Intra-Task Dynamic Cache Reconfiguration*

Hadi Hajimiri, Prabhat Mishra
Department of Computer & Information Science & Engineering
University of Florida, Gainesville, Florida, USA
{hadi, prabhat}@cise.ufl.edu

ABSTRACT

Optimization techniques are widely used in embedded systems design to improve overall area, performance and energy requirements. Dynamic cache reconfiguration (DCR) is very effective to reduce energy consumption of cache subsystems. Finding the right reconfiguration points in a task and selecting appropriate cache configurations for each phase are the primary challenges in phase-based DCR. In this paper, we present a novel intra-task dynamic cache reconfiguration technique using a detailed cache model, and tune a highly-configurable cache on a per-phase basis compared to tuning once per application. Experimental results demonstrate that our intra-task DCR can achieve up to 27% (12% on average) and 19% (7% on average) energy savings for instruction and data caches, respectively, without introducing any performance penalty.

1. INTRODUCTION

Energy conservation has been a primary optimization objective in designing embedded systems. Several studies have shown that memory hierarchy accounts for as much as 50% of the total energy consumption in many embedded systems [1]. Unlike desktop-based systems, embedded systems are designed to run a specific set of well-defined applications (tasks). Moreover, different applications require highly diverse cache configurations for optimal energy consumption in the memory hierarchy. Thus it is possible to have a cache architecture that is tuned for those applications to have both increased performance as well as lower energy consumption. Traditional dynamic cache reconfiguration (**DCR**) techniques reduce cache energy consumption by tuning the cache to applications need during runtime on task-by-task basis. For each task only one cache configuration is assigned to the task, and it is not changed during the task execution. These techniques are referred as inter-task DCR. Studies have shown that *inter-task* DCR can achieve significant energy savings [2].

Due to task-level granularity, inter-task DCR loses the energy savings opportunity that can be achieved by increasing the reconfiguration granularity. A modern processor executes billions of instructions per second and a program's behavior can change many times during that period. The behavior of some programs changes drastically, switching between periods of high and low performance, yet system design and optimization typically focus on average system behavior. Instead of assuming average behavior, it is highly beneficial to model and optimize phase-based program behavior. *Intra-task* tuning techniques tweak system parameters for each application phase of execution. Parameters are varied during execution of an application, as opposed to keeping fixed as in an application-based (inter-task) tuning methodology. Furthermore, inter-task DCR is not beneficial in a single-task environment (or in a multi-task environment where execution time of one task is dominant) because the cache configuration is determined on a per task basis. Since many small-size embedded-mobile applications are based on a single-task model, inter-task DCR cannot provide the best possible energy savings for such systems.

These limitations lead to the idea of intra-task DCR where a given task is partitioned into several phases, and different cache configurations are assigned for each phase. There have been limited attempts [3] [4] for developing an intra-task DCR but they provide no systematic methodology of selecting the best program locations where DCR can be applied (phase detection). Furthermore, these approaches either perform exhaustive exploration (can be infeasible in many scenarios) or select suboptimal cache configurations. In this paper, we propose an intra-task DCR approach based on static analysis of a target application achieving significant improvements in energy consumption. It also can be applied to a single-task environment since it reconfigures the cache within each task. We propose a phase detection technique that fully exploits drastic changes in program behavior and finds boundaries between phases of high and low performance. In addition, we propose a dynamic programming based cache assignment algorithm that finds the optimal cache solution and reduces the time complexity of design space exploration.

The rest of the paper is organized as follows. Section 2 provides an overview of related research activities. Basic notations, cache and energy model are described in Section 3. Our proposed intra-task DCR methodology is presented in Section 4. Experimental results are discussed in Section 5. Finally, Section 06 concludes the paper.

2. RELATED WORK

DCR has been extensively studied in several works [5] [6] [7]. The problem is to determine the best cache configuration for a particular application. Most such methods configure cache size, line size, and associativity for only a single level of cache. Existing techniques can be classified into dynamic and static analysis. By dynamic analysis different cache configurations are evaluated on-line (i.e., during runtime) to find the best configuration. However, it introduces significant performance/energy overhead which may not be feasible in many embedded systems with real-time constraints. During static analysis, variety of cache options can be explored thoroughly and the best cache configuration is chosen for each application [5]. Regardless of the tuning method, the predetermined best cache configuration can be stored in a look-up table or encoded into specialized instructions [5]. The reconfigurable cache architecture proposed by Zhang et al. [8] determines the best cache parameters by using Pareto-optimal points trading off energy consumption and performance. Chen and Zou [9] introduced a novel reconfiguration management algorithm to efficiently search the large space of possible cache configurations for the optimal one.

