A Non-Uniform Cache Architecture for Low Power System Design

Ishihara, Tohru
Fujitsu Laboratories of America

Fallah, Farzan
Fujitsu Laboratories of America

出版情報：Proc. of International Simposium on Low Power Electronics and Design, pp.363-368, 2005-08. International Simposium on Low Power Electronics and Design
バージョン：
権利関係：

A Non-Uniform Cache Architecture for Low Power System Design

Tohru Ishihara
Fujitsu Laboratories of America
Sunnyvale, California, 94085
Toru.Ishihara@us.fujitsu.com

Farzan Fallah
Fujitsu Laboratories of America
Sunnyvale, California, 94085
Farzan.Fallah@us.fujitsu.com

ABSTRACT
This paper proposes a non-uniform cache architecture for reducing the power consumption of memory systems. The non-uniform cache allows having different associativity values (i.e., the number of cache-ways) for different cache-sets. An algorithm determines the optimum number of cache-ways for each cache-set and generates object code suitable for the non-uniform cache memory. The paper also proposes a compiler technique for reducing redundant cache-way accesses and cache-tag accesses. Experiments demonstrate that our technique can reduce the power consumption of memory systems by up to 76% compared to the best result achieved by the conventional method.

Categories and Subject Descriptors
C.3 [**Special-Purpose and Application-Based Systems**]: Microprocessor/microcomputer applications

General Terms
Algorithms, Performance, Design.

Keywords
Microprocessor, Cache Memory, Compiler, Embedded Systems

1. INTRODUCTION
On-chip cache memories are one of the most power hungry components of today's microprocessors. For example, ARM920T™ microprocessor dissipates 25% of the power in its instruction cache [1][2]. StrongARM SA-110 processor, which specifically targets low power applications, dissipates about 27% of the power in its instruction cache [3]. Many techniques have been proposed for optimizing cache configuration considering tradeoff between energy consumption of off-chip memory and cache memory [4][5] [6][7][8]. All these works use the fact that while a bigger cache consumes more energy per access, it can reduce the number of cache misses and as a result can reduce the energy consumption of the off-chip memory. Furthermore, the aforementioned works use a uniform cache architecture in which all cache-sets have the same associativity (i.e., the number of cache-ways). We show that by relaxing the cache uniformity constraint, the dynamic power consumption and the leakage current of the cache memory can be substantially reduced. Figure 1 shows the conventional uniform cache architecture and the non-uniform one.

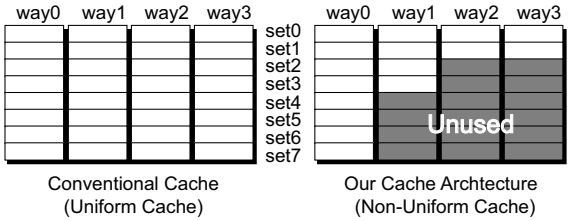

Figure 1. Uniform and non-uniform cache architectures

One of the most effective compiler techniques to reduce the power consumption of off-chip memories is the code placement technique [9][10][11][12][13]. The idea is to modify the place of basic blocks or procedures in the address space so that the number of cache conflict misses is minimized. This can significantly reduce the number of cache misses and improve a program's execution time. In this paper, we also propose a method for simultaneous cache sizing and code placement for the non-uniform cache architecture.

In summary the main contributions of this paper are introducing,

1) the non-uniform cache architecture,
2) a new code placement technique for reducing the power consumption of set associative caches by reducing the number of tag lookups and cache-ways accessed , and
3) an algorithm for simultaneously optimizing cache architecture and performing code placement to reduce both the number of cache accesses and cache misses.

The rest of the paper is organized as follows. In Section 2, we motivate the problem and present our approach to reducing the power consumption of memories. The formal definition of the problem and our algorithm for solving it are presented in Section 3. Section 4 presents experimental results. The paper concludes in Section 5.

2. MOTIVATION AND OUR APPROACH

2.1 Motivational Example
Assume the energy consumption for each cache access and off-chip memory access is 60pJ and 250pJ, respectively (see Figure 2) [1][2][14][15]. If we optimize the number of cache-ways and the code placement for an application, the power consumption of memory hierarchy may be reduced significantly.

