

Exploiting Critical Data Regions to Reduce Data Cache Energy Consumption

Ananda Vardhan
kanandv@gmail.com

Y N Srikant
srikant@csa.iisc.ernet.in

ABSTRACT

In this paper, we address the problem of power consumption in a data cache by focusing on the latency tolerance of data regions. Latency tolerance of a data region indicates how critical the data region to the overall performance of the program. We use a profile-based technique focusing on context-as well as path-sensitive analysis to identify critical and non-critical data regions. We deploy criticality analysis to drive a power-aware optimization technique which allocated a split data cache, operating in normal and drowsy modes, to critical and non-critical data regions respectively. This technique saves around 30% of total power and 20% of leakage power in the data cache without any significant performance penalty.

1. INTRODUCTION

A memory subsystem is one of the primary energy consuming components in modern embedded systems. In memory subsystems, several parameters like per-memory access latency, per-memory access power consumption, and die area have a direct and significant impact on the overall power efficiency of embedded computing systems [8]. Various subsystems like data caches, RAM and the disks constitute the memory subsystem.

Cache memories emerge as significant contributors to the leakage power problem because they constitute a significant proportion of on-chip transistors in embedded devices [14]. To quote few examples, on chip caches occupy 43% of total chip area in the SA-110 [4] and about 75% in Itanium-2 [12]. They consume 30% of the total power in the processor cores like StrongARM and 16% of the total power in Alpha 21264. This attracts considerable attention of designers and justifies the need to achieve circuit and architectural optimizations in order to reduce leakage power. The simplest and most frequently used method of reducing such leakage power is to place unused cache lines into low leakage mode and keep the rest of the cache lines active by exploiting some form of locality [11]. Several circuit techniques and management schemes exist in literature that indicate the timing and procedure for turning individual cache lines on or off [11].

In this paper, we address the problem of power consumption in a data cache by focusing on the latency tolerance of the data regions. Latency tolerance of a data region indicates how critical a data region is to the overall performance

of the program. Figure 1(a) provides a brief overview of our approach.

We implement a context sensitive, profile-driven analysis, to identify disjoint data regions. The length of the calling context and dynamic path length are taken into account.

We now define criticality of the data regions identified in the previous step (profile-driven analysis). We apply a critical path model of a superscalar out-of-order pipeline and quantify latency tolerance of all the memory operations. A memory operation can be either a hit or a miss at some level of the cache hierarchy. This means that the latency of a memory operation could vary in value depending on the particular architectural event. If the impact of this latency changes the instructions per cycle (IPC), then the memory operation is considered as *critical*. By considering the latency tolerance of these memory operations, we analyze the criticality of a disjoint data region. For our research, we define critical data regions as, *the regions of data that are accessed by critical memory operations*.

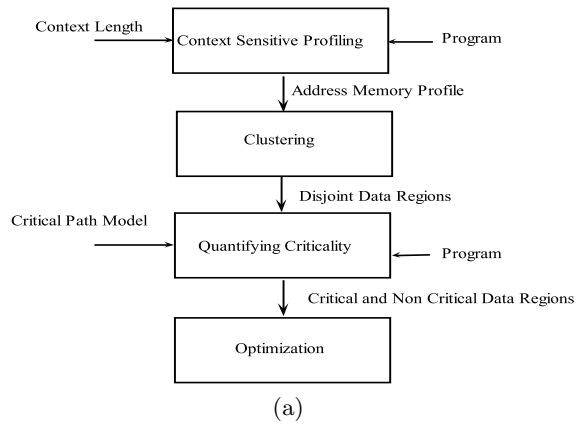
To the best of our knowledge, our work is the first one that describes power-aware optimization using critical data regions.

1.1 Organization

The subsequent sections of the paper are ordered as follows. Section 2 briefly provides the background and discusses the significance of a data region in the context of power savings. Section 3 elaborates the technique to identify disjoint data regions. Section 4 explains the quantification of criticality and in section 5 the optimization technique that employs this criticality analysis to reduce power consumption is described along with the experimental design and our power saving results. Section 6 describes related work and section 7 concludes.

2. BACKGROUND

Most of the compiler-directed power optimization techniques proposed in the literature operate at the instruction level. Any decision made by an optimization based on a per-instruction analysis should be independent of the decisions already made for other instructions. This non-interference based approach can yield efficient optimizations by avoiding frequent variations in the tuning of the parameters. However, per-instruction based techniques do not capture the higher level semantics of programs as they operate at a very fine grain level of program execution. Programmers implement meaningful procedures and data structures in their software designs. Such coarse-grained information on



```

for(k=0; k<num_nets_affected; k++) {
1)   inets=nets_to_update[k];
2)   if(net_bloc_moved[k] == FROM_AND_TO) {
3)       net[inet].tempcost = -1;
4)       continue;
5)   }
6)   bb_coords[inet] = bb_coord_new[bb_index];
7)   if(net[inet].num_pins > SMALL_NET)
8)       bb_num_on_edges[inet]=bb_edge_new[bb_index];
9)   bb_index++;
10)  net[inet].ncost=net[inet].tempcost;
11)  net[inet].tempcost=1;
}

```

(b)

Figure 1: Overview of the approach. An example from the benchmark *vpr*. Boxed statements are more critical.

software designs can be utilized in designing optimizations which can result in significantly more gains as compared to per-instruction based optimizations.

An Example

Consider an example drawn from the *vpr* benchmark shown in figure 1(b). Let us first consider the scenario when per-instruction based optimizations are to be carried out. As shown in the figure, a significant portion of the total memory accesses are concentrated towards two data structures *net_bloc_moved* and *net*. The branch instruction in line 7 depends on the memory accesses to *net*. A branch instruction is a critical instruction. Therefore, these accesses can be considered as more critical when compared to other accesses to the same data structure as depicted in lines 3, 10 and 11. When the memory operation at line 7 is encountered, the dynamic criticality predictor (derived from the critical path model as explained in [6]) considers that access as critical. Consider that all the fields of *net* belong to the same cache line. When this cache line is accessed at 10 and 11, the predictor considers these accesses to be non-critical. During the execution of the program, this continuous change in the state of the instruction could lead to sub-optimal behavior of an optimization. For example, a cache controller

may continuously change the state of cache lines considering the criticality of data accessed by these individual instructions. This demonstrates that considering instruction criticality in isolation for the purpose of classifying data as critical or non-critical is not sufficient for an optimal power-saving technique. We need to take into account the region of memory that gets accessed by non-critical data accesses and apply an optimization on the *entire region* (for example place the entire region into a low-power state). In order to design such an optimization, the data regions that show uniform criticality behavior need to be identified first. In the rest of this section, we explain the techniques that consider a region of data for power optimization and lead to significant power savings.

2.1 Attributing Criticality to a Data Region

The criticality of a data region can be mainly attributed to memory access latencies and branch-load dependencies. For instance, a memory access to a data region can hit a cache or go through a series of misses. This delay may vary across different memory accesses. Similarly, in the case of branch-load dependencies, a branch misprediction can delay a memory access to a particular data region. Hence, we can correlate memory access latencies and branch mispredictions with the criticality of a data region. We identify these critical data regions and comprehend their behavior from a complete program point of view. This helps us to drive power-aware optimizations at the data structure level, which are semantically meaningful data regions rather than contemplating on per-instruction basis.

