Compiler-Directed Code Restructuring for Reducing Data TLB Energy

M. Kandemir
Dept. of Computer Science & Engr.
Pennsylvania State Univ., USA
kandemir@cse.psu.edu

I. Kadayif
Dept. of Computer Engineering
Canakkale Onsekiz Mart Univ., TR
kadayif@comu.edu.tr

G. Chen
Dept. of Computer Science & Engr.
Pennsylvania State Univ., USA
guilchen@cse.psu.edu

ABSTRACT
Prior work on TLB power optimization considered circuit and architectural techniques. A recent software-based technique for data TLBs has considered the possibility of storing the frequently used virtual-to-physical address translations in a set of translation registers (TRs), and using them when necessary instead of going to the data TLB. This paper presents a compiler-based strategy for increasing the effectiveness of TRs. The idea is to restructure the application code in such a fashion that once a TR is loaded, its contents are reused as much as possible. Our experimental evaluation with six array-based benchmarks from the Spec2000 suite indicates that the proposed TR reuse strategy brings significant reductions in data TLB energy over an alternate strategy that employs TRs but does not restructure the code for TR reuse

Categories and Subject Descriptors
D.3.4 [**Programming Languages**]: Processors –*compilers.*

General Terms
Performance, Experimentation.

Keywords
Code restructuring.

1. BACKGROUND AND MOTIVATION
The TLB provides a translation from a virtual address (provided by the load/store instruction) to a physical address. Depending on the time at which it is performed, this operation can get in the critical path. Moreover, even in the cases that it is not in the critical path, it can consume a significant amount of dynamic power during execution as it is accessed at each load operation.

TLBs are generally designed very carefully at the circuit level to minimize their access times and hit rates. Consequently, they are usually highly associative structures. This characteristic, combined with the fact that they are accessed at each memory reference, can result in significant amount of dynamic power consumption. For example, according to data sheets, in both Hitachi SH-3 and Intel StrongARM, instruction and data TLBs together can consume over 15% of the overall on-chip budget [16, 7]. In addition, since TLBs are small in size, their power densities tend to be high [9]. Because of these factors, it is important to optimize TLB power consumption.

CODES+ISSS'04, September 8–10, 2004, Stockholm, Sweden.

Optimizing TLB performance has a long history [10, 3]. Prior research in reducing TLB energy considered both circuit level and architectural techniques [8, 13, 4, 5, 11, 12]. A recent work has proposed the use of translation registers (TRs) to reduce the energy spent in data TLB (dTLB) accesses [9]. The idea can be summarized as follows.

The architecture provides n-1 translation registers, whose format is:

[Virtual Page Number, Physical Frame Number, Protection Bits].

These registers are manipulated by a special loadTR instruction:

loadTR Virtual Address, TR_i.

When this instruction is issued, the hardware uses the virtual page number that is given, and goes to the dTLB to get the corresponding entry (or to the page table if it is not in the dTLB). It, then, puts this entry (which consists of the physical frame number, protection and other book-keeping bits such as modified/referenced, etc.) into the specified TR_i. The program is not allowed to modify the physical address or the protection bits that get loaded into the TR, as that would compromise protection. When a load/store is issued, some bits of the address (indicated in the load/store) are used to specify whether to go through the dTLB or whether to find the translation within a specific TR. Specifically, one can use the top log n bits (where n-1 is the number of TRs) to indicate whether the address should go through the dTLB for a translation (which corresponds to all these log n bits being 0) or whether to take the physical frame number from a TR, and if so, which specific one. Note that, while it is also possible to associate these log n bits with the instruction itself, this presents at least two drawbacks. First, this can increase the decode logic complexity as a result of an increase in instruction sizes. Second, this prevents a load operation in the program from using different TRs at different execution phases.

In our approach, the compiler is responsible for setting these bits. Whenever the compiler knows for sure that a TR_i has the translation that is needed, it forces these bits to correspond to that TR_i. Otherwise (i.e., if the compiler is not 100% sure), it will conservatively set them to 0 so that the translation goes to the dTLB.

