Code Transformations for TLB Power Reduction*

Reiley Jeyapaul, Sandeep Marathe and Aviral Shrivastava

Compiler and Microarchitecture Laboratory, Arizona State University, Tempe, AZ 85281 USA
Email : {reiley, sandeep.marathe, aviral.shrivastava}@asu.edu

Abstract

The Translation Look-aside Buffer (TLB) is a very important part in the hardware support for virtual memory management implementation of high performance embedded systems. The TLB though small is very frequently accessed, and therefore not only consumes significant energy, but also is one of the important thermal hot-spots in the processor. Recently, several circuit and microarchitectural implementations of TLBs have been proposed to reduce TLB power. One simple, yet effective TLB design for power reduction is the Use-Last *TLB architecture proposed in [9]. The Use-Last TLB architecture reduces the power consumption when the last page is accessed again. While very effective for instruction TLB, this technique is not as effective for the data TLB. In this paper, we propose compiler techniques (specifically, instruction and operand reordering, array interleaving, and loop unrolling) to reduce the page switchings in data accesses. Our comprehensive page-switch reduction algorithm results in an average of* 39% *reduction in the data-TLB page switching, and therefore power with negligible variation in performance on benchmarks from MiBench, Multimedia, DSPStone and BDTI suites.*

1. Introduction

Power, energy and thermal issues in current and near future digital systems form the crux of the biggest challenge that the semiconductor industry faces today. In high-end computing, power consumption limits the amount of achievable performance because of exorbitant increase in the cost of heat removal mechanisms. In battery operated portable systems, the battery is the single largest factor in device cost, weight, recharging time, frequency and ultimately the usability of the system. Translation Look-aside Buffer or TLB is an important component of high-end multi-tasking embedded processors, like the Intel XScale. The TLB performs virtual to physical address translation and determines page access permissions. Most modern processors, including the Intel XScale implement virtually-addressed caches, in which the cache lookup is directly performed on the virtual address provided by the processor, and therefore the TLB lookup comes in the critical path. Elkman et al. [11] note that the TLBs can consume 20-25% of the total $L1$ cache energy. Kadayif et al. [14] observed high power densities of the data-TLB, as compared to the data-$L1$ cache. Thus reducing the power consumption of TLBs is an important research problem.

Several TLB designs have been proposed to trade-off the TLB lookup delay, area and power consumption [12, 15]. One simple, yet effective technique for TLB power reduction proposed in [10, 9], is the *Use-Last* TLB architecture. Observing that there is a high probability that instruction access will refer to the same page as the last one, they store the previous page translation information into a latch, and thereby reduce the TLB lookup power. The Use-Last TLB architecture is able to reduce the instruction TLB power by 75%. However, since data accesses do not exhibit as high locality as instructions, this microarchitectural technique was not effective for data TLBs.

We develop compiler techniques to reduce the power consumption of the Use-Last TLB architecture by improving the locality of data accesses. We propose a novel instruction scheduling and operand reordering technique, heuristic for deciding when to perform array interleaving, and loop unrolling to minimize the page switchings between consecutive TLB accesses while minimizing performance loss. Our combined technique can reduce the TLB switches by 39%, with minimal performance impact on benchmarks from MiBench, Multimedia, DSPStone and BDTI suites. Note that this improvement is above and beyond what the Use-Last hardware technique alone could achieve.

2. Related Work

Several researchers have proposed efficient circuit-level, microarchitectural and software techniques to reduce the power consumption of the TLB and the Memory Management Unit.

2.1. Compiler based Approaches

A compiler-directed array interleaving technique [17] was proposed to save energy in multi-bank memory architectures with power control features. In this, the arrays used in separate banks are interleaved such that only one of the banks is active and the other can be powered down, thus saving energy. The energy reduction achieved by this technique does not account for the leakage power of the SRAM cells during standby mode. Parikh et al in [18] schedule instructions within a block based on the minimum obtainable value for a weighted cost function:*circuit-state cost*. One recent work is [19], where energy reduction is achieved through effective utilization of resources by switching between two processor modes based on the cache misses.

