Code Transformations for TLB Power Reduction

Reiley Jeyapaul and Aviral Shrivastava

Compiler and Microarchitecture Laboratory,
Arizona State University, Tempe, AZ 85281 USA
Email : {reiley.jeyapaul, 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 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 [1]. The *Use-Last* TLB architecture reduces the power consumption when the last page is accessed again. In this work, we develop code transformation techniques to reduce the page switchings in data cache accesses and propose an efficient page-aware code placement technique to enhance the energy reduction capabilities achieved by the *Use-Last* TLB architecture for instruction cache accesses. Our comprehensive page switch reduction algorithm results in an average of 39% reduction in the data-TLB page switching, and our code placement heuristic results in an average of 76% reduction in the instrucion-TLB page switchings with negligible impact on the performance. The reduced page switch count in the cache accesses of an application achieves an equivalent power savings over and above the reduction achieved by the *Use-Last* TLB architecture. We have demostrated this energy reduction through experiments on benchmarks from MiBench, Multimedia, DSPStone and BDTI suites.**

I. 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. [2] note that the TLBs can consume $20-25\%$ of the total $L1$ cache energy. Kadayif et al. [3] 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. In [3], researchers show that the iTLB architecture has a power density of $7.820\ nW/mm^2$ compared to 0.975 and $0.670\ nW/mm^2$ for $iL1$ and $dL1$, respectively.

Several TLB designs have been proposed to trade-off the TLB lookup delay, area and power consumption [4], [5]. One simple, yet effective technique for TLB power reduction proposed in [1], [6], 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.

For a modified processor with the inclusion of the *Use-Last* TLB architecture for both the instruction and data TLB structures, we present here, compiler directed code transforamtion techniques to reduce the processor power consumption, by improving the page locality of data and instruction cache accesses. We first 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 data-TLB accesses, while minimizing performance loss. Our comprehensive algorithm can reduce the data-TLB page switches by 39%, with minimal performance impact experimented over benchmarks from MiBench, Multimedia, DSPStone and BDTI suites. We then propose a novel page-aware code placement heuristic to enhance the page locality of instruction cache accesses and thereby reduce the power consumption of the instruction-TLB by an average of 76% with less than 1% variation in perforamnce over benchmark applications from the MiBench suite. It should be noted here that this power reduction obtained through the code transformations is above and beyond what the *Use-Last* hardware technique alone could achieve.

II. Related Work

TLB power reduction is important not only to reduce the total energy consumed by the processor, but also to alleviate the high power density (hotspot) of TLB in the processor. Several researchers have proposed efficient circuit-level, microarchitectural and software techniques to reduce the power consumption of the TLB and the Memory Management Unit.

A. Hardware Approaches

Several researchers have proposed efficient circuit and microarchitectural techniques to reduce the power consumption of the TLB and the Memory Management Unit. A fully associative TLB architecture with (Content Addressable Memory) CAM implementation has been proved to be efficient in terms of performance and power consumption. [7] proposes a banked associative design for TLBs (BA-TLB) which consumes less power than a fully associative TLB through the use of a banked design such that only half the CAM entries are looked up during each access to the TLB. In [8] the TLB is constructed as multiple banks with a small filter-bank buffer located above its associated bank. Through the use of selective filtering and banking mechanism, the number of entries accessed is reduced and it therefore proves to be highly efficient in embedded processors.

Hyuck,.et.al. [9] in their work propose a two-level TLB architecture that integrates a 2-way banked filter TLB with a 2-way banked main TLB design. This architecture aims at reducing the power consumption of the TLB in embedded processors by distributing the accesses to TLB entries across the banks in a balanced manner. Chang [10] in his paper, presents a real-time filter scheme to remove redundant TLB accesses by distinguishing them as soon as the virtual address is generated. This in combination with two adaptive banked TLB designs is proved to effectively improve the energy delay product of data TLBs. [11] introduces translation registers (TR) to store the most frequently used TLB translations. During subsequent virtual address entries, these TRs are looked up and if present the information stored is used. This saves the switching activity of the register files in mapping the virtual address to the physical address. It should also be noted that the granularity at which these hardware architectures alone achieve power reduction is limited by the number of registers or only succesive access. The power savings achieved by such hardware techniques are also limited by the area and power overheads involved in the implementation.

B. Software Approaches

A compiler-directed array interleaving technique [12] 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 [13] schedule instructions within a block based on the minimum obtainable value for a weighted cost function:*circuit-state cost*.One recent work is [14], where energy reduction is achieved through effective utilization of resources by switching between two processor modes based on the cache misses. These software approaches though achieve power reduction in the TLB are limited by the applicability to broad spectrum of applications and also by the compatibility to underlying architecture.

C. Hybrid Approaches

Kayadif,.et.al. in [3] present a set of software only, hardware only and integrated hardware-software techniques to achieve reduced instruction TLB energy consumption. In this journal, the authors demonstrate through analysis and experiments the efficiency and advantages of a hybrid hardware-software technique to reduce TLB power. A hybrid approach has the critical advantage of modifying the software in such a way that the architectural modification is efficiently used through architecture aware software optimizations. In this work, we propose such a hadware-software hybrid apporach for TLB power reduction using the *Use-Last* TLB architecture in the caches.