We assume that arrays in the code are aligned to page boundaries, which can be ensured using existing compiler directives (e.g., SGI MIPSPro compiler has a directive to enforce this). The idea behind our scheme is to load TRs whenever we notice that this translation is going to be heavily used in the near future. Consider the code fragment below:

```
for i: 1, N
  …U1[i]…U2[f(i)]…
```

where f() is not a compiler-analyzable function. In this case, the compiler restructures the computation around U_1 using a loop transformation called strip-mining [17]:

```
for s: 1, N, P
  load translation for U1[s] to TRj
  for i: s, max(s+P,N)
    access U1[i] through TRj
    access U2[i] through dTLB
```

In this code sketch, P is the data page size. The first loop iterates over the pages, and before starting to process a new page, TR_j is loaded (updated) from the dTLB with the associated translation. The inner loop, on the other hand, iterates over the current page using the translation in TR_j (for U_1). That is, during the execution of the inner loop, the accesses to array U_1 do not lookup dTLB. Array U_2, on the other hand, is accessed through dTLB.

It is to be observed that satisfying most of the address translations from the TRs can bring large power benefits since (1) each TR has a very small per access power consumption compared to a dTLB, and (2) we do not perform any tag comparison when a TR is accessed. Consequently, it is important to make sure that most of the translation requests are satisfied from the TRs. Satisfying this goal depends on two major factors. First, the number of TRs can make a difference as more TRs means less dTLB visits (since we can capture more address translations, e.g., coming from different arrays). Second, the data access pattern determines the order in which address translations are requested. While the first factor is not something that can be changed easily, the second factor can be controlled by performing source code level modifications to the application. In particular, an optimizing compiler can play a crucial role here as the code/data optimizations it performs can change the data access pattern entirely. This paper proposes a fully automatic *compiler-based* strategy that increases the effectiveness of the TRs by satisfying a majority of the translation requests from them. It achieves this by restructuring the code (data access pattern) in such a way that the resulting code reuses the contents of a given TR as much as possible. In other words, the objective here is to increase TR reuse. However, since we are restructuring the code, the impact of our transformations on other aspects of the code (e.g., cache behavior) should be studied as well. In particular, an important question is how our code modifications interact with other compiler optimizations.

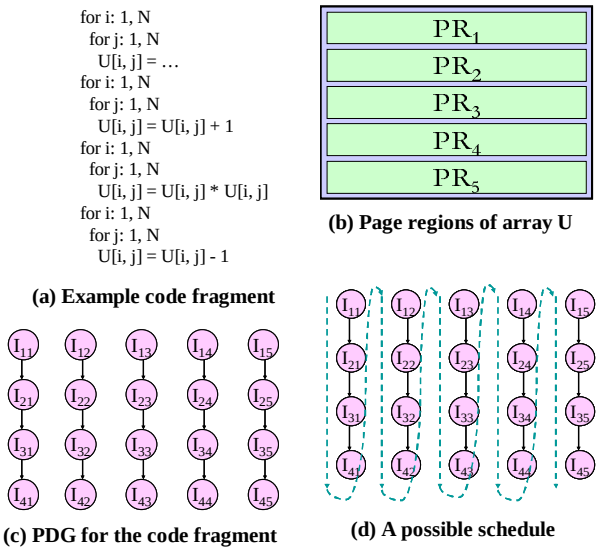
Our focus in this paper is on array-intensive applications. Our experiments with a set of six Spec2000 array benchmarks indicate that the proposed compiler-based scheme cuts the number of dTLB accesses significantly, thereby reducing the energy spent in address translation. Even more importantly, the proposed scheme outperforms a technique that uses TRs but does not restructure the application code for TR reuse. The experimental analysis also shows that the proposed scheme performs consistently well under a wide variety of simulation scenarios.

The rest of this paper is organized as follows. Section 2 presents our code restructuring strategy in detail. Section 3 introduces our experimental setup, and Section 4 discusses experimental results. Section 5 concludes the paper with a summary.

2. DATA-CENTRIC CODE RESTRUCTURING

We propose a compiler-directed data-centric code restructuring strategy for utilizing the TRs. We first focus on single array case, and then later show how our approach extends to multiple arrays case.

A *page region* of an array is the set of consecutive elements that map to the same data page. Our approach operates on a graph



(a) Example code fragment **(b) Page regions of array U**

(c) PDG for the code fragment **(d) A possible schedule**

Figure 1.

structure called the *page dependence graph* (PDG). In this graph, each node corresponds to a set of loop iterations that access the elements in a particular page region. The edges between the nodes indicate *data dependences*. A traversal of this graph corresponds to the execution of the iteration spaces of all the loop nests in the code. A *legal traversal* is the one that respect all data dependences. Consider, for example, the code fragment shown in Figure 1(a), where four different loop nests access an array (U). Figure 1(b) shows how the array is divided into five page regions, and Figure 1(c) gives the PDG under this page region partitioning. To obtain this graph, each loop nest is divided into five sets. The iterations in the set (represented by a PDG node) marked I_{ij} represent the loop iterations of nest i that access the page region j.

