A Cache-Defect-Aware Code Placement Algorithm for Improving the Performance of Processors

Ishihara, Tohru
Fujitsu Laboratories of America, Inc.

Fallah, Farzan
Fujitsu Laboratories of America, Inc.

A Cache-Defect-Aware Code Placement Algorithm for Improving the Performance of Processors

Tohru Ishihara and Farzan Fallah

Advanced CAD Technology
Fujitsu Laboratories of America, Inc.
Sunnyvale, California 94085, USA

Abstract— **Yield improvement through exploiting fault-free sections of defective chips is a well-known technique [1][2]. The idea is to partition the circuitry of a chip in a way that fault-free sections can function independently. Many fault tolerant techniques for improving the yield of processors with a cache memory have been proposed [3][4][5]. In this paper, we propose a defect-aware code placement technique which offsets the performance degradation of a processor with a defective cache memory. To the best of our knowledge, this is the first compiler-based technique which offsets the performance degradation due to cache defects. Experiments demonstrate that the technique can compensate the performance degradation even when 5% of cache lines are faulty. In some cases the technique was able to offset the impact even in presence of 25% faulty cache-lines.**

I. INTRODUCTION

Most of today's microprocessors including embedded ones employ one or more on-chip caches. Currently in most chips, caches occupy a large percentage of the area and the percentage is expected to increase in future. For example in StrongARM SA-110 processor, one half of the total chip area is devoted to the two 16KB caches [6]. Since cache arrays are designed with the tightest feature and scaling rules available in a given technology, they are more susceptible to faults compared to logic blocks [7][8]. Thus, the yield of microprocessors with on-chip caches can be enhanced considerably if cache defects are tolerated without noticeable performance degradation. Many techniques have been proposed for disabling faulty cache blocks in partially good chips [3][4][5][9]. Since using a smaller cache memory does not affect the correct operation of a processor, the processor can still be used after some changes even if the cache contains manufacturing defects. Therefore, the most straightforward solution to fix a chip with a faulty cache is to disable the entire cache. For set-associative caches, a possible solution is to disable the cache-way which contains the defect [9]. A more sophisticated way is disabling only the defective cache-line because the existence of a defect in a cache-line does not affect other cache lines. This can be done by using an extra bit added to the flag bits associated with each block and using the added bit for marking a faulty cache block [10]. If the bit is one, the corresponding cache block will not be used for replacement in case of a cache miss. Thus, accessing the block will always cause a cache miss. Otherwise, the block is non-faulty and will be used. In this paper, we refer to the extra bit as *FT-bit* (Fault-Tolerance bit) [11]. Other names used in literature are: *availability bit* [3], *the second valid bit* [4] and *purge bit* [5]. The idea of adding the FT-bit was initially proposed in [10].

In this paper, we propose a defect-aware code placement technique which reduces the performance degradation of a processor with a partially good cache memory. We used FT-bits in our technique. Our approach is to modify the placement of basic blocks or functions in the address space so that the number of cache misses is minimized for a given defective cache. To the best of our knowledge, this is the first compiler technique which reduces the performance degradation of a partially good cache memory.

The rest of the paper is organized as follows. In Section 2, we summarize previous work and our approach. The 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.

II. PREVIOUS WORK AND OUR APPROACH

A. Techniques for Recovering Cache Performance

A replacement technique called the *Memory Reliability Enhancement Peripheral* (MREP) is presented in [12]. The idea is to have several spare words in a small set associative cache which may replace faulty words in the main memory. A technique similar to MREP is presented for on-chip caches in [13]. A very small fully associative spare cache is added to a direct-mapped cache and is used as a spare for the disabled faulty blocks. The experimental results in [13] show that one or two spare blocks are sufficient to avoid most of the extra misses caused by a few (less than 5%) faulty blocks. However, for a large number of faults, a few spare blocks will not be sufficient.

Sohi [3] investigated the application of a Single Error Correcting and Double Error Detecting (SEC-DED) Hamming code in an on-chip cache memory. The idea of the SEC-DED approach is to use extra bits per word to store an

error correcting code and correcting data if a single error is detected [14]. The approach is effective for a single bit error per word (e.g., a single defect on a bit line) only.

