PAPER

Dynamic Reconfiguration of Cache Indexing in Embedded Processors

Junhee KIM[†], **Sung-Soo LIM**[††], ***Nonmembers***, ***and*** **Jihong KIM**[†a)], ***Member***

SUMMARY Cache performance optimization is an important design consideration in building high-performance embedded processors. Unlike general-purpose microprocessors, embedded processors can take advantages of application-specific information in optimizing the cache performance. One of such examples is to use modified cache index bits (over conventional index bits) based on memory access traces from key target embedded applications so that the number of conflict misses can be reduced. In this paper, we present a novel fine-grained cache reconfiguration technique which allows an intra-program reconfiguration of cache index bits, thus better reflecting the changing characteristics of a program execution. The proposed technique, called dynamic reconfiguration of index bits (DRIB), dynamically changes cache index bits in the function level. This compiler-directed and fine-grained approach allows each function to be executed using its own optimal index bits with no additional hardware support. In order to avoid potential performance degradation by frequent cache invalidations from reconfiguring cache index bits, we describe an efficient algorithm for selecting target functions whose cache index bits are reconfigured. Our algorithm ensures that the number of cache misses reduced by DRIB outnumbers the number of cache misses increased from cache invalidations. We also propose a new cache architecture, Two-Level Indexing (TLI) cache, which further reduces the number of conflict misses by intelligently dividing indexing steps into two stages. Our experimental results show that the DRIP approach combined with the TLI cache reduces the number of cache misses by 35% over the conventional cache indexing technique.

key words: *cache indexing, cache organization, dynamic reconfiguration, embedded processor, microprocessor architecture*

1. Introduction

As embedded systems are employed to support more intelligent services with sophisticated functions, more powerful embedded processors are necessary. In high-performance embedded processors, as with general-purpose processors, reducing performance gap between the CPU and memory system is critical in achieving the maximum performance potential of the embedded processors. Since most high-performance embedded processors are based on cache memory, cache performance optimization becomes an important design goal in building high-performance embedded processors. Unlike general-purpose processors, embedded processors are designed with a set of target applications in mind. Therefore, it is possible to further optimize the design of cache architecture in embedded processors based on application-specific information. For example, several research groups have investigated cache reconfiguration techniques to improve the performance and energy efficiency of the cache memory [1]–[6].

Improving on-chip cache memory performance naturally involves modifications of indexing or hashing mechanism for cached data. Especially for embedded system applications with limited caching behaviors, such modifications could show significant impact. A number of researchers have proposed cache hashing techniques to obtain better cache performance in general-purpose processors [7]–[9], [14]. As even more aggressive techniques, the recent works proposed techniques to decide better cache hashing functions based on memory access traces of target embedded applications [1], [2]. In these techniques, the optimal cache index bits are extracted by analyzing memory traces. As expected, the index bits selected are quite different from the conventional least-significant bits. However, these techniques are still limited in that a single set of statically chosen index bits is used for target embedded applications. Since the memory access patterns can be drastically different even in the execution of a single application, a more fine-grained reconfiguration technique can be more effective than these techniques.

In this paper, we propose a novel cache indexing technique called dynamic reconfiguration of index bits (DRIB) which reconfigures index bits in a function-level granularity. Based on cache simulation results, we first select a set of functions in a program for which cache index bits are modified for better cache performance. We call the selected functions Index Bits Configured Functions (IBCF). The optimal index bits for each IBCF are extracted from the simulation results. One factor to consider in selecting the IBCFs is the potential performance degradation caused by frequent cache invalidations during cache index bits changes. We describe an algorithm to select the IBCFs which ensures that the number of cache misses reduced by DRIB outnumbers the number of cache misses increased from cache invalidations.

We also propose a new cache indexing architecture, called Two-Level Indexing (TLI) cache, which divides cache index bits into two levels (L1 and L2) so that the index bits are configured more aggressively depending on input memory traces. The L1 cache index bits are first configured based on the target applications and then the L2 cache index bits are configured depending on the sub-traces of target applications. Since the TLI cache increases the flexibility of

Manuscript received December 28, 2005.
Manuscript revised July 2, 2006.

[†]The authors are with the School of Computer Science and Engineering, Seoul National University, Seoul, 151–742 Korea.
[††]The author is with the School of Computer Science, Kookmin University, Seoul, 151–742 Korea.
a) E-mail: jihong@davinci.snu.ac.kr

DOI: 10.1093/ietisy/e90–d.3.637

cache index bits during executions of target applications, the combination of the TLI cache and DRIB mechanism gives a synergetic effect on improving the cache performance.

In order to evaluate the effectiveness of the proposed techniques, we have performed several experiments using a cache simulator. When the DRIB technique is used in the TLI cache, experimental results show that our approach can reduce the number of cache misses up to 35% over the conventional cache indexing technique. Compared to the static cache reconfiguration techniques [1], [2], the proposed dynamic approach reduced the number of cache misses up to 23%.

The rest of the paper is organized as follows. In Sect. 2, we introduce a dynamically reconfigured cache organization. Section 3 describes our compiler-directed DRIB approach. Section 4 presents an overview of the TLI organization while Sect. 5 shows the experimental results. Finally, we conclude with a summary in Sect. 6.

2. Overview of Dynamic Reconfiguration of Cache Indexing

2.1 Background

Givargis et al. proposed the idea of cache index bits selection based on target applications for the fist time [1]. In the study, optimal index bits are extracted from a given memory access trace using a heuristic algorithm and the cache index bits setting is performed by hard-wiring between core address lines and cache address lines [1]. This hard-wiring does not allow dynamic reconfiguration of cache index bits.

Patel et al. improved and extended the idea of Givargis et al. in [2]. They inserted multiplexors between the core and cache to support the reconfigurability of index bits as depicted in Fig. 1. This allows reconfiguration of cache index bits depending on target application sets. They implemented $n:1$ mux in Fig. 1 with NMOS pass transistors to minimize timing delay in [2]. Though the method improved the reconfigurability of cache index bits, the cache index bits configuration granularity is a single application. During the execution of a single application, each fragment in the program could show different caching behavior and thus the possibility of further optimization in cache index bits selection could exist. Our idea of finer-grained cache reconfiguration is based on such reasoning.