Peng and Sun [3] introduced a phase-based self-tuning algorithm, which can automatically manage the reconfigurable cache on a per-phase basis. Their method used dynamic profiling of applications and limited to only four choices of cache configurations for L1 cache. Gordon-Ross et al. [4] proposed an intra-task DCR where each task is partitioned into fixed-length timeslots. It [4] shows limited improvements in the energy reduction (only 3% on average). Moreover, it provides no systematic methodology for selecting the best program locations where DCR can be profitable (programs divided into equal phases). These techniques solved the cache assignment either by performing exhaustive exploration (can be infeasible in many scenarios) or selecting suboptimal cache configurations. Our methodology outperforms existing approaches using novel phase detection and cache selection algorithms.

* This work was partially supported by NSF grant CCF-0903430 and SRC grant 2009-HJ-1979.

3. BACKGROUND AND MOTIVATION

3.1 Inter-task versus Intra-task DCR

Fig. 1 illustrates how energy consumption can be reduced by using inter-task (application-based) cache reconfiguration in a simple system supporting three tasks. In application-based cache tuning, dynamic cache reconfiguration happens when a task starts its execution or it resumes from an interrupt (either by preemption or when execution of another task completes). Fig. 1 (a) depicts a traditional system and Fig. 1 (b) depicts a system with a reconfigurable cache. For the ease of illustration let's assume cache size is the only reconfigurable parameter of cache (associativity and line size are ignored). In this example, Task1 starts its execution at time P1. Task2 and Task3 start at P2 and P3, respectively. In a traditional approach, the system always executes using a 4096-byte cache. We call this cache as the **base cache** throughout the paper. This cache is the best possible cache configuration (in terms of energy consumption) for this set of tasks. In Fig. 1(b), Task1, Task2, and Task3 execute using 1024-byte cache starting at P1, 8192-byte cache starting at P2, and 4096-byte cache starting at P3, respectively.

Although inter-task DCR provides significant energy savings compared to using only the *base cache*, it has several practical limitations as discussed in Section 1. Hence it may be more efficient in terms of energy consumption to utilize different cache configurations in different phases of a task. Fig. 1 (c) depicts intra-task DCR where reconfiguration can be done per phase basis. A task may need larger cache size for only a small phase of execution. Increasing the cache size for this phase would boost performance and decrease both cache misses and energy consumption. However, in some of the program phases the application may need a lower cache size thus the cache size can be reduced without loss of performance to produce savings in energy consumption. In these cases, intra-task DCR is able to fulfill cache needs of application perfectly while minimizing the energy consumption.

3.2 Energy Model

In this subsection, we describe the energy model for the reconfigurable cache. We assume that DCR is available in the target system. Specifically, we have a highly configurable cache architecture, with reconfigurable parameters including cache size, line size and associativity, which can be tuned to m different configurations C = {c$_1$, c$_2$, c$_3$, ... , c$_m$}. Cache energy consumption consists of dynamic energy E_{cache}^{dyn} and static energy E_{cache}^{stat} [10]: $E_{cache} = E_{cache}^{dyn} + E_{cache}^{stat}$. The number of cache accesses *num_accesses*, cache misses *num_misses* and clock cycles *CC* are obtained from simulation using SimpleScalar [11] for any given task and cache configuration. Let E_{access} and E_{miss} denote the energy consumed per cache access and miss, respectively. Therefore, we have:

$$E_{cache}^{dyn} = num\_accesses \,.\, E_{access} + num\_misses \,.\, E_{miss}$$

$$E_{cache}^{stat} = P_{cache}^{stat}.\, CC.\, t_{cycle}$$

Where P_{cache}^{stat} is the static power consumption of cache. We collect E_{access} and P_{cache}^{stat} from CACTI [12] for all cache configurations and adopt E_{miss} and other numbers for other parameters from [8].

