

Compiler-Directed Instruction Cache Leakage Optimization

W. Zhang, J. S. Hu, V. Degalahal, M. Kandemir, N. Vijaykrishnan, M. J. Irwin
Microsystems Design Lab
Pennsylvania State University
University Park, PA 16802
mdl@cse.psu.edu

Abstract

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.

1. Introduction

With the increasing number of transistors employed in current microprocessors and the continued reduction in threshold voltages of these transistors, leakage energy consumption has become a major concern [1]. 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 [14].

There have been several efforts [9, 13, 16, 7, 14, 20] 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 [14] 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 [8]. This technique monitors the periods of inactivity in cache lines by associating saturating counters with them. In [20], 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 [5]. 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. [6] 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 [8] 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 [5] 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 [5] using 0.07 micron technology [2]. 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 tradeoffs when optimizations that target data locality are applied.

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 [15], 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. Finally, we present our conclusions in Section 6.

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 is retained or not. When the normal supply voltage of 1.0V is reduced below 0.3V (for a 0.07 micron process), we observe that the data in the cells are no longer retained. Thus, we select a 0.3V supply voltage for the state-preserving leakage control mode. However, if state preservation is not a

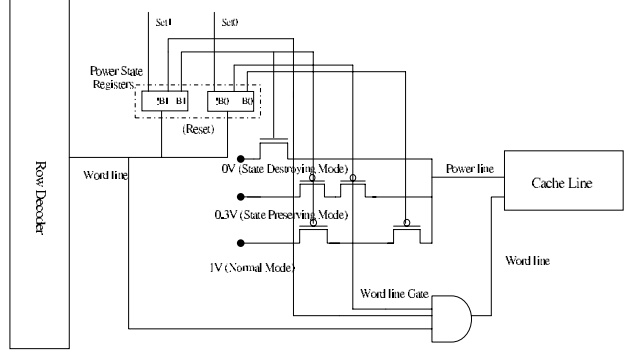


Figure 1. Leakage control circuitry.

consideration, we switch the supply voltage to 0V 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 [5]. 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 0.3V (0.0V) 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 1.0V 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 are set to 1.0V, 0.3V or 0V, respectively, for the three modes when using caches designed with 0.07 micron Berkeley predictive technology [2]. 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 1.0V. 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

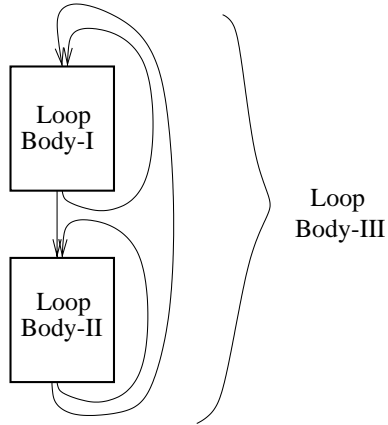


Figure 2. A code fragment that contains three loops.

instruction to place cache lines in state-preserving or state-destroying mode. This is in contrast to the approach used in [5] where a periodic timer is used to place all cache lines in a state-preserving (drowsy) mode. We refer to the variants of the scheme proposed in [5] 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*.

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. 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.

The conservative strategy is based on determining the last usage of instructions. 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 turn-off instructions inserted in the code. Also, such frequent turn-off instructions themselves would consume considerable dynamic energy. Consequently, in this work, we turn off instructions at the *loop granularity level* (We treat the streamline code as a loop which iterates once). More specifically, when we exit a loop and we know for sure that this loop will *not* be visited again, we turn off the

cache lines that hold the instructions belonging to the loop. While ideally we would want to issue turn-offs 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 which itself would consume energy, or a software support which would be very slow). As a result, in this work, when we exit the loop, we turn off *all* cache lines.

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, we *cannot* turn off the cache lines occupied by it; because, there is an outer loop that will re-visit this loop body; 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 turn off 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.

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 turn off 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 turn off 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 turn off 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

¹While 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 non-array applications) take a long time to execute.

	Conservative	Optimistic
State-Destroying	Strategy I	Strategy II
State-Preserving	Strategy IV	Strategy III

Figure 3. Four different implementation choices depending on the leakage control mechanism (mode) used and the compiler strategy employed.

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.