Shirvani et al. proposed a new cache architecture called PADded cache [11] which requires neither spare blocks nor error correcting bits. The idea is to recover the performance degradation due to faulty cache lines by mapping them to existing non-faulty cache lines. This is made possible by the use of programmable address decoder. Consider the direct mapped cache shown in Figure 1.
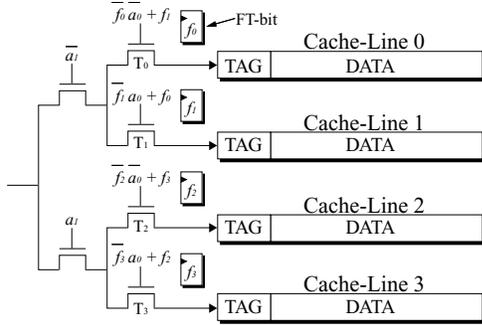

Figure 1.   Programmable Address Decoder (PAD)

The number of sets is 4. Let a_1a_0 be the cache-line address. If $a_1a_0 = 00$, *cache-line 0* is selected. If $a_1a_0 = 01$, *cache-line 1* is selected, and so forth. Assume that *cache-line 0* is faulty. The gates of transistors T0 and T1 are controlled using the FT-bit such that when *cache-line 0* is marked faulty, T0 is always off and T1 is always on. As a result, all references to *cache-line 0* are re-mapped to *cache-line 1*. Therefore, the addresses mapped to *cache-line 0* are still cached, but they will have conflict misses with *cache-line 1*. By adding another FT-bit to the cache-line, it can be indicated whether both *cache-line 0* and *cache-line 1* are faulty. Thus, the second level of the decoder can be controlled and *cache-line 0* and *1* can be mapped to *2* and *3* or vice-versa. The approach can be extended for set associative caches. Note the fault tolerance is achieved at the cost of higher complexity for the decoder which makes the cache-line access slower. Furthermore, there is a large area overhead due to adding FT-bits to each block. Note that FT-bits must be implemented using flip-flops. One way to avoid adding extra bits is using the unused combination of the already available flag bits (e.g., valid bit = 0 and lock bit = 1) to indicate a fault in the cache-line. The lock bit is used to lock some or all contents of the cache in place. This feature is available on several commercial processors including PowerPC 604e, 405 and 440 families [22][23], Intel-960, some Intel x86 processors, ARM 10 family [24], NEC V830R processor [25] and so on. However, this results in performance degradation because it requires one extra cycle to read the FT-bits from the tag memory and configuring the address mapping.

Unlike previous approaches, our approach needs neither any spare memory blocks nor error correcting codes.

Furthermore, no complicated hardware support like a Programmable Address Decoder (PAD) is required.

B.   Conventional Code Placement

We first explain the idea behind the conventional code placement technique. Consider a direct-mapped cache of size C ($= 2^m$ words) whose cache line size is L words, i.e., L consecutive words are fetched from the main 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 be mapped onto the same cache line if the following condition holds,

$$\left(\left\lfloor \frac{M_i}{L} \right\rfloor - \left\lfloor \frac{M_j}{L} \right\rfloor \right) \bmod \frac{C}{L} = 0 \qquad (1)$$

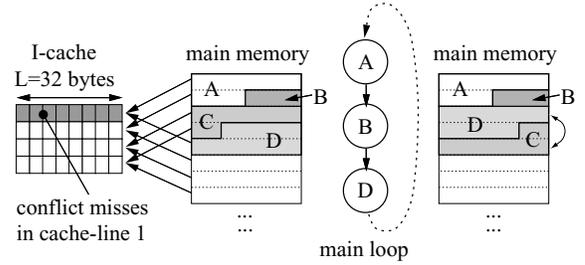Several code placement techniques have used the above formula [16][17][18][19][20][21].

Figure 2.   An Example of Code Placement

Assume a direct mapped cache with 4 cache-lines, where each cache-line is 32 bytes. Functions A, B, C and D are placed in the main memory as shown in the left side of Figure 2. If functions A, B, and D are accessed in a loop, conflict misses occur because A and D are mapped onto the same cache line. If the locations of C and D are swapped as shown in the right side of Figure 2, the cache conflict is resolved. Code placement modifies the placement of basic blocks or functions in the address space so that the total number of cache conflict misses is minimized.
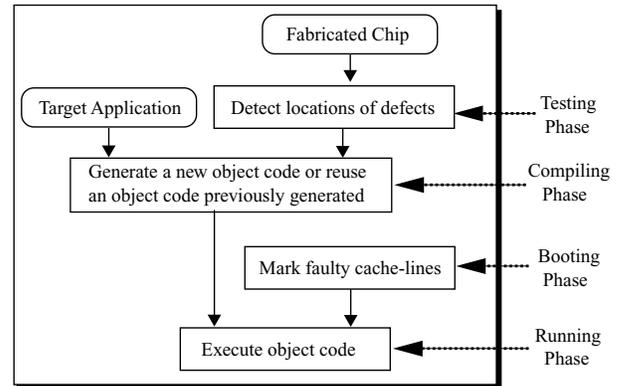
C.   Our Approach

