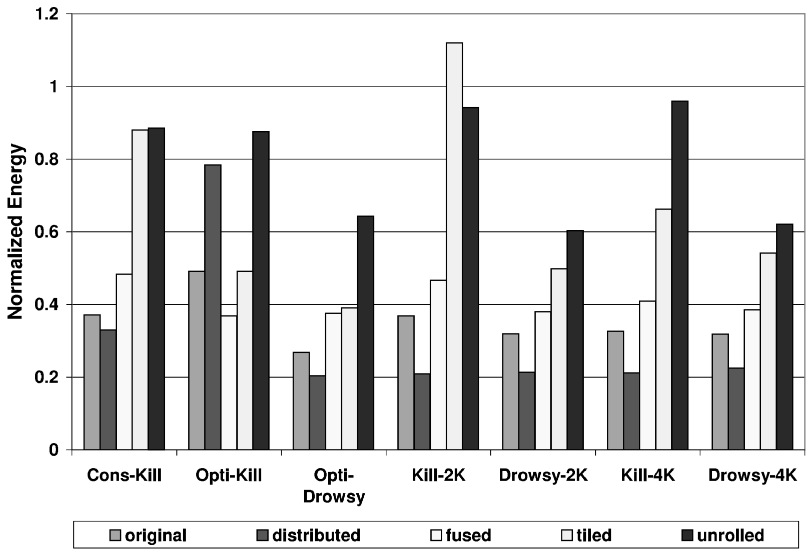


Fig. 18. Instruction cache leakage energy impact of optimizations on adi. All values are normalized with respect to the case without power management.

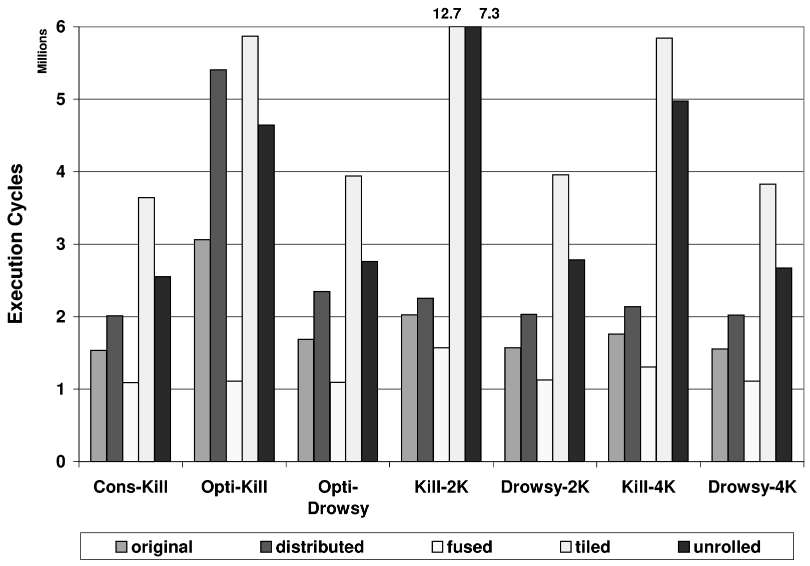


Fig. 19. Performance impact of optimizations on adi.

6. INFLUENCE ON SOFT ERROR RATES

Technology scaling has made it important for one to consider reliability along with energy and performance optimizations. Further, many energy optimizations have a negative impact on reliability of the system. While the previous sections have looked at the influence of our technique on energy and performance behavior, it also has implications on reliability. Consequently, in this