2.2. Closest Approach

The work closest to our approach, is by Kandemir et al. [13]. Their compiler technique is to increase the effectiveness of a previously proposed architectural technique that uses *Translation Registers* or TRs. The addition of TRs requires changing the ISA, which may not be desirable in many cases. In contrast,

*This work was partially funded by grants from Raytheon and Startdust Foundation.

our approach is to improve the effectiveness of Use-Last TLB architecture, which exists in the Intel XScale processor. They have to profile the code to find out which page will be accessed frequently in the near future, and then generate code to load the translations to that page into TRs. In comparison, our approach is a static technique. We do not need/use profile information. Not only that profile-based compilation is limited in application and scope, it has huge overhead in terms of compilation time. Our technique does not have any such overheads. Finally, in their technique, the code is modeled as nodes which represent loop nests that access data from a particular page region. Code transformations to enhance the use of TRs are directed at scheduling these loop nests (nodes that access data from a particular page region) together. In contrast, our approach is to schedule and transform instructions so that the accesses to the same page are grouped together. Our technique operates at a finer granularity than theirs, and could therefore co-exist, and enhance the effectiveness of each other.

3. *Use-Last* TLB Architecture

Our compilation approach enhances the effectiveness of an already effective and popular TLB architecture, the *Use-Last* TLB architecture. Proposed in [9], the Use-Last TLB architecture utilizes a modified TLB-CAM structure. The virtual address input is matched with the TLB tag through the CAM cells (which has reduced power consumption). The TLB tag is then used to retrieve the mapped physical address from the register files. The lookup on the register files is a power consuming process because of the bit-line and word-line drivers and other associated circuitry involved in its operation. The key factor in this architecture design is the latch used to store the tag address of the previously accessed address. If the two TLB tag addresses match, the page address and access information at the output is the same for both. In this case, the word line (WL) of the register files are not activated and the switching energy of the RF cells and associated circuitry is eliminated. The effectiveness of this technique was demonstrated on instruction TLB, and it was shown to reduce the power consumption by 75%. However, this technique was deemed un-useful for data caches, as data accesses in general do not exhibit high data locality as compared to instruction TLB. Our work aims to enhance the effectiveness of this architectural technique on data caches through code transformations and achieve power savings through reduction in the number of page-switches during successive data accesses.

4. Experimental Setup

We explore and develop compiler techniques for the Intel XScale processor [8] on which the *Use-Last* architecture was implemented(Section 3). Intel XScale is an out-of-order, 7-stage superpipelined high-end embedded processor, which runs at up to 1 GHz. The Intel XScale uses TLBs to implement virtual memory support. The Intel XScale is intended to be used in wireless and handheld applications and therefore we execute benchmarks from MiBench [3], MultiMedia [6], DSPstone [4], Spec2000 [5], and the BDTI [7] benchmark suites. The *sim-outorder* cycle-accurate simulator of the SimpleScalar toolset [16] was modified to model the Intel XScale memory configuration and to determine the total number of page switches in the data TLB in a program.

The remainder of the paper is organized as follows: Section 5 describes our instruction scheduling and operand reordering technique. Section 6 describes our array interleaving implementation. Section 7 describes the conditions for our implementation of loop unrolling. Section 8 forms a comprehensive algorithm for TLB page switch reduction.

5. Page Switch-Aware Instruction Scheduling

Instruction scheduling can aggregate instructions that access the same pages consecutively, thereby reducing page switches in the data TLB. In addition, for commutative operations, it is also possible to reorder the operands, and effect the memory access pattern. We develop a combined instruction scheduling and operand reordering technique to reduce TLB page switching.