ISLPED'05, August 8–10, 2005, San Diego, California, USA.

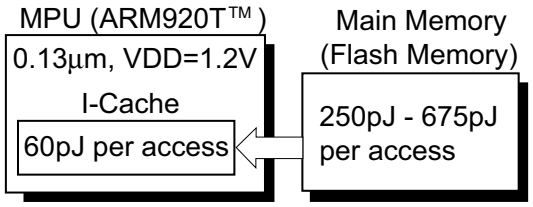

Figure 2. ARM920T™ example

For example, based on our experiment, the optimal number of cache-ways for the SPEC95 benchmark program, "Compress", is 8 as shown in the left-side of Figure 3. Note that the leakage power of the cache memory is assumed to be 10% of its dynamic power consumption. If we employ a non-uniform cache and optimize the number of cache-ways and the code placement simultaneously, the total power consumption of memories can be reduced by 22%, compared to the power consumption of the optimized uniform cache configuration as shown in Figure 3.

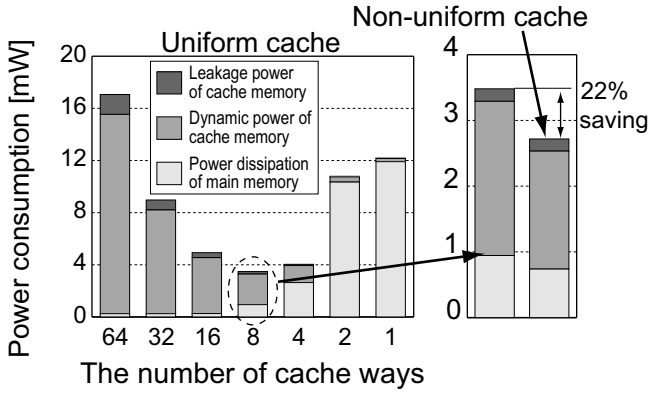

Figure 3. A comparison between the power consumption of uniform and non-uniform caches

2.2 Non-Uniform Cache Architecture

We determine the optimum number of cache-ways for each cache-set at design time. Although the number of active cache-ways can be changed dynamically by using a sleep transistor during the course of running an application program, we do not consider it in this work. The power supply of unused cache-ways (the gray portion of Figure 4) can be disconnected by eliminating vias used for connecting the power supply to memory cells. Unused memory cells can also be disconnected from bit and word lines in the same fashion.

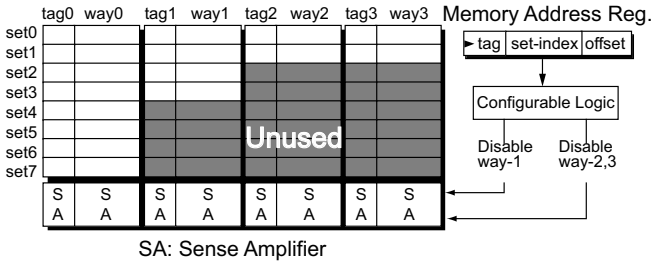

Figure 4. Deactivating sense amplifiers

One possible way of marking unused cache blocks is to use a second valid bit [16]. If the bit is one, the corresponding cache block will not be used for replacement in case of a cache miss. Accessing an unused block will always cause a cache miss. To reduce the dynamic power consumption of the non-uniform cache, it is possible to deactivate sense-amplifiers of cache-ways which are marked as unused for the accessed cache-set. This can be easily implemented by checking the set-index field of the memory address register. For example in Figure 4, sense-amplifiers for tag1 and way1 are deactivated when the target cache-set is 4, 5, 6, or 7. Similarly, sense-amplifiers for tag2, way2, tag3, and way3 are deactivated when one of sets 2-7 is accessed.

2.3 Reducing Redundant Cache Accesses

In [17], Panwer et al. have shown that cache-tag access and tag comparison do not need to be performed for all instruction fetches. Consider an instruction *j* executed immediately after an instruction *i*. There are three cases,

1. Intra-cache-line sequential flow

This occurs when both *i* and *j* instructions reside on the same cache-line and *i* is a non-branch instruction or an untaken branch.

2. Inter-cache-line sequential flow

This case is similar to the first one, the only difference is that *i* and *j* reside on different cache-lines.