In the next step, we schedule the nodes in this graph in such a way that the nodes that access the same page region are scheduled successively (i.e., one after another) as much as possible. The rationale behind such a schedule is to utilize the current TR contents as much as possible; i.e., to maximize TR reuse.

Returning to our running example, Figure 1(d) depicts a possible schedule (dashed curve) determined by our approach (which will shortly be presented). Note that, in a sense, such a schedule (which starts with I_{11} and ends with I_{45}) represents the ideal scenario where a page region is fully utilized by all loop nests before moving to the next page region. That is, once a TR is loaded a translation, that translation is reused as much as possible. This means that, even if we have more TRs, we could not reduce the number of dTLB lookups. As a second example, Figure 2 shows another PDG and a possible schedule determined by our approach. It is to be noted that the extra dependences in this example prevent us from fully utilizing the current page region. Consequently, each page region needs to be visited twice, which may mean extra TR updates (loads), depending on how many TRs we have.

Formally, let us assume, without loss of generality, that the program to be optimized has s loop nests, and I_1, I_2, …, I_s denote the iteration sets of these nests (each iteration set contains the iterations executed by a loop nest). We define the *computational space* of the program, I_c, as:

$$I_c = \cup_k I_k,$$

where $\cup$ denote set union and $1 \leq k \leq s$. As has been discussed earlier, we use I_{ij} to denote the set of iterations from loop nest i that access the page region j. In formal terms, $l \in I_{ij}$ if and only if the following holds:

$$\exists \text{ R and } \exists d \in PR_j \text{ such that } R(l) = d,$$

where R is a reference in nest i, and PR_j indicates the page region j. That is, iteration l (of nest i) should access a data element from page region j. Note that:

$$\cup_i \cup_j I_{ij} = I_c.$$

We say that there is a *data dependence* between I_{ij} and I_{mn} if an iteration that belongs to I_{mn} depends on (the result generated by) an iteration of I_{ij}. As will be discussed shortly, in our approach, the nodes of this graph are scheduled using a heuristic algorithm, which is based on *list scheduling*, a scheduling paradigm used by compilers [14] and high-level synthesis tools [6]. Specifically, the algorithm first divides the PDG nodes into *chains* and schedules one chain at a time. In mathematical terms, if possible, we want the execution to move always from I_{ij} to I_{kj}. That is, we want to stay within the same page region as long as possible. As has been discussed earlier, this helps improve TR reuse and reduce the number of TR updates/dTLB references. Therefore, our chain-building strategy tries to place such nodes into the same chain.

It should be observed that this approach requires only one TR if all the nodes that access a given page region can be scheduled one after another. For example, considering the schedule depicted in Figure 1(d), we can traverse all the nodes (which is the entire computational space) using only a single translation register; that is, the TR is updated only once per page region (in the first access). However, when we have a schedule like the one illustrated in Figure 2, we have two options. If we use only a single TR, we need to update the TR 10 times (including the initial loads for page regions). Alternately, we can use multiple TRs to cut the number of TR updates. For example, if we have 5 TRs, we can assign a TR to each page region, and can cover the entire computational space with just 5 TR updates (all of them being initial loads). Consequently, the number of TRs can play a crucial role in overall behavior of the application. As a result, effective use of available TRs is critical.

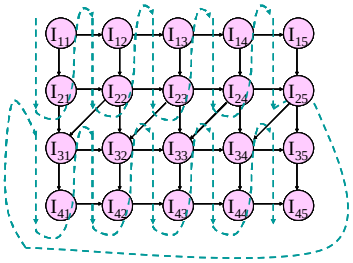


Figure 2. An example PDG and a possible schedule.