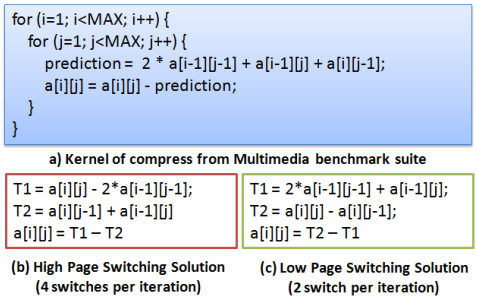

Figure 1. Impact of code generation on TLB page switching

5.1. Motivation

We motivate the applicability and effectiveness of fine-grain instruction and operand reordering on TLB page switches using a kernel from the *compress* benchmark, shown in Figure 1(a). The kernel accesses elements from a two-dimensional array. If the array size is much larger than the page size(which is typically small in embedded systems), elements from the higher dimensions may reside in different pages. In this example, there are high chances that $a[i]$, and $a[j]$ may be in different pages, if $i \neq j$. Assuming this, the two code sequences generated by the compiler, illustrated in Figure 1(b) and (c), may result in the same performance, they may differ significantly in the number of TLB switches they cause. When executed, the code in Figure 1(b) will result in accesses in the sequence: $a[i][j]$, $a[i-1][j-1]$, $a[i][j-1]$, $a[i-1][j]$, and $a[i][j]$, which will result in 4 page switches per iteration, while the code in Figure 1(c) will result in only 1 page switch per iteration. Note that depending on the cache size and page size, the page switches can vary, but if there is no performance impact, it will be better to generate the code as in Figure 1(c). In the rest of this section, we first formulate the problem of minimizing the page switches by instruction scheduling and operand reordering. Finding the problem to be NP-complete, we propose a heuristic for the same.

5.2. Problem Formulation

Input: Data Flow Graph (DFG) is a directed acyclic graph (DAG) $D = (V, E)$ of a code sequence. The nodes $v \in V$ represent instructions $i \in I$. An instruction i is represented by a ordered $(k+2)$-tuple $i =< op, d, s_1, s_2, ...s_k >$, where op is the opcode, d is the destination, and there are k source

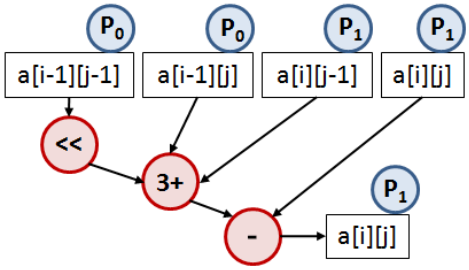

Figure 2. DFG and page mapping of compress kernel

operands, $s_1, ...s_k$, $d, s_1, s_2, ...s_k \in O$, where O is the set of program variables, or operands. There is a directed edge $e = (v_1, v_2) \in E, \ni v_1, v_2 \in V$, from v_1 to v_2 if the destination of the instruction represented by node v_2, is the same as any of the source operands of the instruction represented by node v_1. i.e., $(v_1.i.d = v_2.i.s_1) \vee (v_1.i.d = v_2.i.s_2) \vee ... \vee (v_1.i.d = v_2.i.s_k)$. The data flow graph will also have nodes at the beginning of the graph, representing loading of operands, and nodes at the end of the graph, representing storing of operands, or intermediate values that will be carried over to the next loop. The DFG of the compress kernel is illustrated in Figure 2.

Output: Instruction Sequence represented by the function $Time : I \rightarrow \mathbb{N}$ such that all data dependencies are maintained. i.e., if there is an edge from instruction i_a to i_b, then $Time(i_a) < Time(i_b)$.

Objective: Minimize Page Switches in the instruction sequence. To estimate page switching at the compiler level, we define a function $Page : O \rightarrow P$, which maps operands $o \in O$ to pages $p \in P$, where P is the set of all the pages accessed by the application. A source operand may be a scalar, or an array, and can be defined in a local scope or a global scope. We define $Page(s)$ thus:

- $Page(s) = undefined$ if the operand s is a local scalar variable. This is because most probably all the local scalar variables will be allocated to registers and therefore will not involve in memory access.
- $Page(s) = p_0$ if s is a global scalar variable. We assume that all the global scalars are allocated to a single page.
- For the global or local arrays, we assume that each array, irrespective of it's size is mapped to exactly one unique page.