3. Non-sequential flow

In this case, *i* is a taken branch instruction and *j* is its target.

In the first case (*intra-cache-line sequential flow*), it is easy to detect that *j* resides in the same cache-way as *i*. Therefore, there is no need to perform a tag lookup for instruction *j* [1][17][18]. On the other hand, a tag lookup and a cache-way access are required for a non-sequential fetch such as a taken-branch (*non-sequential flow*) or a sequential fetch across a cache line boundary (*inter-cache-line sequential flow*). As a consequence, the power consumption of the cache memory can be reduced by deactivating memory modules of tags and cache-ways in case of the *intra-cache-line sequential flow*. Several embedded processors including ARM [1][18] use this technique. We refer to this technique as *Inter-Line Way Memoization* or *ILWM*. We use *ILWM* in our approach.

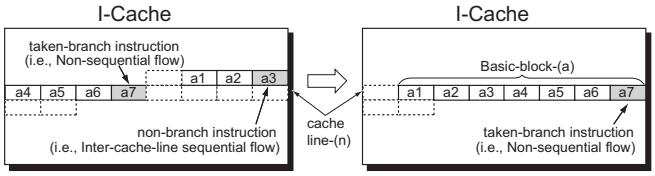

Figure 5. A code placement technique for reducing redundant cache-way and cache-tag accesses

Assume a basic block "a" consists of 7 instructions and its last instruction, a7, which is a taken-branch resides in the fourth word of the cache line "n" (see Figure 5). Further, assume the last instruction of the cache line "n" is not a branch instruction. A tag lookup is required when a3 or a7 is executed because in either case it is not clear whether the next instruction resides in the cache or not. However, if the location of the basic block "a" in the address space is changed so the basic block "n" is not located across a cache-line boundary, the cache and tag accesses for

instruction a3 can be eliminated (see Figure 5). Therefore, we change the placement of basic blocks in the main memory so frequently accessed basic blocks are not located across a cache-line boundary. To the best of our knowledge, this is the first code placement technique which reduces the number of redundant cache-way and cache-tag accesses.

Figure 6 shows the power breakdown for a cache. For example in "JPEG_enc", the *inter-cache-line sequential flow* is responsible for 10% of cache accesses. Note that for *inter-cache-line sequential flows*, all cache-ways and cache-tags are activated. Therefore, the power consumption of the cache memory due to the *inter-cache-line sequential flow* is large especially for highly associative caches. Assuming a 16-way set associative cache, more than 50% of the cache power in "JPEG_enc" is due to the *inter cache-line sequential flow*. Therefore, decreasing the number of the *inter cache-line sequential flow* substantially reduces the cache power consumption. Another way of reducing the number of times the *inter cache-line sequential flow* occurs is increasing the size of cache-lines. However, increasing the cache-line size increases the number of off-chip memory accesses in case of a cache miss. Our algorithm presented in the next section takes this trade-off into account and explores different cache-line sizes to minimize the total power consumption of the memory hierarchy.

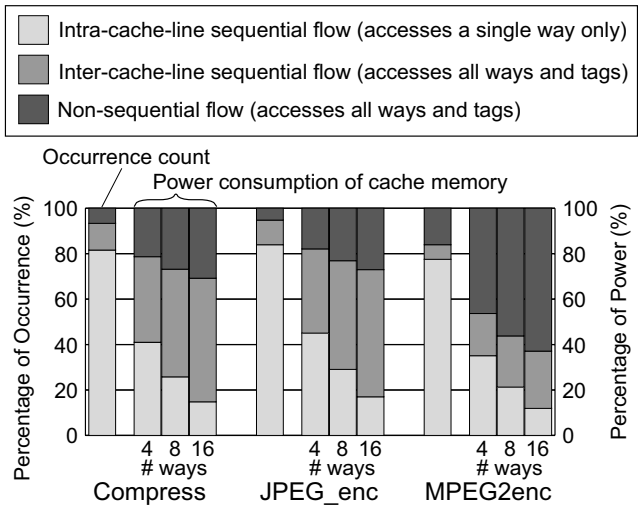

Figure 6. Power break down for a cache

2.4 Concurrent Optimization of Cache Configuration and Code Placement

We first explain the idea behind the conventional code placement technique. Consider a direct-mapped cache of size C ($C = 2^m$ words) whose cache line size is L words, i.e., L consecutive words are fetched from the memory on a cache read miss. In a direct-mapped cache, the cache line containing a word located at memory address M can be calculated by ($\lfloor M/L \rfloor$ mod C/L). Therefore, two memory locations M_i and M_j will map onto the same cache line if the following condition holds,

$$\left(\left\lfloor \frac{M_i}{L} \right\rfloor - \left\lfloor \frac{M_j}{L} \right\rfloor\right) \bmod \frac{C}{L} = 0$$