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 Figure 3. Among the choices we have, Strategy IV does not make much sense since being conservative means that we do not turn off 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: I, II, and III, and compare them with fixed period strategies Kill-M and Drowsy-M [5]. 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 turn-off 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) [15]. A program flows through IMPACT, Elcor, and the cycle-level simulator. IMPACT applies 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. Further, the increase in code size due to the inserted turn-off 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 Figure 4. The energy values reported are based on circuit simulation. In our evaluations, we report results using the basic block based scheduling.

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 Figure 5. The benchmark source is indicated in the second column. The third column in this figure 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 rawaudio are typical media applications. The last three benchmarks (adi, btrix, 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 due to executing turn-off instructions. Since some of our strategies increase execution cycles, the extra leakage energy consumption 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 turn-offs/ons and due to any additional L1 instruction cache accesses. 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 6 the percentage time that cache lines spend in leakage control mode for different optimiza-

Parameter	Value
Feature size	0.07 micron
Supply voltage	1.0V
L1 instruction cache	16KB direct-mapped cache
L1 instruction cache latency	1 cycle
L1 data cache	32KB 2-way cache
L1 data cache latency	1 cycle
Unified L2 cache	512KB 4-way cache
L2 cache latency	10 cycles
Memory latency	100 cycles
Clock speed	1GHz
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 (excluding miss latency)
L1 dynamic energy per access	0.11nJ
L2 dynamic energy per access	0.58nJ

Figure 4. Default parameters used in our simulations.

Benchmark	Source	Lines	Input	Execution Cycles	Instr. Cache Energy (nJ)
129.compress	SpecInt95	1939	test.in	42784111	13627628 (53%)
139.li	SpecInt95	7597	train.lsp	918252701	230649411 (67%)
idea	Mediabench	1232	/	335180	97343 (58%)
mpeg2dec	Mediabench	9832	mei16v2.m2v	140735320	46702867 (51%)
paraffins	Trimaran Distribution	388	/	523363	111058 (79%)
polyphase	Mediabench	542	polyphase.IN	587442	181888 (54%)
rawdaudio	Mediabench	314	clinton.adpcm	7479483	2870793 (44%)
adi	Livermore	46	274.68MB	1490229	435725 (58%)
btrix	Specfp92	135	202.53MB	27056699	6337571 (72%)
vpenta	Specfp92	114	14.42MB	141445594	29645742 (81%)

Figure 5. 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.

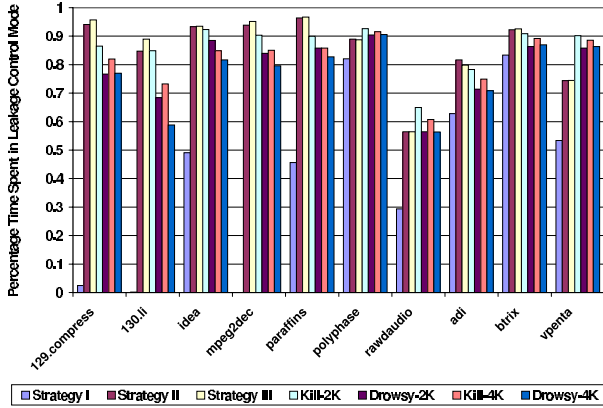


Figure 6. Percentage of times where cache lines are in leakage control mode.

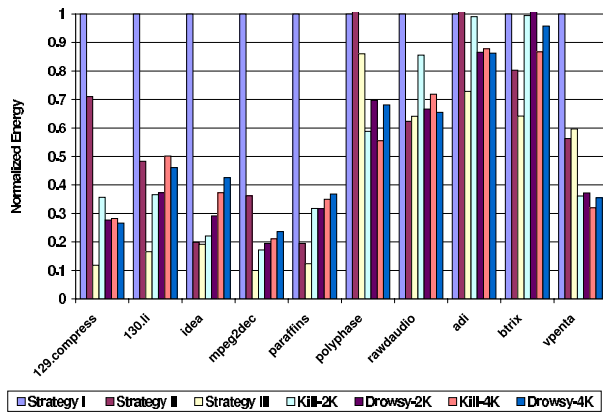


Figure 7. Normalized (w.r.t. Strategy I) energy consumption with different leakage saving strategies.

tion strategies. One thing to note in this graph is that, for some versions, this time is very high. Specifically, the percentage of time that cache lines are in a leakage control mode for Strategy-I, Strategy-II, Strategy-III, 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. Since only a hypothetical optimal strategy would achieve 100% execution time spent in the leakage control mode, we believe that some of our strategies perform really well in placing cache lines into leakage control modes.