Page Switch Model In addition, we also need a page switch model, i.e., given a sequence of instructions, how many page switches will occur. We assume that when an instruction i executes, its operands are accessed in the order $\{i.s_1, i.s_2, ..., i.s_k, i.d\}$. Assuming that the page accessed just before the execution of an instruction i is p, then, we define the page switching function, $PS_I(p, i_1, ...i_n)$ to be the number of page switches when a sequence of instructions $i_1, ...i_n$ is executed.

$$\begin{aligned} PS_I(p, i_1, ...i_n) &= PS_O(p, i_1.s_1, i_1.s_2, ..., i_1.s_k, i_1.d, \\ &= i_2.s_1, i_2.s_2, ..., i_2.s_k, i_2.d, \\ &= ..., \\ &= i_n.s_1, i_n.s_2, ..., i_n.s_k, i_n.d) \end{aligned}$$

The total page switch count between operands can be recursively computed,

$$\begin{aligned} PS_O(p, o_1, ..., o_m) &= PS_O(p, o_1) \\ &+ PS_O(LP_O(p, o_1), o_2, ..., o_m) \end{aligned}$$

where $PS_O(p, o) = 1$, when both p and $Page(o)$ are defined, and $p \neq Page(o)$. $LP_O(p, o)$ is the last page accessed when operand o_1 is accessed after accessing page p. The last page function $LP(p, o) = Page(o)$, if $Page(o)$ is defined, otherwise, it is p.

5.3. Solution for Page Switch Minimization

To minimize page switches by instruction scheduling and operand reordering, we define a Page Switching Graph $PSG\_full = (I, S)$, which is a directed graph, whose vertices are instructions $i \in I$, and there is an edge from instruction i to instruction j if instruction j can be scheduled immediately after instruction i. We attach a weight attribute to each edge $w(i, j)$, which is the minimum increase in the page switches when instruction j is scheduled immediately after instruction i. Thus,

$$w(i,j) = \begin{cases} min \begin{cases} PS_O(p, j.s1, j.s2, j.d) \\ PS_O(p, j.s2, j.s1, j.d) \end{cases} & \text{if } j.op \text{ is comm} \\ PS_O(p, j.s1, j.s2, j.d) & \text{otherwise} \end{cases}$$

where p is the last page that has been accessed after instruction i is executed. We add a dummy source node, and a sink node so that there is an edge from the source node to all the instructions that do not have any predecessors in DDG, and there are edges all nodes that do not have successors in DDG to the sink node. Dummy nodes access only $undefined$ pages.

The problem of finding the instruction sequence and operand ordering that minimizes the number of page switches is exactly equal to the problem of finding the shortest hamiltonian path from source node to sink node. This implies that if we can solve the problem of page switch minimization in polynomial time, we can also solve the hamiltonian problem, which is a well known NP-Complete problem in polynomial time. This is quite unlikely, therefore the problem of scheduling for page switch minimization is NP complete. Therefore we focus our efforts on developing scheduling heuristics for page switch minimization.

5.4. Heuristic for Page Switch Minimization

For heuristics, we first construct a Page-Not-Switching Graph $PNSG = (I, D, S)$, where the nodes (I) are instructions, and there are two kinds of edges, first is the set of data dependence edges D, and the second S is the set of inter-instruction page not-switching edges. Thus there is a an edge $s = (i, j) \in S$ between two instructions: $i, j \in I$, if there is NO inter-instruction page switch when instruction j is scheduled immediately after instruction i. In other words, $(i, j) \in S, \forall i, j \in I$, iff $Q_{ps} \geq 1$, where

$$Q_{ps} = \begin{cases} min \begin{cases} PS_O(p, undefined, i.d, j.s1) \\ PS_O(p, undefined, i.d, j.s2) \end{cases} & \text{,if } j.op \text{ is comm} \\ PS_O(undefined, i.d, j.s1) & \text{otherwise} \end{cases}$$

An example of a $PNSG$ is shown in Figure 3. The nodes 1 through 7 are instructions, and the solid edges represent data dependencies. The dashed edges represent the inter-instruction page not-switching edges. We now perform our scheduling on this graph representation. We first developed a greedy algo-

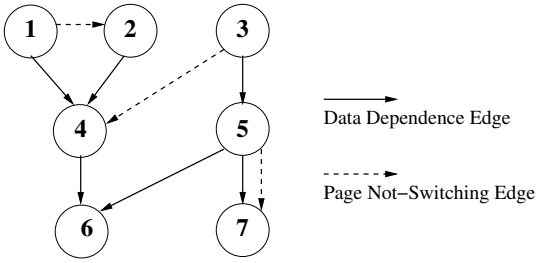

Figure 3. Problem in greedy solution

rithm. In the greedy algorithm, in every iteration, the *last scheduled instruction*, l is maintained, and list of instructions that are now ready to be scheduled, R is created. If there is a page-not-switching edge between l and any instruction $r \in R$, then r gets priority, as it minimizes the page switches. Thus suppose instructions $1, 2$ and 3 are scheduled, with $l = 3$. Then R can be computed as $R = \{4, 5\}$. Out of these, the greedy heuristic will pick up instruction 4.