One hybrid approach closest in semblance to ours, is by Kandemir et al. [15]. 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, 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.

III. Architecture Description

A. *Energy consumption in the conventional TLB*

In a VI-VT (Virtually Indexed Virtually Tagged) or VI-PT (Virtually Indexed Physically Tagged) cache architecture, all data/instruction accesses by the processor are to virtual addresses. On a cache access, the virtual address is compared with any existing tag entries in the CAM based table of the TLB. If the entry exists, it indicates a cache-hit and therefore, the mapped physical address (of the data entry) and the corresponding access permissions are transferred to the output of the TLB, which inturn is used to address the specific location in the cache. If the data is not present in the cache, the tag comparison at the TLB returns a cache-miss and therefore the data is retrieved from the memory, and the corresponding physical address and the processor accessing the data is updated as an entry in the TLB table. It should be noted here that for every data/instruction access the TLB lookup is activated and the CAM structure is switched to access the table entries on every access. This switching energy consumed on every TLB access is directly proportional to the size of the TLB.

B. *The* Use-Last *TLB Architecture*

Proposed in [1], the Use-Last TLB architecture Fig. 1 utilizes a modified TLB-CAM structure. The virtual address input is matched with the TLB tag through the CAM cells (which is optimized for power consumption). The TLB tag is then used to retrieve the mapped physical address from the lookup table (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. This newly introduced latch clearly differentiates the two stages of the TLB operation on every data access. In the first stage, the CAM cells are activated on every access to check for the presence of an existing TLB entry. These CAM cells have been optimized for reduced power consumption [1] because of the requirement that the tag comparison has to be performed on every access. At the second stage of the TLB operation, the compared tag for a cache-hit access, is used to lookup for the corresponding physical cache address and access permissions from the table (a register file array). On a data access, the output of the latch(previously accessed tag) is compared with the current tag entry and if found to be equal, the register file array is not looked up, as the physical address and access permissions from the previous lookup remain unaltered at the output of the TLB structure. If the tag entries are not equal, the latch stores this tag entry and the register file array is switched to lookup for the corresponding physical address and access permissions.

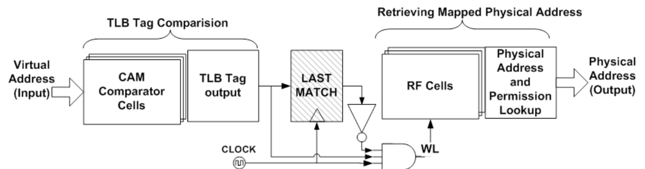

Fig. 1. Representative block diagram of the *Use-Last* TLB Architecture [1]

Since, for every successive data accesses to the same page, where the tag entries are the same, the register files (word and bit lines) are not activated, the switching energy of the RF cells and associated circuitry is eliminated. The effectiveness of this technique was demonstrated on a 90-nm virtually addressed microprocessor cache memory subsystem functioning at $2.5\ GHz$ with $32KB$ of instruciton and data cache structures. The instruction TLB demostrated 75% power savings while the data TLB showed 42%. As can be observed, this technique was inefficient for data caches, as data accesses in general do not exhibit high data locality as compared to instruction TLB. Our primary 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. A similar compiler-directed approach is used to reduce the page-switches during instruciton accesses in the instruction-TLB and significant energy reduction is demonstrated in this journal.

IV. Experimental Setup

We explore and develop compiler techniques for the Intel XScale processor [16] on which the *Use-Last* architecture was implemented(Section III). 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 [17], MultiMedia [18], DSPstone [19], Spec2000 [20], and the BDTI [21] benchmark suites. The *sim-outorder* cycle-accurate simulator of the SimpleScalar toolset [22] was modified to model the Intel XScale memory configuration and to determine the total number of page switches in the data and instruciton TLB of a program.

V. Organization of Contents

The remainder of this paper is organized into two broad parts. In the first part, we develop and demonstrate our page aware code transforamtion techniques for data-TLB page switch redcution. Section VI-A describes our instruction scheduling and operand reordering technique.

Section VI-B describes our array interleaving implementation. Section VI-C describes the conditions for our implementation of loop unrolling. Section VI-D then describes our comprehensive algorithm for data-TLB page switch reduction. In the second part, we describe in detail our page aware code placement heuristic in Section VII and demonstrate its efficiency in page switch reduction for the instruction cache TLB through experiments. We then conclude and summarise our work in Section VIII.