3.3. Energy and Performance Results

Figure 7 shows the energy consumptions of the strategies (given as fractions of the energy consumption of Strategy I). Note that though not given in the graph, Strategy I 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 I 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 Strategy III and Strategy II (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 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 I does not perform well because it acts very late in turning off cache lines. While Strategy II 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 III also turns off cache lines quickly (after each inner loop); however, it uses state-preserving leakage mode. In contrast to Strategy II, this scheme does not have the additional overhead of instruction cache misses. Moreover, Strategy III can obtain most of the leakage savings provided by Strategy II (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 Strategy II and Strategy III will not achieve very good results as it will take a long time before they can turn off the corresponding cache lines. However, the fixed period strategies can turn off the cache lines in such a loop (maybe several times depending on the execution time).

Figure 8 divides the energy consumption in the instruction cache into different components for Strategies II and III. For Strategy II, 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 III, 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

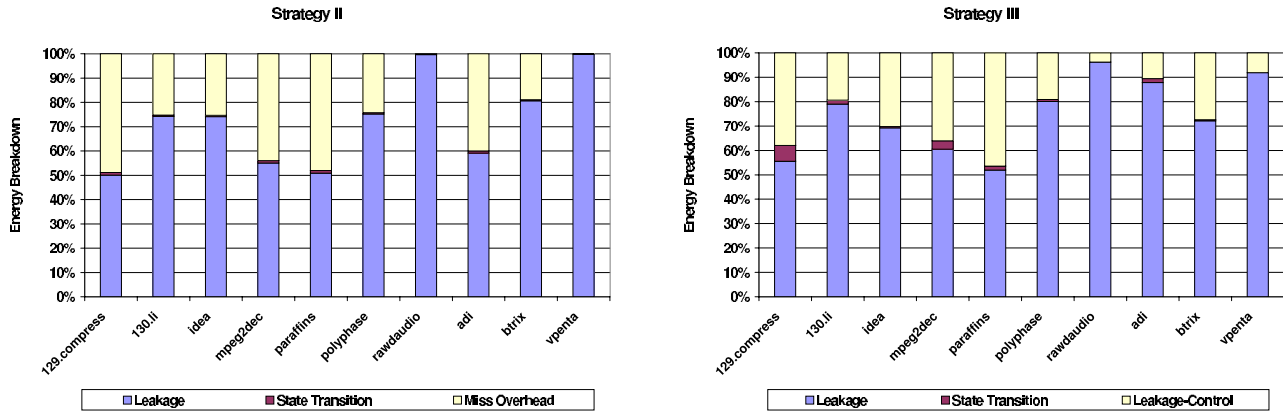


Figure 8. Energy breakdown for Strategy II (left) and Strategy III (right).

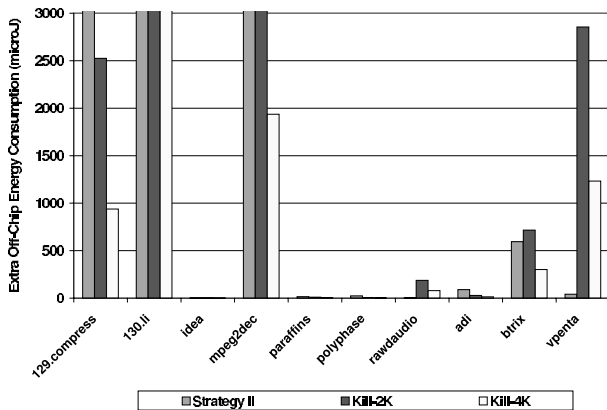


Figure 9. Extra dynamic energy consumption in off-chip L2 and memory. Only the strategies that employ state-destroying mode incur this penalty.

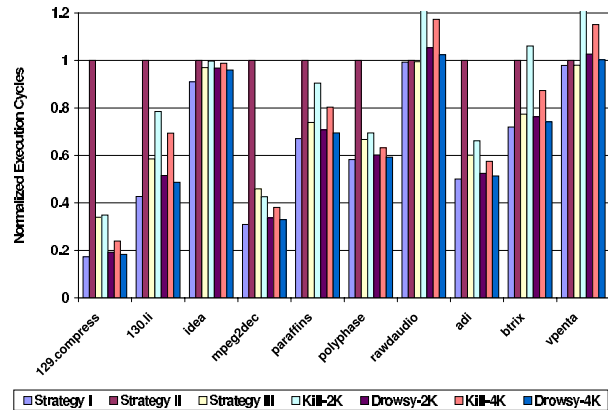


Figure 10. Normalized (w.r.t. Strategy II) number of execution cycles with different leakage saving strategies.