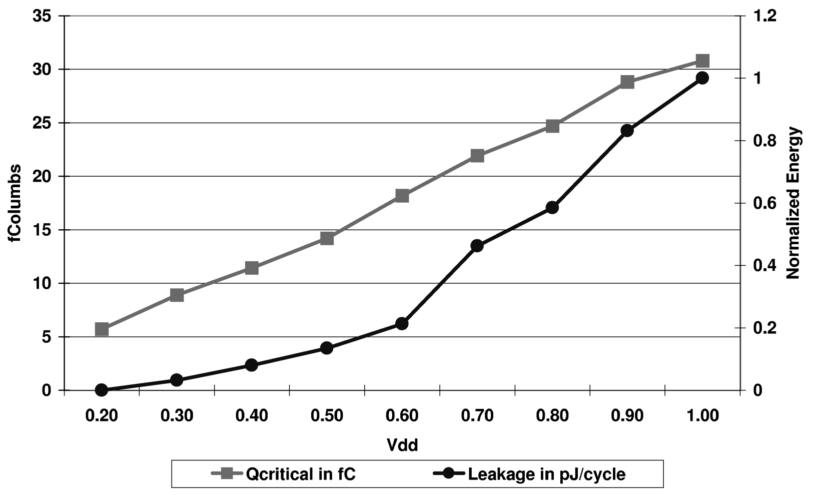


Fig. 20. Leakage energy versus Q_{critical} variation with supply scaling.

section, we examine the effect of reducing supply voltages for leakage energy savings on soft error rate (SER) in the caches.

Soft errors [Hareland et al. ; Hazucha and Svensson 2000; Seifert et al. 2001; Sivakumar et al. 2002] are circuit errors caused due to excess charge carriers induced primarily by external radiations. Radiation directly or indirectly induces a localized ionization capable of upsetting internal data states. Thus, soft errors are undesirable for memory elements as they flip the bit values stored. For a soft error to occur at a specific node in a circuit, the collected charge Q at that particular node should be more then a value defined as critical charge, Q_{critical}. This concept of critical charge is generally used to estimate the sensitivity of SER. In Hazucha and Svensson [2000], a method to estimate the soft error rate in CMOS SRAM circuits was developed. The model was verified for 600 nm design and was scalable. In this model, an exponential dependence of SER on critical charge was shown as $SER\ \alpha\ N_{\text{flux}}.CS.\exp(Q_s/Q_{\text{critical}})$, ($\alpha$ denotes proportional to) where N_{flux} is the intensity of the neutron flux, CS is the area of the cross section of the node, and Q_s is the charge collection efficiency (it is strongly dependent on doping). Q_{critical} is proportional to the node capacitance and the supply voltage and can also represented as $Q_{\text{critical}} = \int_0^{T_f} I_d\ \delta t$, where I_d is the drain current induced by the ion, T_f is the flipping time which is defined as the point in time when the feedback mechanism of the back-to-back inverter will take over from the incident ion's current. To investigate the effect on Q_{critical} due to change in the supply voltage, the parasitic capacitance was extracted from the layout of the SRAM cell used in our cache line. The particle strikes were modeled with a piecewise linear current waveform to account for funneling and diffusion charge collection. Q_{critical} was calculated by numerically integrating the drain current obtained through circuit simulation for different supply voltages. The corresponding T_f was found by simulation.

The results of the simulation are plotted in Figure 20. The Q_{critical} has linear relation with the supply voltage. The leakage of the SRAM array was also