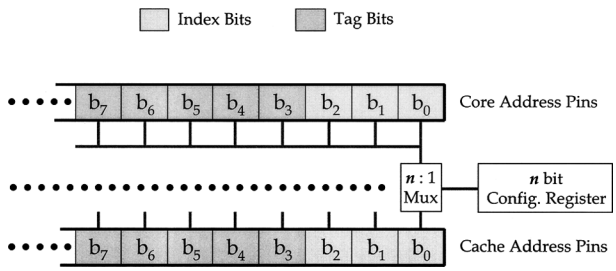

Fig. 1 Patel et al.'s cache index-bit reconfiguring circuit.

2.2 Basic Idea

Each of the program fragments has a different memory access pattern that requires different optimal index bits. Therefore, it would be better if the cache index bits could be dynamically reconfigured during program execution depending on the memory access pattern of each program fragment. DRIB is a mechanism that configures cache index bits dynamically according to the current execution context while running a single application. We call the existing caching indexing approach proposed in [2] static reconfiguration of index bits (SRIB) for distinction.

In order to illustrate the potential advantage of the DRIB over SRIB, consider Fig. 2 which shows the number of cache misses for 'cjpeg' application (a JPEG encoder) under different cache indexing approaches. The y-axis of Fig. 2 indicates the number of cache misses obtained from cache simulation under our experimental environment described in Sect. 5. 'SRIB for the whole program' means that optimal index bits are extracted from the memory trace of the whole program and SRIB is used based on the trace. This does not reduce cache misses at all compared to 'traditional indexing,' which means that the optimal index bits for the whole program trace are the same as the LSBs used for conventional cache indexing for 'cjpeg'. By analyzing the memory trace, we observe that the function 'pre_process_data' and its calling functions have significantly different memory access patterns from the rest of 'cjpeg', thus requiring a different set of index bits for cache performance optimization. Therefore, we would get totally different caching performance depending on the memory traces for which the cache indexing bits are configured: if index bits are only configured for 'pre_process_data', the caching performance for the rest of the functions in 'cjpeg' would be significantly degraded. However, in DRIB, different optimal index bits can be configured for both 'pre_process_data' and the rest of the functions dynamically and thus minimizes cache misses as shown in Fig. 2.

2.3 Index Bits Reconfiguration Issues

Although DRIB can offer additional opportunities for optimizing cache performance, it's not always helpful to use DRIB for all the program fragments. DRIB has an important side effect: cache invalidation. Whenever cache index bits are reconfigured during the execution of a program, all the existing cache lines must be invalidated because the existing cache lines will map to different memory blocks under the new cache indexing.

Figure 3 depicts an example of an incorrect cache operation that could happen in reconfiguring index bits. Let's assume that there is an 8-bit address machine and that this machine has an 8-set direct mapped cache. The address '00100011' is mapped to set '011' by original index bits b_0, b_1 and b_2 before reconfiguration, as seen in (a). After this

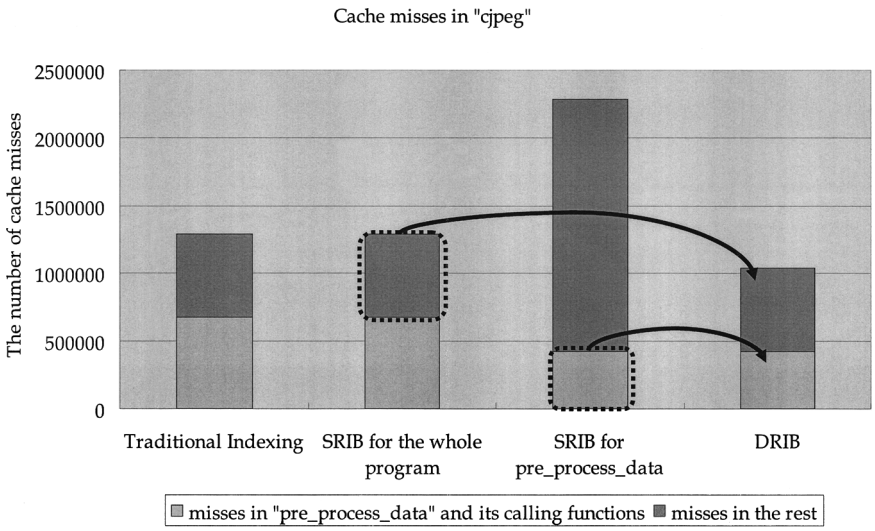

Fig. 2 Number of cache misses for 'cjpeg' under different cache indexing approaches.

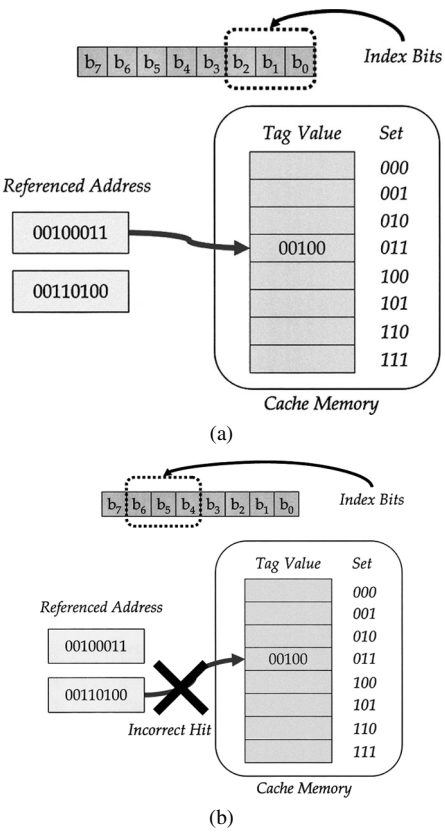

Fig. 3 An example of incorrect cache operations under the cache indexing reconfiguration: (a) before reconfiguration, (b) after reconfiguration.

reference, cache index bits are reconfigured as b_4, b_5 and b_6. Under these new index bits, the address '00110100' is also mapped to set '011'. These two references have the same tag value of '00100', which produces an incorrect cache hit, as seen in (b).