We now provide a summary of critical path model of an out-of-order processor [6]. The performance characteristics of a program including the architectural events and inherent data dependencies are modeled by a dynamic dependency graph. Each node in a dynamic dependency graph represents a mode of instruction. An instruction can be in the dispatch mode *D*, the execution mode *E* or in the commit mode *C*. These three nodes denote events within the machine pertaining to the instruction. The model captures different dependencies between these modes across the instructions. For each instruction there is a dependency between *D* and *E* followed by *E* and *C*. This indicates the execution of the instruction, i.e. dispatch followed by execution and finally commit. The model also captures dependencies between two instructions. A *DD* edge captures in-order dispatch, *CC* captures in-order commit and *EE* edge between two instructions captures data dependency. A mis-predicted branch's execution outcome is needed to dispatch instructions from the right target. Such control dependencies are captured by *ED* edge. A reorder buffer dispatches instructions in program order. The size of this buffer places a constraint on the dispatch of new instructions unless the instruction at the head of the buffer commits. Such resource constraints are captured in the form of *CE* dependency edge. Each of these edges is attributed by a latency indicating the time taken for the architectural event. The commit of last instruction ends this dependency graph. By traversing backwards on the longest path, we can identify the critical path of the program. The instructions on the critical path are the critical operations. The regions of data accessed by critical memory operations are critical data regions.

Fields et al. [6] also predict the criticality of in-flight instructions by planting tokens through the dependency edges. The

critical path predictor aids in the accurate prediction of the critical path. This further helps in tuning architectural parameters which harmonize with the critical instructions thereby improving the performance.

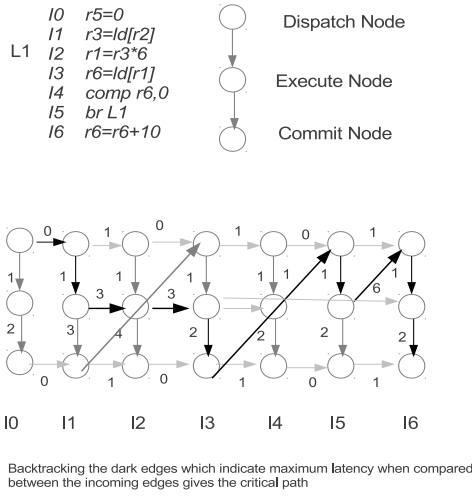


Figure 2: Fields critical path model. This Example shows a dynamic trace of instructions and dynamic dependency graph.

An example of this model is shown in Figure 2. The Figure shows a dynamic trace of a program. Each instruction is represented by three nodes as described earlier. The instructions *I2* and *I1* have *EE* edge because of data dependency. Instructions *I1*, *I3* and *I5* are constrained by reorder buffer (ROB) size of length 2. Instructions *I5* and *I6* have a branch mis-prediction dependency. By traversing backwards from the last instruction’s commit node and taking the maximum latency edges, the critical path can be identified. The darked edges indicate the critical path in the instruction trace. Using a trace variant of this model, where we generate the trace of the program and feed it to a backward pass model as explained above, we identify the critical path and critical instructions. We attribute disjoint data regions as critical or non critical by measuring the criticality of these data regions whenever they get accessed on the critical path.

The Fields model also has a predictor model, where by maintaining history of the event latencies, instructions are predicted either to be critical or non-critical. This model need significant amount of hardware to maintain the history and prediction tables.

3. IDENTIFYING DISJOINT DATA REGIONS

Data structures that are either heap-allocated or stack-allocated are considered as ideal candidates for criticality analysis. Such data structures comprise more than 90% of total accesses during program execution. Most of the dynamic power consumption in a memory subsystem is due to the per-access power consumption. Hence it is profitable for

any power-aware optimization to be designed for such data accesses.

A dynamically allocated data structure can be viewed as a set of data addresses distributed across the memory space. These data blocks are not contiguous. The memory, which is required by a data structure, is allocated (using malloc or any other custom memory management method) and accessed at different points of execution. We profile SPEC benchmarks for memory allocation sites and memory operations such as loads and stores using context sensitive profiling. Identification of disjoint data regions is done in two stages.

First, we profile allocation sites and memory operations such as loads and stores. By profiling these allocation sites and collecting the access behavior of these allocated regions, we generate an undirected graph where each node is a static memory instruction (PC value of memory accessing instruction). This undirected graph is called Data Relationship Graph (DRG). Next, we use a clustering algorithm to create disjoint data regions. Each disjoint data region can be mapped to a data structure in a program or a logical data region of a data structure.

3.1 Context-Sensitive Profiling

The LLVM compiler provides data structure analysis for automatic pool allocation [10], where data regions corresponding to a particular data structure (for e.g., a linked list) are allocated contiguously in memory. Their analysis is a context-sensitive, flow-insensitive, alias analysis. It captures heap-allocated data structures. They perform alias analysis at each of the address references. By using the alias analysis results as well as heap allocation tables, they map malloc sites to the references. They do not consider program paths.

However, for some benchmarks, program paths also impact the allocation criteria significantly. Moreover, static program analysis is conservative and is an expensive technique both with respect to computing as well as memory footprint when used to perform a context, flow and path analysis. Static program analysis conservatively creates large data regions, thereby impeding the opportunities. Considering the above constraints, we suggest that profiling an application with representative inputs before performing this analysis could be a more acceptable and scalable technique. We follow this methodology in order to analyze data structures.

More recently, data partitioning techniques [3][16] have been used to map memory operations to data addresses. They distribute data into different cache partitions by forming a relationship between memory operations and individual data addresses. The authors do not consider context-sensitive profiling which is crucial for SPEC integer benchmarks, where dynamically allocated data significantly depends on the context as well as the path in which it was allocated. Moreover, it should be noted that considering individual data addresses and creating relationships could lead to finer granular regions, where applying criticality based techniques becomes infeasible.

An Example: In the *parser* benchmark, a table is used for most of the computation. This table consists of linked-lists to store words along with the attributes that are associated with these words. Each entry of the table stores a single sentence and the table is initialized for each sentence. The