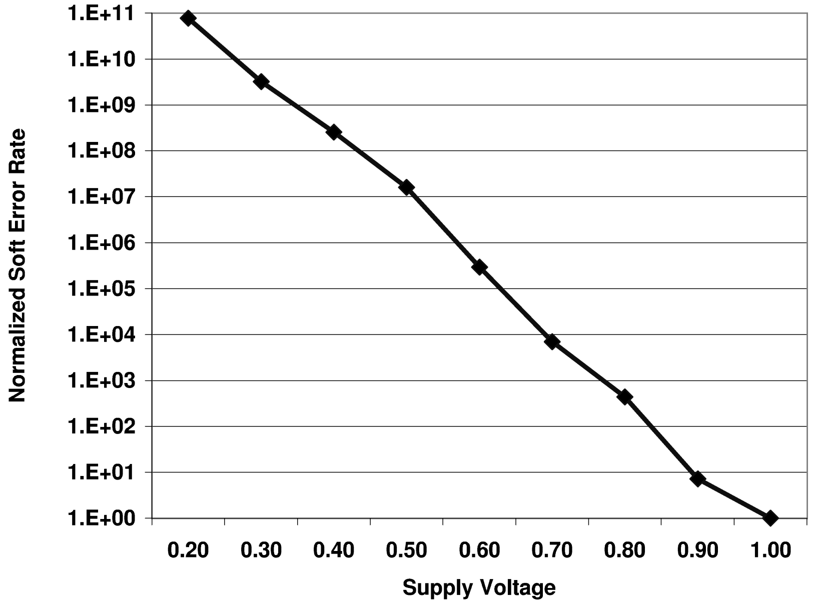


Fig. 21. Soft error rate (per bit) variation with supply scaling. Soft error rates are normalized with respect to the soft error rate at 1 V operation.

simulated using the same cell for different supply voltages. These results are also shown in Figure 20. It can be seen that even though there is a large reduction in leakage energy but there is a correspond loss of immunity to soft errors. Now for the same circuit and different supply voltages all parameters influencing SER are the same except Q_{critical}. This impact on SER is shown in Figure 21. Here we can see that for the same circuit and neutron flux, there is an exponential increase in the SER for reduction in supply voltage. Thus, leakage energy savings need to be balanced with concerns of reliability.

In order to investigate this further, let us consider a single SRAM cell from our cache. Its SER expressed in FIT/bit is 2.5E−05 (Failures in time, FIT, unit expresses the number of errors in 10^9 h). This implies that there is one error/bit for every 4E+13 h of operation at 1.0 V supply voltage. Now considering the same cell operating at the state-preserving mode at VDD=0.3 V and using the observations from Figure 21, we find that the error rate increases to one error per 2000 hours of operation. This would imply that a 32 byte cache line can have one soft error for every 7.8 h of operation. While a single bit error can be easily handled using ECC hardware commonly found in many current off-chip memory systems, multiple error bits can pose a serious problem. A single particle strike upsetting multiple bits can be avoided by intelligent interleaving of the cache line bits. However, two successive strikes on the same cache line can still be a problem. Since a word is checked and corrected only when it is read, if there are two or more particle strikes between two successive read operations, they may cause multiple bit errors. A simple solution to this problem is to periodically read all the bits of the memory forcefully or to periodically invalidated certain cache lines that are not used often. In fact, the second option is attractive as the state-destroying mode of power control may be preferable to state-preserving when the duration between accesses for the cache line are many hours.

Thus, we anticipate that this leakage control mode will be viable even considering the adverse impact of soft errors. Our simulation results also reiterate earlier observations performed in Flautner et al. [2002] that the cells in state-preserving leakage mode are stable when adjacent bitlines or wordlines are swinging.

7. CONCLUSIONS AND FUTURE WORK

This work presents a new approach to controlling leakage energy using the compiler to insert power mode instructions that control the supply voltage for the cache lines. This work is in contrast to prior techniques that have focused on hardware-based schemes. Also, one of our approaches dynamically chooses between state-preserving and state-destroying leakage modes. The experimental evaluation using a set of benchmarks indicates that the proposed compiler-based approach is competitive in terms of energy and energy delay as compared to one of the recently proposed hardware-based leakage control schemes. Further, our analysis reveals that compiler optimizations can have a significant impact on the effectiveness of the leakage control mechanisms. Finally, we quantitatively evaluated the increased soft error rates due to the voltage scaling technique used in our state-preserving leakage control mechanisms and conclude that soft error rates will potentially not impede the adoption of this technique.

As part of our future work, we plan to devise compiler optimizations to increase effectiveness of leakage control modes. The current work on instruction caches can also be extended to data caches. We are in the process of extending the compiler-based strategy for controlling leakage in the data cache. Further, we plan to integrate the hardware-based and compiler-based strategies in a unified optimization framework.