VI. Part I: Data-TLB Power Reduction

A. 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.

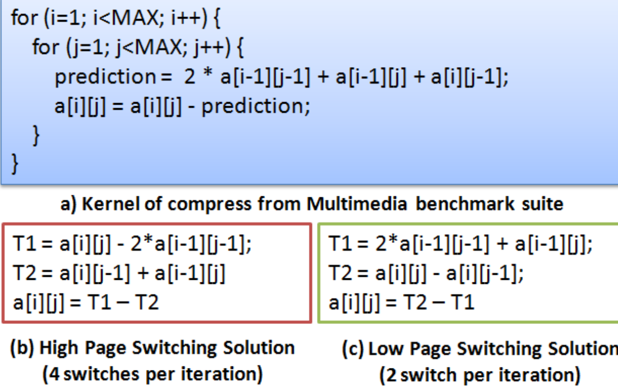

Fig. 2. Impact of code generation on TLB page switching

1) Technique Overview: 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 Fig. 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 Fig. 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 Fig. 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 Fig. 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 Fig. 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.

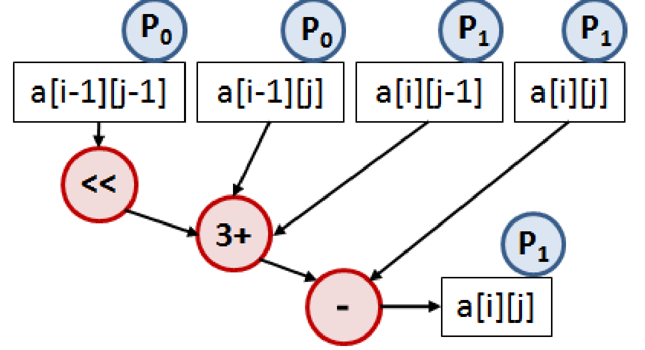

Fig. 3. DFG and page mapping of compress kernel

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 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 Fig. 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.

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,

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

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.

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

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

An example of a $PNSG$ is shown in Fig. 4. 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.

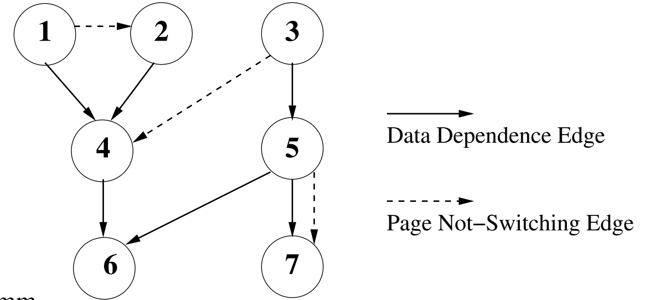

Fig. 4. Problem in greedy solution

We first developed a greedy algorithm. 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.

Fig. 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) Experiments and Results: We have implemented this page-aware instruction rescheduling algorithm as a compiler post-pass [23]. 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 [22] simulator to count the page switches for an application execution. Fig. 5 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 Fig. 5. 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.

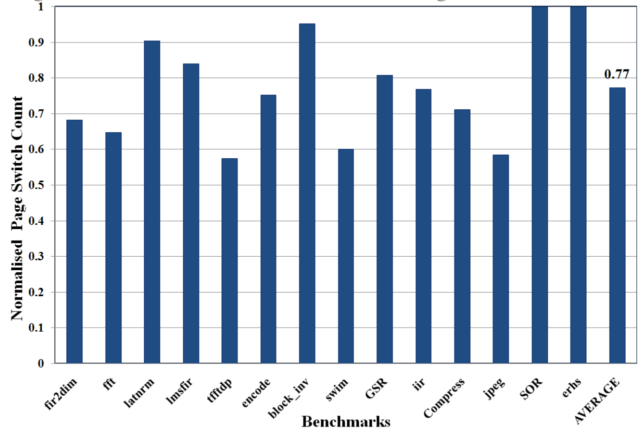

Fig. 5. Impact of Instruction Scheduling on Page Switch Count

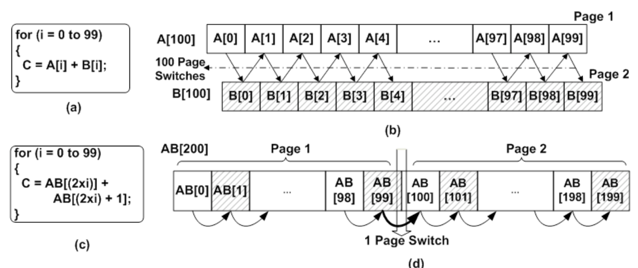

Fig. 6. Array Interleaving through example: (a)Example loop (b)Array allocation and access pattern (c)Loop block with interleaved arrays (d)Array allocation and access pattern of interleaved array

B. Page-Switch Aware Array Interleaving

1) Technique Overview: Fig. 6 shows how array interleaving can reduce the TLB page switching over data accesses in the program. The code in Fig. $6(a)$ shows a loop which accesses elements from two different arrays A and B, which are mapped to different pages. Fig. $6(b)$, shows that when this loop executes, there is a page switch between consecutive memory accesses in the program. Fig. $6(c)$ shows the transformed code after interleaving. Array interleaving places the elements of the two arrays as alternate elements of the array AB. Fig. $6(d)$ shows that there is no page-switching between consecutive access to AB.

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.

3) Array 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 mth 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 mth array $f_m = A_m[a_m i + b_m j + c_m k + d_m]$ by $f_m = A[a_m i + b_m j + c_m k + d_m].m$.