Figure 3 illustrates one problem with this simple approach. In the first iteration, the greedy solution can pick up either instruction 1, or instruction 3. Picking up instruction 3 is a bad choice, because it is not possible to schedule instruction 4 as the second instruction. Instruction 3 should only be scheduled only if instruction 4 can be scheduled next. We fix this problem by adding that - when picking an instruction which is the source of a page-not-switching edge, we pick up a pair of instructions to schedule; plus, we give priority to pick up instructions that are not connected through page-not-switching edges. This gives us more opportunities to pick up instruction pairs with page-not-switching edges.

5.5. Experiments

We have implemented this page-aware instruction rescheduling algorithm as a compiler post-pass [1]. We compile our benchmarks with GCC -O3 optimization, to ensure that the benchmarks are compiled and scheduled for the maximum performance. We disassemble the generated object file, discover the basic blocks, and re-create the control flow graph (CFG), and the data flow graph. We perform this modified list scheduling heuristic on basic blocks. This fine grain instruction scheduling approach is applicable to any program. The effectiveness of this approach could be increased by performing our scheduling on hyperblocks, and/or superblocks. We observed that our scheduling gains from performing local reordering of load instructions. There is not much increased opportunity to move load instructions across basic blocks, because of tight data dependencies.

We modified the sim-outorder [16] simulator to count the page switches for an application execution. Figure 4 plots the page switch count, after implementing our page-aware instruction scheduling and operand re-ordering transformations normalized to the baseline page switch count. On an average, our technique achieves 23% reduction in the page switch count as indicated by the right-most bar in Figure 4. As a matter of fact, we observed an average performance improvement of 4%. This reduction in page switches directly translate into 23% power savings in the Use-Last TLB. Note that this is over and above what Use-Last TLB architecture achieves on its own.

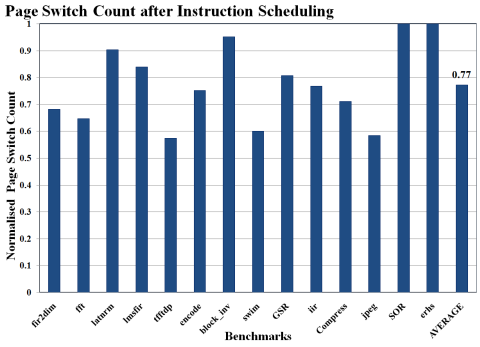

Figure 4. Impact of Instruction Scheduling on Page Switch Count

6. Page-Switch Aware Array Interleaving

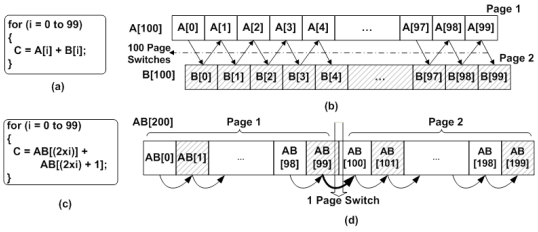

Figure 5. Array Interleaving through example: (a)Example loop (b)Array allocation and access pattern (c)Loop block with interleaved arrays (d)Array allocation and access pattern of interleaved array

6.1. Motivation

Figure 5 shows how array interleaving can reduce the TLB page switching over data accesses in the program. The code in Figure 5(a) shows a loop which accesses elements from two different arrays A and B, which are mapped to different pages. Figure 5 (b), shows that when this loop executes, there is a page switch between consecutive memory accesses in the program. Figure 5(c) shows the transformed code after interleaving. Array interleaving places the elements of the two arrays as alternate elements of the array AB. Figure 5(d) shows that there is no page-switching between consecutive access to AB.