The above equation can be written as:

$$(n \cdot C - L) < (M_i - M_j) < (n \cdot C + L) \quad (1)$$

where n is any integer. If basic blocks B_i and B_j are inside a loop whose iteration count is N and their memory locations M_i and M_j satisfy condition (1), cache conflict misses occur at least N times when executing the loop [19]. This can be extended for a W-way set associative cache. A cache conflict miss occurs in a W-way set associative cache if more than W different addresses with distinct $\lfloor M/L \rfloor$ values satisfy condition (1) are accessed in a loop; note M is the memory address. Therefore, the number of cache conflict misses can be easily calculated from cache parameters (i.e., cache-line size, the number of cache-sets and the number of cache-ways), the location of each basic block in the memory address space and the iteration count for each closed loop for a target application program [11]. Several code placement techniques have used the above before [9][10][11][12][13] and many cache optimization techniques have been proposed for reducing the sum of the power consumption of the off-chip memory and the cache [4][5][6][7][8]. However, to the best of our knowledge, there is no technique for simultaneously performing code placement and cache configuration optimization. Therefore, conventional cache optimization flows need to perform several steps iteratively to find the optimum cache configuration and generate object code for it (see Figure 7).

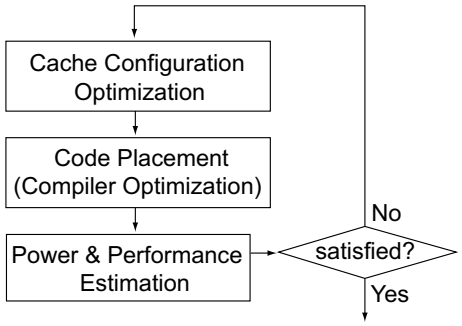

Figure 7. A conventional cache optimization flow

Our approach optimizes cache configuration and code placement simultaneously to reduce the dynamic and leakage power consumption of cache memory and off-chip memory for a given performance constraint. We used the method presented in [11][13] to analytically calculate the number of cache misses based on (1). Our algorithm calculates the number of cache conflicts in each cache-set for a given associativity (i.e., the number of ways).

3. PROBLEM DESCRIPTION

3.1 Notation

- E_{memory}, E_{way}, and E_{tag}: The energy consumption per access for the main memory, a single cache-way, and a cache-tag memory, respectively.
- P_{static}: The static power consumption of the main memory.
- TE_{memory} and TE_{cache}: The total energy consumption of the main memory (i.e., the off-chip memory) and the cache (i.e., cache-tag and cache-way), respectively.
- $P_{leakage}$: The leakage power consumption of a 1-byte cache memory block.
- $TE_{leakage}$: The total energy consumption of the cache memory due to leakage.
- W_{bus}: The memory access bus width (in byte).
- W_{inst}: The size of an instruction (in byte).

- S_{cache}: The number of sets in a cache memory.
- C_{access}: The number of CPU cycles required for a single memory access.
- C_{wait}: The number of wait-cycles for a memory access.
- F_{clock}: The clock frequency of CPU.
- n_{line}: The line size of the cache memory (in byte).
- a_i: The number of ways in the i^{th} cache-set.
- N_{miss}: The number of cache misses.
- N_{inst}: The number of instructions executed.
- X_i: The number of "full-way accesses" for the i^{th} cache-set. In the "full-way" access, all cache-ways and cache-tags in the target cache-set are activated. A "full-way access" is necessary in case of an *inter-cache-line sequential flow* or a *non-sequential flow*. Otherwise, only a single cache-way is activated.
- T_{total}, and T_{const}: The total execution time and the constraint on it.
- P_{total}: The total power consumption of the memory system.