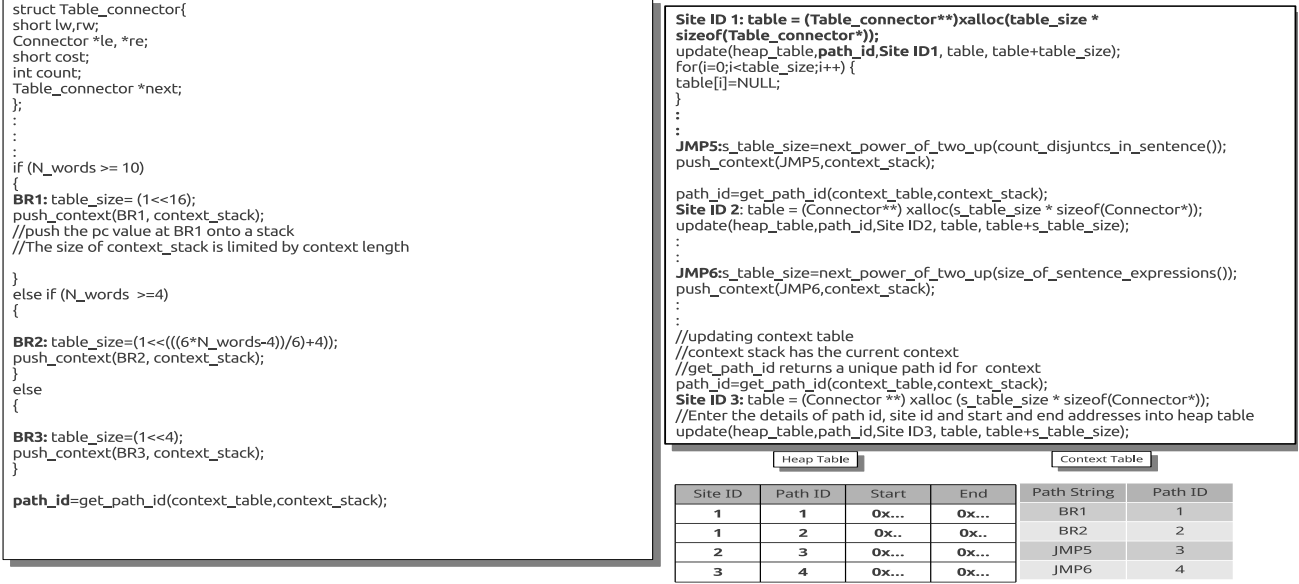


Figure 3: The example considers malloc sites whose characteristics change depending on the context. This is a code snippet extracted from the *parser* benchmark. The path to the malloc site alters the way in which the created data is accessed.

connector variables in each table entry are updated through other functions. Each of these variables points to a list of attributes that vary across sentences. During the dictionary lookup and parsing of the sentence, the *Table-connector* data structure is accessed frequently. The computation logic for each of the sentence varies depending on the grammatical notations.

Consider the example shown in figure 3 derived from the *parser* benchmark. The program is instrumented for collecting data addresses allocated by each *malloc* call, represented as a **SiteID**. In addition, we maintain an abstract heap table which comprises of the start address, malloc site¹, and the size of the data allocated. Whenever there is an access to a certain data address, a lookup on the heap table provides the malloc site id which is associated with the data address. Thus we create a mapping between *malloc site id* and *program counter value (PC)* of the memory operation. We use a program instrumentation and analysis tool, ATOM [5], for profiling. *Note that the free routines which deallocate memory seldom have an impact due to rare occurrences of address reuse.*

The function indicated in figure 3 allocates data for the *parser* benchmark. The context of the *malloc* (in this case *xalloc*) site, which can be obtained either by a string of procedure calls (calling-context) or a string of branches (path), varies during the execution of the program. Here we obtain the context of the malloc site through a string of branches. The instrumentation points for branches and memory allocation sites are indicated in bold letters. As seen in figure 3, **BR1**, **BR2**, **BR3**, **JMP5**, **JMP6** indicate branches. Here we consider only the last branch before the allocation happens, as the context.

¹An identification code based on program counter value of the malloc call.

The **first** table is a context table which is maintained to record the context. Each entry has a context and we associate a *path id* with it. If the context of the *malloc* site is obtained by a string of function calls (calling context), we can associate a *context id* to each of these strings. For example, in the Figure the path string **BR1** is associated with **Path ID 1**.

The **second** table presents the heap table. This table stores each of the memory allocation sites and the range of (beginning and ending) addresses of the blocks allocated in an instance of execution. During the execution of the program, we encounter a series of branches. As maintained in the heap table, the context at a particular execution point indicates the corresponding context id or path id, depending on the type of the context. A data region is represented by a set of ranges of data addresses at each malloc site along with a corresponding path id or a context id. The **SiteID 1** has two dynamic instances as shown in the heap table. These two instances are identified with the path IDs **1** and **2**, due to **BR1** and **BR2**. There are three *tables* in this program. We can identify these three tables from the three malloc sites.

The above example indicates that the data regions allocated at various points of execution are dependent on the input sentence in the *parser* benchmark. By profiling these allocation sites and collecting the access behavior of these regions, we generate groups or clusters of data regions that can be deployed for further analysis. This motivates us to consider context-sensitive profiling.

3.2 Clustering Algorithm for Identifying Disjoint Data Regions

The primary goal of the clustering algorithm is to create disjoint regions of data allocated during program execution and form clusters of static memory operations (PCs) access-

Algorithm 1 Clustering Algorithm

Input: an instrumented program P**Output:** set of disjoint data regions

```
1: C Context length initialized to 1
2: D Number of disjoint data regions
3: DRG is Data Relationship Graph
4: includeDRG( $O_i$ ): Procedure to include a node in DRG
5: createEdgeDRG( $O_p, O_q$ ): Procedure to create an edge
   between  $O_p, O_q$  in DRG
6: pushContext: Procedure to push a branch up to length
   C
7: ContextTable: A map between context and  $M_j$ 
8: HeapTable: A map between malloc site and data address
9:  $i = 1, 2, 3 \dots O_i$  indicates memory operations
10:  $j = 1, 2, 3 \dots M_j$  indicates malloc site instances
11: while D keeps changing do
12:   createDRG(P)
13:    $D = \text{createConnectedComponents}(\text{DRG})$ 
14:    $C = C + 1$ 
15:   if Limitation Guidelines Encountered then
16:     report warning
17:     exit
18:   end if
19: end while
20: Procedure CreateDRG(P)
21: for each dynamic Instruction of P do
22:   if Instruction is a Control instruction and jump to
   Address then
23:     if Considering path then
24:        $\text{currentContext} = \text{Push}(\text{Context}, \text{Address})$ 
25:     end if
26:   else if Instruction is a function call to an Address
   then
27:     if Considering call string then
28:        $\text{currentContext} = \text{Push}(\text{Context}, \text{Address})$ 
29:     end if
30:   else if Instruction is a malloc call  $M_p$  then
31:      $\text{ContextTableInsert}(\text{currentContext}, M_p)$ 
32:      $\text{HeapTableInsert}(\text{mallocSite}, \text{createdData})$ 
33:   else if Instruction is a memory operation  $O_p$  then
34:      $\text{effadd} = \text{EffectiveAddress}(\text{Instruction})$ 
35:      $\text{context of } O_p = \text{ContextTableQuery}$ 
36:     ( $\text{HeapTableQuery}(\text{effadd})$ )
37:     if  $O_p \in \text{DRG}$  then
38:       if  $\forall O_q$ : context of  $O_q$  = context of  $O_p$  then
39:          $\text{createEdgeDRG}(O_p, O_q)$ 
40:       end if
41:     else
42:        $\text{includeDRG}(O_p)$ 
43:     end if
44:   end if
45: end for
46: Procedure createConnectComponents()
47: N is set of nodes in DRG
48: C is set of connected components
49:  $C_i$  is connected component i, set of nodes
50: while  $N \neq \emptyset$  do
51:    $\text{node} \in N$ 
52:    $N_{\text{node}}$  is set of nodes reachable from node
53:    $\forall n \in N_{\text{node}}$ 
54:   if  $\text{node} \in C_i$  then
55:      $C_i = n \cup C_i$ ;  $N = \{N\} - C_i$ 
56:   else
57:     create new component  $C_{\text{new}}$ 
58:      $C_{\text{new}} = n \cup C_{\text{new}}$ ;  $N = \{N\} - C_{\text{new}}$ 
59:   end if
60: end while
```

ing these disjoint regions. While particular instance of a memory operation (a memory access) maps to a single *malloc* site, a static memory operation may map to more than one *malloc* site. In order to capture this complex relationship between malloc site instances and memory operation instances, we first introduce a **Data Relationship Graph (DRG)**. DRG is defined as follows:

Definition 3.1. A data region is a set of data addresses.