An important question that needs to be answered, though, is how can we schedule a PDG with r translation registers ($1 \leq r \leq N$, where N is the total number of page regions) such that the number TR updates is minimized? Note that minimizing the number of TR updates is important as each such update increases the code size, and causes extra energy consumption in datapath and instruction cache/memory (in addition to the dTLB visit it entails). We address this issue by adopting a strategy, which works as follows. In the first step, we identify a set of *threads* (*data dependence chains*, or simply *chains*) in the PDG. Each such chain consists of set of nodes and edges, and does not share a node or edge with the other chains, except for the start or end nodes of the chain. We use s to denote the number of such chains. If $r \geq s$, that is, the number of TRs is larger than the number of independent chains, we assign a private TR to each chain. Note that, with such an assignment, the contents of a TR are updated only for the initial loads. Therefore, the total cost of such an assignment is s. A TR is said to have *completed* its chain when it reaches the end node of the chain. Figure 3(a) depicts this situation for an example PDG with r=5.

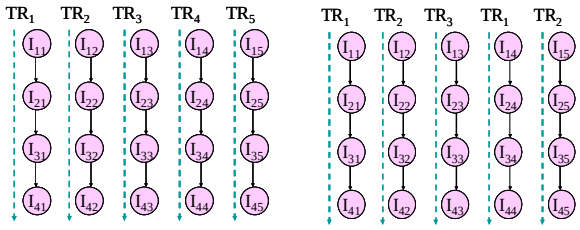


Figure 3.Two different TR assignments. (a) 5 TRs. (b) 3 TRs.

Note that, in this example, we have 5 independent chains, i.e., s=5. A more interesting scenario occurs when $r \leq s$. In this case, we still assign one TR per chain. However, a TR is reassigned to another chain (that has not been assigned a TR yet) when it completes its current chain. An example of this scenario is illustrated in Figure 3(b), assuming r=3 (and, s=5 as before). However, this is still a preferable scenario since the only TR loads we make are the ones necessary when we move from one data page to another (i.e., the initial loads).

It should be observed that once a chain is scheduled we may not be able to execute all its nodes one after another. This can happen as a result of the dependences between the chains. That is, if there is an edge entering to the chain from a node outside that chain, it is not possible to complete the chain without executing (scheduling) the mentioned (outsider) node. In this case, such a chain is treated as not a monolithic chain but a sequence of *subchains*; and, our algorithm schedules each subchain separately. However, the algorithm also tries to use the same TR for all subchains of a given chain as much as possible. The sketch of our algorithm is given in Figure 4. In this algorithm, at each iteration of the while loop, a new subchain is selected and scheduled. Note that this is a compiler algorithm and executed off-line, and in the last step, it builds the code for each subchain. The resulting (restructured) code is then executed at runtime.

Before giving examples to illustrate how this algorithm works in practice, let us explain our chain/subchain-building process in more detail. Our algorithm for constructing chains is rather simple. We first start with a node I_{ij} and expand it with nodes I_{kj} ($k \neq i$) until we reach the point where it is not possible to expand it further. The selected nodes (that start with I_{ij}) form a chain. Then, we select a new node (which is different from the ones that have already been placed into a chain), and repeat the process. This continues until all the nodes have been assigned to chains.

Our algorithm for constructing subchains is similar to the chain construction algorithm described in the previous paragraph. Basically, we iterate over each chain determined by the chain construction algorithm. For each chain, we identify the points where there is a dependence from an outsider node to a node in the chain. All such nodes in the chain delimit the subchain boundaries. However, once a subchain is detected, all its nodes and connected edges are removed from the PDG, before starting to search for the next subchain.

As an example, let us re-consider the PDG in Figure 1. Our algorithm identifies five chains, and since these chains are not connected, we have no subchains. Within the while loop of the algorithm in Figure 4, at each iteration, we select a chain and assign a TR to it. If we do not have enough TRs to assign a private TR to each chain, we reuse TRs (compare the cases in Figures 3(a) and 3(b)). Note that this does not cause any additional overhead for this example as, no matter what TR is used, each chain requires one TR update (the initial load). As a second example, consider the PDG in Figure 2. In this case, our algorithm identifies the same chains as before. However, this time the chains are

```
Build computational space
Build the PDG (page dependence graph)
Determine the chains in the computational space
Divide chains into subchains if necessary
While (there are subchains to schedule)
 Select a schedulable subchain
 Assign a TR to the subchain based on the following two constraints:
   1. where possible, use the same TR as the other subchains of the chain
      this subchain belongs to.
   2. where possible, do not use a TR assigned to an unrelated subchain
EndWhile
Generate code for each subchain based on data dependence between subchains
```

Figure 4. Subchain scheduling algorithm.

dependent on each other. Consequently, the algorithm also identifies subchains. Note that each subchain contains a set of nodes, with the characteristic that the entire subchain can be executed without interruption when the first node starts executing. In this case, we divide each chain into two subchains, and schedule the subchains observing the data dependences between them. However, we are also careful in assigning the TRs in such away that both the subchains of a given chain use the same TR.