We assume E_{memory}, E_{way}, E_{tag}, P_{static}, $P_{leakage}$, W_{bus}, W_{inst}, S_{cache}, F_{clock}, C_{access}, C_{wait}, and T_{const} are given parameters. The parameters to be determined are n_{line} and a_i. N_{miss}, X_i, and T_{total} are functions of the code placement, W_{bus}, W_{inst}, n_{line} and a_i. We can find N_{miss}, N_{inst}, and X_i by using a method presented in [11][13]. Since the cache is usually divided into sub-banks and only a single sub-bank is activated per access [20], E_{way} is independent of n_{lines}.

3.2 Problem Formulation

The problem can be formally defined as follows:

"For given values of E_{memory}, E_{way}, E_{tag}, P_{static}, $P_{leakage}$, W_{bus}, W_{inst}, S_{cache}, F_{clock}, C_{access}, C_{wait}, and the ***original object code***, determine **code placement**, n_{line} and a_i to minimize P_{total}, the total power consumption of the memory hierarchy under the given time constraint T_{const}."

Note that T_{total}, TE_{memory} TE_{cache}, $TE_{leakage}$, and P_{total} can be calculated using the following formulas:

$$T_{total} = \frac{1}{F_{clock}} \cdot \{N_{inst} + N_{miss} \cdot (C_{access} \cdot \frac{n_{line}}{W_{bus}} + C_{wait})\}$$

$$TE_{memory} = E_{memory} \cdot N_{miss} \cdot \frac{n_{line}}{W_{bus}} + P_{static} \cdot T_{total}$$

$$TE_{cache} = E_{way} \cdot N_{inst} + E_{way} \cdot N_{miss} \cdot \frac{n_{line}}{W_{inst}} + E_{tag} \cdot N_{miss} + E_{way} \cdot \sum_{i=0}^{S_{cache}} \{(a_i - 1) \cdot X_i\} + E_{tag} \cdot \sum_{i=0}^{S_{cache}} (a_i \cdot X_i)$$

$$TE_{leakage} = P_{leakage} \cdot T_{total} \cdot n_{line} \cdot \sum_{i=0}^{S_{cache}} a_i$$

$$P_{total} = (TE_{memory} + TE_{cache} + TE_{leakage}) / T_{total}, \quad T_{total} \le T_{const}$$

3.3 Algorithm

Our algorithm starts with an original cache configuration (n_{lines}=32, S_{cache}=8, a_i=64). In the next step, our algorithm finds the optimal location of each block of the application program in the address space. This is done by changing the order of placing functions in the address space and finding the best ordering. For each ordering, the algorithm greedily reduces the energy by iteratively finding a cache-set for which reducing the number of ways by a factor of 2 gives the largest power reduction. The power consumption (P_{total}) and the run-time (T_{total}) are found by calculating the number of cache misses for a given associativity using the technique presented in [11][13]. The calculation can be done without simulating the cache and by analyzing an iteration count of each loop and the location of each basic block in the address space for the application program. The ordering which gives the minimum energy is selected along with the optimal number of ways for each cache-set. The algorithm performs the above for different cache-line sizes and continues as long as the power consumption reduces. Note the ordering of functions is fixed when the cache-line sizes are changed. This is a good simplification because the optimum ordering of functions usually does not change widely when cache-line sizes vary by a factor of 2. The computation time of the algorithm is quadratic in terms of the number of functions and linear in terms of the number of loops of the application program.

Procedure *MinimizePower*
Input: E_{memory}, E_{way}, E_{tag}, $P_{leakage}$, W_{bus}, W_{inst}, S_{cache}, F_{clock}, C_{access}, C_{wait}, T_{const}, P_{static}, and *original object code*
Output: n_{line}, a set of a_i, and order of functions in the *optimized object code*