Definition 3.2. Let M_i be a set of data regions allocated from an allocation site S at different instances of execution. Suppose a data region $D \in M_i$. Let a and b be program counter values. Let M_a and M_b be the set of data addresses accessed by a and b respectively. We define the data region D as a common data region, if $M_a \cap M_b \subset D$.

Definition 3.3. A data relationship graph is defined as $G=(V,E)$, where V is the set of nodes corresponding to the static memory operations, i.e., program counter values of memory accessing instructions and E is the set of edges defined as follows: Between a node $a \in V$ and a node $b \in V$, \exists an edge $e \in E$: a and b have a common data region.

Algorithm 1 describes the high level clustering routine for generating disjoint data regions. An instrumented program, as described earlier, is used to generate these data regions. We consider a context in the form of a sequence of branches or string of function calls. We increment the length of the context until we reach a fix point with respect to the number of disjoint regions.¹ The inner loop of the algorithm creates the data relationship graph for a particular context length.

3.3 Algorithm for the Creation of Data Relationship Graph

Algorithm 1 describes the creation of a data relationship graph. During instrumentation, we instrument branch, jump and memory instructions and also malloc sites. The memory operations are indicated as O_i and malloc sites as M_i . The instrumentation routine inserts profiling functions before each of these primitives. Here we can recall that the heap table and context table discussed earlier in this section maintain mappings between data addresses, malloc site instances and current contexts.

The algorithm performs different operations for each type of Instruction as described below:

- **If the instruction is a branch**

During the execution of a program, we may encounter a conditional jump or a branch. If we consider a path as an indicator of the context, we push the encountered branch target address onto a context-stack. The size of the context-stack is pre-defined in the outer loop.

- **If the instruction is a function call**

If we encounter a function call, which is a jump instruction, we push the callee address onto a context-stack in case we consider call string as an indicator of the context.

- **If the instruction is a call to a malloc M_p :**

¹Here we increment the context length by one, but in general depending on the availability of the memory one can choose to increment it by more than one

If we encounter a call to the malloc routine, we maintain the data addresses created at that instance in the *HeapTable*. For the corresponding malloc instance, we consider the current context from the context stack and update the *ContextTable*.

- **If the instruction is a memory operation O_p :** During execution, each memory operation is instrumented to capture the effective data address. Effective data address is the actual data address obtained by adding the base address and the offset. With this effective data address we lookup the *HeapTable*. This yields us the malloc instance and this is used to obtain the context from the *ContextTable*. The query operations, *ContextTableQuery* and *HeapTableQuery* provide us with the context at which the corresponding data address was created.

While executing the memory operations, if the context identifier of any operation matches with the context identifier of the current operation, *it indicates that these two operations have accessed data created at the same context*. Therefore we insert an edge between these two operations. Each operation can have more than one context identifier, since it can access multiple data addresses during execution at different times.

3.4 Identifying Disjoint Data Regions using DRG

The DRG captures the mapping between memory operations and accessed data regions. The connected components of DRG are identified using a variant of Depth First Search (DFS) based algorithm as described in Algorithm 1. During DFS, we check if a node already belongs to a component. If it does, all the nodes reachable from that node associate with the same component identifier. During DFS, if a node is visited, we annotate the node with the component id and include the node into the set of nodes of the component id. We obtain a forest of disjoint components. Each component can now be considered as a **disjoint data region**.

3.5 Relevance of Context Length

One of the important parameters to the algorithm used for context sensitive profiling is the context length. Context length directly affects the number of disjoint regions that are formed using profiling and clustering. Evidently, as we increase the context length, the number of edges in the DRG increases and then decreases. Hence, identifying the right set of disjoint regions (considering context length) is crucial for the efficient operation of any optimization technique. In this section, we elaborate with the help of an example, how the number of disjoint regions varies with respect to the context length.

3.5.1 Example

Figure 4 shows an example of code for memory allocation that usually occurs in the benchmarks. The figure depicts four paths that lead to the actual allocation site S1. At S1, the program calls a custom allocator, which gets fixed values for the variables *p* and *q* in all the four paths. In this example, the path for the creation of data structure *p* is S1, S2, S4 where as that for the data structure *q* is S1, S3, S4. There are two functions *foo* and *bar* which access these data

structures. The function *bar* has two loads from *q*, whereas *foo* has four loads from *p*. There are three cases to consider:

When the path length is zero: In this case the graph has too many disjoint regions and it cannot capture the essence of data regions. This gives scope for a lot of interference across the operations since the data addresses share a memory block. For example, all the addresses of data structure *p* might be available in a few other memory blocks as well. This does not meaningfully capture the meaning of data regions. The resulting DRG is shown in figure 5(a).

When the path length is one: The heap table associated with site S4 captures all the data that is allocated at that site. In other words, at S4 both *p* and *q* are allocated. The DRG formed by using path length 1 is shown in the figure 5(b). All the memory operations in both the functions are connected to each other. This is because of the following reason. When we look up a data address and associate the context for that instance, we get S4 as the context identifier for all the accesses. The context S4 along with the site, identifies all the data allocated for both *p* and *q*. **When the path length is two:** By increasing the path length to 2, we are able to distinguish between the two allocations. The paths now considered are S2,S4 and S3,S4. The DRG in this case is shown in figure 5(c).

In figures 6(a) and 6(b), DRGs for Parser benchmark with path lengths 2 and 5 respectively are shown. In figures 6(c) and 6(d), we show DRG generated when call string is considered in *perl* benchmark. These DRGs indicate an increase in disjoint regions when either path length or call string length are increased.