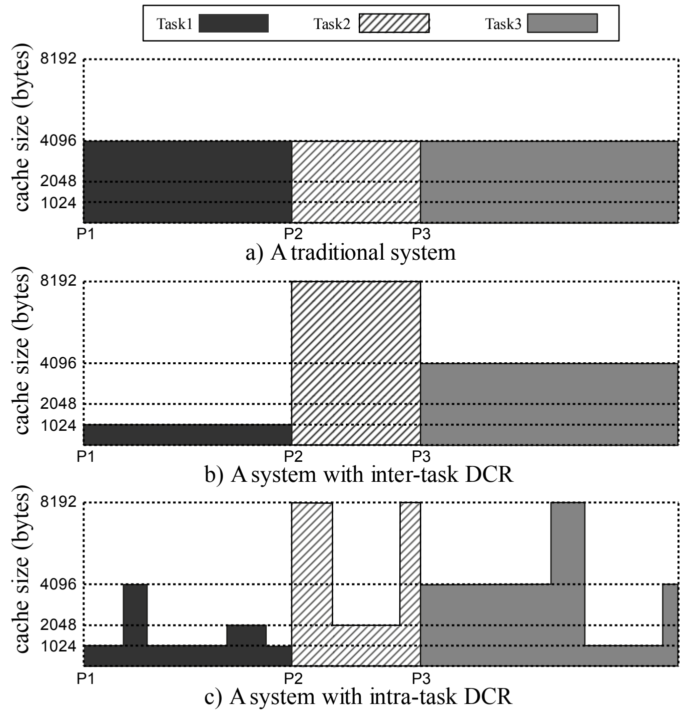

Fig. 1: DCR for a system with three tasks

4. INTRA-TASK DCR

We define a phase as a set of intervals (or time slices) within a program's execution that has similar behavior. The key observation for discovering phases is that the cache behavior of a program changes greatly during execution. We can find this phase behavior and classify it by examining the number of cache misses in each interval. We collect this information through static profiling of the program. We begin the analysis of phases with an illustrative example of the time-varying behavior of *epic-encode* from MediaBench [13]. To characterize the behavior of this program, we have simulated its execution using a 1024-byte cache with one-way associativity and 32-byte line size. Fig. 2 shows the cache behavior of the program, measured in terms of cache miss statistics using two cache configurations (C_1 and C_2).

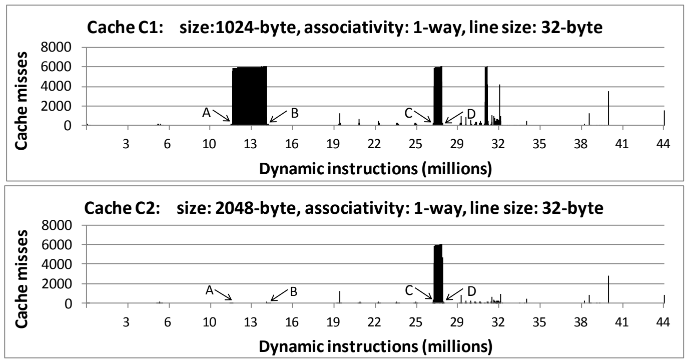

Fig. 2: Instruction cache miss for *epic-encode* benchmark

Each point on the graph represents the frequency of instruction cache misses taken over 100,000 instructions of execution (an *interval*). Two important aspects can be observed from this graph. First, average behavior does not sufficiently characterize a program's behavior in all phases of execution. For example, in *epic-encode* the number of instruction cache misses varies by several orders of magnitude. Second, the program can exhibit stable behavior for millions of instructions and then suddenly change. As a result, *epic-encode*'s behavior alternates greatly between phases. These two aspects, imply that significant energy savings can be achieved by accurately reconfiguring the cache to satisfy long-term execution behavior.

For *epic-encode* benchmark, we first need to find cache miss statistics in order to find potential reconfiguration points. Note that the least energy cache configuration for *epic-encode* benchmark is a 2048-byte cache with associativity of 1 and line size of 32 (cache C$_2$ in Fig. 2) chosen by inter-task cache configuration method. From Fig. 2, it can be observed that up to point *A* (around the dynamic instruction 12 million) miss rates are nearly the same for both caches C$_1$ and C$_2$. Starting from *A* to point *B* the miss rates are greatly different. We find

A and *B* as potential reconfiguration points for this example. For the ease of illustration, let's assume only configurations C_1 and C_2 are available. Since C_2 is larger than C_1, C_2 is beneficial for performance and dynamic energy but detrimental for leakage energy compared to C_1. To reduce energy consumption we can run the program using configuration C_1 up to *A* then reconfigure the cache and use configuration C_2 from *A* to *B* and then again reconfigure the cache back to C_1.

In this paper, with the aim of energy optimization, we present a method to enable automatic partitioning of a program's execution into a set of phases that will quantify the changing behavior over time. The goal is that after finding phases, each phase would use a specific cache configuration suitable for that phase to reduce energy consumption without performance loss. We define the following terms that we will use in the rest of the paper:

- An *interval* is a section of continuous execution, a time slice, within a program. We chose intervals of equal length, as measured by the number of instructions executed during program execution. In this paper we choose 100,000 instructions as the length of intervals[2].
- A *phase* is a set of consecutive intervals within a program's execution that have similar and stable behavior. Boundaries of each phase are determined by reconfiguration points. For example, Fig. 2 has three phases; *start of execution* to *A*, *A* to *B*, and *B* to *end of execution*.
- A *potential reconfiguration point* is a point in the execution of a program at which a noticeable and sudden change in program behavior happens going from one phase to another phase. For example, *A* and *B* are potential reconfiguration points.
- The *profitability* of a reconfiguration point is a metric that shows how well a reconfiguration point can distinguish two different phases of a program. We use this metric for building a spectrum of energy savings while the number of reconfigurations is limited. We describe this metric in Section 4.1.