Figure 3.   Compiler Optimization Flow

An overview of our approach is depicted in Figure 3. First, we detect the locations of defects in a cache memory. Next, our code placement technique generates the object code such that the number of cache misses will be lower than a given number. Therefore, we perform recompilation only if the original object code does not satisfy the required performance for a specific chip. If the original object code or an object code previously generated for another defect can satisfy the required performance, we use it. Every time the chip is turned on, it executes an initialization step during which based on the information collected during test, faulty cache-lines are marked using lock and valid bits. Then the chip executes the compiled code.

Our approach exploits an unused combination of existing flag bits (i.e., *valid bit*=0 and *lock bit*=1) to indicate a fault in a specific cache-line. Assuming a 4-way set associative cache with lock and valid bits, if the lock bit of the *way1* in the 5th cache-set is "1" as shown in Figure 4, the corresponding cache-line will not be used for replacement in case of a cache miss. If the valid bit of *way1* in the 5th cache-set is "0", accessing to the corresponding block will always cause a cache miss. Therefore, this mechanism guarantees the correctness of the processor operation even in presence of defects in tag or data memory.
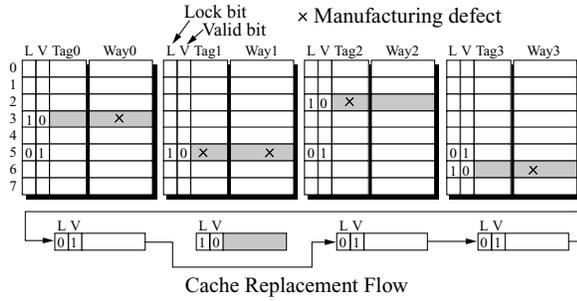

Figure 4.   Cache Locking Function

Although most existing commercial processors do not allow to set the lock bit for an invalid cache-line (i.e., *valid bit*=0 and *lock bit*=1 is not a valid combination), supporting this combination is possible by a minor modification of the cache control circuitry. Note that for a set associative cache, the associativity of the cache-set which includes the faulty cache-lines is reduced by one. In a direct mapped cache, every access to the cache-set which includes the faulty cache-line will cause a cache miss. Our method modifies the placement of functions in the address space so as to compensate the increased number of cache misses due to faulty cache-lines. The most important feature of our approach is that it does not have any hardware overhead if the target processor has the cache lock function.

III.   PROBREM DEFINITION

A.   Cache Miss Calculation

We first generate an instruction trace corresponding to an execution of the application program as follows,

$$(o_0, o_1, o_3, o_5, o_6, o_7, o_3, o_5, o_7, o_3, o_5, o_6, o_7, o_3) \qquad (2)$$

where o_i represents the ith instruction of the original object code. Assume a_i represents the address of o_i. Each instruction o_i is mapped to a memory block whose address is $\lfloor a_i/L \rfloor$. From the instruction trace we generate the trace of memory block addresses accessed, $TMB = (b_1, b_1, b_2, b_3, \ldots)$

We define $X(b_i)$ as,

$$X(b_i) = \begin{cases} 1 & \text{if there are at least } W_s \text{ accesses} \\ & \text{to } s = (b_i \bmod S) \text{ between an access to} \\ & b_i \text{ and its next access} \\ 0 & \text{otherwise.} \end{cases}$$

where S and W_s are the number of cache-sets and the number of non-faulty cache-lines in the sth cache-set, respectively. The total number of misses can be calculated by,

$$M_{total} = \sum_{\forall\, b_i \in TMB} X(b_i). \qquad (3)$$

The above formula, which takes into account the effect of faulty cache-lines, is an extension of the formulae derived in [18]. Although the approach [18] results in many gaps in the object code (i.e., the memory requirement increases), our approach does not increase the size of object code.

