Exploiting Critical Data Regions to Reduce Data Cache Energy Consumption

Ananda Vardhan  
Indian Institute of Science  
kanandv@gmail.com

Y N Srikant  
Indian Institute of Science  
srikant@csa.iisc.ernet.in

ABSTRACT
In this paper we propose an energy aware optimization that exploits latency tolerance of data regions in programs. We propose techniques to identify data regions and rate their criticality using a dynamic critical path model. We compare latency tolerance of data regions to existing characteristics like access frequency and size of data regions. We leverage previously proposed drowsy cache lines to design an optimization that can reduce energy consumption in a data cache. We target this optimization to a simplified single-core with a private cache and single-threaded system which can be part of any type of a multi-core processor. We compare this technique to existing optimizations that use drowsy caches. We experimentally show that this technique can yield total power savings close to 38% and leakage power savings of 20% in the data cache when compared to a baseline configuration without any significant performance penalty.

Categories and Subject Descriptors
D.3.4 [Programming Languages]: Processors—compilers, optimization

General Terms
Algorithm

Keywords
Profiling, Critical Data Regions, Energy Consumption

1. INTRODUCTION
Modern embedded systems like smartphones and tablets provide a platform to run computation and memory intensive applications. The underlying architecture and software layers support the application to execute efficiently both in terms of power as well as performance. As the applications become data intensive, its imperative to reduce power on the data fetch and storage mechanisms. The current multi-core embedded processors employ a multi-level cache hierarchy with last level sharing across the cores. The front line caches like L1 (or in some cases L0) operate on data private to a thread or a single task. The design of such caches is extremely critical to mitigate overall cache power consumption. One of the significant cache designs in terms of power is a drowsy cache. The drowsy cache has the capability to maintain and preserve the data between different power states and also requires few cycles to switch between these power states [9] [11].

In this paper we present a novel software-centric mechanism to utilize the drowsy cache with support from program profiling and cache hints in the instructions. The majority of the traditional data related optimizations focus mainly on size of the data, temporal and spatial locality and frequency of accesses to a data region. We introduce a new program-level characteristic that captures the latency tolerance of data regions. This characteristic brings into perspective the criticality or the impact of different data regions to the program performance which fundamentally differs from the other traditionally used characteristics.

The subsequent sections of the paper are organized as follows. Section 2 briefly describes the concepts underlying our approach and the significance of a data region in the context of power savings. Section 3 elaborates our approach of identifying critical data regions. In section 4 our optimization technique is described along with the experimental results comparing various configurations. Section 5 describes related work and we conclude the paper in section 6.

2. BACKGROUND
Most of the compiler-directed power optimization techniques proposed in the literature operate at the instruction level. While per-instruction based techniques perform effectively in driving fine-grained optimizations, they do not capture the coarser level semantics of programs. Programmers implement meaningful procedures and data structures in their applications. Such coarse-grained information can be utilized in optimizations and can gain significant power savings.

Consider an example drawn from the vpr [19] benchmark shown in figure 1(a). 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 instruc-
2.1 Attributing Criticality to a Data Region

Memory access latencies and branch-load dependencies are primary reasons for causing a bottleneck in the system. For instance, a memory access to a data region may be a hit or go through a series of misses. This latency may vary for 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 such events define the criticality of a data region. We identify these critical data regions through profiling. This helps us to drive power-aware optimizations at the data region level.

In a nutshell the critical path model of an out-of-order processor proposed by [7] can be summarized as follows. 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 $E$ and between $E$ 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 by a $CE$ dependency edge. Each of these edges is attributed with a latency, indicating the time taken for the architectural event. The commit of the last instruction ends this dependency graph.

By traversing backwards on the longest path, we can identify the critical path of the program. The memory instructions on the critical path are the critical memory operations. We run different programs on this model and identify critical memory operations. The regions of data accessed by these critical memory operations are defined as critical data regions.