Consequently, the reconfiguration of index bits must accompany cache invalidation. Miss reduction by using DRIB might be outnumbered by miss increment resulting from cache invalidation in some program fragments; overall performance would worsen in this case. Thus DRIB must be used carefully considering its side effects. Therefore, a method to decide which program fragments should be selected for DRIB and how much is the benefit of selecting the fragments is required. We propose a compiler-directed approach for the program mentioned above in this paper.

3. Proposed DRIB Approach

3.1 Index Bits Configured Functions

In our DRIB approach, the unit of program fragments in a program to change cache indexing bits dynamically is a function. Therefore, to apply our DRIB, we need to select the functions (IBCFs) where the cache index bits could be reconfigured and the performance improvement due to the reconfiguration would be still profitable considering the cache invalidations caused by the reconfigurations. The selection is performed at compilation level and the compiler inserts cache index reconfiguration codes into the IBCFs selected. We call the optimal index bits for each of the IBCFs its *native index bits*. A typical example of IBCF is shown in Fig. 4. In function Q, the cache invalidation and index bits configuration codes are inserted by the compiler. For simplicity, we assume that execution returns to its caller function only at the end of the function body.

3.2 Algorithm for Selecting IBCFs

3.2.1 Selection Criteria for IBCFs

In this section, the detailed algorithm to select IBCFs among the functions in a program is presented. Before describing the detailed algorithm in a formal fashion, the selection criteria for IBCFs are intuitively introduced in this subsection. Since the selection criteria is directly related to

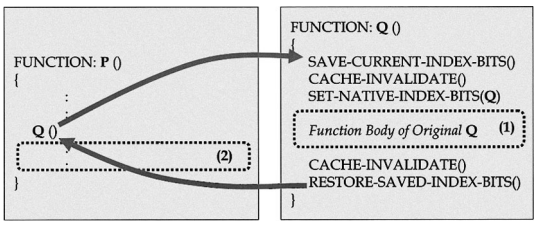

Fig. 4 An example of calling typical index bits configured function.

the additional cache misses caused by DRIB, we categorize the criteria into two different conditions depending on the scope of the effect of DRIB: the amount of additional cache misses in calling function caused by DRIB in called function (*inter-function memory reference relationship*) and the amount of additional cache misses in the called function (*intra-function memory reference relationship*). A key general condition is that the cache misses newly introduced due to the application of DRIB should be smaller than the cache miss reduction due to the reconfiguration of cache indexing by DRIB. For inter-function memory reference relationship, the cache misses by memory references newly introduced by DRIB due to the cache invalidation in called function (IBCF) should not be significant. For intra-function memory references, the cache misses in called function when DRIB is applied should not be larger than the cache misses when DRIB is not applied.

We describe the selection criteria using the example in Fig. 4. The inter-function memory references would be the memory references in (2) of P to the addresses that have existed in cache lines before the call of Q, which results in new cache misses by cache invalidation in Q. Cache miss increases in (2) is mainly due to these new misses[†]. In order to eliminate the possibility of significant cache miss increases for inter-function memory references, we could safely select the IBCFs so that the memory references in the IBCFs are sufficiently large to evict the cache contents filled in caller function even when DRIB is not applied. This condition is conservative, but safe. From this, we make the first condition for a function Q to be an IBCF as follows:

Condition 1: The number of memory references from Q while running the body of Q must be larger than the given threshold value, $Access_{threshold}$ which is obtained compared with the number of memory references in caller function.

The condition for intra-function memory references could be summarized in the following:

Condition 2: For the memory references occurring while running the body of Q, the number of cache misses when DRIB is used in Q must be smaller than when DRIB is not used in Q.

In the following subsection, we define the two conditions in formal way.

3.2.2 Terminology and Definition for the Proposed Algorithm

Our algorithm takes as inputs the function call graph without cycles, the symbol table, and the cache memory trace of a target application. Though original function call graphs may contain cycles, to be used in our IBCF selection algorithm, the call graphs should be converted to DAGs first using the method proposed in [10]. Because each function in a target application corresponds to one of the nodes in this DAG, we use functions and nodes interchangeably in the following explanation.

In our algorithm, memory trace is expressed as a 'set' of memory references, whose elements have the form of (i, a_i). The value i is an index and means that the reference is i-th reference in the whole memory trace T, thus there can be discontinuity in indexes of elements for sub-traces of T. The value a_i denotes the memory address of the corresponding memory reference. So total memory trace T is expressed as $T = \{(1, a_1), (2, a_2), \ldots, (L, a_L)\}$ where L is the number of total memory references.

First, we define the following Boolean values using terms of activation record and run-time stack in [15] when i-th memory reference of T occurred, for a node A and an edge e.

$$NODE\_CALL_A(i) = \begin{cases} true & \text{(if activation record of } A \text{ is in runtime stack)} \\ false & \text{(otherwise)} \end{cases}$$

$$EDGE\_CALL_e(i) = \begin{cases} true & \text{(if activation record of } e\text{'s tail function is in runtime stack and just over it in runtime stack, is activation record of } e\text{'s head function)} \\ false & \text{(otherwise)} \end{cases}$$

As shown in the definitions, *NODE_CALL* and *EDGE_CALL* must be obtained from runtime information. In order to determine these values using only cache simulation, we must also gather instruction cache memory traces even if the target cache is only the data cache. We can know the points when calling or returning of functions is occurred by comparing current and last instruction memory addresses while all the memory references are processed sequentially. That is, we can say that either calling or returning of functions is occurred when two functions whose area include current and last instruction address respectively on symbol table are not the same. In these cases, we can push a corresponding function to runtime stack if current instruction

[†]There are also new misses from memory references in (2) to the addresses that mapped to cache lines in (1). But they are relatively small.

address is one of entry addresses of functions on symbol table because calling for that function is occurred. Otherwise, we can do pop operations until the function includes current instruction address is on the top of stack because returning to that function is occurred.