B.   Trace Compaction

For a given trace, it is usually possible to generate a shorter trace which results in the same number of misses, but can be processed faster.
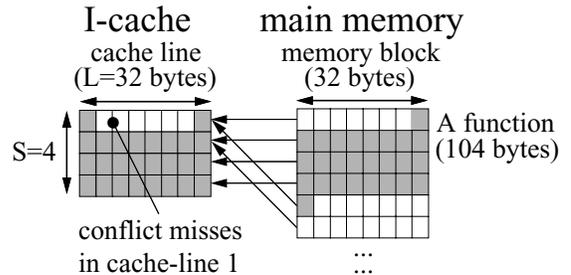

Figure 5.   An Example of a Self-Conflict Miss

Consider a function whose size is smaller than the cache size as shown in Figure 5. Left shows a direct mapped cache with four 32-byte lines. If a function whose size is 104 bytes is placed in the main memory, cache conflict misses occur in the cache-line 1 because the first and last words of the function are mapped onto the same cache-line. If the size of the function is 100 bytes, no self-conflict miss will occur no matter where in the address space the function is placed. Therefore, the condition for the self-conflict free function is as follows,

The size of the function $\leq L \times (S-1) + I$ (4),

where L, S, an I represent cache-line size in byte, the number of cache-sets and instruction word size in byte, respectively.

If a function satisfies the condition (4), two instructions o_i and o_j of the function will never evict each other no matter where in the address space the function is placed. As a result, when a loop is executed if all instructions in the loop belongs to the function and the loop is executed n times, $n-1$ instances can be deleted from the trace.

C. Problem Formulation

We summarize notations used in the problem formulation as follows,

S: The number of cache-sets.

L: Cache-line size in byte.

W_i: The number of non-faulty cache-ways in the ith cache-set.

T_c: The compacted trace of memory block addresses accessed.

F: The list of functions in the target program sorted in descending order of their execution counts.

M_{total}: The total number of cache misses.

M_{bound}: The upper bound of the number of cache misses. This parameter is given by the user.

The problem can be formally defined as follows:

"For a given S, L, a set of W_is, T_c, F, and M_{bound}, find **an order of functions,** for which M_{total} is less than M_{bound}."

D. Algorithm

```
Procedure Defect_Aware_Code_Placement
Input: S, L, a set of Wis, Tc, F and Mbound
Output: order of functions in the optimized object code
  Mmin = infinity;
  repeat
    for (t=0; t<|F|; t++) do
      p = F[t];   BESTlocation = p;
      for each p' ∈ F and p'≠ p do
        Insert function p in the place of p';
        Update Tc according to the locations of functions;
        Calculate Mtotal using (3);
        if (Mtotal ≤ Mmin) then
          Mmin = Mtotal;
          BESTlocation = p';
        end if
      end for
      Put function p in the place of BESTlocation
    end for
  until (Mmin < Mbound or Mmin stops decreasing)
  Output order of functions
end Procedure
```

Our algorithm starts from an original object code and finds the optimal location of each function 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 updates the trace of memory block addresses executed (T_c) according to the locations of functions and calculates the total number of cache misses (M_{total}) using (3). The ordering which yields

the minimum number of cache misses is selected. The algorithm continues as long as the number of cache misses reduces and is no less than M_{bound}. The computation time of the algorithm is quadratic in terms of the number of functions in the application program.

IV. EXPERIMENTAL RESULTS

We used three benchmark programs; *Compress* version 4.0, *JPEG encoder* version 6b, and *MPEG2 encoder* version 1.2. All programs are compiled with "–O3" option. We used GNU C compiler and debugger for ARMv4T architecture to generate address traces. Table I shows the number of functions, basic blocks and instructions for each benchmark program. The trace of each benchmark program is one million instructions long.

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

Figure 6 shows the cache access statistics for *JPEG encoder*. The y-axis shows the number of accesses to each cache-set. We used a 32Kb direct mapped cache. The number of cache-sets and cache-line size are 128 and 32 bytes, respectively. As one can see, accesses are not evenly distributed. Therefore, the increase in the number of cache misses depends on the location of defects. If a frequently accessed cache-set contains the defect, cache performance will degrade substantially. To take this into account, we considered three scenarios as follows,
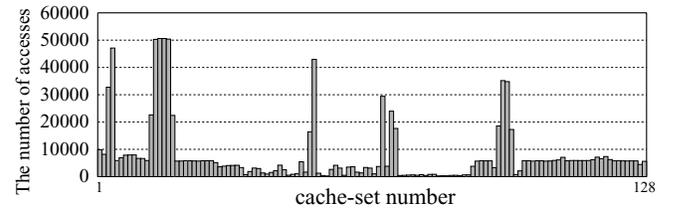

Figure 6.   Cache Access Statistics of JPEG encoder

- **Best Case**

  In this case, the number of accesses of any faulty cache-set is smaller than or equal to that of any non-faulty cache-set.