4) Experiments and Results: We translate the source code into the FORAY format [24], 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.

Fig. 7 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 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

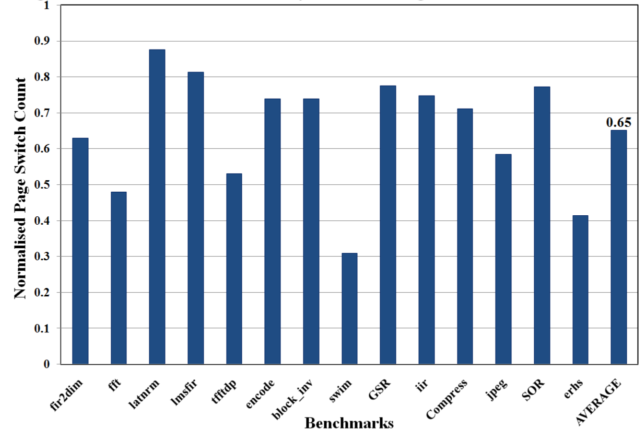

Fig. 7. Impact of Array Interleaving and Instruction Scheduling on Page Switch Count

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.

C. 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,

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

1) Experiments and Results: We have implemented our page-switch aware loop unrolling transformation also as source code transformation. Fig. 8 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

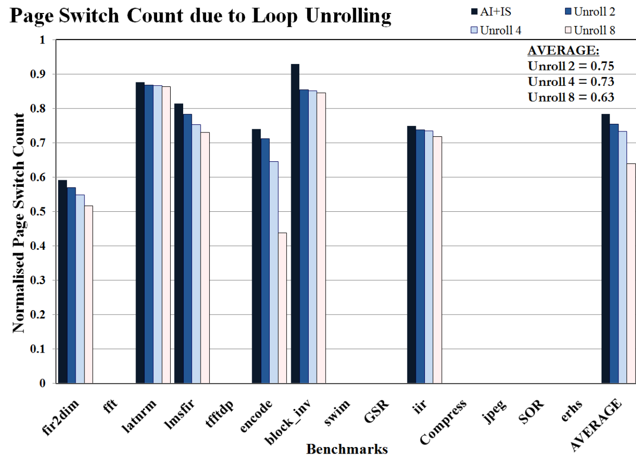

Fig. 8. Impact of Loop Unrolling on Page Switch Count

(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 Fig. 8 indicate the average values for the cases plotted. On an average, 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.

D. 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.

1) Experiments and Results: The dark bars on the left in Fig. 9 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 Fig. VI-D1 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.

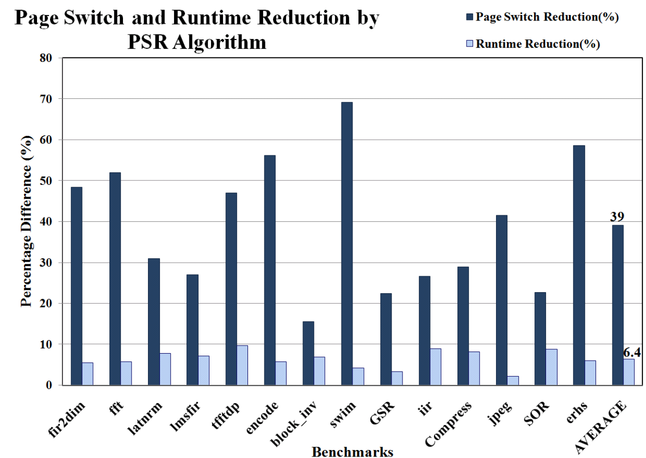

Fig. 9. Page Switch Count and Runtime reduction by our Page-Switch Reduction Algorithm

VII. Part II:Instruction-TLB Power Reduction

A. Instruction TLB Power

The instructions of a program are predominantly a sequence of words placed consecyutively in the instruction memory. This inherently means that majority of the instruction cache accesses are to accesses within the same page when the next subsequent instruction accessed at all times. However, $(i)$when the loop block of a program extends across a page boundary or $(ii)$when the called function and the call-site, are each in different pages of the instruction memory, the instruction control will cross page boundaries and thus cause page switches. When successive accesses to the instruction cache are to two different pages, the last stored permission and tag data stored by the *Use-Last* [1] latch cannot be used and thus cause switching of the TLB register file structures and therefore leading to power consumption. In this work, we develop a code placement heuristic that places the various function blocks of the program such that the above conditions which case page switches are avoided. This *page-aware code placement heuristic* proposed aims to achieve maximum reduction in the number of instruciton TLB page switches and thereby reduced power consumption by the i-TLB.

B. Mechanics of Code Placement by the GCC Compiler

When an application is compiled by the GCC[1] compiler, the placement of the various functions accessed in the program is in the order of occurrence of the function in the code. When multiple files are involved in one application, the order of the files added in the compilation list for the program is the order in which the functions in that file are added into the binary for that application. It can be noted here that the code placement is not optimized for any requirement (in particular performance) and therefore motivated us to analyze the impact of code placement on the page locality of the instruction memory. This also demonstrates that any change to the placement of the code in an application does not directly affect the performance of the compiler optimized application. *The Code Placement Problem* is defined as the reallocation of the different function blocks in a program such that the conditions for page-switches (as described above), in the instruction memory are avoided. It can be noted here that any code placement optimization intended, is limited to the granularity of the function block which in-turn helps in achieving an efficient greedy heuristic to solve the problem.