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 in this work we focus on on-chip energy consumption, it is also important to know the magnitude of this off-chip energy. Figure 9 gives the extra dynamic energy consumption in the off-chip L2 and memory. It should be seen that among our strategies only Strategy II can cause extra off-chip energy consumption. This is because Strategy I kills the contents of a cache line if and only if it is already dead and Strategy III only employs state-preserving mode. Therefore, Strategy III becomes even more preferable when considering off-chip L2 and memory energy.

Obviously, energy behavior is only part of the picture. To have a fair evaluation of all strategies considered, we need to look at their performance behavior as well. The normalized execution cycles with our base configuration are presented in Figure 10. All values are normalized with respect to that of Strategy II as it is the one which takes the longest time in most of the cases (It should be noted

that the performance degradation of Strategy I is within 1% of the original case). 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 of turn-ons for Strategy I, Strategy II, Strategy III, Kill-2K, Drowsy-2K, Kill-4K and Drowsy-4K are 1448, 22777521, 22777521, 14096265, 10968055, 10086726, and 7428348, respectively. The number of turn-ons in our schemes is typically larger than that of the fixed period schemes (except Strategy I). Note that the performance penalty for the same number of turn-ons for Strategy II is much larger than that of Strategy III due to L2 access latencies. This clearly shows the tradeoff 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 11 shows the normalized energy-delay products for our applications. We see that Strategy III is very

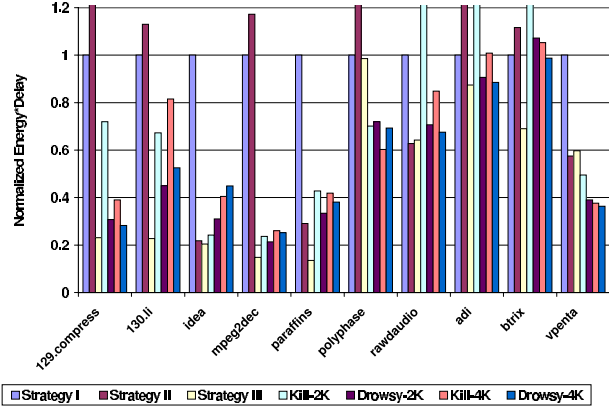


Figure 11. Energy-delay (w.r.t. Strategy I) products with different leakage saving strategies.

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 Strategy III is 0.47 which is 13% better than that of the best fixed-period scheme (Drowsy-2K - 0.54) for the considered applications.

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 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.

The energy-delay product profile of the hybrid strategy is illustrated in Figure 12. 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 Strategy III generate the same energy-delay product. In polyphase and vpenha, 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

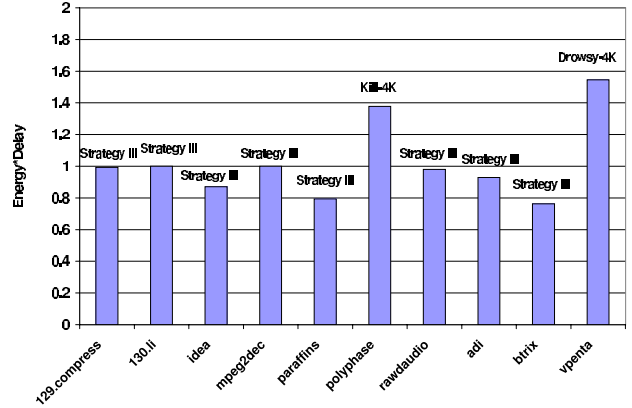


Figure 12. Normalized energy-delay product for the hybrid strategy.

and state-preserving modes. Strategy III 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.

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 [17].

As an example, let us consider the fragment shown in Figure 13(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 13(b). Figures 13(c) and (d), on the other hand, illustrate how the instructions in the fragments in Figures 13(a) and (b), respectively, would map to the instruction cache. In Figures 13(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 13(c). During execution all three blocks (Header, Body-I and Body-II) need to be accessed very frequently, and there will be little opportunity

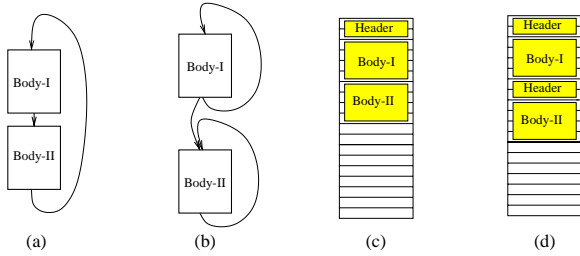


Figure 13. (a) A code fragment with a loop. (b) The distributed version of (a). (c) The instruction cache layout for (a). (d) The instruction cache layout for (b).