In our discussion so far, we omitted an important point, which is the possibility that a loop iteration can touch different page regions. After all, if there is no data elements shared among I_{ij}s of a given nest i, we cannot talk about data dependences among them either. Our approach to this problem can be explained as follows. Note that, when an iteration accesses multiple page regions, I_{ij}s share page regions. For example, consider the following nested loop and the assignment statement within it:

```
for i: 1, N
  for j: 2, N
    U[i, j] = U[i, j-1] +1
```

The iteration point (k_1,k_2) accesses array elements $U[k_1, k_2]$ and $U[k_1, k_2-1]$, and these two elements can be in two different page regions. Now, another iteration point, (k_1,k_2-1), which maps to a different PDG node than (k_1,k_2), accesses array elements $U[k_1, k_2-1]$ and $U[k_1, k_2-2]$. That is, this iteration point shares a page region with (k_1,k_2). Since such problematic iteration points occur only across the page region boundaries, our implementation handles these iteration points separately. Returning to the example above, assuming that iterations (k_1,y) $[1 \le y \le k_2-1]$ access only page region 1, iteration (k_1,k_2) accesses both page region 1 and page region 2, and iterations (k_1,y) $[k_2+1 \le y \le N]$ access only page region 2, and that we have only two page regions, we generate two loop nests in the restructured code, each using its own page region (if we have 2 TRs, each can use a private TR). We will also have a separate statement between these two nests for executing iteration point (k_1,k_2). In this statement, the first array reference will be accessed via the first TR, and the second array reference will be accessed via the second TR. Since the page regions are large, we do not expect too much increase in code size or deterioration in instruction cache performance. In our experiments, we also quantify the increase in the code size as a result of our dTLB optimization.

When we have multiple arrays in the code, it is more difficult to determine a good strategy to traverse the loop iterations in the computational space. Basically, the main problem is to determine an array to restructure the computation around. This array is called the *seed array*. However, one needs to be careful in selecting the seed array as restructuring the entire computation around one array can lead to frequent dTLB visits (TR loads) during the accesses to the remaining arrays. That is, maximizing TR reuse for one array may not be preferable (let alone being optimal) for the performance of accesses to the remaining arrays.

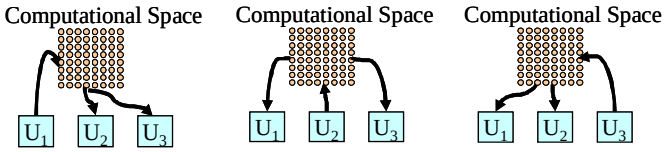


Figure 5. Alternative ways of proceeding for a case with 3 arrays.

Figure 5 illustrates an example where we have three arrays accessed by a code fragment. If we select array U_1 as the seed array and restructure the entire code around this array (as if it is the only array accessed by the application code), we should consider the impact of this on arrays U_2 and U_3. A similar argument can be made for other two alternatives as well (i.e., using U_2 or U_3 as the seed arrays).

To address this problem, we use a strategy based on *estimating* the number of TR updates. This is possible in our application domain where nested loops access multi-dimensional arrays, and the compiler can figure out the data access patterns by analyzing the code being optimized. In addition, even large programs typically have a small number of arrays. Therefore, considering each array in turn (as a potential for using to restructure the code around) is not expected to incur too much overhead at compilation time. Focusing on the example discussed in the previous paragraph, our approach considers each array in turn and determines the total number of TR updates when the array being considered is used to restructure the code. Note that, in calculating the number of TR updates, we consider all the arrays in the code (not just the one being used as the seed array). This process is repeated for each array, and the one that leads to the minimum number of TR updates is selected, and the entire computational space is restructured using that array.

2.1 Discussion

It should be noted that our TR-based execution strategy is safe from the OS viewpoint. The only operation allowed for the programmer/compiler is to load a TR from the dTLB. The TRs themselves cannot be manipulated by normal register operations. As a result, a wrong update of a TR can at most corrupt the address space of the program in question (not the address space of some other program). Also, when context switching from one address space to another, the TRs can be treated as normal registers, i.e., they can be saved and restored. Therefore, they pose no problem even in a multi-programmed embedded environment.