ACKNOWLEDGMENTS

This work was supported in part by a grant from DARPA/MARCO GSRC, NSF CAREER Awards 0093082 and 0093085, and NSF awards 0082064, 0103583, and 0139143. We thank our industrial colleagues from Intel for their input during this project.

REFERENCES

Berkeley predictive technology model. http://www-device.eecs.berkeley.edu.

Trimaran home page. http://www.trimaran.org.

Azizi, N., Moshovos, A., Najm, F., and Falsafi, B. 2002. Exploiting bit value bias to reduce leakage in deep submicron high-performance caches. Tech. rep., Computer Group, ECE, University of Toronto.

Butts, A. and Sohi, G. 2000. A static power model for architects. In *Proceedings of the International Symposium on Microarchitecture*.

Cao, Y., Tomiyama, H., Okuma, T., and Yasuura, H. 2002. Data memory design considering effective bitwidth for low-energy embedded systems. In *Proceedings of the IEEE/ACM International Symposium on System Synthesis*.

CARR, S., DING, C., AND SWEANY, P. 1996. Improving software pipelining with unroll-and-jam. In *Proceedings of the 29th Annual Hawaii International Conference on System Sciences*. Maui HI, 183–192.

CATALDO, A. 2001. Intel does about-face on soi, backs high-k dielectric. *EE Times*.

CHANG, P. P., WARTER, N. J., MAHLKE, S., CHEN, W. Y., AND HWU, W.-M. W. 1991. Three superblock scheduling models for superscalar and superpipelined processors. Tech. Rep. CRHC-91-29, University of Illinois, Urbana, IL.

DAS, K. K. AND BROWN, R. B. 2003. Ultra low-leakage power strategies for sub-1 v VLSI: Novel circuit styles and design methodologies for partially depleted silicon-on-insulator (PD-SOI) CMOS technology. In *Proceedings of the 16th International Conference on VLSI Design (VLSI'03)* (New Delhi, India).

DER MEER, P. R. V. AND STAVEREN, A. V. 2000. Standby-current reduction for deep sub-micron vlsi cmos circuits: smart series switch. In *Proceedings of the ProRISC/IEEE Workshop*, 401–404.

DOYLE, B. ET AL. 2002. Transistor elements for 30nm physical gate lengths adn beyond. *Intel Technology Journal 6*, 2 (May).

FLAUTNER, K., KIM, N., MARTIN, S., BLAAUW, D., AND MUDGE, T. 2002. Drowsy caches: Simple techniques for reducing leakage power. In *Proceedings of the 29th International Symposium on Computer Architecture* (Anchorage, AK).

GUINDI, R. S. AND NAJM, F. N. 2003. Design techniques for gate-leakage reduction in cmos circuits. In *IEEE International Symposium on Quality Electronic Design (ISQED)* (San Jose, CA).

HARELAND, S., MAIZ, J., ALAVI, M., MISTRY, K., WALSTA, S., AND CHANGHONG, D. Impact of cmos process scaling and soi on the soft error rates of logic processes. In *Proceedings of Symposium on VLSI Technology, Digest of Technical Papers*, 73–74.

HAZUCHA, P. AND SVENSSON, C. 2000. Impact of cmos technology scaling on the atmospheric neutron soft error rate. *IEEE Transactions on Nuclear Science 47*, 6 (Dec.).

HEO, S., BARR, K., HAMPTON, M., AND ASANOVIC, K. 2002. Dynamic fine-grain leakage reduction using leakage-biased bitlines. In *Proceedings of the 29th International Symposium on Computer Architecture*, (Anchorage, AK).