C. Problem Formulation

A given program can be represnted in the form of a DCFG as described in Fig. 11 where each function of a program is defined by a 5-tuple of the form $FP =< Id, Pos, Size, Calls, LP[] >$. Here, $LP[]$ is the list of loops within the function and represented by the 5-tuple loop tuple of the form $LP =< Id, Pos, Size, Cnt, FnCall[] >$, where $FnCall[]$ lists the call-sites for each of the function calls from this function and defined by the tuple $FnCall =< FP.Id, Pos >$. This hierarchial representation of a program defines the various components involved in the formation of the problem and also facilitate in deriving a heuristic solution. At the top-level, the program is defined by a list ($FP[]$) of tuples describing the list of functions in the program. The optimized reallocation solution to the problem is given by the page-offset values($FP.Pos$) defined in the form of affine constraints and integer tuple relations. The smallest offset values for each function that satisfy these relations are taken as the solutions to the optimal code placement problem.

D. Page-Aware Code Placement Heuristic

The overall functioning of the heuristic for page-aware code placement, is described in the form of a flowchart in Fig. 10. The program to be analyzed and optimized is first profiled by executing the application and deriving the data required. The data gathered is then processed and stored in the form of tuples datastructure(FP, LP and $FnCall$ arrays) as described above. The values for loop iteration count ($LP.Cnt$), and inter function calls ($FP.Calls$) are sorted and listed in the of decreasing magnitude. At each time, the top of this sorted list is taken and the greedy heuristic is processed. Based on whether the maximum value item is a function pointer (FP) or a loop pointer (LP), the corresponding decisions are made and the overloaded function $Fn_Boundary()$ is called with the respective function parameters accordingly. The idea behind sorting both the loop count values and the function call values as a single list and taking the top-most value, is to ensure that the component(loop or function-call) that dominates the page-switches of the program is given due prominence in the optimization heuristic.

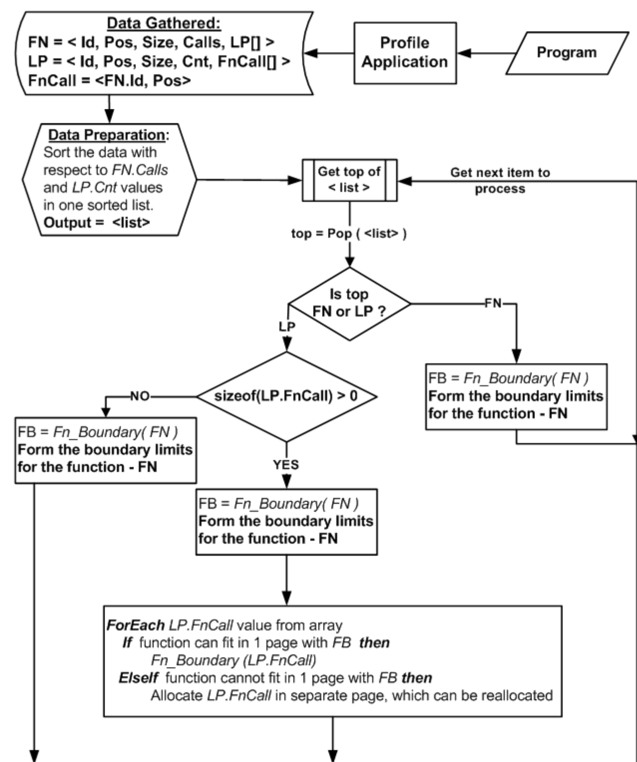

Fig. 10. Flowchart describing the *Greedy Heuristic* for page-aware code placement.

Two basic distinctions are made with respect to the component identified(loop pointer or function pointer). Let us first consider the top-most component is a loop pointer therefore indicating that the loop count is the highest among all values sorted. In this case, the LP tuple values are used to analyze the possible position of the loop within a page and when found to fit within a page, the loop positions are set in the form of affine relations with respect to the page boundaries. These limits on the loop

[1]Analysis in this work was performed using the GCC versions 2.7.2.3 and 3.4.6 only. Unless a release of the GCC compiler alters this functionality, we assume this mechanism to be applicable to all previous and current GCC compiler versions.

position is then translated into equivalent limits on the loop's parent function considering the size of the function. The boundaries derived for the position of the function, within a page, thus gaurantees that the loop with the highest loop count is always placed within a single page and therefore gaurantee that $LP.Cnt$ number of page switches are removed. For the case when the loop pointer identified contains a non-zero number of function calls within its loop block, identified by the $LP.FnCall$ values, another step is performed in addition to the above process of deriving limits. The list of functions called from call-sites within the loop block are noted and the size of each of the function is compared against the limits on the function position (derived earlier). The largest function which satisfies this relationship is updated with its corresponding tuple values with the limits on its position in the page. This additional step for function calls ensures that, any page-switches that may arrise due to control switching within the loop block is avoided. Since this loop block has been identified as that of maximum value, we can assume here that a maximum of $LP.Cnt$ number of page switches are thwarted.