Another important issue is how our optimization affects the impact of other compiler optimizations and how it interacts with other code transformations. It should be noted that our approach tries to obtain data page-level locality as much as possible so that the TR reuse can be increased. However, this does not guarantee good data cache behavior as the unit of transfer between the cache and the memory is a cache line (block), which is much smaller in size than a data page. Consequently, the compiler needs to apply cache

Table 1. Default simulation parameters used in our experiments.

Simulation Parameter	Value
L1 Data and Instruction Caches	16KB, 4-way, 32 byte blocks, 1 cycle latency, write-back policy
Unified L2 Cache	256KB, 4-way, 128 byte blocks, 10 cycle latency, write-back policy
Data and Instruction TLBs	single-level, 128 entries, full-associative, 50 cycle miss penalty
Page Size	4KB
Off-Chip DRAM	128MB (divided into 32MB blocks), 100 cycle latency

Table 2. The benchmark codes used in our evaluation.

Benchmark	dTLB Accesses	dTLB Miss Rate	dTLB Energy (mJ)
177.mesa	1622495502	0.682%	328.7
178.galgel	1334208722	0.418%	266.1
179.art	1880061704	1.003%	390.4
183.equake	1568740411	1.277%	341.4
187.facerec	1066528717	0.930%	304.8
188.ammp	1872011056	1.365%	429.0

locality enhancing transformations following our code optimization. If the code is already modified for data cache locality, this can increase the effectiveness of our approach since there is more page reuse to exploit. However, even in this case, our code transformation still needs to be applied, as we need to make the page boundaries explicit so that our instructions can be inserted in the code. The detailed discussion of the interactions between our approach and conventional code optimizations is beyond the scope of this paper.

3. EXPERIMENTAL SETUP

We enhanced SimpleScalar simulator [2] to perform a detailed performance and energy evaluation of the proposed scheme. The default simulation parameters are given in Table 1. Note that we use 50 cycles (instead of 100 cycles) for the TLB miss penalty, assuming that some of TLB misses (translations) could be satisfied from the cache. The dTLB energy consumptions (to be presented later) have been obtained using a modified version of CACTI [15]. Note that one could potentially have optimized the TLB structure to reduce the per access dTLB energy; but this would not affect our conclusions as we are interested in the reduction in dTLB accesses and percentage dTLB energy savings.

Array accesses are dominant in many applications. In this study, we used six randomly selected benchmark codes from the Spec2000 benchmark suite. Table 2 gives these benchmark codes along with their dTLB accesses/misses and dTLB energy consumptions with the default simulation parameters shown in Table 1. While most of these codes are dominated by global array accesses, there are also nonnegligible stack references. Therefore, in all our experiments, we reserved one TR exclusively for the stack references. This is sufficient as stack references exhibit very good data locality.

We performed experiments with two different optimized versions of each benchmark code in our suite:

1. *Base-Opt:* This version is adapted from [9], and uses the available TRs as efficiently as possible. However, it does this without changing the structure of the code (except for loop strip-mining as explained earlier and loop distribution when appropriate). That is, it represents the best results (the minimum number of dTLB accesses) under the condition that the program structure is not modified for TR reuse.

2. *Structured-Opt:* This is the version described in this paper.

All necessary code restructurings have been automated using the SUIF infrastructure [1]. All the performance and energy savings are given (in the following paragraphs) with respect to the original codes that do not make use of TRs. The behavior of this original version is summarized in Table 2. To obtain these results, we fast-forwarded 1 billion instructions, and simulated the next 300 million instructions in detail. While we have performed experiments with both VI-PT (virtually-indexed, physically-tagged) and VI-VT (virtually-indexed, virtually-tagged) L1 caches, due to space concerns, we present only the VI-PT results. Also, since global "scalar" references are not significant in these benchmarks, we do not allocate any registers for them, and they always go through the dTLB.

4. RESULTS

In our first set of experiments, we measure the success of Base-Opt and Structured-Opt in reducing the number of dTLB accesses. The results are given in Figure 6. On the x-axis, the pair (a,b) indicates "a" TRs for the stack and "b" TRs for the array references. As discussed earlier, in our experiments "a" is always 1. The y-axis gives the number of dTLB accesses as a fraction of the original case where we do not use any TRs. We first notice that the applications with large number of stack references benefit from allocating a TR for stack references (that is, the configuration (1,0) on the x-axis). On the other hand, we do not gain much with this configuration in other benchmarks since the global references dominate memory accesses. As we increase the TRs for global arrays, one can observe that the number of dTLB visits reduces significantly for both Base-Opt and Structured-Opt. In particular, with the (1,8) configuration – i.e., a total of only 9 TRs – the normalized dTLB lookups is 54.5% and 32.6%, on the average, for Base-Opt and Structured-Opt, respectively. We also see that, in all benchmark codes considered, Structured-Opt outperforms Base-Opt as the former increases the reuse of TR contents.