In a given call graph, each node has the following node values:

- NE_A: the number of executions of function A while running a target application once.
- NT_A: sub-set of total memory trace T, all of whose element memory references occur while running the body of A. It is expressed as follows:

$$NT_A = \{(i, a_i) | (i, a_i) \in T \text{ and } NODE\_CALL_A(i) = true\} \quad (1)$$

- NM_A: the number of cache misses for memory references in NT_A.
- $NEM_{A,e}$: the number of cache misses for memory references in $NET_{A,e}$, which is expressed as Eq. (2). If an edge e cannot be reached from node A, $NEM_{A,e}$ must be always zero because $NET_{A,e}$ has no element.

$$\begin{aligned} NET_{A,e} = \{(i, a_i) | (i, a_i) \in NT_A \text{ and} \\ EDGE\_CALL_e(i) = true \text{ and} \\ e \text{ can be reached from } A\} \end{aligned} \quad (2)$$

When $NODE\_CALL_A$ is true, it means that corresponding function A is placed in somewhere of stack, not that it must be placed on top of stack. Therefore memory references occurred in other functions that function A calls are also included in NT_A. This means that all the cache memory references (and cache memory misses) in NT_A can be influenced by cache index bits configured in function A. So Using of NEM is very important in our approach. For example, let us assume there is a program that has the following simple call graph.

```
(A) --> (C) <-- (B)
```

As shown above, function C can be called by both A and B. We must know which function's index bits (among A and B) influence cache memory misses occurred in function C for each calling of it. The value of NM is not enough for this purpose because it is defined for only 'node' not 'edge'. Hence we present the concept of NEM.

NT_A and NE_A among node values of A cannot be changed for a fixed target application and memory trace. These are called constant node values and obtained before the beginning of the algorithm†. Unlike those values, NM_A and $NEM_{A,e}$ are dependent on the index bits used in cache simulation. These are called variable node values and must be initialized to zero. Variable node values are determined by two ways. The first is that function A is selected as an IBCF. In this case, NM_A and $NEM_{A,e}$ are obtained by cache simulation using optimal index bits for NT_A. In the second way, function A is not an IBCF, but all the variable node values of its predecessor nodes are already determined, when the node values of A can be obtained from its predecessor nodes by their definitions as follows:

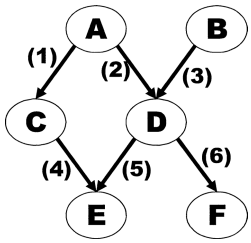

Fig. 5 An example of using node values.

$$NM_A = \sum_{j \in In-Edge_A} NEM_{\text{tail node of } j,\, j} \quad (3)$$

$$NEM_{A,e} = \sum_{j \in In-Edge_A} NEM_{\text{tail node of } j,\, e}$$
(only if edge e can be reached from node A) (4)

where $In-Edge_A$ is a set of all the incoming edges to A. The value of NM_A obtained by this way represents the number of cache misses occurring in NT_A when DRIB is not used in A.

We present a simple example as follows in order to describe usage of these node values on a given DAG. Let us suppose Fig. 5 is a function call graph of a target program. A, B, C, D, E and F are nodes(functions) and (1), (2), (3), (4), (5) and (6) are edges in that graph. If we perform cache simulation in function A and B for fixed index bits, we can get all the node values of A and B for those index bits while simulating. Now let's see function D. Function D is called only when either function A or B is in runtime stack. In other words, all the cache misses during running of function D are occurred when when either function A or B is in runtime stack. So, we can tell as follow.

$$NM_D = NEM_{A,(2)} + NEM_{B,(3)} \quad (5)$$

Of course, the values of NM_C, NM_E and NM_F can be obtained by the same way using node values of their callers'. These values of NM mean the number of cache misses occurred in the corresponding function when it is not an IBCF and caches are operating by using index bits of its caller's (or caller of caller's or so on).

$$NM_C = NEM_{A,(1)} \quad (6)$$

$$NM_E = NEM_{C,(4)} + NEM_{D,(5)} \quad (7)$$

$$NM_F = NEM_{D,(6)} \quad (8)$$

However, in case of NM_E and NM_F, the values of $NEM_{C,(4)}$, $NEM_{D,(5)}$ and $NEM_{D,(6)}$ are prerequisite. Therefore we must get the values of NEM also while processing

†$NET_{A,e}$ can also be one of constant node values. But we don't have to save it because it can be easily obtained from NT_A and $EDGE\_CALL_e(i)$.

```
SELECT-INDEX-BITS-RECONFIGURING-FUNCTIONS()
{
00  get constant node values for every node
01  set Checked[i] as false for each node i in a given call graph
02  CurrentNode = entry node of a given call graph
03  NM_CurrentNode = ∞
04  while (true)
    {
05    if ( n(NT_CurrentNode)/NE_CurrentNode ≤ Access_threshold ) goto line 11
06    IndexBits = GET-OPTIMAL-INDEX-BITS (NT_CurrentNode)
07    NumMiss = SIMULATE-CACHE-BY-GIVEN-INDEX-BITS (IndexBits, NT_CurrentNode)
08    if ( NumMiss + Overhead_reconfig. < NM_CurrentNode )
      {
09      SET-AS-INDEX-BITS-CONFIGURING-FUNCTION(CurrentNode, IndexBits)
10      update all the variable node values of CurrentNode by the values obtained in line 7
      }
11    Checked[CurrentNode] = true
12    if ( Checked[i] is true for all the nodes in a given call graph ) break
13    choose a node that has no unchecked predecessor node and set it as CurrentNode
14    compute NM_CurrentNode and NEM_CurrentNode using the node values of its predecessor nodes
    }
}
```

Fig. 6 Algorithm for selecting IBCFs.

function C and D as follows.

$$NEM_{C,(4)} = NEM_{A,(4)} \tag{9}$$

$$NEM_{D,(5)} = NEM_{A,(5)} + NEM_{B,(5)} \tag{10}$$

$$NEM_{D,(6)} = NEM_{A,(6)} + NEM_{B,(6)} \tag{11}$$

As shown in this example, exploiting these node values is very important and helpful because it can decide the number of cache misses without cache simulation in case of non-IBCF.