As seen in the above three cases, the total number of disjoint components tends to increase with the context string length (length=0 is an exception). The reason for this increase can be attributed to the connectivity of the DRG. Increasing the context length beyond certain limit removes more and more edges from the DRG. However there is a maximum limit to the number of components that can be obtained by increasing the context length. Beyond this threshold limit, once the maximum number of components is attained, this number remains a constant notwithstanding any subsequent increase in the context length. In some benchmarks, a large number of paths to an allocation site tends to generate more components, as in *parser* and *vpr*. For other benchmarks, these paths do not guarantee improved contexts unless a very long path is considered. In such programs, we consider *call graph profiles* as the context for the *malloc* sites. For this purpose we experiment with both path and calling context, with different context lengths and we choose the one that generates more number of components. In some cases, for example *gzip*, memory allocation and usage happen in a loop body. So, unless we consider the entire program path, we cannot capture the disjoint regions. One may choose a trial-and-error method for various path lengths or calling contexts, but this does not guarantee reaching the fixed point of Algorithm 1 (for maximum number of meaningful data regions). A set of *Limitation Guidelines* to identify such workloads is to track heap table and disjoint data regions, and exit from Algorithm 1 are, if the size of the heap table, i.e. number of entries, is small and not changing with context length. The number of disjoint regions would also remain constant. This behavior is observed at small context lengths. The observation is usually true even at small context lengths, the algorithm generates significant disjoint

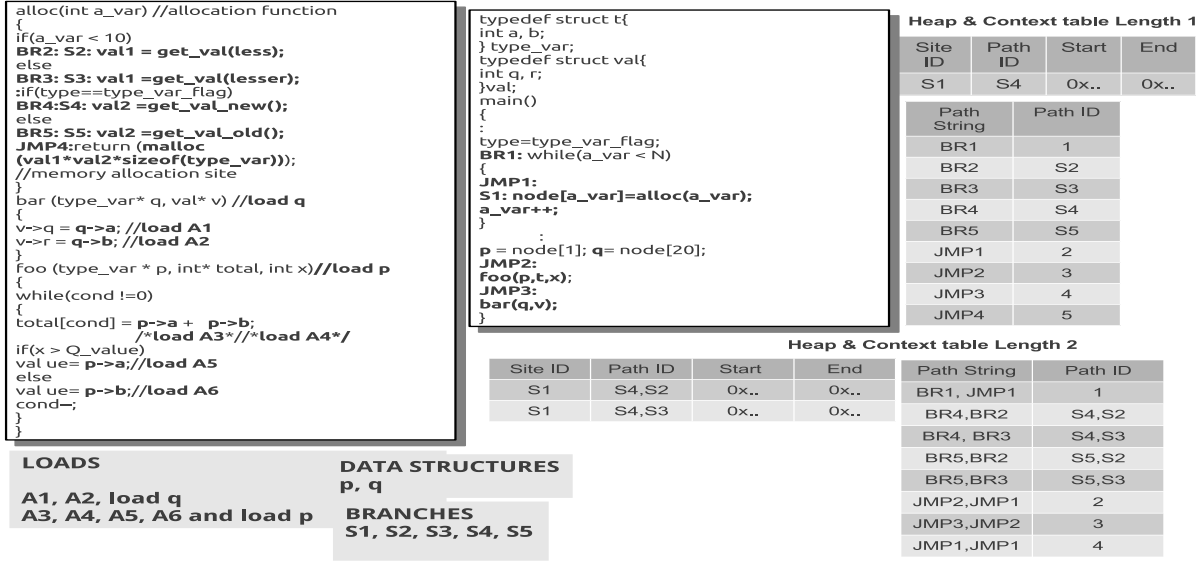


Figure 4: This example considers malloc sites whose characteristic would change depending on the context. The path to the malloc site changes the way the created data is accessed.

regions. In such cases, it is better to use the Full Drowsy configuration with simple policy as explained in section 5. As gzip falls under this class, we do not consider it for our experiments.

4. QUANTIFYING CRITICALITY OF DISJOINT DATA REGIONS

As discussed in an earlier section, dynamic data structures that do not impact the overall execution time can be used as candidates for designing power-aware optimizations. Due to complex pipelines, quantifying the criticality or the latency tolerance of these data regions is challenging. We quantify the criticality of the memory operations using Fields criticality model [6] as described in the sub section 2.1. Further, we use a pipeline simulator [19] to obtain the frequency of execution of each memory instruction and the total number of critical instances of the instruction.

We first define the criticality of a data region using equation 1.

$$C_{op} = T_{Critop} / Total \quad (1)$$

$$C_{data} = \sum_{vop} C_{op} \quad (2)$$

C_{op} : criticality of operations

C_{data} : criticality of data region

T_{Critop} : critical instances of operations

$Total$: total number of instances

During simulation, we gather the total number of times a particular instruction was commit critical, i.e., T_{Critop} . The section $Total$ is the total number of times that the instruction was executed. The summation over all the critical operations that access a particular data region is given as C_{data} . Higher the ratio of C_{op} , more critical the component

is.

Applying equation 1, we get graphs as shown in the figure 7. As we can observe from the figure 7, the data regions can be named as *critical* or *latency tolerant* adhering to a particular threshold value. For the benchmarks *vpr* and *eon* there are two components that are critical as compared to others. It is observed from the experiments that these components are not always hot components, i.e., components with a high frequency of access. *Thereby we can say that a hot component is not necessarily a critical component.* Other benchmarks also show a partition between critical and non-critical components. We also observe that the latency-tolerant data regions comprise of 30 to 50% of total memory accesses. Further, the footprint of these data regions varies across the benchmarks. It has been observed that footprints, criticality and frequency are not always coherent for these data regions. This is shown in the figure 8. This indicates that in order to design new optimizations, criticality of data regions becomes a vital parameter.

5. POWER AWARE OPTIMIZATION

In order to design an effective power-aware optimization, we first spilt the L1 cache into drowsy cache and normal cache. The drowsy cache has varying latency and its parameters are given in the Table 1.

We assume an additional bit to annotate each instruction to be either critical or non-critical. Each memory operation is categorized as critical or non-critical based on its criticality. During instruction fetch, the instruction decoder dynamically identifies it as either critical or non-critical. When critical memory instructions are missed data is fetched into the normal cache. When non-critical instructions are missed, data is fetched into the drowsy cache. Note that latency of the accesses to the drowsy cache is higher than that of the normal cache. This causes considerable delay in the pipeline.