(or energy benefit) in placing the cache lines in question into leakage control mode. If we consider the picture in Figure 13(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 13(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 tradeoff here between the number of cache lines occupied and the time duration during which they are active.

A similar tradeoff 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 loop (e.g., going from the code in Figure 13(b) to the code in Figure 13(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 [17]. 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 [17, 10] 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.

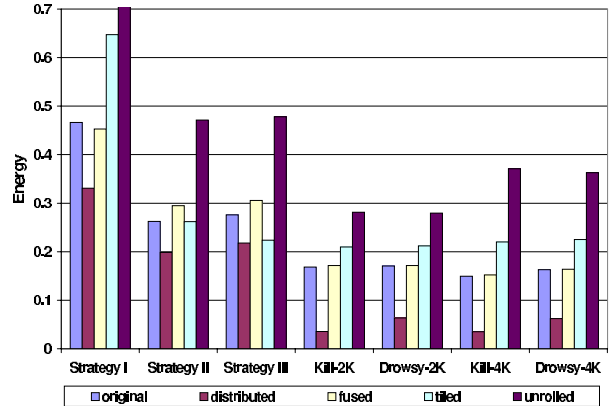


Figure 14. Instruction cache leakage energy impact of optimizations on vpentia. All values are normalized w.r.t. to the case without power management.

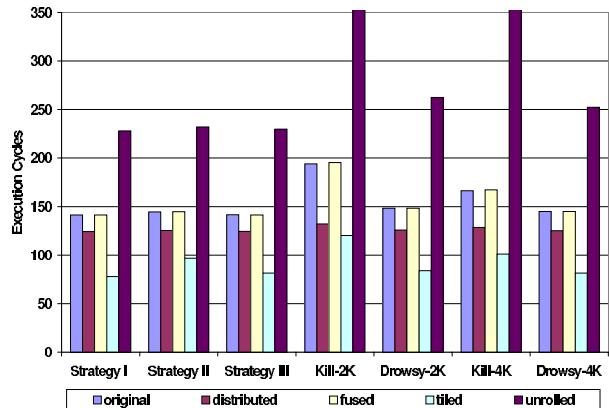


Figure 15. Performance impact of optimizations on vpentia. Y-axis is expressed in million cycles.

Figures 14 and 15 show, respectively, the instruction cache leakage energy and performance behavior of different versions of vpentia. 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 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.

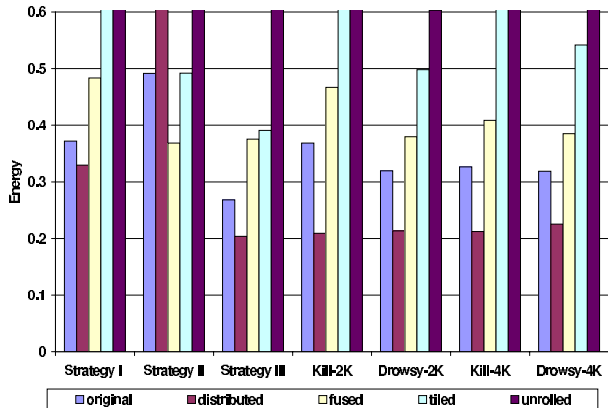


Figure 16. Instruction cache leakage energy impact of optimizations on adi. All values are normalized w.r.t. to the case without power management.

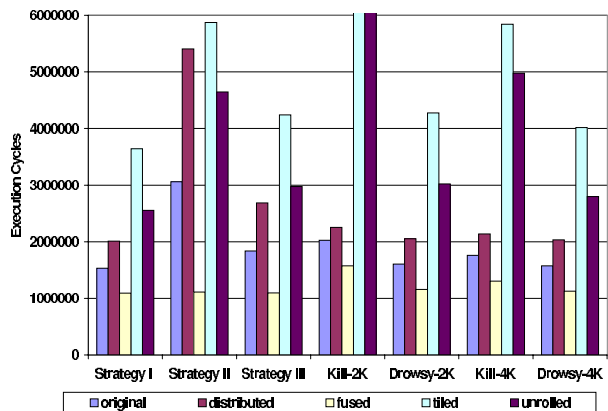


Figure 17. Performance impact of optimizations on adi.