Fig. 3 shows an overview of our intra-task DCR approach. Our approach has two major steps (represented by ovals): phase detection and cache assignment. During a program's lifetime it can execute millions or billions of instructions each of which can be a reconfiguration point. The challenge is to choose a small number of profitable points from these millions of points. Moreover, the reconfiguration overhead is not constant and is different based on the point where the reconfiguration happens and can be found by actually reconfiguring and flushing the cache at that point during simulation. Thus finding the best set of reconfiguration points that is capable of separating program phases and guarantees energy savings is a difficult problem. We instead, find a set of potential reconfiguration points. Next, we choose if reconfiguring the cache is feasible at each point and if yes to what cache configuration. In order to find the potential reconfiguration points we compare frequency of misses in each interval. In addition, the energy consumption of a phase using a particular cache can vary depending on whether the previous phase has executed using the same cache (reconfiguration is needed if the cache is different). These are the main challenges we address in our approach. In the remainder of this section we explain each of these steps in detail.

4.1 Phase Detection

A phase is a set of consecutive intervals determined by two reconfiguration points (starting interval and ending interval). Finding best possible set of potential reconfiguration points is the objective of this step. First, we generate cache miss statistics (using simulation) for all possible cache configurations and find frequency of misses in each interval. Next, we compute the difference of frequency of misses (for all possible pairs of cache configurations) to discover the potential reconfiguration points. The statistics for data and instruction caches are gathered separately. Miss data is then used to calculate the frequency of cache misses in each interval of 100,000 dynamic instructions.

[2] We chose the interval length to be small (100,000 instructions) to increase granularity of cache miss information.

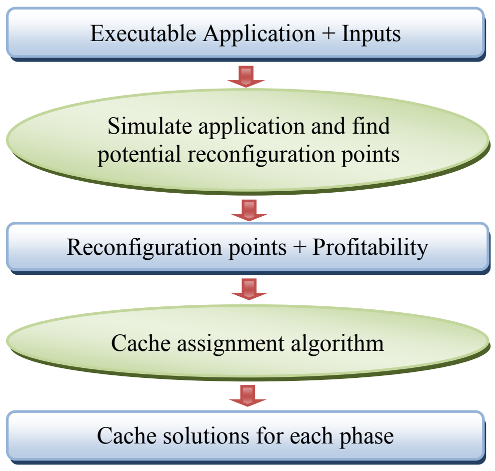
Executable Application + Inputs
Simulate application and find potential reconfiguration points
Reconfiguration points + Profitability
Cache assignment algorithm
Cache solutions for each phase

Fig. 3: Overview of our intra-task DCR

We use the example in Fig. 2 to explain our phase detection algorithm. Fig. 2 shows the miss frequency of the application *epic-encode* using cache configurations C_1 and C_2. Every point in the chart represents the frequency of misses (in thousands) in an interval. For example, the frequency of misses at *A*, for cache configuration C_1 is 6000 while it is nearly zero using C_2. We compare frequency of misses for cache configurations C_1 and C_2 to discover potential reconfiguration points. We include the edge of the regions in which the magnitude of the difference is greater than the threshold (we choose threshold to be 1000 in this example) into the set of reconfiguration points. For example, the magnitude of the difference in intervals *A* to *B* is greater than 1000 so we take the edge points of this region (the first instruction in *A* and the last instruction in *B*) as potential reconfiguration points. Considering the edge points of *A* to *B* as our potential reconfiguration points will create the phases, P_{start} (*start of execution* to *A*), P_2 (*A* to *B*), and so on.

Analyzing a miss frequency by itself may not necessarily lead to finding reasonable reconfiguration points since changes in cache miss frequencies may happen for all caches due to the cache behavior of a program. For instance, in Fig. 2, at the interval *C* to *D* we observe a significant change in cache misses. However this change is nearly the same in both cases. Since both cache configurations have the same behavior these points are not good candidates for a reconfiguration point. We find reconfiguration points as phase boundaries so that we would reconfigure the cache and use a different cache configuration. If all of the caches exhibit the same behavior this means program can continue with the same cache it was executing before. For this reason we compare miss frequencies of different cache configurations instead of scrutinizing frequencies solely.

