Compiler-Guided Code Restructuring for Improving Instruction TLB Energy Behavior*

I. Kadayif[1], M. Kandemir[2], and I. Demirkiran[3]

[1] Canakkale Onsekiz Mart University, Canakkale, Turkey
kadayif@comu.edu.tr
[2] Pennsylvania State University, University Park, PA, USA
kandemir@cse.psu.edu
[3] Syracuse University, Syracuse, NY, USA
idemirki@eecs.syracuse.edu

Abstract. This paper presents a compiler-directed strategy for reducing energy consumption of instruction TLBs. The main idea is to restructure the code to increase the chances that one can generate virtual-to-physical address translation without going through the instruction TLB. The preliminary experimental results are promising.

1 Introduction and Motivation

TLB (translation lookaside buffer) is a crucial component that maintains recent virtual-to-physical address translations. Optimizing energy consumption of TLB is critical due to two main reasons. First, since this component is accessed at each memory reference, it can contribute to a significant fraction of on-chip energy budget. For example, instruction and data TLBs are known to contribute to over 15% of on-chip energy consumption in SH-3 and Intel StrongARM [14]. Second, since TLB is a very small component, its power density can be quite high as a result of frequent accesses. Therefore, reducing TLB energy consumption can be very important. TLB optimization has been focus of several circuit and architecture level studies. For example, Juan et al [7] proposed modifications to the basic cells and to the structure of TLBs that led to15% improvement in per access energy consumption. Choi et al [3] proposed a two-way banked filter TLB and a two-way banked main TLB. Balasubramonian et al [2] and Delaluz et al [5] proposed changing the TLB configuration dynamically, based on the need of the application at a given execution phase.

In a recent work, Kadayif et al [8] has demonstrated that using an optimizing compiler and a suitable help from the hardware, one can generate most of virtual-to-physical address translations automatically, thereby reducing the frequency of iTLB accesses. Their compiler-directed strategy operates as follows. Overall philosophy of this strategy is to perform the translation for a page once, and subsequently keep reusing it directly without going to the iTLB, as long as it does not change. This is achieved through the use of a register called Current Frame Register (CFR), whose format is of the form:

<Virtual Page Number, Physical Frame Number, Protection/Other Bits>.

* This research is partly supported by NSF Career Award #0093082.

M. Danelutto, D. Laforenza, M. Vanneschi (Eds.): Euro-Par 2004, LNCS 3149, pp. 304–309, 2004.

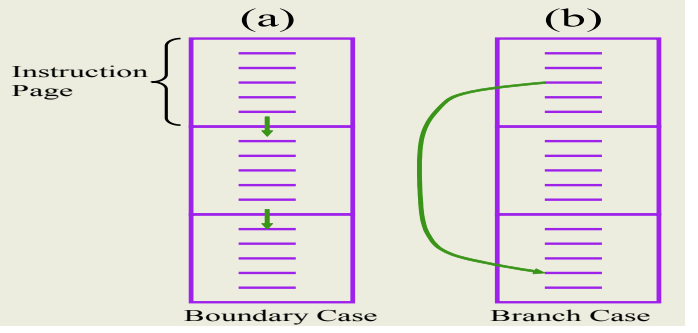

Fig. 1. Two possible instruction page transitions during execution. (a) Boundary case. (b) Branch case.

Basically, this register holds the current virtual-to-physical address translation. As long as we are sure that the current address translation does not change (i.e., we are within the same page), we take the translation from the CFR without going to the iTLB. Whenever there is a page change, we need to re-load (update) the CFR. This occurs in two scenarios in normal execution as shown in Figure 1: (a) two successive instructions, which are on page boundaries (we refer to this as the boundary case), i.e., one is the last instruction of a page, and the next is the first instruction of the next page (we assume that instructions are aligned so that a single instruction does not cross page boundaries), and (b) explicit branch instructions whose target is in a different page (we call this the branch case).

In the compiler-based scheme proposed in [8], the compiler is used to determine both the boundary and branch cases. The branch cases are handled as follows. The compiler assumes that the branch target is within the same page as the branch instruction if the static analysis of the code by the compiler can reveal (with 100% accuracy) that this indeed is the case (note that this typically occurs when branch targets are given as immediate operands or as PC relative operands). Otherwise, if the branch is unanalyzable or its target is proven to go outside the current instruction page, the compiler conservatively assumes a page change and updates the CFR contents via an iTLB access. The necessary compiler support for implementing this involves checking whether the target of a statically analyzable branch is on the same page of the branch itself. To handle the boundary case, the compiler inserts an explicit branch instruction at the end of each instruction page, with the target being the very next instruction (the first one on the next page). Note that this mechanism does not much affect iL1 and L2 hits or misses, and thus it does not affect the rest of the memory system energy consumption.

While the experimental results given in [8] indicate significant energy benefits without much performance overhead, note that *one can achieve even better energy savings by restructuring the code.* Specifically, what we need to do is to increase the number of branches whose targets can be proven to be within the same page (as the branch instruction itself). In the rest of this paper, we describe a compiler-directed strategy to achieve this.

