Instruction Cache Leakage Reduction by Changing Register Operands and Using Asymmetric SRAM Cells

Goudarzi, Maziar
System LSI Research Center, Kyushu University

Ishihara, Tohru
System LSI Research Center, Kyushu University

Instruction Cache Leakage Reduction by Changing Register Operands and Using Asymmetric SRAM Cells

Maziar Goudarzi, Tohru Ishihara
System LSI Research Center, Kyushu University, Fukuoka, Japan
{goudarzi, ishihara}@slrc.kyushu-u.ac.jp

ABSTRACT

Share of leakage in cache memories is increasing with technology scaling. Studies show that most stored bits in instruction caches are zero, and hence, asymmetric SRAM cells which dissipate less leakage when storing 0, effectively reduce leakage with negligible performance penalty. We show that by carefully choosing register operands of instructions, it is possible to further increase the number of 0 bits, and hence, increase leakage savings in instruction cache. This compiler technique is performed off-line and introduces absolutely no delay penalty since processor registers are all the same. Experimental results of our benchmarks show up to 33% (averaging 30.35%) improvement in leakage.

Categories and Subject Descriptors

B.3.1 [**Semiconductor Memories**]: *Static memory (SRAM).* B.3.2 [**Design Styles**]: *Cache memories.* D.3.4 [**Processors**]: *Compilers, Optimization.*

General Terms

Algorithms, Measurement, Design, Experimentation.

1. INTRODUCTION

Cache memories are a major source of power consumption in processor-based embedded systems and consume up to half of total power [6]. With technology scaling, dynamic power reduces but static (leakage) power increases, and hence, leakage comprises an increasingly larger portion of cache power consumption. This motivates techniques to reduce cache leakage power especially when noting that cache memories occupy the largest area in today embedded processors (e.g.70% of StrongARM [6]).

Cache-decay [4] and drowsy caches [2] turn off or put into sleep mode those parts of the cache that are not likely to be accessed by current computation; they, however, cannot reduce leakage of those parts that are being accessed. Another way to reduce cache leakage is to increase threshold voltage (V_{th}), but this slows down the cells unless supply voltage (V_{dd}) is also increased which raises dynamic power consumption. The large bias of stored bits toward zero [7] has been the motive in [1] to design asymmetric SRAM cells that employ higher V_{th} for the transistors that leak when the cell stores 0. Consequently, such asymmetric cells dissipate far less leakage in the 0 state; the delay penalty is minimized by careful design of the sense-amplifiers. We present a software-level technique that needs no change to the above asymmetric-cell cache, but reinforces its benefits in instruction caches.

We change the register operands of instructions in the binary executable of the application such that number of 0 bits is increased in the instruction-cache; since asymmetric SRAM cells dissipate less leakage when storing a 0, this results in less leakage compared to the initial register assignment. This is a low-cost technique that is applied off-line to the binary executable of the applications and reduces the leakage corresponding to register operands up to 33% at no performance cost because all processor registers have the same speed, and moreover, this technique does not affect cache miss ratio. Modifying register operands changes the switching activity on the processor-cache bus and also on cache to off-chip memory bus, and hence, potentially changes the dynamic power; our experiments to evaluate this effect are on-going but their results are not reported in this preliminary work. While dynamic register-renaming is a well known technique in high-performance computing, to the best of our knowledge it has not been used to reduce *leakage* power in the past, although it has previously been proposed for bus *dynamic* power reduction [9].

2. RELATED WORKS AND OUR APPROACH

Several compiler techniques reduce power [3][8][9][11]-[14] but most of them focus on dynamic power, not leakage power. Compiler-inserted special instructions are used in [14] to deactivate (put into low-leakage mode) those cache lines of data-cache whose data are not used by the current computation. But this requires that each cache-line can be individually put in low-leakage mode, and furthermore, the processor core needs to be extended by special instructions to activate/deactivate cache-lines. In addition, it targets only unused cache lines. Assuming the cache is already implemented using asymmetric cells (due to their advantage in leakage power and minimal or no impact on other quality factors), our approach needs no hardware modification while it reduces leakage even in the cache lines being used.