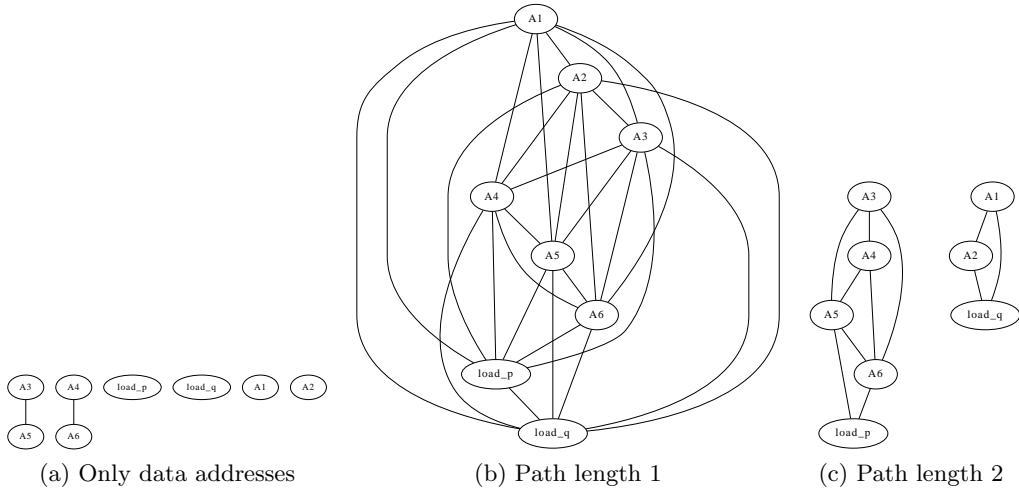


Figure 5: Three cases depicting different path lengths

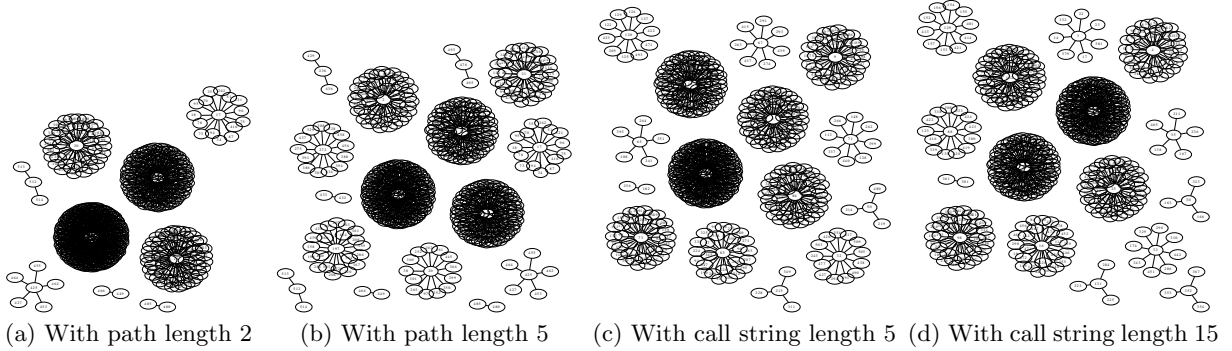


Figure 6: Relevance on context length in Parser and Perl benchmarks. We can observe that there is an increase in the number of disjoint regions with context length.

But since these accesses emanate from non-critical data regions or latency-tolerant data regions, this delay is absorbed by the pipeline, thereby minimizing the impact on performance.

An important limitation of our study is that the profile-driven technique that is employed here does not guarantee precision in finding disjoint data regions. In other words, parts of critical regions may not be critical at all and similarly, parts of latency-tolerant regions may be critical. Hence it may be necessary to check both the partitions in the case of a miss. This operation may not guarantee reduction in the power consumption since the per-access power may either remain the same or even slightly higher than the baseline. Addressing this concern we devise an access policy for a miss.

Case 1: When there is a miss due to a critical memory operation

The optimization performs a tag-search in the drowsy cache. If it is found, the memory operation is performed. However this additional penalty especially for critical memory operations can hamper overall performance. Hence our technique of power optimization simultaneously searches both drowsy cache as well as the L2 cache, causing a significant reduction in access latencies. This mitigates the above problem and minimizes performance penalty. In our power con-

sumption model, we have considered the additional power consumption due to simultaneous searches of both drowsy as well as the L2 cache.

Case 2: When there is a miss in the drowsy cache

In this case, a tag look up in the normal cache is performed. If we encounter a miss, an L2 cache access is performed. In this case, the access latency is higher than the access latency observed in case 1. However, since the access emanates from non critical memory operations, the additional latency is absorbed.

5.1 Experimentation

For experimentation, we consider benchmarks from *SPECINT2k*: *parser*, *perl*, *vpr*, *twolf*, *eon*, pointer intensive benchmarks like *bc* and *olden* benchmarks: *em3d*, *bh*. It is not possible to experiment with other benchmarks because they have less number of disjoint regions. Moreover due to custom memory allocation, especially *gcc*, it is difficult to instrument the memory allocation points. In spite of a custom memory allocator in *parser*, the allocation happens through a single function which could be instrumented. For our experiments, we modified the *Hotleakage simulator* [19] to add extra cache and measured the total leakage savings using the power model provided in the simulator. *Hotleakage* implements various low leakage controls like drowsy cache,

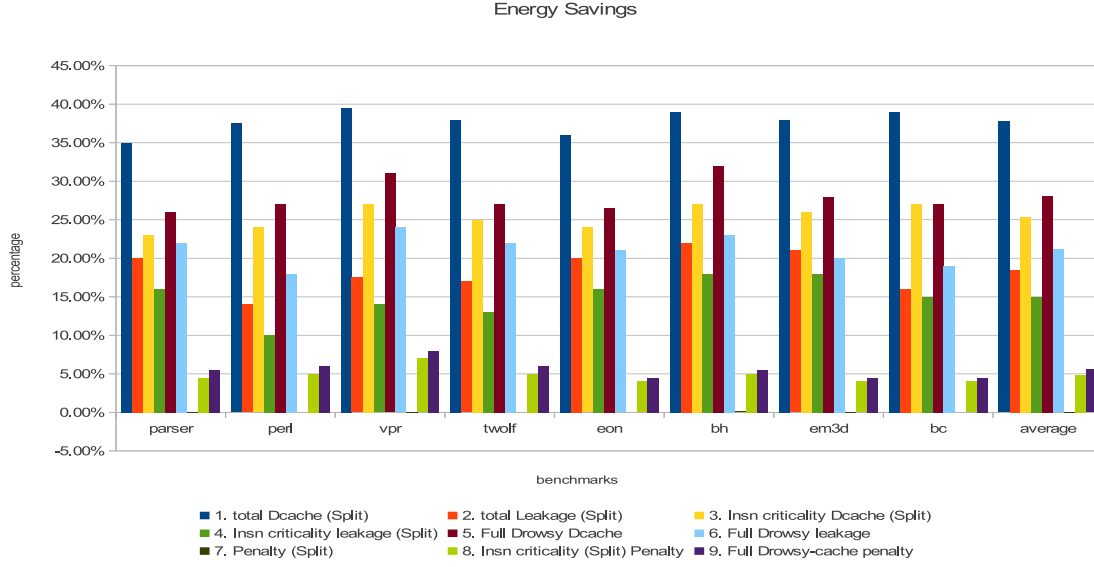


Figure 9: The total power savings compared against a baseline processor with no leakage control. The graph shows total savings in the data cache, both L1 and L2. The penalty due to this optimization is also shown

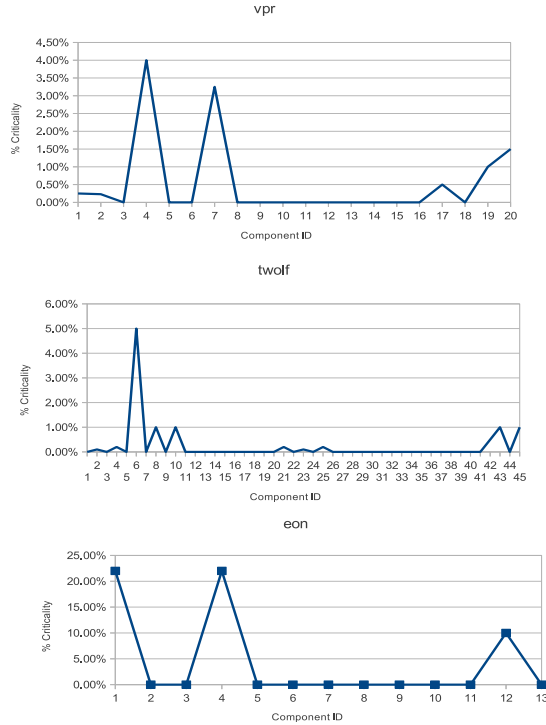


Figure 7: Criticality of vpr, eon and twolf using equations 1 are shown here.