Now we can convert the two conditions presented in a previous sub-section to the following forms using notations defined in this sub-section:

Condition 1:

$$n(NT_A)/NE_A > Access_{threshold} \tag{12}$$

Condition 2:

$$NumMiss_{new\_indexing}(NT_A) + Overhead_{config.} < \sum_{j \in In-Edge_A} NEM_{tail\ node\ of\ j,\ j} \tag{13}$$

For our experiments, the value of $Access_{threshold}$ is set equal to the size of the target cache (in bytes). In Eq. (13), $NumMiss_{new\_indexing}(NT_A)$ means the number of cache misses resulting from cache simulation[†] for NT_A using its optimal index bits, and $Overhead_{config.}$ means the timing overhead for reconfiguring the cache. Therefore, the left side of Eq. (13) means the value of NM_A when A is assumed to be an IBCF and DRIB is used in A. On the contrary, the right side of Eq. (13) is the value of NM_A when A is not an IBCF, as seen in Eq. (3), and DRIB is not used in A. All the variable node values of A's predecessor nodes must be decided in advance to apply Eq. (13) to A. For this reason, our algorithm must be a top-down approach that begins in the entry node and proceeds to leaf nodes in a given call graph.

3.2.3 Overall Algorithm for Selecting IBCFs

Our algorithm for selecting IBCFs from among all the functions is described in Fig. 6. Our algorithm proceeds by examining each of the functions one by one from the entry node to see whether or not it satisfies the two conditions for IBCFs. In Fig. 6, the function 'GET-OPTIMAL-INDEX-BITS(T)' extracts and returns optimal index bits of a given memory reference trace T. This function can be implemented using algorithms in [1] or [2]. The function 'SIMULATE-CACHE-BY-GIVEN-INDEX-BITS(IBs, T)' simulates caching behavior with given index bits *IBs* for a memory reference trace T and returns the number of cache misses as a result. And the function 'SET-AS-INDEX-BITS-CONFIGURING-FUNCTION(N, IBs)' sets a function N and index bits *IBs* as one of IBCFs and its native index bits, respectively.

4. Two-Level Indexing Cache

The benefit of cache index bits reconfiguration would be more significant once we could utilize the diverse caching behaviors in a program more aggressively. In order to further improve the caching performance by cache bits reconfiguration, we propose more flexible cache indexing architecture called Two-Level Indexing (TLI).

[†]Cache simulation for the sub-trace is a little different from that for total trace T. Cache must be invalidated whenever there is discontinuity between the indexes of a last reference and a current reference while simulating each memory reference in the sub-trace by ascending order of its index. This is because cache re-configuring instructions must be executed at these discontinuous points in real execution using DRIB.

4.1 Concept

Cache optimization by using application-specific optimal index bits is based on the fact that different memory traces may have different optimal index bits. A key objective of TLI architecture is to maximally utilize the different caching behaviors of the memory traces and to apply different index bits for each memory trace as much as possible. This mechanism divides input memory reference of a program into sub-traces which are even finer-grained unit than the functions used until the previous section; the next step is to extract different index bits for each sub-trace. At the same time, the index bits in the TLI cache is divided into two steps, *L1 index bits* and *L2 index bits*. The set of memory references that is mapped to the same L1 index bit value is called the *super-set* for the L1 index bit value. In the TLI cache, each memory reference is mapped to one of the super-sets in the first-level indexing. For example, assume that b_0, b_1, b_2 and b_3 are index bits for a 16-set cache. In this case, sets 0, 1, 2 and 3 belong to the same super-set if b_2 and b_3 are L1 index bits because they have the same values of zero for b_2 and b_3. Within the same super-set, the memory references could be further divided into several groups depending on the values of L2 index bits.

Consequently, the total memory reference trace is divided into $2^{\textit{the number of L1 index bits}}$ sub-traces inside the cache. So if we extract and use different optimal L2 index bits for the sub-traces of each super-set, we can reduce cache conflict misses more than conventional one-level indexing, although all the super-sets have the same L1 index bits.

4.2 Motivational Example

The following example shows how the TLI cache could improve the cache indexing performance compared with conventional one-level indexing cache. Let us suppose that memory accesses occur in the order of index values shown in Table 1 assuming an 8-bit address machine.

Under the conventional one-level indexing scheme based on the algorithm presented in [1], [A_0, A_4, A_5, A_3, A_1, A_2, A_6, A_7] is selected as an optimal index bit order. If the target cache has 4 sets, A_0 and A_4 should be chosen for index bits according to this order and memory accesses in Table 1 should be hashed to each set as shown in Fig. 7 (a).

We can get better hashing results by using the TLI cache. Let us assume that a 4-set TLI cache has only one bit for L1 and L2 index bit, respectively. Memory accesses are divided into two groups as shown in Fig. 7 (b) when A_0 is used for L1 index bit. Then A_4 is used for L2 index bit in group of ([2] [5] [6] [8]) and A_3 is used in group of ([1] [3] [4] [7]). As shown in the example, memory accesses are distributed to each set more uniformly in the TLI cache.

4.3 Structure of the TLI Cache

The structure of the TLI cache is depicted in Fig. 8. In this figure, 'super-set' means the group of cache sets resulted from the first-level indexing. Each cache set included in the same super-set has the same value of first-level index bits. Each super-set has its own second-level row decoder, which is implemented using AND gates and linked to different L2 index bits. The notations $b_{L1,k}$, $b_{L2-s,k}$ and $b_{tag,k}$ mean the k-th first index bit, the k-th second index bit of super-set s and the k-th tag bit, respectively, in Fig. 8. Programmable index bit selectors proposed in [2] are inserted between the core and cache address pins of the TLI cache for reconfigurability, which are depicted as n:1 muxes in Fig. 8.