Special instructions for dynamic voltage scaling and adaptive body biasing are inserted by compiler in [3] to reduce total power consumption including leakage. Our work does not need any extra actions or core-control instructions during program execution; it is a one-off task at compile-time needing no run-time action, and hence, imposes no overhead.

Register-renaming is a well known technique in high-performance computing to eliminate false dependence among instructions that otherwise could not have been executed in parallel. It is usually applied at run-time, but we change register operands statically to increase number of zero bits. This has been used to reduce bus switching and reduce dynamic power [9] but to the best of our knowledge, has not been used in the past for leakage reduction.

2.1 Asymmetric SRAM Design

Noting the strong bias toward 0 in bit content of instruction and data caches [7], Azizi et al. proposed to design SRAM cells that dissipate less leakage when storing a 0 [1]. They observed that *(i)* subthreshold leakage is the major contributor to leakage power in SRAM, and *(ii)* disjoint sets of transistors contribute to leakage when storing 0 compared to storing 1; Fig. 1 shows the leakage paths when the cell stores a 0 (note that bit lines are precharged to V_{dd}): M1 has no voltage across its drain to source and M2 and M5 are not in the subthreshold region, and hence, do not leak.

Subthreshold leakage is the main contributor to leakage in cache memories [4][2]. Detailed simulations using BSIM3v3/BSIM4 predictive models of 90nm, 65nm, 45nm, and 32nm technologies also show that gate leakage as well as other leakage components are very low even in 32nm process and that subthreshold leakage is the dominant factor [10].

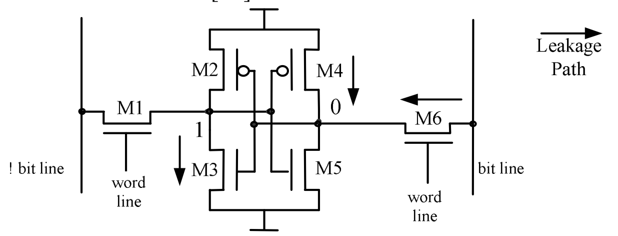

Figure 1. Leakage paths when SRAM cell is storing a 0.

Since subthreshold leakage can be effectively reduced by increasing V_{th}, [1] used higher V_{th} for M3, M4, and M6 in one of their designs to reduce leakage when storing 0; they also modified sense-amplifier to compensate the reduced speed of high-V_{th} transistors. The reduced leakage power is achieved at the cost of a marginal increase in cell delay or its stability. They designed and examined a family of such asymmetric cells in [1]; characteristics of better cells are summarized in Table I [7]. The *Leakage* columns are normalized leakages when storing 0 and 1. Δdelay is the performance penalty. ΔSNM (increase in signal-to-noise margin) and $\Delta I_{trip}/I_{read}$ (current necessary to trip the cell value) are stability measures; in these two columns, a positive percentage suggests an increase in stability. Dynamic power of the cells is the same for 0 and 1 since sizes of cell transistors are symmetric.

Table I. Characteristics of asymmetric cells [7]

	Leakage (%)		Δdelay	ΔSNM	$\Delta I_{trip}/I_{read}$
	(0)	(1)			
Regular V_{th} (symmetric cell)	100	100	0%	0%	0%
Leakage Enhanced	1	14	5%	7%	-5%
Speed Enhanced	14	50	0%	-6%	15%
Stability-Leakage Enhanced	14	43	5%	23%	-7%
Stability-Speed Enhanced	50	53	0%	9%	13%

2.2 Motivational Example

M32R Instruction Format. We implemented our technique on a 32-bit RISC processor, M32R [5], and the motivational example also uses that processor. The instruction word is 32 bits and can consist of either two 16-bit instructions or one 32-bit instruction; left-most bit determines which one is the case. The left register operand designates the destination register. M32R has 16 general purpose registers two of which serve special functionality: R14 is link register for procedure calls and R15 is stack pointer.