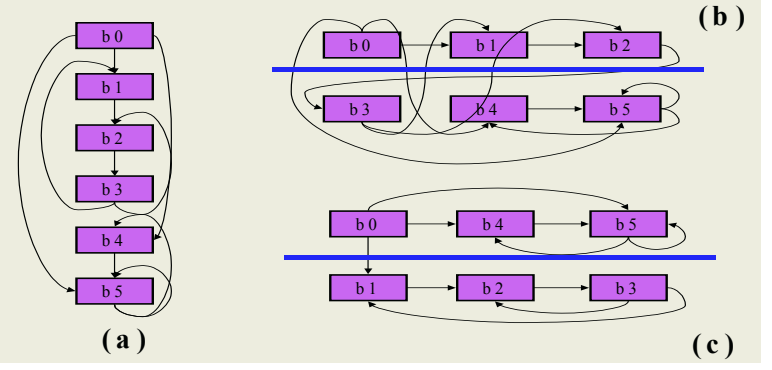

Fig. 2. (a) An example code fragment. (b-c) Two alternate page assignments.

2 Code Restructuring for Translation Reuse

We can define our problem as follows. Consider the code layout shown in Figure 2(a). Let b_i represent the size of basic block i, and P be the page size. We use B to denote the set of all basic blocks in the code. The connections between basic blocks (which correspond to the edges in the control flow graph representation of the program) are indicated using edges $e_{ij} = (b_i, b_j)$. Each edge e_{ij} also carries a weight, denoted w_{ij}, which indicates the importance of satisfying the edge. In this context, strictly speaking, "satisfying edge e_{ij}" means colocating b_i and b_j within the same instruction page. However, we will use a relaxed version of this definition, which says if b_i and b_j are stored in memory one after another, e_{ij} is satisfied. Given an assignment of basic blocks to memory (denoted using mapping M), it is likely that some edges will remain inside a page, whereas some other edges will cross the page boundaries; these two sets of edges are referred to as *in-page edges* and *out-page edges* in the rest of this paper. Then, the *cost* of such an assignment can be formulated as: $C(B,M) = \Sigma_i \Sigma_j w_{ij}$ such that e_{ij} is an out-page edge. That is, the total weights of all out-page edges give us the cost of mapping M. Obviously, our objective is to place basic blocks into pages such that C(B,M) is minimized. In other words, we want to find an M that minimizes C(B,M).

In this paper, we propose an algorithm for solving this problem of determining M (a memory assignment) for all blocks in B as shown below:

This strategy, which is actually a heuristic, operates in a page size (P) oblivious manner. Consequently, it adopts the relaxed definition of the concept of satisfying an edge, as described above. This heuristic is similar to the Kruskal's spanning tree algorithm [4], and is given below. In this algorithm, E is the ordered set of edges (according to their weights), and T is the (spanning) tree to be built.

```
1.   Order the edges according to non-increasing weights
2.   While (there is a node to be included in T)
  2.1.   Select the edge with the largest weight
  2.2.   If (adding the selected edge to T does not increase
           the degree of an node in T to 3 AND does not
           create a cycle in T) then
         add the selected edge to T
3.   Traverse T and store the nodes (basic blocks)
     that are directly connected consecutively in memory
```

Basically, this algorithm builds a spanning tree (which includes all nodes in B and a subset of edges in E), with the property that no node (basic block) is connected to more than 2 neighbors. This is done so to guarantee that each pair of nodes with a connected edge will be stored in memory consecutively (since in memory layout a basic block can have only two neighbors). This algorithm is similar to the one presented in [9] for assigning program variables to memory locations in DSPs. The main difference is that we work on basic blocks not variables since our objective is to restructure code layout for iTLB energy savings. Note also that this algorithm is completely different from prior work on instruction cache optimization [13, 6, 11, 10] as we use a different representation and solution method. It is to be noted, while this algorithm does not take into account the page size (P), we do not expect this to be a major problem in practice. This is because the algorithm will fail to optimize only in cases where the two basic blocks with high affinity fall into different pages. Given the large page sizes, we do not expect this to occur very frequently.

Figures 2(b) and (c) illustrate two possible page assignments, assuming, for simplicity, that each page can hold three basic blocks. The first alternative (shown in (b)) is the straightforward one, whereas the second (the one in (c)) is the one generated by our approach (under the assumption of same weights for all edges between the basic blocks). In both the cases, the thick line delineates the page boundary. Note that, in our case, there is only one out-page edge, while we have five out-page edges in the straightforward option.