In the TLI cache, each super-set must have different tag bits essentially because the set has different L2 index bits; tag bits are the rest of the bits, except the offset bits and index bits. But a complex circuit must be inserted into the comparator parts to use different tag bits for each super-set. To avoid this complexity, we use all the L2 index bits as parts of the tag bits also, which makes a single cache line needs additional tag bits of the number of L2 index bits. So we limit the number of L2 index bits to two at most in consideration of this space penalty. However, there is no timing penalty in cache hit time of the TLI cache because a summation of the operating times of L1 decoder and L2 decoder for each super-set is the same as the operating time of a row decoder used in the conventional one-level indexing cache. And there is no additional delay in comparator due to increments of tag bits because the comparator checks all

Table 1 A sample address trace.

index	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
1	0	0	0	0	1	0	1	1
2	0	0	0	0	1	1	1	0
3	0	0	0	1	0	0	1	1
4	0	0	1	0	1	0	1	1
5	0	0	0	0	1	1	0	0
6	0	0	0	1	1	1	0	0
7	0	0	0	0	0	0	1	1
8	0	0	1	1	1	1	0	0

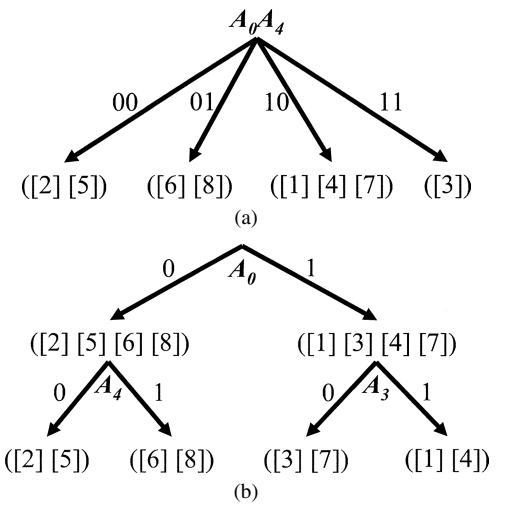

Fig. 7 An example of cache hashing results: (a) conventional one-level indexing scheme, (b) two-level indexing scheme.

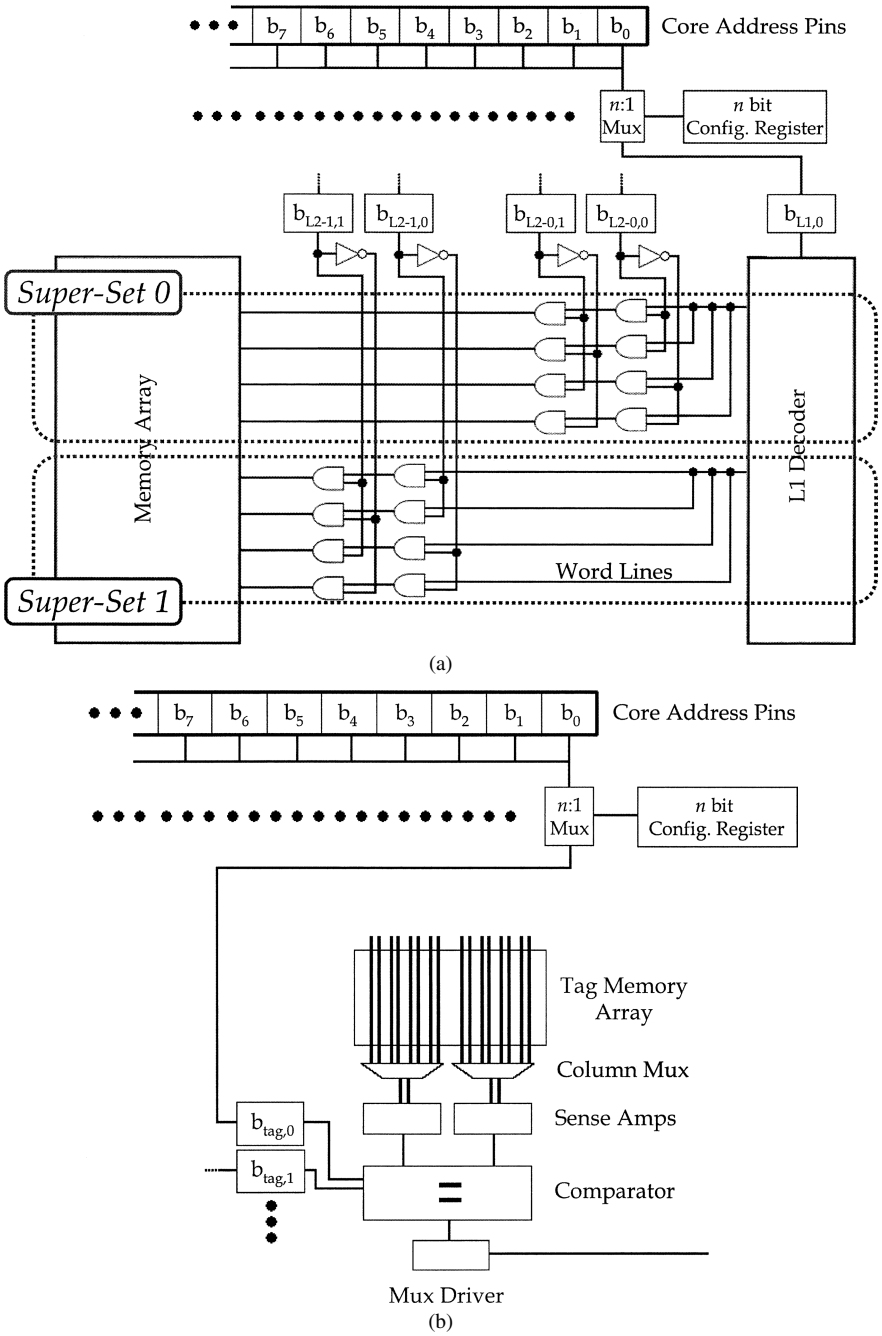

Fig. 8 Structure of a two-level indexing cache: (a) row decoder parts, (b) comparator parts.

the values of the tag bits simultaneously according to the CACTI model [11].

4.4 Method for Selecting Optimal Index Bits in the TLI Cache

Selecting optimal index bits in the TLI cache requires significantly complicated additional steps compared to the algorithm used for one-level indexing cache. In this paper, we devise a simplified algorithm for optimal index bits selection in TLI cache with modification of the algorithm used for the one-level cache: First, follow the steps in the conventional cache's index-bits-selecting algorithm for the total memory trace assuming that there is a direct-mapped cache that has $2^{\textit{the number of L1 index bits}}$ sets. Index bits selected by this are L1 index bits. Next, divide the total memory trace into sub-traces of each super-set using these L1 index bits. Finally, follow the steps in the conventional cache's index-bits-selecting algorithm for each sub-trace assuming that there is a cache the number of whose sets is $2^{\textit{the number of L2 index bits}}$ and other parameters that are the same as the real cache. As a result, all the L2 index bits for each super-set are obtained.