Let L be the list of functions in the target program sorted in descending order of their execution counts;
  P_{min} = T_{min} = infinity;
  for each $n_{line} \in$ {32,64,128,256,512} **do**
    P_{init} = P_{min}; T_{init} = T_{min};
    repeat
      P_{min} = P_{init}; T_{min} = T_{init};
      for (t=0; t<|L|; t++) **do**
        p = $L[t]$;
        for each $p' \in L$ and $p' \neq p$ **do**
          Insert function p in the place of p';
          Set all a_i to 64 and calculate P_{total} and T_{total};
          repeat
            1. Find a cache-set for which reducing the number of cache ways by a factor of 2 results in the largest power reduction;
            2. Divide the number of cache-ways for the cache-set by 2 and calculate P_{total} and T_{total};
          until ((P_{total} stops decreasing) or ($T_{total} > T_{const}$))
          if ($P_{total} \le P_{min}$ & $T_{total} \le T_{min}$) **then**
            P_{min} = P_{total}; T_{min} = T_{total}; $BEST_{location}$ = p';
          end if
        end for
        Put function p in the place of $BEST_{location}$
      end for
    until (P_{min} stops decreasing)
    if (P_{init} == P_{min} & $T_{init} \le T_{const}$) **then**
      Output $BEST_{line}$, $BEST_{ways}$ and $BEST_{order}$; **Exit**;
    else
      $BEST_{line}$ = n_{line}; $BEST_{ways}$ = a set of a_i;
      $BEST_{order}$ = order of functions;
    end if
  end for
end Procedure

4. EXPERIMENTAL RESULTS

4.1 Experimental Setup

We calculated E_{memory}, E_{way}, E_{tag}, $P_{leakage}$, and P_{static} for the system in Figure 2. The cache size, the number of cache sets, the number of cache-ways, the cache line size, and the clock frequency of the original CPU are assumed to be 16KB, 8, 64, 32-byte, and 250MHz, respectively [2]. Since the outputs of flash memories used in [14] and [15] are 16-bit, we assumed W_{bus}=16 in our experiment. The bit width of instructions is 32, therefore, W_{inst} = 4. The number of CPU cycles required for a memory access and the number of wait-cycles for a memory access are assumed to be 4 (i.e., $C_{access} = C_{wait} = 4$), since the clock frequency of the flash memory and the processor core are 80MHz and 250MHz, respectively. We considered two scenarios, low leakage and high leakage with the leakage power of the cache memory equal to 5% and 10% of the dynamic power, respectively. We used three benchmark programs; *Compress* version 4.0, *JPEG encoder* version 6b, and *MPEG2 encoder* version 1.2. Table I shows the number of functions, basic blocks and instructions for each benchmark program. We used GNU C compiler and debugger for ARMv4T architecture to generate address traces.

TABLE I. Specification of benchmark programs

	# Functions	# Basic blocks	# Instructions
Compress	160	2,281	10,716
JPEG_enc	353	6,451	30,867
MPEG2enc	256	6,428	33,850

4.2 Results

We compared the following four techniques, **(a)** performing cache sizing for uniform cache, **(b)** performing cache sizing for uniform cache and the conventional code placement after that, **(c)** performing our code placement and cache sizing for a uniform cache concurrently, and **(d)** concurrent optimization for non-uniform cache. Redundant cache-way and cache-tag access elimination *(ILWM)* [17] was used for all four techniques. The number of cache-sets in all experiments was 8.

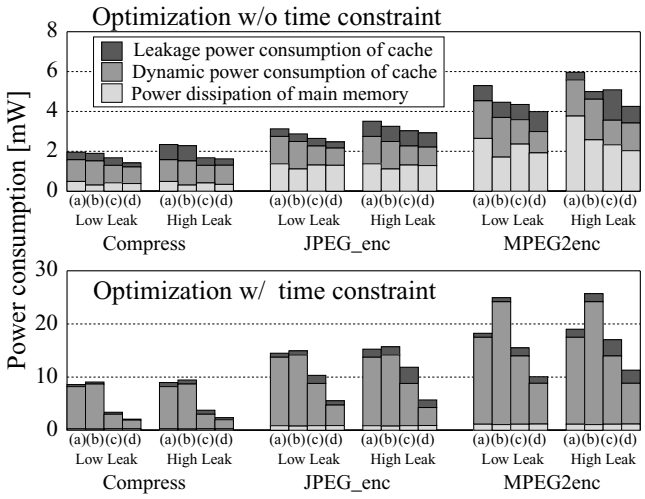

Figure 8. Comparison of different techniques

The power consumption results optimized without and with a time constraint are shown in Figure 8. The time constraint T_{const} is set to the execution time of the target application program with the original cache configuration. "Low Leak" and "High Leak" in Figure 8 correspond to low- and high-leakage scenarios, respectively.