3 Preliminary Experiments

We have implemented the proposed strategy and performed experiments with several Spec95 applications. The experiments have been performed using SimpleScalar [1], and the energy numbers have been obtained through CACTI [12]. In Figure 3(a), we present the *normalized* energy consumptions for two different strategies. The first strategy (marked opt-1) is the one presented in [8]. It uses the CFR to the fullest extent possible, but does not restructure code for the CFR reuse. The second strategy (marked opt-2) represents the results obtained through the compiler-based code restructuring strategy discussed in this paper. For each benchmark, both the bars are normalized with respect to the iTLB energy consumption of the original (default) case, where we do not make use of CFRs, and all instruction accesses go through iTLB. Note that in both opt-1 and opt-2, all extra energy consumptions due to CFR accesses have been included in the results. We observe from these results that, in five out of six benchmarks, our approach improves the iTLB energy consumption beyond the method presented in [8]. That is, restructuring application codes for CFR reuse is useful in practice. The performance (execution cycle) results are given in Figure 3(b). As before, all the bars are *normalized* with respect to the default version, where no iTLB energy-saving technique is used. We see that, while opt-1 generates (almost) the same results with the default version, opt-2 slightly improves performance as well. This is because our approach also enhances instruction cache locality by bringing the blocks with temporal affinity together. Therefore, we can conclude that the proposed scheme brings both energy and performance benefits. Our on-going work involves implementing an optimal scheme for basic block re-ordering (based on integer linear programming), and comparing it against the scheme proposed in this paper.

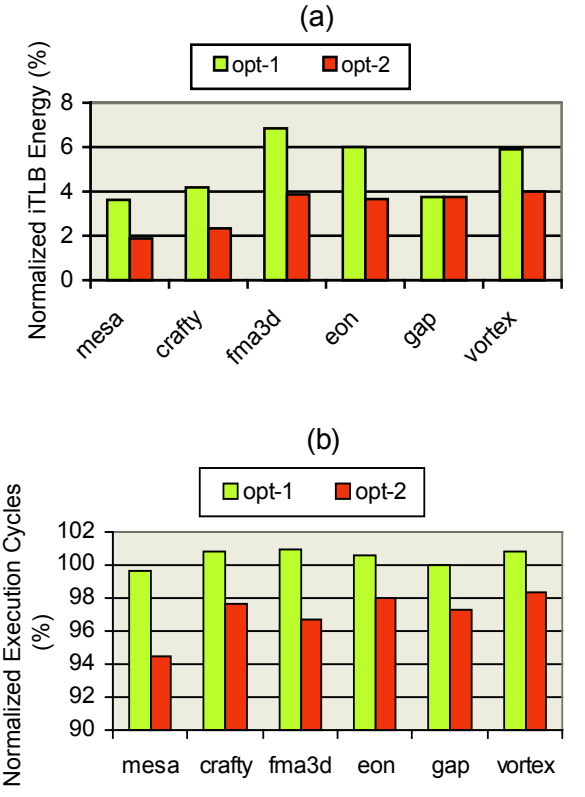

Fig. 3. (a) Normalized iTLB energy. (b) Normalized execution cycles.

References

1. T. Austin, E. Larson, and D. Ernst. SimpleScalar: an infrastructure for computer system modeling. *IEEE Computer Magazine,* pp. 59-67, Feb 2002.
2. R. Balasubramonian, D.H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proc. 33rd International Symposium on Microarchitecture,* pp. 245--257, December 2000.
3. J-H. Choi, J-H. Lee , S-W. Jeong , S-D. Kim , and C. Weems. A low-power TLB structure for embedded systems. *IEEE Computer Architecture Letters,* Volume 1, January 2002.
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms,* Second Edition, The MIT Press, 2001.
5. V. Delaluz, M. Kandemir, A. Sivasubramaniam, M. J. Irwin, and N. Vijaykrishnan. Reducing dTLB energy through dynamic resizing. In *Proc. the 21st International Conference on Computer Design,* San Jose, California, October, 2003.
6. N. Gloy, T. Blackwell , M. D. Smith, and B. Calder. Procedure placement using temporal ordering information. In *Proc.the 30th ACM/IEEE International Symposium on Microarchitecture,* p.303-313, December 01-03, 1997.

7. T. Juan, T. Lang, and J. J. Navarro. Reducing TLB power requirements. In *Proc. International Symposium on Low Power Electronics and Design,* 1997.
8. I. Kadayif, A. Sivasubramaniam, M. Kandemir, G. Kandiraju, and G. Chen. Generating physical addresses directly for saving instruction TLB energy. In *Proc. International Symposium on Microarchitecture,* Istanbul, Turkey, November 2002.
9. S. Liao, S. Devadas, K. Keutzer, S. W. K. Tjiang, and A. Wang. Storage assignment to decrease code size. In *Proc. International Symposium on Programming Language Design and Implementation,* pp. 186-195, 1995.
10. S. McFarling, Procedure merging with instruction caches. *ACM SIGPLAN Notices*, v.26 n.6, p.71-79, June 1991.
11. K. Pettis and R. C. Hansen. Profile guided code positioning, *ACM SIGPLAN Notices,* v.25 n.6, p.16-27, June 1990.
12. G. Reinman and N. P. Jouppi. CACTI 2.0: an integrated cache timing and power model. *Research Report 2000/7,* Compaq WRL, 2000.
13. A. D. Samples and P. N. Hilfinger. Code reorganization for instruction caches. *Technical Report UCB/CSD 88/447,* University of California, Berkeley, October 1988.
14. SH-3 RISC processor family. http://www.hitachi-eu.com/hel/ecg/products/micro/32bit/sh_3.html.