5. Experiments

5.1 DRIB Simulator

In order to evaluate the effectiveness of our proposed techniques, we have implemented a DRIB simulator. Figure 9 shows the overall structure of our DRIB simulator as well as the IBCF selector. In our DRIB simulator, the symbol table and call graph are extracted from the target application. The binary image of the target application is given for the input to SimpleScalar/ARM [12] in order to obtain the cache memory reference trace. The symbol table, call graph, and memory trace are used as the inputs for the IBCF selector. The IBCF selector produces a list of selected IBCFs and their native index bits.

The IBCF selector consists of three important modules: the main IBCF selecting module, the cache simulator, and the optimal index bits extractor. The main IBCF selecting module is the main part of the IBCF selector that runs the algorithm described in Fig. 6. It calls the other two modules while processing the algorithm. The cache simulator contained in the selector supports both conventional one-level indexing and our two-level indexing. The optimal index bits extractor is the module to extract optimal cache index bits from a given memory trace, which can be implemented using algorithms in [1] or [2]. We have implemented a new algorithm to extract the optimal cache index bits compensating the limitations of the existing algorithms in [1] and [2].

The outputs of the IBCF selector program, a list of IBCFs and their native index bits, can be used by the compiler for creating binary images that include cache index setting codes for DRIB. Instead of modifying a compiler and generating modified binary images from target applications, we used cache simulator using the IBCF selection information to simulate the caching behavior when DRIB is applied.

5.2 Results

We performed the experiments for our DRIB approach and the TLI cache for embedded applications from the PowerStone suite [13] (adpcm, des, fir, pocsag, ucbqsort and v42) and real multimedia applications, including the JPEG encoder (cjpeg), JPEG decoder (djpeg), MPEG audio decoder (mad) and MPEG-4 complaint video codec decoder (xvid). In our experiments, the target cache is a direct-mapped data cache whose total size is 1 KB and line size is 16 bytes and we set the value of $Overhead_{config.}$ as zero. Though the proposed DRIB method and the TLI cache architecture could be used for other cache configurations other than direct-mapped cache, we only show the results of direct-mapped cache configuration for the simplicity.

Depending on which approaches are chosen to be applied, there are four different implementation choices for cache indexing as summarized in Fig. 10. Among these,

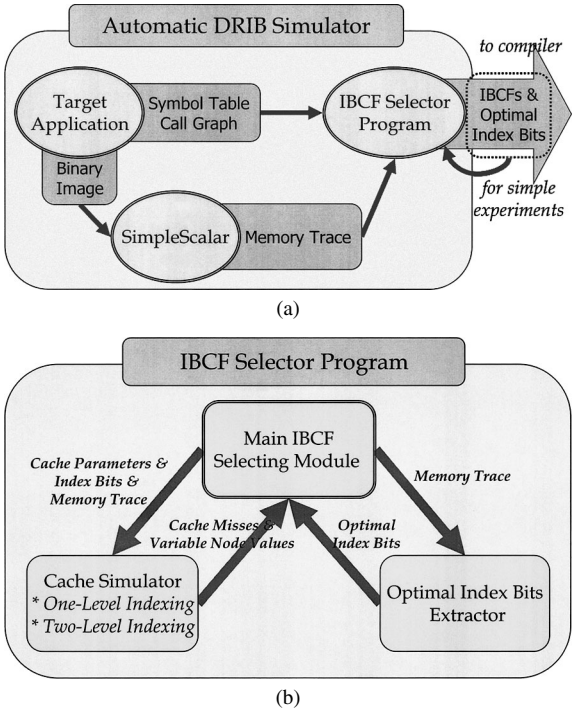

Fig. 9 Structure of automatic DRIB simulator: (a) overall structure, (b) IBCF selector program.

	SRIB	DRIB
One-level indexing cache	SRIB ONE-LEVEL	DRIB ONE-LEVEL
Two-level indexing cache	SRIB TWO-LEVEL	DRIB TWO-LEVEL

Fig. 10 Four implementation choices for cache indexing.

'SRIB ONE-LEVEL' means the approaches in [1], [2] because it is the case that optimal index bits are statically configured when a conventional one-level indexing cache is used.

Figure 11 shows the number of cache misses of the four choices in Fig. 10 normalized to traditional indexing that uses the LSBs for index bits.

Our DRIB approach could obtain more miss reduction than SRIB in a conventional one-level indexing cache, as shown in Fig. 11. Even in some cases when SRIB cannot reduce misses at all like 'des', 'cjpeg' and 'mad', our approach improves cache performance. In addition, Fig. 11 shows that our TLI cache reduces conflict misses in both SRIB and DRIB. In particular, we obtain the best results when our DRIB approach is used in the TLI cache, which is about 23% and 35% of miss reduction compared to static index bits configuring and traditional indexing in a conventional one-level indexing cache, respectively.