Let us now consider the top-most component is a function pointer indicating that a particular function is called maximum number of times. In such a scenario, we can assume that for every call to the function, notwithstanding the components within the function, its sequential set of instructions are executed for sure. If these set of sequential instructions were such that they crossed a page boundary, we would experience a page switch for every call to this function. In order to avoid this, considering the size of the function, the function can be gauranteed to be placed within the same page through the use of function position limits over a single page. This process this ensures the removal of $FP.Calls$ possible page switches in the program. If for any top-most element of the list, none of the above conditions, for possible optimizations, satisfy, the element is deemed *reallocatable* and the next element is chosen. Since at all times, we only take the top-most element from the sorted list, and deduce a position limit for the functions based on that value we call this a greedy heuristic. It should also be noted here that none of the already defined limits are redefined during the process and therefore optimal page-switch reduction is achieved for the instruction-TLB accesses.

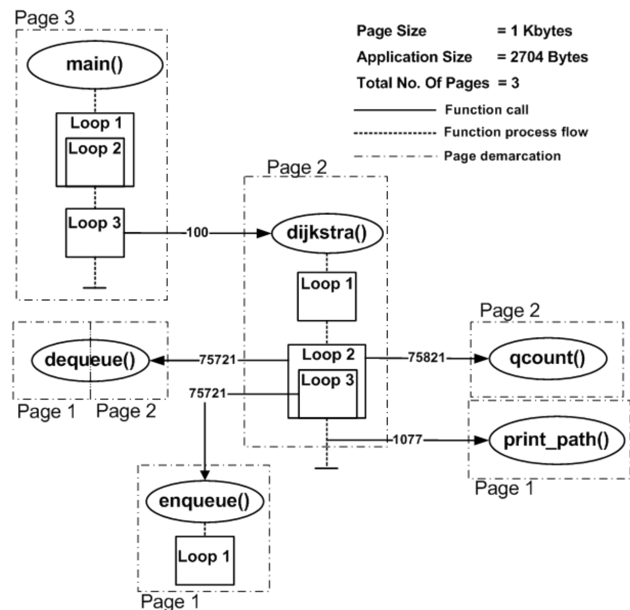

Fig. 11. Original DCFG of *dijkstra* program with page demarcations.

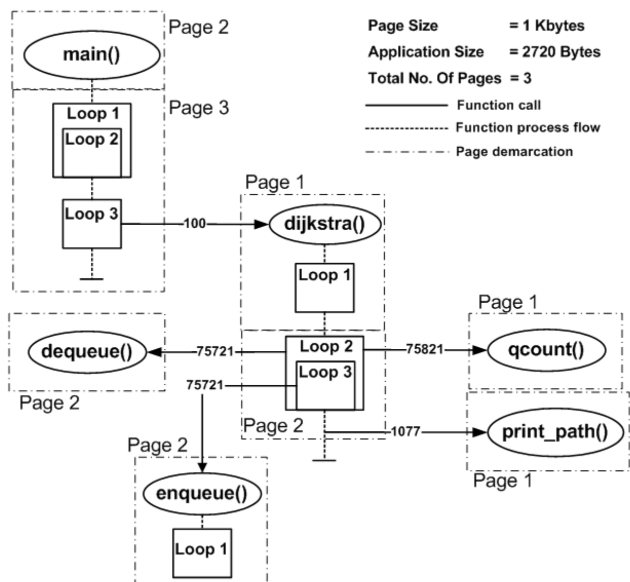

Fig. 12. Optimized DCFG of *dijkstra* program with page demarcations.

E. Heuristic Demonstration Through Example

We use the *dijkstra* application (from the MiBench [17] benchmark suite) as an example to demostrate the application of our *page-aware code placement heuristic* described above. Fig. 11 describes the data control flow graph (DCFG) of the original *dijkstra* benchmark program. Each oval in the figure represents a function and the dotted line marks the process-line for that function. The loops in each of the functions are labelled by rectangular boxes and the nested loops are marked by overlapping rectangular boxes, denoting the level of each loop. A function call is indicated by the use of a solid arrow to another function, and the weight on that arrow indicates the number of function calls executed, throughout the operation of the program. The page boundaries or the pages to which each function (or a part of the function) resided in the memory is indicated with the help of corrugated lines and labelled with the page number.