Next, we focus on adi. As can be seen from the results given in Figure 17, only loop fusion is useful for adi.² When we look at the (instruction cache) energy results (Figure 16),

²It 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 [10]. A wrong tile size can lead to increased execution time. Similarly, unrolling factor is a very critical parameter in loop unrolling [3]. In this work, we have used these optimizations without trying to tune their parameters.

however, the picture changes. As compared to the original code, the fused code incurs much larger energy consumption (except for Strategy II). 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 tradeoff 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).

6. Conclusions

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.

Acknowledgment

This work was supported in part by a grant from GSRC, NSF CAREER Awards 0093082 and 0093085, and NSF Awards 0082064, 0103583 and 0139143.

References

- [1] J. A. Butts and G. Sohi. A static power model for architects. In Proc. the International Symposium on Microarchitecture, December 2000.
- [2] Berkeley predictive model. <http://www-device.eecs.berkeley.edu>
- [3] S. Carr, C. Ding, and P. Sweany. Improving software pipelining with unroll-and-jam. In Proc. the 29th Annual Hawaii International Conference on System Sciences, Maui HI, January 1996, pp. 183–192.
- [4] P. P. Chang, N. J. Warter, S. Mahlke, W. Y. Chen, and W. M. W. Hwu. Three superblock scheduling models for superscalar and superpipelined processors. Technical Report

CRHC-91-29, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, December 1991.

- [5] K. Flautner, N. Kim, S. Martin, D. Blaauw, T. Mudge. Drowsy Caches: Simple techniques for reducing leakage power. In Proc. the 29th International Symposium on Computer Architecture, Anchorage, AK, May 2002.
- [6] S. Heo, K. Barr, M. Hampton, and K. Asanovic. Dynamic Fine-Grain Leakage Reduction using Leakage-Biased Bit-lines. In Proc. the 29th International Symposium on Computer Architecture, Anchorage, AK, May 2002.
- [7] H. Kawaguchi, K. Nose and T. Sakurai. A super cut-off CMOS scheme for 0.5V supply voltage with pico-ampere standby current. *Journal of Solid-State Circuits*, Vol 35, No. 10, October 2000, pp. 1498–1501.
- [8] S. Kaxiras, Z. Hu, M. Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. In Proc. the 28th International Symposium on Computer Architecture, Sweden, June 2001.
- [9] T. Kuroda and T. Sakurai. Threshold-voltage control schemes through substrate-bias for low-power high-speed CMOS LSI design. *Journal of VLSI Signal Processing Systems*, Vol. 13, No. 2/3, pp. 191–201, August 1996.
- [10] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In Proc. the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, April 1991.
- [11] C. Lee and M. Potkonjak, and W. H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In Proc. the International Symposium on Microarchitecture, pp. 330–335, 1997.
- [12] S. S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, San Francisco, CA, 1997.
- [13] S. Mutoh et al. 1-V power supply high-speed digital circuit technology with multi-threshold-voltage CMOS. *IEEE Journal of Solid State Circuits*, Vol. 30, No. 8, pp. 847–854, August 1995.
- [14] M. D. Powell, S. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. Reducing Leakage in a High-Performance Deep-Submicron Instruction Cache. *IEEE Transactions on VLSI*, Vol. 9, No. 1, February 2001.
- [15] Trimaran home page, <http://www.trimaran.org>
- [16] P. R. Van der Meer and A. Van Staveren. Standby-current reduction for deep sub-micron VLSI CMOS circuits: smart series switch. In Proc. the ProRISC/IEEE Workshop, pp. 401–404, December 2000.
- [17] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.
- [18] Y. Ye, S. Borkar, and V. De. A new technique for standby leakage reduction in high-performance circuits. In Proc. the Symposium on VLSI Circuits, pp. 40–41, 1998.
- [19] W. Zhang, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, D. Duarte and Y. Tsai. Exploiting VLIW Schedule Slacks for Dynamic and Leakage Energy Reduction. In Proc. the 34th Annual International Symposium on Microarchitecture, Austin, December 2001.
- [20] H. Zhou, M. C. Toburen, E. Rotenberg, and T. M. Conte. Adaptive mode control: a static power-efficient cache design. In Proc. the 2001 International Conference on Parallel Architectures and Compilation Techniques, September 2001.