Since conventional code placement techniques reduce the number of cache misses only, they may increase the number of cache-way and tag accesses if the processor uses the *ILWM* technique [17]. For example, compare case (a) and (b) in the time-constrained optimization results in Figure 8. On the other hand, our methods (c) and (d) always reduce the dynamic power consumption of the cache memories. Optimizing without a time-constraint reduced the power consumption for "Compress" by 29% (17% on an average), while in presence of a time-constraint, up to 76% (52% on an average) reduction in the power consumption was achieved. The reason for better results for time-constrained case is that it requires higher number of ways. Therefore, there is more opportunity for our method to reduce the average number of cache-ways accessed.

Table II shows the number of ways, cache-line size and cache size (in byte) in the high-leakage case in our experiment. As one can see, in many cases our approach (d) reduces the effective size (the total size of blocks used) of the cache memory as well.

TABLE II. The cache configuration results

		Compress		JPEG_enc		MPEG2enc	
		w/o T_{const}	w/ T_{const}	w/o T_{const}	w/ T_{const}	w/o T_{const}	w/ T_{const}
(a) and **(b)**	n_{line}	256	32	128	32	64	32
	a_i	4	32	8	64	8	64
	size	8K	8K	8K	16K	4K	16K
(c)	n_{line}	256	64	256	64	512	64
	a_i	4	16	4	64	4	64
	size	8K	8K	8K	32K	16K	32K
(d)	n_{line}	256	64	256	64	128	64
	$a_{0...7}$	1,4, 4,4, 4,4, 4,2	8,8, 8,8, 8,8, 8,8	4,4, 4,4, 4,4, 2,4	64,64, 32,16, 32,32, 16,16	8,8, 16,16, 8,8, 2,4	32,32, 64,32, 64,64, 64,64
	size	3.38K	4K	7.5K	17K	8.75K	26K

Since the behavior of a program depends on its input values, an object code and cache configuration optimized for a specific input value is not necessarily optimal for the other input values. To see the effect of changing the input value on the cache behavior, we calculated the power consumption of memory systems for different input values. We calculated the following three values for six different input values:

1. the power consumption for the original object code executed with a uniform cache optimized for Data0.
2. the power consumption (P_{total}) for the optimized object code executed with a non-uniform cache optimized for Data0.

3. the total execution time (T_{total}) for the optimized object code running on a processor with the non-uniform cache. The performance value is normalized to the performance for Data0.

Figure 9 shows the results for six different input values for each benchmark program. The left and right vertical axes represent the power consumption of memories and the normalized performance of a processor with the non-uniform cache, respectively. The object code and cache configuration were optimized for Data0 using our algorithm for non-uniform caches. As one can see, the object code and the cache configuration optimized for Data0 achieve very good results for other input values as well.

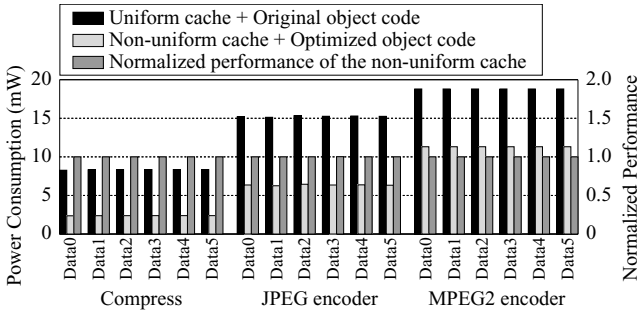

Figure 9. Input Data Dependency

Table III shows computation time (in second) of four optimization methods executed on an UltraSPARC-II dual CPU workstation running Solaris8 at 450MHz with 2GB of memory. Since the optimization time in some cases is very large, our future plan is substantially reducing it.