Algorithm 1 outlines our heuristic to find a set of reconfiguration points. We compare frequency of misses (*f1* and *f2*) for all pairs of cache configurations using a dynamic threshold to find potential reconfiguration points with their profitability. Every element in arrays *f1* and *f2* keeps the frequency of misses in an interval (for example $f1_k$ represents the number of misses in the k^{th} interval). We treat the frequency of misses as a pattern (a time-varying quantity). So basically we use (compare) the intersection of two patterns and exploit their differences to discover potential reconfiguration points. The array *Profitability* (in Algorithm1) is determined by the magnitude of the differences between two frequencies of misses and is used as a metric that represents effectiveness of a point in discovering boundaries of

```
Algorithm 1: Finding potential reconfiguration points
Input: Cache miss statistics for each cache configuration
Output: List of potential cache configuration points
Begin
    th = the starting threshold;
    n = number of intervals;
    li = an empty list to store potential reconfiguration points;
    for i=0 to 17 do
        for j=i to 17 do
            f1 = array of frequency of misses for cache $C_i$ for all intervals
            f2 = array of frequency of misses for cache $C_j$ for all intervals
            Profitability = differences of f1 and f2;
            for k=0 to n do
                if ($Profitability_k > th$) then
                    add the pair ($po_k$ , $Profitability_k$) to li;
            end for
        end for
    end for
    return li;
end
```

phases (for instance *A* and *B* in Fig. 2). We include a point, po_k, (starting point of the k^{th} interval) into the set of reconfiguration points only if the profitability of this interval, $Profitability_k$, is greater than the threshold. Note that these points are potential reconfiguration points and reconfiguration may not actually happen at these points.

Finding potential reconfiguration points may seem simple; however, there are several challenges in finding points that are beneficial in practice. First, the absolute number of misses can be significantly different for each cache configuration not because of changes in interactions between cache and program but due to the difference in cache/line size. For example total number of misses for a 1024-byte cache may be several orders of magnitude greater than number of misses for an 8192-byte cache. This makes comparison of frequencies unfair and biased towards the smaller cache. Second, since the length of intervals is relatively small, reconfiguration points may be chosen very close to each other (with a distance of one interval). Reconfiguring cache after such a short period of time does not seem reasonable due to the reconfiguration overhead. Third, when a program is not in a stable phase it may have a chaotic cache behavior; many ups and downs will be present in the miss frequencies. Thus comparison of miss patterns is prone to fluctuation from glitches in frequency of misses that will result in numerous unnecessary reconfiguration points that are ineffective in separating program phases.

To cope with these challenges we carried out several improvements to the algorithm. First, in order to perform an unbiased comparison between frequencies we normalize them before comparing so that the sum of number of misses over all program execution is a constant value for all caches. This way we ignore the absolute number of misses while we keep the information about the behavior of cache. Second, we limit the minimum distance between two reconfiguration points to be at least 500,000 instructions. This will be the minimum length of a phase that is reasonable to reconfigure the cache. This length is mainly determined by the reconfiguration overhead. Therefore, if there are multiple reconfiguration points that are close to each other (in the minimum distance range) we choose the one with the highest profitability. Third, we ignore the short-term fluctuations in frequency of misses (cache behavior) when comparing cache miss patterns.

4.2 Cache Assignment Algorithm

Finding the best cache configuration for each program phase using potential reconfiguration points from the previous step is the goal of this step. We call this problem as cache assignment since we are assigning a cache for each of the program phases. In this step we employ a dynamic programming based algorithm for optimal cache assignment which significantly reduces the time complexity of cache selection. Solution for instruction cache can affect the energy of data cache by increasing/decreasing the execution time of a phase. This will change the static energy of the chosen cache. However, according to simulation results reconfiguration overhead for data/instruction caches mostly consist of dynamic energy hence we can solve the cache selection problem for data and instruction cache independently.

After finding the potential reconfiguration points we need to find the exact energy consumption and the execution time for each phase for all possible cache configurations. We use simulation to obtain cache statistics (time and energy) for the possible 18 cache configurations[3]. We modified SimpleScalar [11] to reconfigure and flush the cache at the reconfiguration points. Reconfiguration overhead for all phases and for instruction and data caches are computed separately. For phase P_i we find energy/time for each of the two cases. Case1 is when the chosen cache for the phase is the same as the selected cache for phase P_{i-1}. In this case the cache is not flushed and will keep the data (no reconfiguration). Case2 is when the chosen cache for this phase is different from the selected cache for phase P_{i-1}, where reconfiguration takes place and the cache should be flushed. Therefore simulation starts this phase with an empty cache (accounts for reconfiguration overhead).

Table 1: Notations

Symbol	Representing
E_{P_i,C_j}	Energy consumption of phase P_i using cache C_j while P_{i-1} also used C_j (no reconfiguration)
$E^f_{P_i,C_j}$	Energy consumption of phase *Pi* using cache C_j starting with a flushed cache (includes reconfiguration overhead), i.e., P_{i-1} does not use C_j.
$S_{i,C_j} = \{C_{P_{start}}, \dots, C_{P_i} = C_j\}$	The most profitable solution for the set of consecutive phases P_{start} to P_i assigning $C_{P_{start}}$ to $P_{start}, \dots, C_{P_{i-1}}$ to P_{i-1} and C_j to P_i, i.e., the last phase uses C_j
$E_{S_{i,C_j}}$	Energy consumption of solution S_{i,C_j} (for phases P_{start} to P_i)