6. Conclusions

We proposed a compiler-directed and function-level cache indexing optimization technique called DRIB in which cache index bits are reconfigured based on the memory access behavior of each function. In order to make the

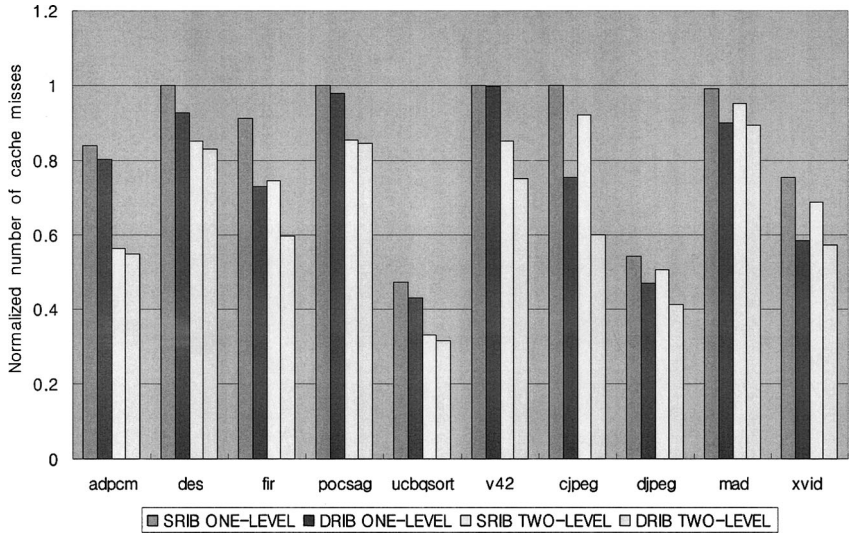

Fig. 11 The number of total cache misses normalized to traditional indexing.

function-level cache reconfiguration profitable, the cache miss reduction from the modified cache indexing should be bigger than the cache miss increment due to the cache invalidation for index bits reconfiguration. We described an efficient algorithm to select functions for which cache index bits are dynamically reconfigured. Our algorithm considers both program structure and memory access patterns.

In order to further reduce the number of conflict misses, we proposed a novel cache architecture, called the TLI cache. The TLI cache is based on two indexing stages. The first stage index divides cache accesses into several super-sets while the second stage index decides a cache set using its own row decoder and second-level index bits. So if we extract and use different optimal second-level index bits for each super-set, we can obtain better cache performance.

From the experiments based on simulation, we showed that our dynamic index bits reconfiguring technique and the TLI cache can be effective in reducing the number of conflict misses over the conventional cache indexing technique. For example, the DRIB approach combined with the TLI cache reduces the number of conflict misses by up to 35% over the conventional cache indexing technique.

Acknowledgments

This work was partly supported by the Brain Korea 21 Project and the MIC (Ministry of Information and Communication), Korea, under the ITRC (Information Technology Research Center) support program supervised by the IITA (Institute of Information Technology Assessment). The ICT (Institute of Computer Technology) at Seoul National University provides research facilities for this study.

References

[1] T. Givargis, "Improved indexing for cache miss reduction in embedded systems," Proc. IEEE 2003 DAC, pp.875–880, Anaheim, CA, June 2003.

[2] K. Patel, L. Benini, E. Macii, and M. Poncino, "Reducing cache misses by application-specific reconfigurable indexing," Proc. IEEE 2004 ICCAD, pp.125–130, San Jose, CA, Nov. 2004.

[3] R. Balasubramonian, D.H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general purpose processor architectures," Proc. IEEE 2000 MICRO, pp.245–257, Monterey, CA, Dec. 2000.

[4] P. Petrov and A. Orailoglu, "Towards effective embedded processors in codesigns: Customizable partitioned caches," Proc. IEEE 2001 CODES, pp.79–84, Copenhagen, Denmark, April 2001.

[5] C. Zhang, F. Vahid, and W. Najjar, "A highly configurable cache architecture for embedded systems," Proc. IEEE 2003 ISCA, pp.136–146, San Diego, CA, June 2003.

[6] D.H. Albonesi, "Selective cache ways: On-demand cache resource allocation," Proc. IEEE 1999 MICRO, pp.248–259, Haifa, Israel, Nov. 1999.

[7] A. Seznec, "A case for two-way skewed-associative caches," Proc. IEEE 1993 ISCA, pp.169–178, San Diego, CA, May 1993.

[8] A. Agarwal and S.D. Pudar, "Column-associative caches: A technique for reducing the miss rate of direct mapped caches," Proc. IEEE 2003 ISCA, pp.179–180, San Diego, CA, May 1993.

[9] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee, "Using prime numbers for cache indexing to eliminate conflict misses," Proc. IEEE 2004 HPCA, pp.288–299, Madrid, Spain, Feb. 2004.

[10] S. McFarling, "Program optimization for instruction caches," Proc. IEEE 1989 ASPLOS, pp.183–191, Boston, Mass, April 1989.

[11] S.J.E. Wilton and N.P. Jouppi, "CACTI: An enhanced cache access and cycle time model," IEEE J. Solid-State Circuits, vol.31, no.5, pp.677–688, May 1996.

[12] T.M. Austin, "The SimpleScalar/ARM Toolset," http://www.eecs.umich.edu/taustin/simplescalar

[13] A. Malik, B. Moyer, and D. Cermak, "A low power unified cache architecture providing power and performance flexibility," Proc. IEEE 2000 ISLPED, pp.241–243, Rapallo, Italy, July 2000.

[14] A. Gonzalez, M. Valero, N. Topham, and J.M. Parcerisa, "Eliminating cache conflict misses through XOR-based placement functions," Proc. IEEE 1997 ICS, pp.76–83, Vienna, Austria, July 1997.

[15] A.V. Aho, R. Sethi, and J.D. Ulman, Compilers - Principles, Techniques, and Tools, Addison-Wesley, 1986.

Junhee Kim received the B.S. and M.S. in computer science and engineering from Seoul National University, Korea in 2003 and 2005, respectively. He is currently an engineer at the Samsung Advanced Institute of Technology. His research interests include low-power systems, embedded systems, and computer architecture.

Sung-Soo Lim received the B.S., M.S., and Ph.D. in electrical and computer engineering from Seoul National University, Korea in 1993, 1995, and 2002, respectively. He is currently an assistant professor in the School of Computer Science, Kookmin University, Korea. Previously, he was the Chief Technical Officer in PalmPalm Technology Co. Ltd., Korea from 2001 to 2004. His research interests include real-time embedded systems, mobile handheld device architecture, real-time operating system, and computer architecture. He is a member of the IEEE.

Jihong Kim received a B.S. in computer science and statistics from Seoul National University, and an M.S. and Ph.D. in computer science and engineering from the University of Washington. He is currently an associate professor in the School of Computer Science and Engineering, Seoul National University, Korea. His research interests include computer architecture, embedded systems, and multimedia and real-time systems. He is a member of the IEEE and ACM.