In modern application more than 90% of the total accesses are to either heap-allocated or stack-allocated data regions. Hence it is profitable for any power-aware optimization to be designed for such data accesses. A dynamically allocated data structure is a set of data addresses distributed across the memory space in a non-contiguous manner. 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. The initial task is to identify data regions whose accesses do not originate from the same set of memory instructions. We define
them as disjoint data regions (DDRs). In our technique we utilize this disjoint nature of accesses during profiling and apply criticality model. During execution, we exploit this to reduce power state variations in the cache.

3. IDENTIFYING DISJOINT DATA REGIONS

Identification of disjoint data regions (DDR) 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 DDR 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 [12], 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.

Data partitioning techniques [3][15] 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 [19] 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 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 2 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 site\(^1\), 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 2 allocates data for the parser benchmark. The context of the malloc (in this case malloc) 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. In figure 2, BR1, BR2, BR3, JMP5, JMP6 indicate branches. Here we consider only the last branch before the allocation happens, as the context.

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 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 Data Relationship Graph

Here we describe an algorithm to create disjoint regions of data allocated during program execution and form clusters of static memory operations (PCs) accessing these disjoint regions. While a particular instance of a memory opera-
Figure 2: 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.

A data region is a set of data addresses.

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 $E = \{(a, b) | a, b \in V, \text{and } M_u \cap M_v \neq \emptyset\}$

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 fixed point with respect to the number of disjoint regions. The inner loop of the algorithm creates the data relationship graph for a particular context length. It is to be noted that differentiating a call-site can be achieved with increase in the context length. The context length is limited for a program run, though it may impede system resources. Therefore this algorithm guarantees that we reach a fixed point. We explain an exit criteria that can help the execution of algorithm practical without stressing the system resources in 3.4. Such an exit criteria can easily be tuned for different benchmarks.

### 3.3 Algorithm for the Creation of DRG

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 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 $M_i$**

  If we encounter a call to the malloc routine, we maintain the data addresses created at that instance in the...
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_i
8: HeapTable: A map between malloc site and data address
9: i = 1, 2, 3...O_i indicates memory operations
10: j = 1, 2, 3...M_j indicates malloc site instances
11: while D keeps changing do
12: createDRG(P)
13: D = createConnectedComponents(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: currentContext = Push(Context, Address)
24: end if
25: else if Instruction is a function call to an Address then
26: if Considering call string then
27: currentContext = Push(Context, Address)
28: end if
29: else if Instruction is a malloc call M_p then
30: ContextTableInsert(currentContext, M_p)
31: HeapTableInsert(mallocSite, createdData)
32: else if Instruction is a memory operation O_p then
33: effadd = EffectiveAddress(Instruction)
34: if context of O_p = ContextTableQuery(O_p) then
35: createEdgeDRG(O_p, O_q)
36: (HeapTableQuery(effadd))
37: if O_p in DRG then
38: if ∀O_q: context of O_q = context of O_p then
39: createEdgeDRG(O_p, O_q)
40: end if
41: else
42: includeDRG(O_p)
43: end if
44: end if
45: end for
46: Procedure createConnectedComponents()
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 ≠ ∅ do
51: node ∈ N
52: N_{node} is set of nodes reachable from node
53: ∀n ∈ N_{node}
54: if node ∈ C_i then
55: C_i = n ∪ C_i; N = {N} - C_i
56: else
57: create new component C_new
58: C_new = n ∪ C_new; N = {N} - C_new
59: end if
60: end while

- 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. 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.

### 3.4 Relevance of Context Length

One of the important parameters to the algorithm and also used for context sensitive profiling is the context length. Context length directly affects the number of DDRs 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 DDRs (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 DDRs varies with respect to the context length.

An example of code for memory allocation that usually occurs in the benchmarks is shown in figure 3. 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
Figure 3: 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.

data regions. The resulting DRG is shown in figure 4(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 4(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 4(c).

In figures 5(a) and 5(b), DRGs for Parser benchmark with path lengths 2 and 5 respectively are shown. In figures 5(c) and 5(d), we show DRG generated when call string is considered in perl benchmark. These DRGs indicate an increase in DDRs when either path length or call string length are increased. It is to be noted that there is a significant increase in the DDRs from path length 2 to 5, whereas not much if we consider call string. Essentially, the call sites can be differentiated with a small path length in perl. Also, the incremental change is small after certain path-length or call-string length.