We present a recursive approach to find the optimal solution for each of the phases. Table 1 includes a set of notations we use in the rest of this section. In our recursive approach, in the general case, when there are *m* phases and *n* available cache configurations, we can find the best cache configuration for the phases P_{start} to P_i using the following formula:

$$\begin{cases} E_{S_{i,C_1}} = min(E_{S_{i-1,C_1}} + E_{P_i,C_1}, \dots, E_{S_{i-1,C_n}} + E^f_{P_i,C_1}) \\ \vdots \\ E_{S_{i,C_n}} = min(E_{S_{i-1,C_1}} + E^f_{P_i,C_n}, \dots, E_{S_{i-1,C_n}} + E_{P_i,C_n}) \end{cases} \quad \textbf{(Eq. 1)}$$

with the initial state: $\begin{cases} E_{S_{start,C_1}} = E^f_{P_{start},C_1} \\ \vdots \\ E_{S_{start,C_n}} = E^f_{P_{start},C_n} \end{cases}$

We observe that storing all possible cache combinations is not needed (for finding the optimal solution) in each iteration. We only need to keep the one with the lowest energy consumption from all possible solutions ending with a particular cache. All other combinations ending with the same cache can be discarded.

[3] In our work we use a 4KB L1 cache architecture proposed in [16]. Since the reconfiguration of associativity is achieved by way concatenation, 1KB L1 cache can only be direct-mapped as three of the banks are shut down. For the same reason, 2KB cache can only be configured to direct-mapped or 2-way associativity. Therefore, there are 18 (=3+6+9) configuration candidates for L1.

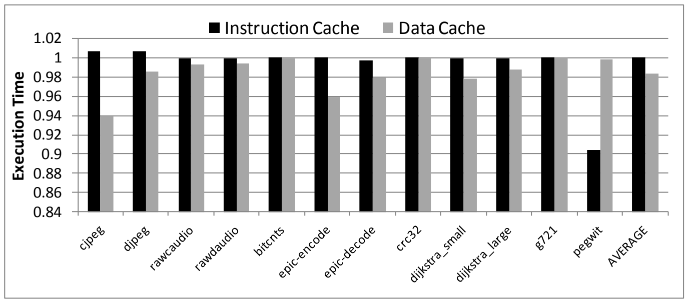

Fig. 5: Execution time normalized to the least-energy cache configuration found by inter-task DCR

Algorithm 2 shows an iterative implementation of our cache assignment approach. In each iteration, we evaluate C_1 for the phase P_i considering all of the solutions found from P_{start} to P_{i-1} in the last iteration. By comparing these solutions we find the best solution for phases P_{start} to P_i ending with cache C_1. For each of the possible cache configurations we find the minimal energy option ending with that cache (chosen for P_i) and keep it for next iteration discarding the other solutions. Similar computation is done for caches C_2 to C_n. For our final least energy cache solution we use:

$$E_{S_{final}} = min(E_{S_{end,C_1}}, \dots, E_{S_{end,C_n}}) \qquad \text{(Eq. 2)}$$

In the general case, suppose m is the number of phases (number of potential reconfiguration points) and n is the number of possible cache configurations (18 in our case). Having n different cache options for each phase we can count the total number of possible solutions for cache assignment:

$$n^m = \underbrace{n \times n \times \dots \times n}_{m\ times}$$

Therefore, finding the optimal cache assignment in a brute force manner (trying all possible solutions), takes the time complexity of $O(n^m)$. In our approach, $E_{S_{end,C_1}}$, …, $E_{S_{end,C_n}}$ are computed in m iterations starting with the initial state. Computing (Eq. 1) in each iteration needs $n \times n$ comparisons. Therefore, in the recursive approach we reduce the time complexity of finding the optimal solution to $O(mn^2)$.

It should be noted that since the length of each phase is relatively long we can assume that reconfiguration (flushing) at the beginning of phase P_i only has an impact on the energy/time of phase P_i and will fade out for the next phase. In other words, if no reconfiguration occurs at the beginning of phase P_{i+1} we can assume no reconfiguration has happened prior to phase P_{i+1} in estimating time/energy of this phase. Therefore, energy/time of a cache for phase P_{i+1} is only dependent on the cache selected for the previous phase, P_i. Reconfiguration should be done when the selected caches for two consecutive phases are different and reconfiguration overhead should be accounted.

5. EXPERIMENS

5.1 Experimental Setup

In order to quantify effectiveness of our approach, we examined *cjpeg*, *djpeg*, *epic* (*encode* and *decode*), *adpcm* (*rawcaudio* and *rawdaudio*), *pegwit*, *g.721* (*encode*) benchmarks from the MediaBench [13] and *dijkstra*, *crc32*, *bitcnt* from MiBench [14] compiled for the PISA [11] target architecture. All applications were executed with the default input sets provided with the benchmarks suites.