6.2. Which Arrays to Interleave

The problem of reducing TLB page switching is localized to consecutive memory accesses, therefore interleaving of arrays

need only be directed to decrease the page switching in the innermost loop. Consider a nested loop of 3 levels, whose iterators are i, j, and k, in which are there are references to arrays A and B. Suppose in the innermost loop, the reference functions are affine functions of the iterators, i.e., the access function can be represented as a linear combination of the iterators, $f_A = a_0 + a_1 i + a_2 j + a_3 k$, and similarly $f_B = b_0 + b_1 i + b_2 j + b_3 k$.

We consider two arrays A and B as interleaving candidates only if i)the access functions of the arrays are the same. Thus, $a_0 = b_0, a_1 = b_1, a_2 = b_2, a_3 = b_3$ ensuring minimized page switches after interleaving. ii)the arrays of the same size. For example, we will interleave an array of integers with another same size array of integers. It is important to note that while it is possible to interleave arrays with slightly different access patterns also, it results in an overhead in terms of extra addressing instructions. However, the innermost loop may contain several references to the same array. Two arrays will be interleaving candidates if the conditions are satisfied for any pair of references to the arrays. We perform this analysis on all the important loops of the application, and find pair of arrays, which are interleaving candidates, we take the union of interleaving candidates. Thus if arrays A and B are found to be interleaving candidates from one loop, while B and C are interleaving candidates from some other, then all the three arrays will be interleaved.

6.3. Interleaving

The process of interleaving r arrays of the same data type $A_1, A_2, ...A_r$ is a three step transformation. The first is to replace the individual array declarations with a single array A of r times the size of each array, and second is to fix the access functions of all the array references. The access function $f_m = A_m[a_m i + b_m j + c_m k + d_m]$ of the m^{th} array is replaced by $f_m = A[r \times (a_m i + b_m j + c_m k + d_m) + (m-1)]$ in three-level nested loop. At the end of the day, it is important to schedule the instructions that access the interleaved array in the same pattern consecutively. This is done by moving the result of the first instruction in a new temporary variable, and replacing all its uses by the temporary variable. Interleaving of r arrays of different data types is done by declaring a new structure, say s, which contains an element from each of the arrays. We then declare an array A of the same size as all the previous arrays consisting of elements of data type s. Then we replace the access function of the m^{th} array $f_m = A_m[a_m i + b_m j + c_m k + d_m]$ by $f_m = A[a_m i + b_m j + c_m k + d_m].m$.

6.4. Experiments

We translate the source code into the FORAY format [2], which essentially consists of just the loop structure and the array access functions as affine functions of the loop iterators. We analyze the code in this format, and, perform our page-aware array interleaving transformations in this format, and then convert it back to the source code. The application is compiled again, and our instruction scheduling for page switch minimization is applied to enhance the impact of array interleaving. Figure 6 plots the page switch count after performing array interleaving and instruction scheduling on all the benchmarks. The plot thus shows that our page-aware array interleaving is a very effective transformation, and reduces the data-TLB page-swith count by an

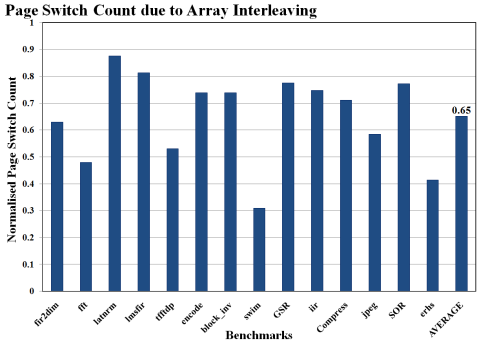

Figure 6. Impact of Array Interleaving and Instruction Scheduling on Page Switch Count

average of 35% (indicated by the right-most bar) with an overall average of 11% increase in performance. This performance improvement is inherent to array interleaving, as it inherently increases the spatial locality of data, leading to improved cache behavior. In *swim*, two global arrays and 5 local arrays were accessed together in the loop bodies. Interleaving was possible on all the arrays, thereby forming two interleaved arrays (one global, and other local). This transformation enhanced the opportunities for instruction scheduling and therefore 70% page switch reduction was observed. Since the TLB power is directly proportional to the number of accesses, we can expect a concomitant 35% reduction in TLB power due to the combined impact of array interleaving and page switch-aware scheduling.