Motivational Example. This is a small code excerpt of MPEG2 encoder compiled for M32R showing respectively hex address, instruction encoding, and instruction in assembly-like language:

```
27fe30:  e6 36 d6 68   load r6,36d668 <gptr>
27fe34:  e7 30 88 d4   load r7,3088d4 <image>
27fe38:  27 46 06 a7   store r7,@r6 -> add r6,r7
```

The "->" sign shows the right-hand 16-bit instruction is executed after the left-hand one. The register operands are in bold underlined font in the instruction encoding.

There are 51 zero bits in the above three instructions. Recalling Table I that shows the asymmetric cells dissipate less leakage when storing a 0, leakage can be further reduced if we can increase the number of zero bits. We propose to do this by changing the register operands of instructions. In the above code excerpt, if we change registers r6 and r7 respectively to r0 and r1, total zero bits increase to 63 and instructions change as follows:

```
27fe30:  e0 36 d6 68   load r0,36d668 <gptr>
27fe34:  e1 30 88 d4   load r1,3088d4 <image>
27fe38:  21 40 00 a1   store r1,@r0 -> add r0,r1
```

Note that although number of 0 bits has improved by 24% in this example, the actual reduction in leakage depends on *(i)* difference between leakage when storing a 0 and a 1, *(ii)* the amount of time each instruction resides in cache; depending on the instruction address and the cache configuration, different instructions spend different times in cache. We consider these items in experiments.

Note that neither the instructions nor their addresses change; only usage of registers is changed. Consequently, there is absolutely no penalty in terms of performance by this approach.

2.3 Our Approach

Obviously the amount of leakage saving using asymmetric cells in an instruction cache depends on the number of 0 bits in each instruction. Thus leakage can be further reduced if the instructions are composed of more 0's. Typically, compilers start from register R0 when assigning variables to registers during code generation, and continue to the highest-numbered available register. This is reasonable since traditionally no register is superior to others; however, this approach is not optimal when an instruction-cache is designed with asymmetric cells. For example, the conventional approach respectively uses R0, R1, R2, and R3 for the first four variables, while it is more beneficial to use R4 instead of R3 since binary representation of 4 has more 0's than that of 3.

This register-assignment strategy can be done at code-generation stage of a compiler. But since in some cases source code of the application is not available to recompile, or source code of the compiler is not at hand or is hard and/or risky to modify, a binary-level modification can be preferential. We applied this technique to the binary executables of M32R processor. Note that control- and data-dependencies among instructions should be considered when applying this technique at binary-level so that the producer-consumer relation among instructions does not change.

Fig. 2 outlines our proposed technique. The application source code is compiled to produce the binary executable. This binary file is processed by our optimization technique (the shadowed gray box) to statically rename register operands; the result is a modified binary executable of the same original size which is stored in the instruction memory of the target embedded system. We assume that in order to reduce leakage power, the embedded system is already equipped with an instruction-cache composed of asymmetric SRAM cells. Nothing changes from the processor point of view; no hardware or run-time change is needed.

Finally, note that since some registers have special functionality, they can neither be changed, nor others can be changed to them. Stack pointer and machine status registers are among such cases.

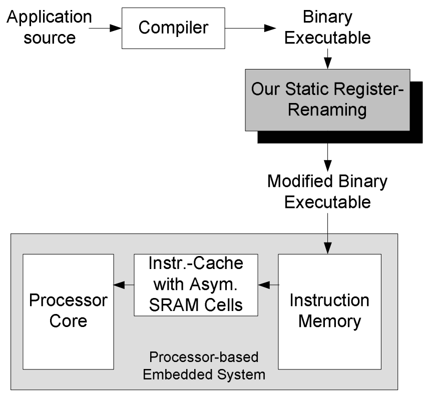
Application source
Compiler
Binary Executable
Our Static Register-Renaming
Modified Binary Executable
Processor Core
Instr.-Cache with Asym. SRAM Cells
Instruction Memory
Processor-based Embedded System

Figure 2. Block diagram of our proposed technique.