- **Worst Case**

  The number of accesses of any faulty cache-set is larger than or equal to that of any non-faulty cache-set.

- **Average Case**

  A defect may be in any cache-set with equal provability.

We first performed conventional code placement for a given application program. The conventional code placement technique tries to minimize the number of cache misses without considering cache defects. Then we analyzed the

above mentioned three cases. Solid lines in Figures 7-9 represent the results. Black dots represent results of our defect-aware code placement. We used the following four types of cache memories:

[Cache-1] A 32Kb direct mapped cache with 128 cache-sets whose cache-line size is 32 bytes.
[Cache-2] A 32Kb 2-way set-associative cache with 64 cache-sets whose cache-line size is 32 bytes.
[Cache-3] A 32Kb 4-way set-associative cache with 32 cache-sets whose cache line size is 32 bytes.
[Cache-4] A 16Kb 2-way set-associative cache with 32 cache-sets whose cache line size is 32 bytes.

In this experiment, we randomly chose faulty cache-lines. After that, we applied our defect-aware code placement. We regarded a chip whose cache miss rate is less than 1.1 times of the original miss rate (i.e., the miss rate of a defect-free cache) as an acceptable chip in this work. Therefore, our algorithm modifies object code such that the cache miss rate becomes less than 1.1x of the original cache miss rate. We tried 480 different patterns of faulty cache-lines for each benchmark program. Figures 7, 8, and 9 show the results for *Compress*, *JPEG encoder*, and *MPEG2 encoder*, respectively. Note that the solid graph for the best-case results is on the x-axis in Figure 7.
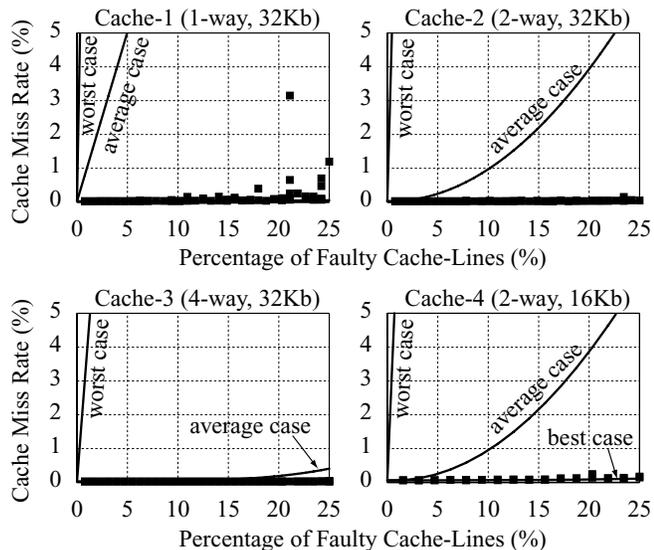

Figure 7. Results for Compress

The results show that our technique can compensate the performance degradation even when 25% of total cache lines are faulty. In the case of *Compress*, faulty cache-lines are tolerated by our code placement technique without noticeable performance degradation when less than 10% of total cache lines are faulty. The results for *MPEG2 encoder* demonstrate that our results are almost equal to or better than the results of the best-case scenario. This is due to the fact that our approach performs code placement after knowing the locations of faulty cache-lines. In this case our technique can offset degradations of miss rates for all types of caches when less than 10% of total cache lines are faulty.
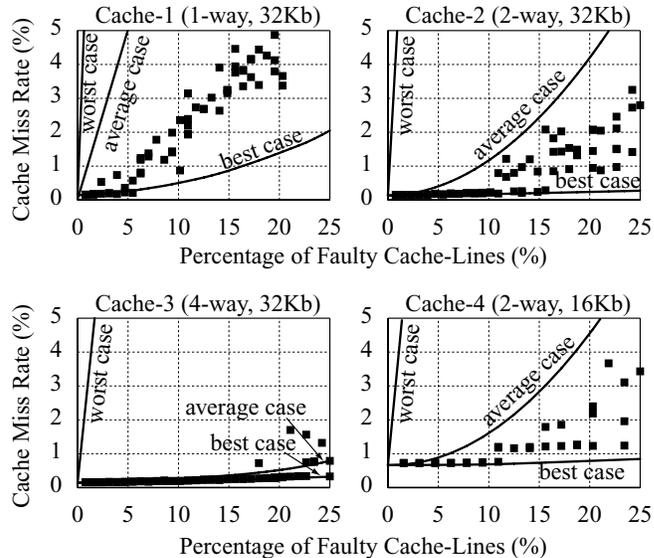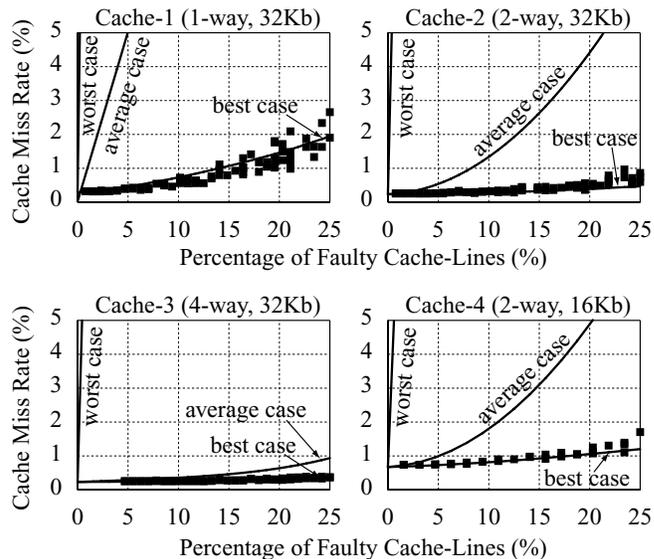