$gatedV_{dd}$ etc. The L1 data cache is split into two halves: a normal mode cache, and another with a leakage control mode. We consider the drowsy cache as the leakage control cache.

Pipeline	O-O-O
Issue width	4
Branch predictor	Combined predictor
L1 - Split cache	16KB, 16KB, 2 cycles
L1 variable voltages	0.8v-0.6v
L1 variable latencies	3-5 cycles
L2 cache	512KB 4 way, 6 cycles

Table 1: Baseline Processor Configuration and Split Data Cache Configuration

Next, we measure the power consumption in two parts. We first measure the total dynamic power consumption. As the L1 data cache is split into two halves, the dynamic power per access changes for each access. The splitting of the cache could alter the behavior of the L2 level data cache, thereby changing the power consumption in the L2 cache. We then measure the total leakage consumption. Usually cache leakage is measured from the tag array and data array parameters. Existing power models like *cacti* [9] compute per-cycle leakage of a cache. We measure the total leakage in the cache as well as total savings in the drowsy cache. The total power savings are in both forms of power consumption: dynamic as well as leakage power.

5.2 Results

Figure 9 presents power savings, both dynamic and leakage compared to a baseline processor for the configuration given in Table 1. The graph shows savings as well as penal-

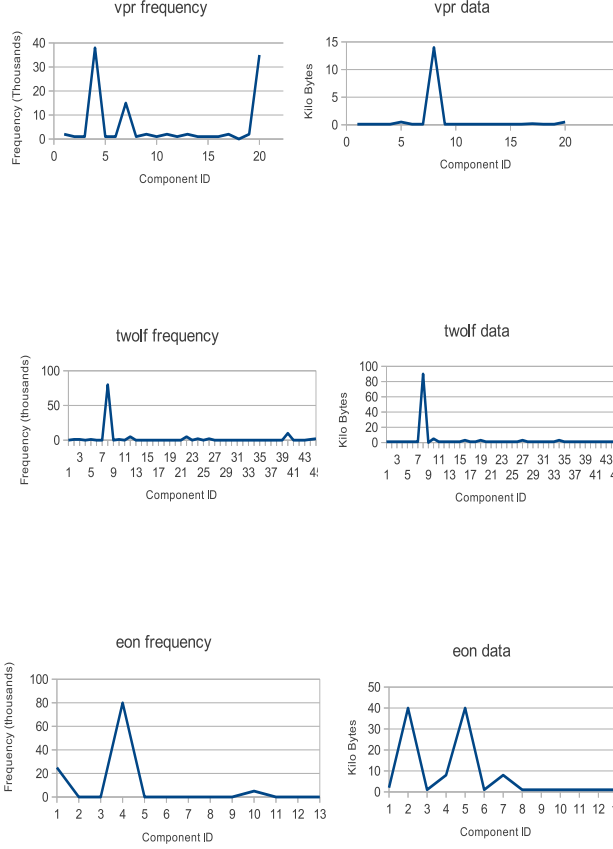


Figure 8: Data regions with footprints and frequencies. The X-axis shows different components ids and Y-axis shows the sizes in terms of bytes and frequency, i.e., number of accesses of the corresponding data regions

ties for 3 different configurations. In the legend, 1, 2 and 7 indicate our savings and penalty due to our technique. *total Dcache* and *total Leakagesplit* indicates our scheme and total data cache dynamic as well as leakage energy savings. The configurations 3, 4 and 8 are based on per-instruction criticality based control policy when we employ Field’s model. The configurations 5, 6 and 9 are due to *Full drowsy* configuration. The graph demonstrates an overall energy savings of 35% to 38% and a leakage energy savings ranging from 13% to 20%. Since L1 cache is split, the per access power consumption is almost halved. By employing drowsy cache, we achieve significant leakage savings. We compare our savings against two policies and configurations. The *Full drowsy* configuration shown in figure 9 indicates employing a 32KB drowsy cache with *Simple policy* [7]. In such a policy the complete drowsy cache is turned off periodically. In our policy we employ a 2000 cycle window. Considering 0.02um technology and 3 cycle latency for bringing a drowsy cache line to active state, we observe an average of 21% savings in the leakage power with 27% of total sav-

ings. These savings are comparable to our technique. The *Simple policy* can reduce the leakage significantly as most of the cache lines are placed in drowsy state. *But the performance penalty of this policy is significantly higher than our technique.* The other configuration is to control the split cache with per-instruction criticality based control policy. This is shown in In such a policy, critical path predictor model as explained in sub-section 2.1 is used. The data fetch policy is similar to our policy, but we use the critical path model to predict whether an instruction is critical or not. In this configuration we observe an average of 25% total savings and 15% of leakage savings. This configuration is moderately less efficient than using the *Full drowsy cache* configuration. But due to critical instruction prediction, the performance penalty is lower than that of Full drowsy cache, i.e. 4% compared to 6% respectively. Leakage savings of our technique are higher and in some cases comparable to *Full drowsy cache*, which is 32KB drowsy cache where in our technique we employ 16KB drowsy cache, but our total savings are significantly higher than both the other configurations. The main reason is very low performance penalty, which is less than 1% even with a split data cache.

Note that as the cache is split, the leakage energy consumption savings are only attributed to the split data cache. The seventh bar indicates the performance penalty due to employing a split cache. This penalty (though very insignificant), is due to the increased latency of the drowsy accesses and to some extent, due to increased L2 level accesses. In some cases like *parser*, there is a slight performance improvement. This is due to a reduction in the number of L2 level accesses and reduction in L1 cache pollution. We do not employ drowsy cache at L2 because it is a shared resource. Also, a minor increase in the access time in L2 can impact the performance significantly.

5.3 Sensitivity analysis

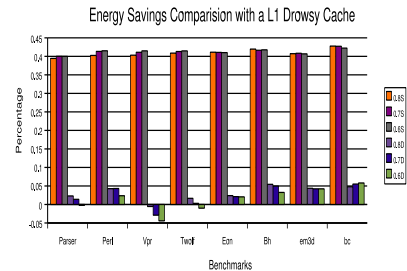


Figure 10: This graph shows the sensitivity of the optimization. The comparison is between a complete L1 drowsy cache and split cache. The power consumption is compared against a baseline processor with no leakage control

Since critical data regions have a higher impact on the performance as compared to non critical regions, we perform a sensitivity analysis of the split cache at various voltage levels. As indicated in the experimental results, the split cache contributes to significant power savings. Now, we vary the voltage levels of L1 data cache. The configurations 0.8S, 0.7S, 0.6S indicate the voltage levels of the drowsy cache pertaining to the split L1 cache. Similarly 0.8D, 0.7D, 0.6D indicate the voltage levels of the total L1 drowsy cache.

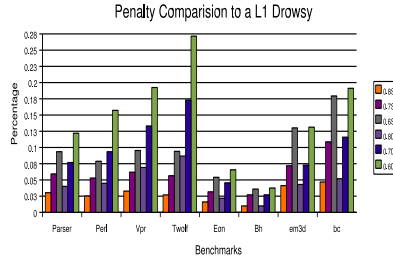


Figure 11: This graph shows the performance penalty between L1 drowsy cache and split cache.