While the reduction in dTLB misses is an important metric, one would ultimately be interested in reducing the dTLB energy consumption. Also, one could ask how our approach compares with respect to a multi-level dTLB structure. To answer these questions, in our next set of experiments, we measured the normalized dTLB energy consumptions. Figure 7 shows these results as a fraction of the monolithic 128-entry dTLB (without any TR). The bars on the left side of each graph show the energy consumption with our scheme using different number of TRs for stack and global references. It should be emphasized that the energy values given in this figure include the energy cost of accessing the TRs (in addition to the dTLB lookups). The bars on the right side of each graph give the total energy consumption for a two-level dTLB structure. In this structure, the second level has always 128 entries; and we vary the entries in the first level from 1 to 8.

We see from these graphs that our scheme (Structured-Opt) brings much more benefits as compared to a multi-level dTLB, with the first level having as many entries as the TRs in our scheme. This is due to three main factors. First, since a TR access in our scheme does not involve any tag comparison cost, it is more efficient than a multi-level dTLB access even if the latter hits in the first level. Second, the per access dynamic energy cost of a multi-level increases when the number of entries in the first level is increased. Third, since the TRs are managed by the software (compiler), they are more effective (in terms of exploiting locality) than LRU-managed dTLBs.

The next metric that we study in our experimental evaluation is performance behavior. Our approach incurs a performance overhead due to TR updates. In our six applications, the maximum increase in execution cycles with respect to the original case (where no TR is employed) is 3.15%, and in three of them the increase in execution cycles is negligible. Overall, these results demonstrate that the proposed scheme can improve dTLB energy consumption significantly without much impacting the original execution cycles.

5. CONCLUDING REMARKS

Virtual-to-physical address translation consumes as much as 16% of the chip power on some processors due to its high associativity and access frequency. As opposed to most prior work that focuses

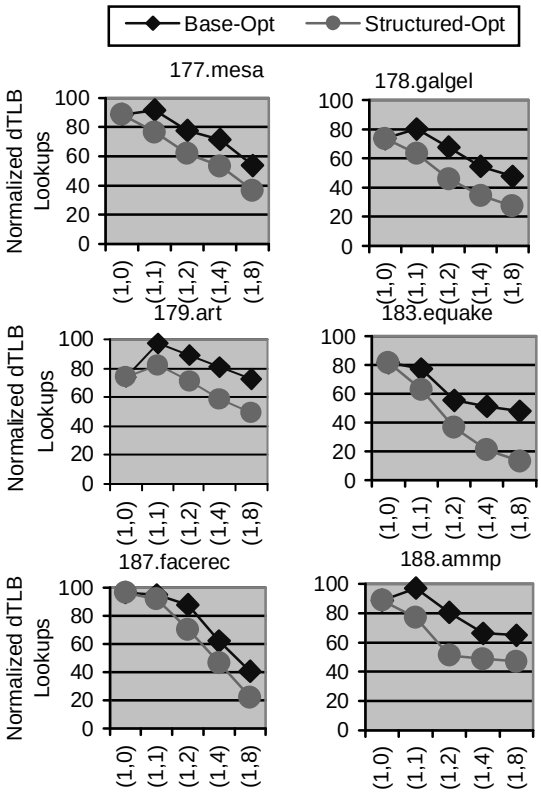


Figure 6. Normalized dTLB lookups for the VI-PT cache lookup mechanism. On the x-axis, the pair (a,b) means "a" TRs for the stack and "b" TRs for the global references.