Figure 8. Results for JPEG encoder

Figure 9. Results for MPEG2 encoder

In practice, the number of defects in a cache is small. Otherwise, other parts of the chip will contain defects and the chip will not work. Therefore, we assumed a single faulty cache-line in the cache. The following Figures show results in this case. Figures 10-11, 12-13, and 14-15 show results for *Compress*, *JPEG encoder* and *MPEG2 encoder*, respectively. The x- and y-axis represent the location of the faulty cache-line and cache miss rate, respectively. In all cases, our approach reduced the cache miss rate compared to the conventional code placement. Especially for the direct mapped cache, our approach drastically reduced the cache miss rate (see Figures 10, 12, and 14). Since we assumed the conventional code placement technique is not aware of the faulty cache-line, the number of cache misses increases drastically if a defect exists on a frequently accessed cache-line. We assumed that a cache whose miss rate is less than

1.1x of the original cache miss rate is acceptable. Under this assumption, our algorithm recovered all chips with a 2-way set associative cache having a single faulty line (See Figure 11, 13, and 15). Supposing *Compress* as the target application executed on a processor with a direct mapped cache, 98% of chips with the single faulty cache-line are recovered by our code placement technique (See Figure 10). For *JPEG encoder* and *MPEG2 encoder*, our approach recovered all chips with defective direct mapped cache (See Figure 12 and 14).
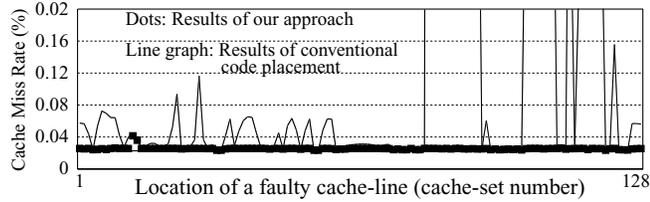

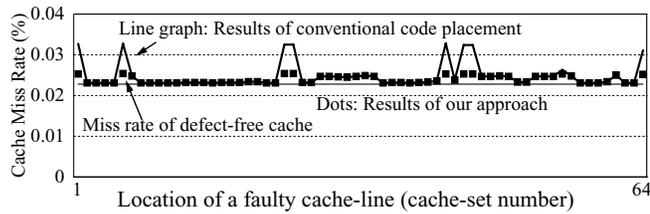Figure 10.   Results for Compress (Cache-1)

Figure 11.   Results for Compress (Cache-2)
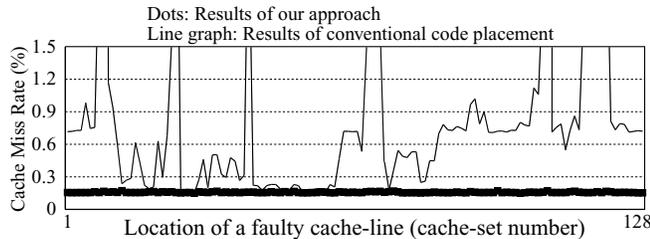