7. Impact of Loop Unrolling

Loop unrolling is a loop transformation in which the loop body is replicated a finite number of times, thereby reducing the loop overhead instructions. It is important to observe that loop unrolling by itself does not reduce TLB page switching, but, it may increase the effectiveness of instruction scheduling, by providing more opportunities to schedule instructions and thereby reduce inter-instruction page switching.

Unrolling a loop may reduce page switches if there is atleast one instruction, such that if we schedule two copies of the instruction belonging to different iterations when scheduled consecutively, will not result in inter-instruction page switching. In other words, loop unrolling can be performed if $\exists i \in I$ such that,

$$\begin{cases} min \begin{cases} PS_O(undefined, i.d, i.s1) \\ PS_O(undefined, i.d, i.s2) \end{cases} & \text{if } i.op \text{ is comm} \\ PS_O(undefined, i.d, i.s1) & \text{otherwise} \end{cases} = 0$$

We have implemented our page-switch aware loop unrolling transformation also as source code transformation. Figure 7 plots the effect of loop unrolling on the page switch count of various benchmark applications. The normalized page-switch count for the case when page-switch aware instruction scheduling and array interleaving are performed is plotted as the dark bar (to the left for each benchmark), and the lighter graphs indicate the page-switch count for unrolling factors of $2, 4$ and 8 times respectively. The right-most set of bars in Figure 7 indicate the average values for the cases plotted. On an average,

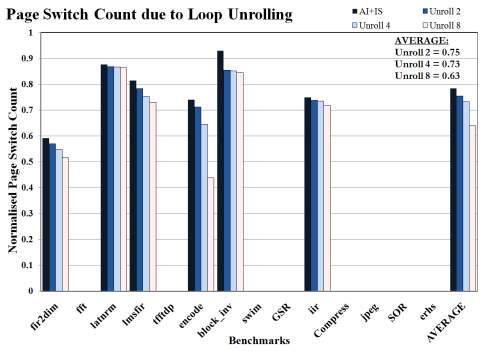

Figure 7. Impact of Loop Unrolling on Page Switch Count

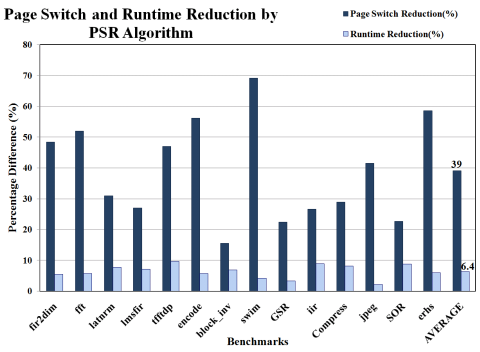

Figure 8. Page Switch Count and Runtime reduction by our Page-Switch Reduction Algorithm

for an unrolling factor of 8, we obtain a reduction of 37% in the page switch count for the applications on which page-aware loop unrolling was possible with 9% performance improvement.

8. Comprehensive Page Switch Reduction

Finally we study the impact of all the three transformations together. The ordering of the transformations is an interesting issue. Instruction scheduling and array interleaving are the fundamental transformations that reduce data TLB page switches. Loop unrolling will be most effective when all the opportunities for page switch reduction achievable after re-scheduling, are exploited. Our page switch-aware instruction scheduling is done at a more fine-grained level, and therefore has to be performed only after array interleaving and unrolling to maximize the effect. We first perform *Page-Switch Aware Array Interleaving* to group the memory allocation of varied arrays together into one overlapped page, and then *Loop Unrolling* on the instructions such that all the instructions capable of being implemented without page-switch are executed together. Our fine-grain instruction scheduling is then performed as a post-pass.