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 a higher 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 are proposed to identify such benchmarks and exit gracefully. The aim is to track heap table and disjoint data regions, and exit from Algorithm 1 at appropriate scenarios. One such scenario is: if the size of the heap table, i.e. number of entries, is small and not changing with context length. Another scenario is: if the number of disjoint regions remaining constant at small caller context lengths while increase linearly with path lengths. As gzip fails under this class, we do not consider it for our experiments.

### 3.5 Quantifying Criticality of DDR

As discussed earlier, 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 [7] as described in the subsection 2.1. Further, we use a
pipeline simulator [20] 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.

\[
C_{op} = \frac{T_{Crit_{op}}}{Total}
\]

\[
C_{data} = \sum_{\forall op} C_{op}
\]

- \(C_{op}\): criticality of operations
- \(C_{data}\): criticality of data region
- \(T_{Crit_{op}}\): critical instances of operations
- \(Total\): total number of instances

During profiling, we gather the total number of times a particular instruction was commit critical, i.e., \(T_{Crit_{op}}\). 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 \(C_{data}\). The criticality of the data region accessed by these operations is higher with the high value of \(C_{op}\).

Applying equation 1, we get criticality data regions across the benchmarks vpr, eon and twolf as shown in figure 6. Here we compare this weighted average of criticality to footprint and hotness of data regions. The graph depicts data footprint, shown as X-axis along with criticality and frequency of accesses as Y-axes. Each point indicates a data region’s size, criticality and frequency of accesses. Size of the data is measured as number of kilo bytes, criticality by applying equation 1 and frequency of accesses as percentage of accesses to this region measured during simulation. As we can observe from figure 6, the data regions can be marked as critical or latency tolerant adhering to a particular threshold value. For example the data regions above 5% of criticality are critical, rest of the regions are latency tolerant. It is to be noted that across these benchmarks more than 80% of the data regions can be classified as non-critical. The threshold value is a design parameter that can be tuned according to the requirements.

Also to be noted is that frequency of accesses, data size and criticality of the data regions do not correlate with each other. Except for the point P1, \(8, 22\%, 80\%\), most of the data regions dominantly exhibit either one or two of these characteristics. For example the point P2, \(40, 1\%, 80\%\), depicts a data region which is of significant size with large number of accesses. But the criticality is very low. Thereby we can say that a hot component or a large size data region is not necessarily a critical region. Other benchmarks also show a notable partitioning between critical and non-critical data regions. We also observe that the latency-tolerant data regions comprise of 30 to 50% of total memory accesses.

This indicates that in order to design new optimizations, criticality of data regions becomes a vital parameter.
4. POWER AWARE OPTIMIZATION

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

We assume an additional bit to annotate each instruction to be either critical or non-critical. Once we identify the critical and non-critical data regions, we use DRG to remap these data regions to memory operations. We now annotate all the memory operations accessing critical data regions as critical. Note that some of these operations might not have been considered critical earlier during profiling. The intuition behind this method is, by using the disjoint nature of these data regions we are minimizing variations in power state of cache lines.

Here we describe our cache policy split data cache with data region policy: Split data cache with Data Region based Policy (SDRP), of handling critical and non-critical data accesses. 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. 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 a policy for misses.

**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 rest of memory operation is executed by fetching the data. 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 consumption 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 due to non-critical memory operation**

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.

We compare SDRP with a baseline configuration where L1 cache of size 32KB is not split and is a normal cache. A 32KB drowsy cache is employed in Full Drowsy cache Simple Policy (FDSP). This policy was proposed in [9]. In FDSP a complete drowsy cache is turned off periodically. In FDSP we employ a 2000 cycle window. Considering 0.02um technology and 3 cycle latency for bringing a drowsy cache line to active state. As we discussed earlier, we assume a simplified model to validate our technique. In most of the current multi-core architectures, a private L1 cache is employed which caters to the single application. We assume such a cache which is small in size and associativity. We presume a large, scaled up cache architecture can also be employed with SDRP by rerunning the whole methodology.