From the DCFG in Fig. 11, it is evident that function

calls to *dequeue* and *enqueue* are the major contributors to the page-switch count of the application. In addition, it can be noted that the function *dequeue* is called from outer loop *Loop* 2 of the *dijkstra*, and the *enqueue* function is called from the inner loop *Loop* 3 of the *dijkstra*. Therefore, in order to optimize the *dijkstra* application for reduced page switches, the function were rearranged as in Fig. 12. Here, we observe that the code size of the application has increased by 16 Bytes which was used as padding to position the function calls and the loops optimally. The functions were rearranged in the instruction memory such that the basic block within loops *Loop* 2 and *Loop* 3 of *dijkstra* were in one page. In addition, since each of these loops involved function calls to *dequeue* and *enqueue*, these functions were placed immediately below the *dijkstra* function so that they exist in the same function. Since the collective size of *dequeue*(184 Bytes), *enqueue*(312 Bytes) and *dijkstra*(1408 Bytes) far exceeded the size of one page (1024 Bytes), the *dijkstra* function was placed such that the loops *Loop* 2 and *Loop* 3, the functions *dequeue* and *enqueue* all are placed in the same page. Owing to this code placement order, it can be observed in Fig. 12 that the functions *qcount* and *print_path* originally in the same page as that of *dijkstra* are now located in different pages. The justification for this tradeoff is the fact that the total number of calls to *enqueue* and *dequeue* together is twice that of *qcount* and *print_path* combined. Through experiments we have determined that this optimal code placement for the *dijkstra* program achieves 52% reduced page-switches in line with the justification for the tradeoff involved in the code placement.

F. Experiments

We have implemented this page-aware code-placement heuristic as a profile based compiler post-pass [23] optimization technique. We compile the benchmark first with GCC -O2 optimization, to ensure that the benchmarks are compiled and scheduled for the maximum performance. We modified the sim-outorder [22] simulator to count the iTLB-page switches for an application execution. The individual function calls, and loop counts of the benchmark are extracted through instrumented profiling of the application using the modified sim-outorder simulator. We dissasemble the generated object file, discover the basic blocks, function sizes, loop positions, loop sizes and formulate the input data for the optimization heuristic. The DCFG is formed for the entire application, and the heuristic is applied as in Fig. 10. In order to introduce *nop* stubs into the code for appropriate positioning of the functions and the loops, *no_operation* functions are formed of 8 bytes each and duplicated accordingly.

The optimized code is then built into the binary, and executed using the modified sim-outorder simulator to extract the iTLB-page switch count for the application.

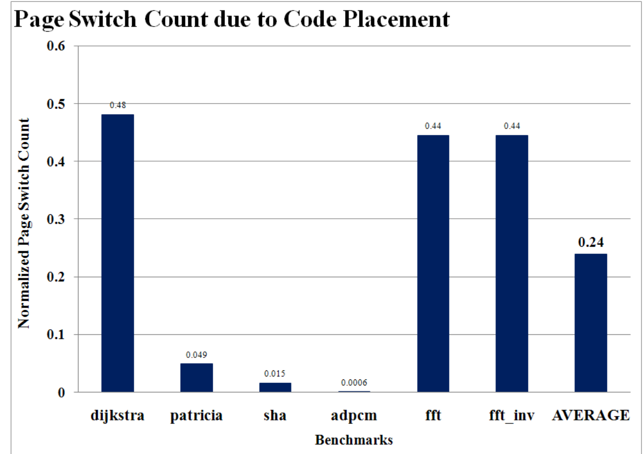

Fig. 13. Impact of Code Placement on Page Switch Count.

In all the benchmarks experimented, only code-placement was performed and no data-TLB page-switch reduction techniques were applied. Fig. 13 plots the page-switch count, after implementing our page-aware code placement heuristic normalzied to the baseline(un optimized code) page-switch count for each benchmark. The benchmarks *patricia, sha* and *adpcm* show significant reduction in the instruction-TLB page-switch count when compared to the other benchmarks, owing to the characterisitics of these applications. *patricia* and *adpcm* are applications which contain large functions, with high number of function calls to smaller functions, and therefore caused high number of page-switches. The heuristic was able to recognize this character of the program, and place the highly used functions such that they were present in the same page as that of the calling instruction from the callee function. Such a page-aware placement of the functions led to the significant page-switch reduction. The *sha* benchmark on the other hand was a large benchmark with many large functions with small number of inter-function calls, but the presence of loops within the functions had a high loop count and were located across page boundaries. Such loops(with maximum-iteration count) were recognized by the heuristic and were placed such that none of these loops were across page boundaries therefore reducing the page-switch count significantly. In all the benchmarks, we observe less than 1% variation in performance and an average of 76% reduction in the page switch count of the applications. This translates into an power saving of 76% over and above the reduction achieved through the *Use-Last* TLB architecture implementation alone.

VIII. Summary

The TLB performs virtual to physical address translation and determines page access permissions. Most mordern processors accomodate virtual addressing and therefore the

TLB is part of the critical path in every access to the cache, and since the $L1$ cache is divided into instruction cache and data cache, and therefore there exist two TLB structures, one for the instruciton (iTLB) and another for the data (dTLB). These TLB structures thus form one of the major hot-spots of the processor, and therefore energy consumption of these structures is a major concern in high general purpose as well as high-performance processors. The *Use-Last* TLB architecture proposed in [1] 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. We have enhanced the application of the *Use-Last* TLB architecture for instruction cache through our page-aware code placement technique. Through our page-aware code placement heuristic, we achieve 76% reduction in the page-switch count of applications with less than 1% variation in performance over a set of control intensive applications. It is to be noted here that the energy reduction achieved through our code tranformations is over and above the energy reduction through the implementation of the *Use-Last* TLB architecture. Our future work in this direction involves further exploration of compiler directed code transformations for energy reduction in the instruction-TLB architecture.