HU, Z., JUANG, P., DIODATO, P., KAXIRAS, S., SKADRON, K., MARTONOSI, M., AND CLARK, D. W. 2002. Managing leakage for transient data: Decay and quasi-static 4t memory cells. In *Proceedings of International Symposium on Low Power Electronics and Design*, (Monterey, CA), 52–55.

KAWAGUCHI, H., NOSE, K., AND SAKURAI, T. 2000. A super cut-off cmos scheme for 0.5v supply voltage with pico-ampere standby current. *Journal of Solid-State Circuits 35*, 10 (Oct), 1498–1501.

KAXIRAS, S., HU, Z., AND MARTONOSI, M. 2001. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proceedings of the 28th International Symposium on Computer Architecture*, (Sweden).

KIM, N., FLAUTNER, K., BLAAUW, D., AND MUDGE, T. 2002. Drowsy instruction caches: leakage power reduction using dynamic voltage scaling and cache sub-bank prediction. In *Proceedings of the 35th Annual IEEE/ACM Symposium on Microarchitecture*.

KURODA, T. AND SAKURAI, T. 1996. Threshold-voltage control schemes through substrate-bias for low-power high-speed cmos lsi design. *Journal of VLSI Signal Processing Systems 13*, 2/3 (Aug.), 191–201.

LAM, M. S., ROTHBERG, E. E., AND WOLF, M. E. 1991. The cache performance and optimizations of blocked algorithms. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*.

LEE, D., KWONG, W., BLAAUW, D., AND SYLVESTER, D. 2003. Analysis and minimization techniques for total leakage considering gate oxide leakage. In *Proceedings of the 40th conference on Design automation*. Anaheim, (CA, USA).

LI, L., KADAYIF, I., TSAI, Y.-F., VIJAYKRISHNAN, N., KANDEMIR, M., IRWIN, M. J., AND SIVASUBRAMANIAM, A. 2002. Leakage energy management in cache hierarchies. In *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques* (Charlottesville, VA).

MUCHNICK, S. S. 1997. *Advanced Compiler Design Implementation*. Morgan Kaufmann, San Francisco, CA.

MUTOH, S. ET AL. 1995. 1-V power supply high-speed digital circuit technology with multi-threshold-voltage cmos. *IEEE Journal of Solid State Circuits 30*, 8 (Aug.), 847–854.

POWELL, M. D., YANG, S., FALSAFI, B., ROY, K., AND VIJAYKUMAR, T. N. 2001. Reducing leakage in a high-performance deep-submicron instruction cache. *IEEE Transactions on VLSI 9*, 1 (Feb.).

SEIFERT, N., MOYER, D., LELAND, N., AND HOKINSON, R. 2001. Historical trend in alpha-particle induced soft error rates of the alpha microprocessor. In *Proceedings of the 39th Annual International Reliability Physics Symposium*. 259–265.

SIVAKUMAR, P., KISTLER, M., KECKLER, S. W., BURGER, D. C., AND ALVISI, L. 2002. Modeling the effect of technology trends on the soft error rate of combinational logic. In *International Conference on Dependable Systems and Networks*.

WOLFE, M. 1996. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Reading, MA.

ZHANG, W., HU, J. S., DEGALAHAL, V., KANDEMIR, M., VIJAYKRISHNAN, N., AND IRWIN, M. J. 2002. Compiler-directed instruction cache leakage optimization. In *Proceedings of the 35th Annual International Symposium on Microarchitecture* (Istanbul, Turkey).

ZHANG, W., VIJAYKRISHNAN, N., KANDEMIR, M., IRWIN, M. J., DUARTE, D., AND TSAI, Y. 2001. Exploiting vliw schedule slacks for dynamic and leakage energy reduction. In *Proceedings of the 34th Annual International Symposium on Microarchitecture* (Austin, TX).

ZHOU, H., TOBUREN, M. C., ROTENBERG, E., AND CONTE, T. M. 2001. Adaptive mode control: a static power-efficient cache design. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*.

Received November 2003; revised December 2003; accepted January 2004