The dark bars on the left in Figure 8 plot the percentage reduction in the data TLB page switch count for each application. The reduction is calculated as compared to the data TLB page switch count when the application is compiled using $GCC - O3$ alone. The rightmost dark bar shows that there is an average 39% data TLB page switch count reduction over all the benchmarks. The light bars on the right in Figure 8 plot the reduction in runtime for all the applications. The rightmost light bar shows that there is an average 6.4% reduction in runtime. In conclusion, the effect of page switch reduction techniques is additive, and the effect is realized after each step of the *Page Switch Reduction* algorithm.

9 Summary

The *Use-Last* TLB architecture proposed in [9] reduces the TLB power consumption, if the same page is accessed successively. This approach was ineffective for data TLB, because data accesses do not exhibit high locality as compared to instructions. In this paper, we have introduced a novel, *page-aware instruction scheduling* algorithm, and proposed heuristics to decide when to perform array interleaving, and loop unrolling to reduce the TLB page switching. Our experiments on benchmarks from MiBench, Multimedia, DSPStone and BDTI suites show a 39% reduction in the TLB page switches with a negligible increase in performance, over what is possible by the GCC compiler. Since the dynamic power of the TLB is directly proportional to the number of page switches in the *Use-Last* TLB architecture, we can expect a concomitant 39% reduction in the TLB power. Our future work is to investigate the impact of other transformations, e.g., instruction selection on TLB power reduction.

References

[1] A. Shrivastava et al., "Operation tables for scheduling in the presence of incomplete bypassing," *In CODES+ISSS*, pages 194199, 2004.

[2] I. Issenin et al., "FORAY-GEN: Automatic Generation of Affine Functions for Memory Optimizations," *In DATE 05*, pages 808813, 2005.

[3] M. R. Guthaus et al., "MiBench: A free, commercially representative embedded benchmark suite," *In WWC 01*, pages 314, 2001.

[4] V. Zivojnovic et al., "DSPstone: A DSP-oriented benchmarking methodology," *In Proceedings of Signal Processing Applications and Technology*, Dallas, 1994.

[5] J. L. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium," *Computer*, 33(7):2835, 2000.

[6] H. Balakrishnan et al., " Multimedia Benchmarks: A Performance Comparison of Multimedia Programs on Different Architectures" *citeseer.ist.psu.edu/233784.html*

[7] BDTI Suite: Berkeley Design Technology Inc. "The BDTI Benchmark suites", *bdti.com/products/benchmark_overview.html*

[8] Intel Corporation, "Intel XScale®Technology Overview", *intel.com/design/intelxscale*

[9] J.R.Haigh et al., "A Low-Power 2.5 GHz 90 nm Level 1 Cache and Memory Management Unit," *In IEEE Journal of Solid State Circuits*, pages 11901199. IEEE Press, 2004.

[10] L. T. Clark et al., "Reducing translation lookaside buffer active power," *In ISLPED 03*, pages 1013, 2003.

[11] M. Ekman et al., "TLB and snoop energy reduction using virtual caches in low-power chip-multiprocessors," *In ISLPED 02*, pages 243246,2002.

[12] X. Zhou et al., "Low-power cache organization through selective tag translation for embedded processors with virtual memory support," *In GLSVLSI 06*, pages 398403, 2004.

[13] M. Kandemir et al., "Compiler-Directed Code Restructuring for Reducing Data TLB Energy," *In CODES+ISSS 04*, pages 98103, 2004.

[14] I. Kadayif et al., "Optimizing instruction TLB energy using software and hardware techniques," *ACM Trans. Des. Autom. Electron. Syst.* 10(2):229257, 2005.

[15] P. Petrov et al., "Energy-effcient physically tagged caches for embedded processors with virtual memory," *In DAC 05*, pages 1722, 2005.

[16] T. Austin. "SimpleScalar LLC". *simplescalar.com*

[17] V. Delaluz et al., "Compiler-directed array interleaving for reducing energy in multi-bank memories," *In ASP-DAC 02*, page 288, 2002.

[18] A. Parikh et al, "Instruction scheduling for low power," *The Journal of VLSI Signal Processing*, 37(1):129149, 2004.

[19] A. Chiyonobu et al., "Energy-efficient instruction scheduling utilizing cache miss information," *SIGARCH Comput. Archit. News*, 34(1):6570, 2006.