REFERENCES

[1] J. R.Haigh, M. Wilkerson, J. Miller, T. Beatty, S. Strazdus, and L. Clark, "A low-power 2.5 ghz 90 nm level 1 cache and memory management unit," in *IEEE Journal of Solid State Circuits*. IEEE Press, 2004, pp. 1190–1199.

[2] M. Ekman, P. Stenstrm, and F. Dahlgren, "Tlb and snoop energy-reduction using virtual caches in low-power chip-multiprocessors," in *ISLPED '02*. New York, NY, USA: ACM Press, 2002, pp. 243–246.

[3] I. Kadayif, A. Sivasubramaniam, M. Kandemir, G. Kandiraju, and G. Chen, "Optimizing instruction tlb energy using software and hardware techniques," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 10, no. 2, pp. 229–257, 2005.

[4] X. Zhou and P. Petrov, "Low-power cache organization through selective tag translation for embedded processors with virtual memory support," in *GLSVLSI '06*. New York, NY, USA: ACM Press, 2004, pp. 398–403.

[5] P. Petrov, D. Tracy, and A. Orailoglu, "Energy-effcient physically tagged caches for embedded processors with virtual memory," in *DAC '05*. New York, NY, USA: ACM Press, 2005, pp. 17–22.

[6] L. T. Clark, B. Choi, and M. Wilkerson, "Reducing translation lookaside buffer active power," in *ISLPED '03*. New York, NY, USA: ACM Press, 2003, pp. 10–13.

[7] S. Manne, A. Klauser, D. Grunwald, and F. Somenzi, "Low power tlb design for high performance microprocessors," 1997. [Online]. Available: citeseer.ist.psu.edu/manne97low.html

[8] J.-H. Lee, G.-H. Park, S.-B. Park, and S.-D. Kim, "A selective filter-bank tlb system," in *ISLPED '03*. New York, NY, USA: ACM Press, 2003, pp. 312–317.

[9] J.-H. Choi, J.-H. Lee, S.-W. Jeong, S.-D. Kim, and C. Weems, "A low power tlb structure for embedded systems," *IEEE Comput. Archit. Lett.*, vol. 1, no. 1, p. 3, 2006.

[10] Y.-J. Chang, "An ultra low-power tlb design," in *DATE '06: Proceedings of the conference on Design, automation and test in Europe*. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2006, pp. 1122–1127.

[11] I. Kadayif, P. Nath, M. Kandemir, and A. Sivasubramaniam, "Compiler-directed physical address generation for reducing dtlb power," in *ISPASS '04*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 161–168.

[12] V. Delaluz, M. Kandemir, N. Vijaykrishnan, M. Irwin, A. Sivasubramaniam, and I. Kolcu, "Compiler-directed array interleaving for reducing energy in multi-bank memories," in *ASP-DAC '02*, 2002, pp. 288–293.

[13] A. Parikh, S. Kim, M. Kandemir, N. Vijaykrishnan, and M. Irwin, "Instruction scheduling for low power," in *VLSI-SP '04*, 2004, pp. 129–149.

[14] A. Chiyonobu and T. Sato, "Energy-efficient instruction scheduling utilizing cache miss information," in *MEDEA '05: Proceedings of the 2005 workshop on MEmory performance*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 65–70.

[15] M. Kandemir, I. Kadayif, and G. Chen, "Compiler-directed code restructuring for reducing data tlb energy," in *CODES+ISSS '04*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 98–103.

[16] Intel Corporation, "Intel XScale®Technology Overview." [Online]. Available: intel.com/design/intelxscale

[17] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14.

[18] Hari Balakrishnan and Rahul Garg, "Multimedia benchmarks: A performance comparison of multimedia programs on different architectures." [Online]. Available: citeseer.ist.psu.edu/233784.html

[19] V. Zivojnovic, J. Velarde, C. Schlager, and H. Meyr, "Dspstone: A dsp-oriented benchmarking methodology," in *Proceedings of Signal Processing Applications and Technology*, Dallas, 1994.

[20] J. L. Henning, "Spec cpu2000: Measuring cpu performance in the new millennium," *Computer*, vol. 33, no. 7, pp. 28–35, 2000.

[21] BDTI Suite: Berkeley Design Technology Inc, "The bdti benchmark suites." [Online]. Available: bdti.com/products/benchmark_overview.html

[22] T. Austin, "SimpleScalar LLC."

[23] Aviral Shrivastava and Eugene Earlie and Nikil D. Dutt and Alexandru Nicolau, "Operation tables for scheduling in the presence of incomplete bypassing." in *CODES+ISSS*, 2004, pp. 194–199.

[24] Ilya Issenin and Nikil Dutt, "Foray-gen: Automatic generation of affine functions for memory optimizations," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 808–813.