We utilized the configurable cache architecture developed by Zhang et al [8] with a four-bank cache of base size 4 KB, which offers sizes of 1 KB, 2 KB, and 4 KB, line sizes ranging from 16 bytes to 64 bytes, and associativity of 1-way, 2-way, and 4-way. The reconfigurable cache was reported to have negligible performance and energy overhead compared to non-configurable cache [8]. For comparison purposes, we used the *base cache* configuration set to be a 4 KB, 2-way set associative cache with a 32-byte line size, a common configuration that meets the average needs of the studied benchmarks [8].

```
Algorithm 2: Finding Cache Assignment
Input: Cache energy and time for all caches for each phase
        /* m = total number of phases */
Output: Cache configuration for each phase
Begin
  S = a 2-dimentional list to store the best caches found;
  for phase j=0 to m do
    for cache i=0 to 17 do
      Find the best cache assignment from phase $P_{start}$ up to $P_j$ for
       cache configuration i using (Eq. 1);
      Update $S_i$;
    end for
  end for
  return the minimal energy solution in S using (Eq. 2);
end
```

To obtain cache hit and miss statistics, we modified the SimpleScalar toolset [11]. The modified version was able to dump dynamic instructions of cache misses as well as energy and time statistics for each program phase for both cases of starting with a flushed cache or a cache keeping previous data. The reconfiguration overhead (energy/time) is computed by flushing the cache at the reconfiguration points. Note that flushing the data cache requires all dirty blocks to be written back to main memory whereas flushing the instruction cache will just reset the valid bits for all cache blocks. The reconfiguration overhead also includes memory access latency/energy of bringing the data/instructions (that were previously in the cache) back to the cache. We applied the same energy model used in [8], which calculates both dynamic and static energy consumption, memory latency, CPU stall energy, and main memory fetch energy.

5.2 Energy versus Performance

Fig. 4 shows energy consumption using our intra-task DCR for all benchmarks normalized to the energy consumption for the least-energy cache found by inter-task DCR. Note that inter-task DCR is shown to achieve up to 53% cache subsystem energy savings in studies [10]. Our intra-task DCR approach achieves up to 27% (12% on average) energy savings compared to inter-task DCR for instruction cache. Energy savings of up to 19% (7% on average) is gained for data cache subsystem using our approach. It should be noticed that only nominal modifications are needed to make a working system using inter-task DCR to benefit from our intra-task DCR.

Fig. 5 demonstrates the execution time for all benchmarks normalized to the execution time for the least-energy cache configuration found by inter-task DCR. It is important to note that intra-task DCR introduces nearly no performance loss compared to the conventional inter-

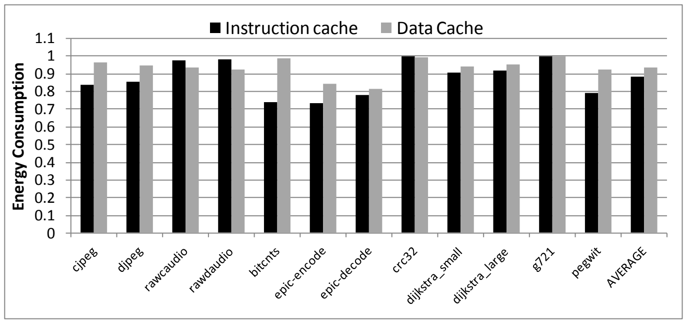

Fig. 4: Energy consumption normalized to the best cache configuration found by inter-task DCR

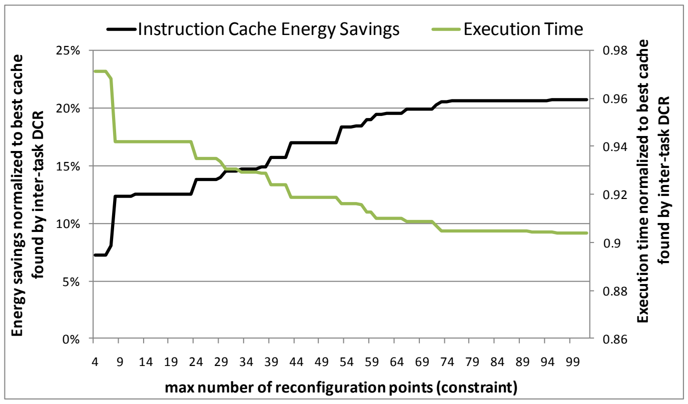

Fig. 6: Energy savings and execution time spectrum for instruction cache using *pegwit* benchmark

task DCR. Less than 1% performance loss observed using intra-task DCR for instruction cache. However, in some cases it actually achieves better performance (10% in the case of *pegwit* benchmark). Interestingly, incorporating intra-task DCR for data cache gains performance by 2% on average (up to 6% using *cjpeg* benchmark). We observed that intra-task DCR does not achieve energy savings compared to inter-task DCR in some applications. By further analysis it turned out that these applications have nearly the same cache behavior (either stable or chaotic) throughout their entire execution. This means that these applications cannot be separated into phases based on their cache behavior. In other words, they consist of only one phase. As expected, our intra-task DCR chooses only one cache configuration in this case which is same as inter-task DCR.