TABLE III. CPU-time for cache optimization (second)

	Compress		JPEG_enc		MPEG2enc	
	w/o T_{const}	w/ T_{const}	w/o T_{const}	w/ T_{const}	w/o T_{const}	w/ T_{const}
(a)	< 1	< 1	1	1	2	2
(b)	15	42	103	474	710	1,628
(c)	62	129	812	1,851	1,331	4,289
(d)	364	351	2,783	5,391	8,329	10,413

5. SUMMARY AND CONCLUSIONS

In this paper, we proposed the non-uniform cache architecture, a code placement technique for reducing the power consumption of caches, and an algorithm for simultaneous cache configuration optimization and code placement. In future we plan to enhance our method by dynamically disabling cache-ways during the course of running an application program. Our current algorithm sets the value of n_{line} and a_i to powers of 2 only. This is done to make the search easy. In future we plan to improve the power saving by relaxing this constraint on the values of n_{line} and a_i.

6. ACKNOWLEDGMENTS

We would like to thank Tom Sidle, the VP of advanced CAD Technology at Fujitsu Laboratories of America for supporting this research.

7. REFERENCES

[1] S. Segars, "Low Power Design Techniques for Microprocessors", ISSCC Tutorial note, February 2001.

[2] ARM Ltd., "ARM Processor Core Overview", http://www.arm.com/products/CPUs/

[3] J. Montanaro et al., "A 160 MHz, 32b 0.5W CMOS RISC Microprocessor", In Proc. of ISSCC, February 1996.

[4] C. Su and A. Despain, "Cache Design Trade-offs for Power and Performance Optimization: A Case Study", In Proc. of ISLPED, pp.63-68, August 1995.

[5] P. Hicks, M. Walnock, and R. M. Owens, "Analysis of Power Consumption in Memory Hierarchies", In Proc. of ISLPED, pp.239-242, August 1997.

[6] Y. Li, and J. Henkel, "A Framework for Estimating and Minimizing Energy Dissipation of Embedded HW/SW Systems", In Proc. of DAC, pp.188-193, June, 1998.

[7] W. T. Shine, and C. Chacrabarti, "Memory Exploration for Low Power, Embedded Systems", In Proc. of DAC, pp.140-145, June, 1999.

[8] A. Malik, B. Moyer and D. Cermak, "A Low Power Unified Cache Architecture Providing Power and Performance Flexibility", In Proc. of ISLPED, pp.241-243, July 2000.

[9] S. McFarling, "Program Optimization for Instruction Caches", In Proc. of Int'l Conference on Architecture Support for Programming Languages and Operating Systems, pp.183-191, April 1989.

[10] W. W. Hwu and P. P. Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler", In Proc. of ISCA, pp.242-251, May 1989.

[11] H. Tomiyama and H. Yasuura, "Optimal Code Placement of Embedded Software for Instruction Caches", In Proc. of European Design and Test Conference, pp.96-101, March, 1996.

[12] A. H. Hashemi, D. R. Kaeli, and B. Calder, "Efficient Procedure Mapping Using Cache Line Coloring", in Proc. of Programming Language Design and Implementation, pp.171-182, June, 1997.

[13] S. Ghosh, M. Martonosi, and S. Malik, "Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior", ACM Trans. on Programming Languages and Systems, vol.21, no.4, pp.703-746, July, 1999.

[14] Micron Technologies Inc., "Low Power Flash", http://www.micron.com/products/flash/lowpower/

[15] Fujitsu Ltd., "128Mbit (x16bit) Burst Mode Flash Memory MBM29BS12DH", http://edevice.fujitsu.com/fj/MARCOM/find/21-2e/

[16] D. A. Patterson, et al., "Architecture of a VLSI instruction cache for a RISC", In Proc. 10th Annual Int'l Symposium on Computer Architecture, vol. 11, no. 3, pp.108-116, June, 1983.

[17] R. Panwar, and D. Rennels, "Reducing the Frequency of Tag Compares for Low Power I-Cache Design", In Proc. of ISLPED, pp.57-62, August 1995.

[18] M. Muller, "Power Efficiency & Low Cost: The ARM6 Family", In Proc. of Hot Chips IV, August 1992.

[19] H. Hill and A. J. Smith, "Evaluating Associativity in CPU Cache", IEEE Trans. on Computers, Vol. 38, No. 12, pp.1612-1630, December, 1989.

[20] K. Ghose and M. B. Kamble, "Reducing Power in Superscalar Processor Caches Using Subbanking, Multiple Line Buffers and Bit-Line Segmentation", In Proc. of ISLPED, pp.70-75, August 1999.