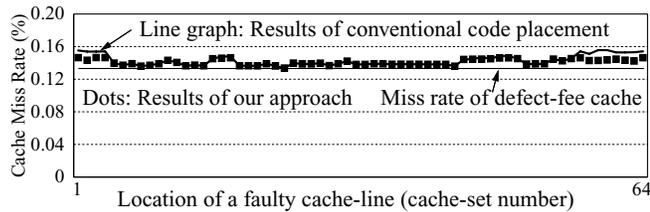Figure 12.   Results for JPEG Encoder (Cache-1)

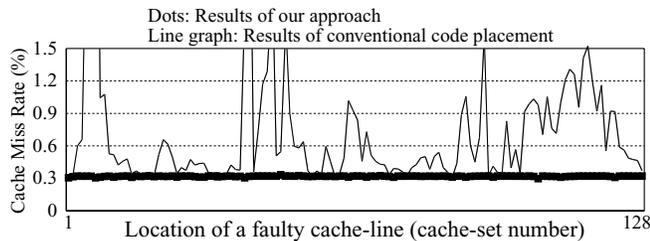Figure 13.   Results for JPEG Encoder (Cache-2)

Figure 14.   Results for MPEG2 Encoder (Cache-1)
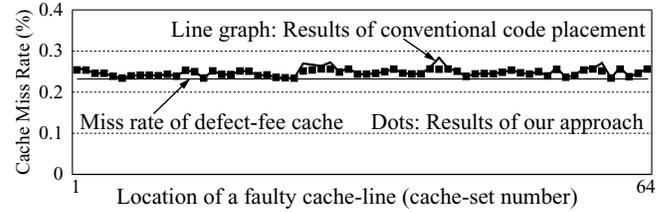

Figure 15.   Results for MPEG2 Encoder (Cache-2)

Assuming a single fault in the cache, we need at most N different object codes, where N represents the number of cache-sets (note an object code generated for a chip with a fault on line i may be good for a chip with a fault on line j as well). For small values of N, it is possible to pre-compile the code for all possible cases and store the object codes. For each chip, depending on the actual location of the fault, one of the object codes can be selected and programmed in the chip.

Table II shows the average compilation time (in seconds) assuming a single fault. We ran the defect-aware code placement algorithm on an UltraSPARC-II dual CPU workstation running Solaris8 at 450MHz with 2GB of memory. We regarded a chip whose cache miss rate is less than 1.1x of the original cache miss rate as an acceptable chip. For example in *MPEG2 encoder*, the original cache miss rate (i.e., the miss rate of a defect-free cache) was 0.3%. Therefore, chips whose cache miss rates were less than 0.33% were accepted as good chips. Assuming 30 cycles for cache miss penalty, the performance degradation of the defective chip will be only 0.83% compared to the performance of a defect-free chip. Similarly, our approach may degrade the performance of processors by 0.07% and 0.43% for *Compress* and *JPEG encoder*, respectively.

TABLE II.   CPU-TIME FOR DEFECT-AWARE CODE PLACEMENT (SECOND)

	Direct map	2-way cache	4-way cache
Compress	10.20	6.45	4.03
JPEG encoder	233.19	109.45	85.81
MPEG2 encoder	518.51	450.52	163.59

Since behavior of a program depends on its input value, an object code optimized for a specific input value is not necessarily optimal for the other input values. To see the effect of input value on the cache behavior, we measured cache miss counts for different input values. Figure 16 shows the results for six different input values for each benchmark program. The vertical axis represents the log of the cache miss rate. The object code was optimized for Data0. We measured the cache miss rate for three different cases as follows:

1.   an object code without considering cache defects (i.e., defect-unaware code) and ran on a processor with a defect-free cache (black bars in Figure 16).

2. the defect-unaware object code ran on a processor with a defective cache(gray bars).

3. a defect-aware code generated using our algorithm and ran on the processor with a defective cache (dark gray bars).