5.3 Overhead versus Energy Savings

The reconfiguration overhead is higher for data cache compared to instruction cache since by flushing the data cache all dirty blocks are needed to be written back to main memory whereas flushing the instruction cache only resets the valid bits for all cache blocks. Depending on the behavior of application in a particular phase, number of dirty blocks can be very high. When number of dirty blocks is relatively high, flushing the cache would result in high energy/performance overhead. Our cache assignment algorithm does not reconfigure cache at the reconfiguration point for a phase starting with high number of dirty blocks. Nonetheless, applications happen to show significant changes in cache requirements that can be exploited using intra-task DCR to compensate for the reconfiguration overhead.

In a given system, if it allows only a limited number of cache reconfigurations, we study the effect of our approach by constructing a spectrum of energy savings. We limit the number of potential reconfiguration points to *l* (using profitability) so that at most *l* number of reconfigurations can happen. Note that this does not mean that reconfiguration will happen at exactly *l* points. Fig. 6 illustrates the number of reconfiguration points – energy savings tradeoffs using *pegwit* benchmark (we observed similar pattern for other benchmarks as well). As we increase *l*, higher energy savings can be achieved. Interestingly, execution time has an inverse relation to energy savings. In other words, increasing *l* increases energy savings and decreases execution time (improves performance). This is expected since one of the best ways to reduce energy consumption is to shrink execution time. Intra-task DCR can reduce execution time by reducing cache misses through assigning larger caches in the cache intensive phases of programs. There are number of saturation points in the graph where increasing the number of potential reconfiguration points (*l*) does not change the result of our intra-task DCR. This is due to the fact that the newly added reconfiguration point is not a good choice (effective phase delimiter) to do the reconfiguration and is ignored by our cache assignment algorithm.

6. CONCLUSION

Optimization techniques are widely used in embedded systems design to improve overall area, energy and performance requirements. Dynamic cache reconfiguration is very effective to reduce energy consumption of cache subsystem. In this paper we presented a intra-task dynamic cache reconfiguration approach using novel phase detection and cache selection algorithms. Our experimental results demonstrated up to 27% reduction (12% on average) and 19% (7% on average) in overall energy consumption of instruction and data cache, respectively, compared to existing inter-task DCR techniques.

REFERENCES

1 A. Malik, B. Moyer, and D. Cermak. A low power unified cache architecture providing power and performance flexibility. *ISLPED* (2000).

2 H. Hajimiri, K. Rahmani, P. Mishra. Synergistic integration of dynamic cache reconfiguration and code compression in embedded systems. *International Green Computing Conference (IGCC)* (2011).

3 M. Peng, J. Sun, Y. Wang. A Phase-Based Self-Tuning Algorithm for Reconfigurable Cache. (), ICDS 07.

4 A. Gordon-Ross, J. Lau, B. Calder. Phase-based Cache Reconfiguration For a Highly-Configurable Two-Level Cache Hierarchy. *GLSVLSI 08.*

5 Gordon-Ross et al. Fast configurable-cache tuning with a unified second level cache. ISLPED (2005).

6 P. Vita. Configurable Cache Subsetting for Fast Cache Tuning. DAC (2006).

7 D. H. Albonesi. Selective Cache Ways: On-Demand Cache Resource *Allocation (2000).*

8 C. Zhang, F. Vahid, W. Najjar. A highly-configurable cache architecture for embedded systems. *ISCA* (03).

9 L. Chen, X. Zou, J. Lei, Z. Liu. Dynamically Reconfigurable Cache for Low-Power Embedded System. *ICNC (2007).*

10 W.Wang and P. Mishra. Dynamic Reconfiguration of Two-Level Caches in Soft Real-Time Embedded Systems. ISVLSI (2009).

11 Burger et al. Evaluating future microprocessors: the simplescalar toolset. University of Wisconsin-Madison, Technical Report CS-TR-1308 (2000).

12 CACTI. HP Labs, CACTI 4.2, http://www.hpl.hp.com/.

13 Lee et al. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. *Micro.* (1997).

14 Guthaus et al. MiBench: A free, commercially representative embedded benchmark suite. WWC (2001).

15 A. Gordon-Ross, F. Vahid, N. Dutt. Automatic tuning of two-level caches to embedded *applications*. *DATE* (2004).