3. PROBLEM FORMULATION

We define the following notation:

B: binary executable of the application.

G: control-data-flow graph of the entire application.

R: set of general-purpose registers of the processor.

The problem can be formally defined as follows:

"For a given processor, choose the register operands of instructions in the binary executable B such that number of zeros in each instruction is maximized, subject to the control- and data-dependencies among the instructions (i.e. G) and available general-purpose registers R."

The following algorithm implements our technique. The binary executable of the application along with its corresponding control-data-flow graph are input to the algorithm. Target is a single-issue in-order RISC processor, M32R—see Section 2.2.

```
Algorithm 1: StaticRegisterRenaming(B, G)
Inputs: (B: Binary executable of the app.)
        (G: control-data-flow Graph of B)
Output: (MB: Modified Binary executable)
1  MB = empty;
2  determine live registers at each node of G;
3  for each instruction i in B do
4    dst = destination register of i;
5    src = first source register of i;
6    S = the set of general-purpose registers
         excluding live ones;
7    if S is not empty then
8      R = the register in S whose binary repre-
        sentation has the highest number of 0's;
9      change dst to R;
10     propagateRegRenaming(G, dst, R);
11     update live-registers information in G;
12     write the modified instruction to MB;
13   else
14     write original instruction i to MB;
15   endif
16 endfor
```

The algorithm iteratively processes all instructions of the binary executable; both 16-bit and 32-bit instructions are processed in the same order as they are executed on M32R. Basically we only rename the destination registers; then the source registers which correspond to the newly renamed register are similarly renamed.

For each instruction, the set of *live* registers are known from control-data-flow graph G; *live* registers are those that are still to be (potentially) read by an instruction down the sequential control flow. Register operands cannot be renamed to live registers otherwise useful data in the live registers would be corrupted. Determining live registers in a sequential code (line 2) is a well known task in compiler and high-level synthesis.

The `S` in line 6 represents the set of all registers to which a register operand can be safely renamed. Note that `dst` register itself may also be live if it is to be read by current instruction; in such case, `S` may be empty, and hence, renaming cannot be applied (line 14). If `S` is not empty, the register in `S` whose binary encoding has the highest number of zeros is selected (line 8), current instruction is modified to use it instead of the original `dst` (line 9), and the new renaming is propagated to all successor instructions (i.e., those which are a consumer of the result produced by this instruction in `dst`)—line 10. Information of live registers is also updated to reflect this new renaming (line 11). Finally the modified instruction is appended to the output binary executable (line 12). Some registers have special functionality (e.g. R14 and R15 in M32R) and can be neither renamed nor renamed to. This is considered in line 6 when generating `S`.

The algorithm has a time- and space-complexity of $O(n^2)$, where n represents number of instructions in the binary executable, since lines 10 and 11 are $O(n)$.

4. EXPERIMENTAL RESULTS

Table II shows the specifications of the benchmarks. The algorithm took only a fraction of a second to complete on a 2.66GHz Pentium-4 processor with 1GB of memory.

Table II. Benchmarks specifications.

Benchmark	No of instructions
MPEG2 encoder ver. 1.2	114162
FFT	86509
JPEG encoder ver. 6b	88679
Compress ver. 4.1	69894
FIR	4176
DCT	2518

Fig. 3 shows the setup of experiments. The benchmarks are compiled by M32R port of GCC, and the generated listing file is processed by our algorithm in our experiments so as to avoid difficulties with manipulating binary executables. The GCC-generated binary is simulated by M32R Instruction-Set Simulator (ISS) to obtain a trace of application execution. This trace is used both to validate correct implementation of the algorithm as well as to obtain leakage energy—see below. Our algorithm (the gray shadowed box) decides the new register operands for each instruction. These are used by the "Trace rewriting" box to modify the original trace file and produce a new modified one. The original and modified traces are simulated by another M32R ISS that receives trace files instead of binary executables (gray boxes in the middle of Fig. 3). Finally the outputs of these two executions are compared (the triangle at the left-hand side of Fig. 3) to make sure the register renaming implementation has not introduced errors. In addition, both trace files are input to a cache simulator to obtain the number of clock cycles that each instruction remains in the cache (***T(i,w)*** values below). Finally, leakage is calculated using Eq. 1 (below) with following notation:

- ***T(i, w)*** or *cache-residence time*: The amount of time that instruction number *i* remains in way *w* of its corresponding cache set. Note that the cache set corresponding to each instruction is fixed, but the cache way may differ over time.
- ***L0:*** Leakage power of asymmetric SRAM cell when storing a 0.
- ***L1:*** Leakage power of asymmetric SRAM cell when storing a 1.

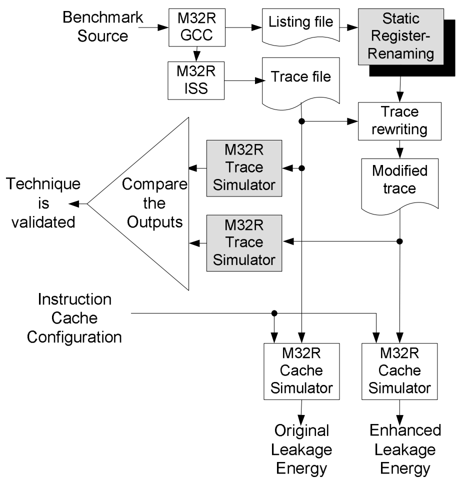

Figure 3. Experiments setup.

Table III. Leakage saving results on different types of asymmetric-cell caches (8KB direct-map cache in all cases)

SRAM type / Benchmark	Leakage Enhanced	Speed Enhanced	Stability-Leakage Enhanced	Stability-Speed Enhanced
MPEG2	26.36%	15.35%	13.64%	0.67%
FFT	26.30%	15.14%	13.43%	0.65%
JPEG	32.86%	18.52%	16.37%	0.78%
Compress	31.79%	18.64%	16.58%	0.83%
FIR	32.35%	18.92%	16.82%	0.83%
DCT	28.45%	16.03%	14.16%	0.67%
Average	30.35%	17.45%	15.47%	0.75%

- ***N_i*:** Total number of instructions in the application.
- ***$N_b(i)$*:** Number of bits of register operands for each instruction.
- ***N_w*:** Number of cache ways.
- ***inst[i][b]*:** value of bit number *b* of register operands in instruction *i* (can be 0 or 1).
- ***E*:** Total leakage energy of register operands in the instruction cache when storing application instructions:

$$E = \sum_{i=1}^{N_i} \sum_{b=1}^{N_b(i)} \sum_{w=1}^{N_w} \big(L1 \times inst[i][b] + L0 \times (1 - inst[i][b]) \big) \times T(i,w) \quad (1)$$

Each term in this summation gives the leakage energy dissipated by bit *b* of register operands of instruction *i* at way *w* of cache.

Experiments Results. Table III shows the reductions in leakage after static register-renaming, as compared to the original register assignment; benchmarks in Table II are examined when the cache is implemented using various types of asymmetric cell in Table I.

The savings highly depend on the leakage in 0 and 1 state in the employed asymmetric cell. The best results are achieved for *Leakage-Enhanced* type where the leakage is minimal and a big difference exists between L0 and L1 (Table I), whereas for *Stability-Speed Enhanced* cell where there is no big difference between L0 and L1, savings are marginal. Also note that although L0 largely differs from L1 in *Speed Enhanced* and also *Stability-Leakage Enhanced* cells, the *percentage* of leakage reduction by our technique is less than *Leakage Enhanced* case since the former cases originally (i.e., before static register-renaming) dissipate more leakage than latter. Table IV clarifies this by giving normalized leakages in each case before applying our technique.