The variation of the voltage levels causes variation in leakage power per cycle and also varies per-access cycle time. A limitation of Cacti [9] is that it can estimate leakage power as well cycle time at a fixed voltage level. Since it does not consider different voltage levels, we assume that the latencies would be 3, 4 and 5 cycles for voltage levels 0.8v, 0.7v and 0.6v respectively.

The results demonstrate that split cache outperforms the total drowsy cache. Even though the per-cycle leakage power in the total drowsy cache L1 is lower than that of the split cache, the extra leakage savings are dominated by an increase in performance penalty. The total number of execution cycles increase due to an increase in the access times. Thus the total power consumption increases with increase in access latencies. Due to increased number of transitions in the total drowsy cache, extra penalty (which is 3 cycles from low to high and 300 cycles from high to low) increases the power consumption. We use the same penalty values for the split data cache. This is further illustrated in figures 10 and 11. As shown in figure 10, *vpr* benchmark consumes more power as compared to the baseline processor. Figure 11 shows the increase in the penalties for both the caches by varying the supply voltage. As we can observe, the total L1 drowsy cache quickly starts showing significant penalties (as for *twolf*), whereas the split cache absorbs the extra latencies due to the split drowsy cache and the penalties do not grow at a similar rate. For benchmarks *bh*, *em3d* and *bc*, we do not notice a remarkable difference as compared to the split cache. This is because the data regions that are fetched into the drowsy cache are tolerant or non-critical. We also observe an extra leakage savings of around 5% as compared to the baseline processor when we vary the supply voltage. These savings can be attributed to the reduced cycles or improved performance of split cache.

6. RELATED WORK

Traditional compiler algorithms statically identify critical memory operations based on the data dependency graph[13]. These techniques associate predefined latencies with the edges of these dependency graphs.

Srinivasan et.al. [17] discussed run-time metrics and other dynamic dependency-based characteristics to identify critical memory operations. The authors demonstrate that these metrics perform well in identifying the critical memory instructions.

As already cited, Fields et.al [6] developed a more rigorous pipeline based critical path model that not only identifies critical instructions but can also be used to predict

the criticality of in-flight instructions. Their model extends the work of Srinivasan et.al. [17] by taking into consideration the resource dependencies and classifies instructions as either fetch critical, execution critical or commit critical.

More recently, newer metrics [20] have been developed to improve the notion of criticality. These distinguished studies signify that using critical instruction predictors may lead to an increased likelihood of non-critical instructions being predicted as critical. These metrics are used to identify critical instructions from program structure. Fields model provides an architecture dependent trace based critical path model which can capture dynamic events.

[15] performs a limit-study on critical loads which are not vital for execution time. In [18] the authors introduce a new metric for quantifying the criticality of instructions. This architecture-dependent metric, tautness, is used to design more efficient critical instruction predictors.

In a nutshell, most of the optimizations designed based on the criticality models worked at the instruction level[2][1]. They conclude that a data cache block gets accessed non-uniformly by critical and non-critical instructions from the time it is available until it gets replaced. This causes inefficiency in the energy savings. All this research points to the need to quantify the criticality of a data region from the program point of view instead of using critical instruction predictors to dynamically tune data cache voltage levels. We go further and analyze chunks of memory regions for their latency tolerance as well.

7. CONCLUSIONS

In this paper we addressed the problem of energy consumption in data caches of modern embedded systems. We used a profile-based technique focusing on context-as well as path-sensitive analysis. The accuracy of the technique depends on the length of the context and the length of the path. We modeled the criticality behavior of data regions at a coarser level than at a single memory operation level.

We used the criticality analysis to drive a power-aware optimization technique which allocated a split data cache, operating at normal and drowsy modes, to critical and non-critical data regions respectively. This technique saves around 30% of total power and 20% of leakage power in the data cache without any significant performance penalty.

8. REFERENCES

- [1] Jaume Abella and Antonio Gonzalez. Power Efficient Data Cache Designs. In *Proceedings of the 21st International Conference on Computer Design (ICCD)*, page 8, 2003.
- [2] Rajeev Balasubramonian and Viji Srinivasan. Hot and Cold: Using Criticality in the Design of Caches. In *Workshop on Power Aware Computing Systems (PACS)*, pages 180–195, 2003.
- [3] Michael Chu, Rajiv Ravindran, and Scott Mahlke. Data access partitioning for fine-grain parallelism on multicore architectures. In *40th International Symposium on Microarchitecture (MICRO)*, pages 369–380, 2007.
- [4] J. H. Edmondson and P. I. Rubinfeld. Internal organization of the alpha 21164, a 300mhz 64-bit quad-issue coms risc microprocessor. *Digital Technical Journal*, 2:119–135, 1995.
- [5] Alan Eustace and Amitabh Srivastava. Atom: a system for building customized program analysis tools. In *ACM SIGPLAN Programming Language Design and Implementation*, pages 528–539, 1994.
- [6] Brian Fields and Ras Bodik. Focusing processor policies via critical-path prediction. In *Proceedings of the 28th annual international symposium on Computer architecture (ISCA)*, pages 74–85, 2001.

- [7] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy Caches: Simple techniques for reducing leakage power. In *29th International Symposium on Computer Architecture (ISCA)*, pages 148–157, 2002.
- [8] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems Cache, DRAM and Disk*. Morgan Kaufmann, 2008.
- [9] Normal P. Jouppi and Premkishore Shivakumar. Cacti 3.0: An integrated cache timing, power and area model. 2001.
- [10] Chriss Lattner and Vikram Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *Programming Language Design and Implementation (PLDI)*, pages 129–142, 2003.
- [11] Yan Meng, T. Sherwood, and R. Kastner. Exploring the limits of leakage power reduction in caches. *ACM Transactions on Architecture and Code Optimization*, 2:221–246, 2005.
- [12] J. Montenaro. A 160mhz 32bit 0.5w cmos risc microprocessor. In *The International Conference on Solid-State Circuits*, 1996.
- [13] Stephen Muchnik. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1998.
- [14] M. Powell, S. H. Yang, B. Falsafi, K. Roy, and N. Vijaykumar. Reducing leakage in a high-performance deep-submicron instruction cache. *IEEE Transactions on Very Large Scale Integration Systems*, 9:77–89, 2001.
- [15] Ryan Rakvic, Deepak Limaye, and John P. Shen. Non-vital loads. In *High Performance Computer Architecture (HPCA)*, pages 165–174, 2002.
- [16] Rajiv Ravindran and Scott Mahlke Michael Chu. Compiler-managed partitioned data caches for low power. In *Conference on Languages, compilers, and tools for embedded systems (LCTES)*, pages 237–247, 2007.
- [17] Srikant Srinivasan and Roy Ju. Locality vs criticality. In *International Symposium on Computer Architecture (ISCA)*, pages 132–143, 2001.
- [18] ES Tune and B Calder. Quantifying instruction criticality. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, page 104.
- [19] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan. Hotleakage: A temperature-aware model of subthreshold and gate leakage for architects. In *Technical Report CS-2003-05, Virginia University*, 2003.
- [20] Craig Zilles and Pierre Salverda. A criticality analysis of clustering in superscalar processors. In *International Symposium on Microarchitecture (MICRO)*, pages 55–66, 2005.