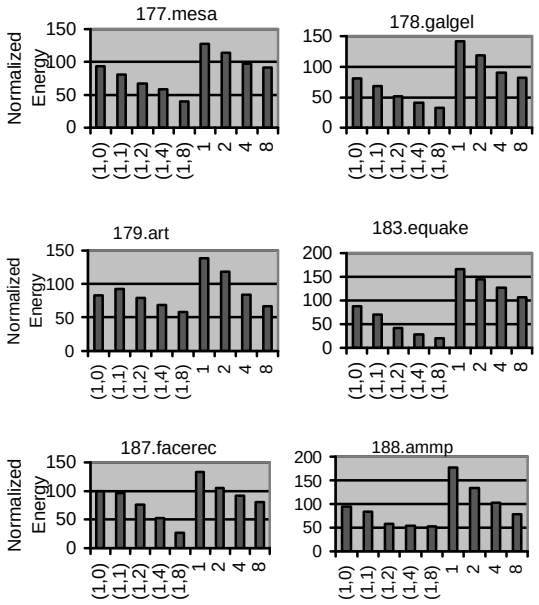


Figure 7. Normalized energy consumptions for the VI-PT cache lookup mechanism. On the left side of the x-axis, the pair (a,b) indicates "a" TRs for the stack and "b" TRs for the global references. The bars on the right side of each graph are for a multi-level dTLB with 1, 2, 4, and 8 entries in the first level and 128 entries in the second level.

mainly on circuit and architecture optimizations for reducing dTLB energy consumption, this paper uses compiler to restructure the application code so that it works better with explicit address translation registers (TRs) that keep frequently used virtual-to-physical address translations. Results with a suite of six Spec2000 applications indicate significant reductions in dTLB accesses as a result of the compiler-based scheme. This results in significant reduction in dTLB energy, at the expense of a very small increase in the execution cycles.

6. ACKNOWLEDGMENTS

This work was supported by NSF Career Award #0093082.

7. REFERENCES

[1] S. Amarasinghe, J. Anderson, C. Wilson, S. Liao, R. French, M. Hall, B. Murphy, and M. Lam. The multiprocessor as a general-purpose processor: a software perspective. IEEE Micro, 1996.

[2] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: the Simplescalar toolset. Technical Report CS-TR-1996-1308, Department of Computer Science, UW, 1996.

[3] T.-C. Chiueh and R. H. Katz. Eliminating. address translation bottleneck for physical address cache. In Proceedings of ASPLOS, 1992.

[4] J.-H. Choi, J.-H. Leek, S.-W. Jeong, S.-D. Kim, and C. Weems. A low power TLB structure for embedded systems. IEEE Computer Architecture Letters, Volume 1, January 2002.

[5] L. T. Clark, B. Choi, and M. Wilkerson. Reducing translation lookaside buffer active power. In Proceedings of the International Symposium on Low-Power Electronics and Design, p.10-13, 2003, Seoul, Korea.

[6] G. De Micheli. Synthesis and Optimization of Digital Circuits. McGraw-Hill, 1994.

[7] Intel StrongArm Processor. http://www.intel.com/design/pca/applicationsprocessors/1110_brf.htm.

[8] T. Juan, T. Lang, and J. J. Navarro. Reducing TLB power requirements. In Proceedings of the International Symposium on Low-Power Electronics and Design, 1997.

[9] I. Kadayif, P. Nath, M. Kandemir, and A. Sivasubramaniam. Compiler-directed physical address generation for reducing data TLB power. In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, March 2004.

[10] J. Knight and P. Rosenfeld. Segmented virtual to real translation assist. IBM Technical Disclosure Bulletin, 27(2):1077-1078, July 1984.

[11] H.-H. S. Lee and C. S. Ballapuram. Energy efficient data TLB and data cache using semantic-aware multilateral partitioning. In Proceedings of the International Symposium on Low-Power Electronics and Design, 2003.

[12] J.-H. Lee, G.-H. Park, S.-B. Park, and S.-D. Kim. A selective filter-bank TLB system. In Proceedings of the International Symposium on Low-Power Electronics and Design, pp. 312-317, 2003, Seoul, Korea.

[13] S. Manne, A. Klauser, D. Grunwald, and F. Somenzi. Low-power TLB design for high-performance microprocessor. Technical Report, Department of Electrical and Computer Engineering and Department of Computer Science, University of Colorado, Boulder, CO, 1997.

[14] S. S. Muchnick. Advanced Compiler Design and Implementation. Morgan-Kaufmann, 1997.

[15] G. Reinman and N. P. Jouppi. CACTI 2.0: an integrated cache timing and power model. Research Report 2000/7, Compaq WRL, 2000.

[16] SH-3 RISC processor family. http://www.hitachi-eu.com/hel/ecg/products/micro/32bit/sh_3.html.

[17] M. Wolfe. High-performance compiler for parallel computing. Addison-Wesley, 1996.