We used a direct mapped cache with 128 sets for JPEG encoder and MPEG2 encoder and a direct mapped cache with 32 sets for Compress (since the direct mapped cache with 128 sets is too large for Compress to see the effect of changing input values). We assumed two faulty cache-lines in each cache memory. As one can see, the object code optimized for Data0 achieves very good results for other input values too.
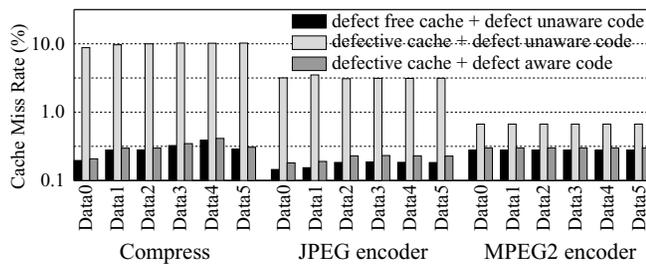

Figure 16.   Input Data Dependency

V.   CONCLUSION

In this paper, we proposed a defect-aware code placement technique. Experiments demonstrated that our code placement technique offset the impact of faults on performance in most cases when less than 5% of cache lines are faulty. In some cases the technique was able to offset the impact even in presence of 25% faulty cache-lines. We plan to extend our technique to handle data caches.

ACKNOWLEDGMENT

REFERENCES

[1] I. Koren and A. D. Singh, "Fault Tolerance in VLSI Circuits", IEEE Computer, special issue in fault-tolerant sisytems, vol.23, pp.73-83, July 1990.

[2] C. H. Stapper, A.N. McLaren and M. Dreckmann, "Yield Model for Productivity Optimization of VLSI Memory Chips with Redundancy and Partially Good Product", IBM Journal of Research and Development, vo.20, pp.398-409,1980.

[3] G. Sohi, "Cache Memory Organization to Enhance the Yield of High Performance VLSI Processors", IEEE Trans. on Computers, vol.38, no.4, pp.484-492, April 1989.

[4] A. F. Pour and M. D. Hill, "Performance Implications of Tolerating Cache Faults", IEEE Trans. on Computers, vol.42, no.3, pp.257-267, March 1993.

[5] X. Luo and J. C. Muzio, "A Fault-Tolerant Multiprocessor Cache Memory", Proc. IEEE Workshop on Memory Technology, Design and Testing, pp.52-57, August 1994.

[6] J. Montanaro, et al., "A 160 MHz, 32b 0.5W CMOS RISC Microprocessor", In Proc. of Int'l Solid-State Circuits Conference, February 1996.

[7] N. R. Saxena, et al., "Fault-Tolerant Features in the HaL Memory Management Unit", IEEE Trans. on Computers, vol.44, no.2, pp.170-179, February 1995.

[8] M. G. Gallup, et al., "Testability Features of the 68040", in Proc. of Int'l Test Conference, pp.749-757, September 1990.

[9] Y. Ooi, M. Kashimura, H. Takeuchi and E. Kawamura, "Fault-Tolerant Architecture in a Cache Memory Control LSI", IEEE Journal of Solid-State Circuits, vol.27, no.4, pp.507-514, April 1992.

[10] 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.

[11] P. P. Shirvani and E. J. McCluskey, "PADded Cache: A New Fault-Tolerance Technique for Cache Memories", In Proc. of 17th IEEE VLSI Test Symposium, pp.440-445, April 1999.

[12] M. A. Lucente, C. H. Harris and R. M. Muir, "Memory System Reliability Improvement Through Associative Cache Redundancy", In Proc. of IEEE Custom Integrated Circuits Conference, pp.19.6.1-19.6.4, May 1990.

[13] H. T. Vergos and D. Nikolos, "Performance Recovery in Direct-Mapped Faulty Caches via the Use of a Very Small Fully Associative Spare Cache", In Proc. of Int'l Computer Performance and Dependability Symposium, pp.326-332, April 1995.

[14] H. T. Vergos and D. Nikolos, "Efficient Fault Tolerant Cache Memory Design", Microprocessing and Microprogramming Journal, vol.41, no.2, pp.153-169, May 1995.

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

[16] 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.

[17] 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.

[18] 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.

[19] P. Panda, N. Dutt, and A. Nicolau, "Memory Organization for Improved Data Cache Performance in Embedded Processors", In Proc. of the 9th Int'l Symposium on System Synthesis, pp.90-95, November 1996.

[20] 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.

[21] 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.

[22] Motorola Inc., "PowerPC 604e RISC Microprocessor Technical Summary", 1996.

[23] IBM Microelectronics Division, "The PowerPC 440 core", 1999.

[24] S. Hill, "The ARM 10 Family of Embedded Advanced Microprocessor Cores", In Proc. of HOT-Chips 13, August 2001.

[25] K. Suzuki, T. Arai, N. Kouhei, and I. Kuroda, "V830R/AV: Embedded Multimedia Superscalar RISC Processor", IEEE Micro, vol.18, no.2, pp.36-47, April 1998.