The other configuration is to control the split cache with instruction criticality based control policy: Split data cache with Instruction Criticality Policy SICP. In SICP, critical path predictor model as explained in sub-section 2.1 is used at run-time. 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. We fetch and place the data into drowsy and normal modes based on the instruction criticality.

4.1 Experimentation

For experimentation, we consider benchmarks from SPEC-INT2k [19] such as parser, perl, vpr, twolf, eon, pointer intensive benchmarks like bc and olden benchmarks: em3d, bb. 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.
We use separate training and test data sets for profiling and for experimentation. For our experiments, we modified the Hotleakage simulator [20] 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, gated\textit{Vdd} 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.

<table>
<thead>
<tr>
<th>Pipeline</th>
<th>O-O-O</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue width</td>
<td>4</td>
</tr>
<tr>
<td>L1 - Split cache</td>
<td>16KB, 16KB, 2 cycles</td>
</tr>
<tr>
<td>L1 variable voltages</td>
<td>0.8-0.6v</td>
</tr>
<tr>
<td>L1 variable latencies</td>
<td>3-5 cycles</td>
</tr>
<tr>
<td>L2 cache</td>
<td>512KB 4 way, 6 cycles</td>
</tr>
</tbody>
</table>

Table 1: 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 \textit{cacti} [10] 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.

4.2 Results

The dynamic and leakage power savings compared to a baseline processor for the configuration given in Table 1 are shown in the figure 7.

The graph depicts that with SDRP an overall power savings of 35% to 38% and a leakage energy savings ranging from 13% to 20% can be achieved. 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 FDSP and SICP. These savings are comparable to our technique. The FDSP 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. In FDSP we observe an average of 21% savings in the leakage power with 27% of total savings.

In SICP we observe an average of 25% total savings and 15% of leakage savings. This configuration is moderately less energy-efficient than using the FDSP configuration. But due to critical instruction prediction, the performance penalty is lower than that of FDSP, which is 4% compared to FDSP which 6%.

Leakage savings of SDRP are higher and in some cases comparable to FDSP, 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 performance penalty (though very insignificant) of SDRP, 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 overall performance because the memory access is already on the critical path.
5. RELATED WORK

Traditional compiler algorithms statically identify critical memory operations based on the weighted data dependency graph[13]. Srinivasan et al. [16] discussed runtime 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. [7] 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. [16] by taking into consideration the resource dependencies and classifies instructions as either fetch critical, execution critical or commit critical. More recently, newer metrics [21] 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.

[14] 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.

In [17] authors propose smart caches where sizes and associativity of caches can be varied. They provide additional index bits in the structure of the cache to choose number of sets there by varying the associativity of the cache. In [6] authors propose an extension to drowsy caches where additional bits control word level voltage control of cache lines. This can enable finer control of voltage supply at word level. In [8] a phase adaptive drowsy cache is proposed. The authors show that by considering the lifetime of a cache line at the various runtime, they either place it in an active cache or a drowsy cache. In [4], the authors propose a L0 cache, which is small in size and can determine whether a hit or miss going to occur prior to L1 cache. L0 and L1 caches are at similar level. They propose a flow cache where in a cache line is maintained for longer duration by sharing it between L0 and L1. They also propose a hit cache, which filters hits to L1 cache and highly reused cache lines are held in L0.

6. 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 the heap as well as path-sensitive analysis to identify disjoint data regions. We analyze the criticality behavior of these data regions at a coarser level than at a single memory operation level.

We used this analysis to drive an energy aware optimization technique. We used a split data cache, operating at normal and drowsy modes, to critical and non-critical data regions separately. We proposed a cache policy to minimize performance penalty. This technique saves around 35% of total power and 20% of leakage power in the data cache without any significant performance penalty on a subset of SPEC benchmarks.

7. REFERENCES