5. SUMMARY AND CONCLUSION

We presented a software-optimization technique that reinforces the leakage-saving advantages obtainable by asymmetric SRAM in an instruction-cache. Since asymmetric cells leak less when storing 0, by increasing the number of 0 bits in the instructions it is possible to further reduce leakage. We did this by statically changing register operands in the binary executable, and showed that up to 32.86% (averaging 30.35%) more leakage can be saved.

It is important to note that since this is a one-off software technique it has very low cost, and moreover, it imposes no delay overhead since nothing changes except the used registers (which have the same speed). This technique can also be used in other asymmetric structures, such as ROM, to reduce dynamic power. Evaluating the effect on dynamic power is part of our future work.

Table IV. Original leakages normalized to *Stability-Speed Enhanced* case (8KB direct-map cache)

SRAM type / Benchmark	Leakage Enhanced	Speed Enhanced	Stability-Leakage Enhanced	Stability-Speed Enhanced
MPEG2	11.08	52.67	47.76	100
FFT	10.79	51.87	47.12	100
JPEG	10.28	50.50	46.02	100
Compress	11.25	53.14	48.13	100
FIR	11.18	52.94	47.98	100
DCT	10.26	50.45	45.98	100
Average	10.81	51.93	47.16	100

6. ACKNOWLEDGMENTS

This work is supported by VDEC, The University of Tokyo with collaboration of STARC, Panasonic, NEC Electronics, Renesas Technology, and Toshiba. This work is also supported by CREST project of JST. We are grateful for their support.

7. REFERENCES

[1] Azizi N., Najm F., Moshovos A. 2003. Low-leakage asymmetric-cell SRAM. IEEE Trans. on VLSI, 11, 4, 701-715.

[2] Flautner K., et al. 2005. Drowsy caches: simple techniques for reducing leakage power. Proc. Int'l Symp. Computer Architecture.

[3] Huang P.K., Ghiasi S. 2006. Leakage-aware intraprogram voltage scaling for embedded processors. Proc. Design Automation Conference (DAC), pp. 364-369.

[4] Kaxiras S., Hu Z., Martonosi M. 2001. Cache decay: exploiting generational behavior to reduce cache leakage power. Int'l Symp. on Computer Architecture, 240-251.

[5] M32R Family 32-bit RISC Microcomputer, http://www.renesas.com

[6] Moshnyaga V.G. and Inoue K. 2005. Low-Power Cache Design. In Low-Power Electronics Design, CRC Press.

[7] Moshovos A., Falsafi B., Najm F., Azizi N. 2005. A case for asymmetric-cell cache memories. IEEE Trans. VLSI, 13, 7.

[8] Panda P. R., et al. 2001, "Data and memory optimization techniques for embedded systems," ACM T. Des. Automat. EL., 6, 2.

[9] Petrov P., Orailoglu A. 2003. Comipler-based register name adjustment for low-power embedded processors. Proc. ICCAD.

[10] Rodriguez S., Jacob B. 2006. Energy/Power Breakdown of Pipelined Nanometer Caches (90nm/65nm/45nm/32nm). Int. Symp. on Low Power Electronic and Design (ISLPED'06), pp.25-30.

[11] Steinke S., Wehmeyer L., Lee B.S., Marwedel P. 2002. Assigning program and data objects to scratchpad for energy reduction. Proc. of Design Automation and Test in Europe (DATE).

[12] Tomiyama H., Yasuura H. 1997. Code placement techniques for cache miss rate reduction. ACM T. Des. Automat. EL., 2, 4.

[13] Verma M., Wehmeyer L., Marwedel P. 2006. Cache-aware scratchpad-allocation algorithms for energy-constrained embedded systems. IEEE Trans. on CAD, 25, 10, 2035-2051.

[14] Zhang W., Kandemir M., Karakoy M., Chen G. 2005. Reducing data cache leakage energy using a compiler-based approach. ACM Trans. Embedded Computing Systems, 4, 3.