Energy-Aware Synthesis of Application Specific MPSoCs

ABSTRACT

Application specifc multi-processor system on chip (MPSoC) can be used to execute streaming applications in a pipelined manner. Each processing element (PE) in the pipeline can be customized depending on the target constraints imposed by the system designer leading to an heterogenous system.

The processing elements can be customized by adding application-specific custom instructions set, selecting appropriate cache configurations and by setting different voltage and frequency. In this paper, we propose a framework to synthesize an heterogeneous application specific MPSoC for the target streaming application. Our framework optimizes for the total energy consumption provided the pipeline period constraint and the silicon area budget. We obtain the optimal points with an efficient branch and bound algorithm. Our results show that the optimally configured ASIP MPSoC by our method acheives significant energy reduction compared to the conventional techniques.

1. INTRODUCTION

MultiProcessor System on Chips (MPSoCs) have significantly proliferated in the embedded system domain due to the scaling of transistor. Furthermore, the MPSoCs can be configured to satisfy the requirements of the target applications, leading to an Application Specific MPSoC. The cores in Application Specific MPSoC can be diverse in nature and may include coprocessors, DSPs, general purpose processors or hardware accelerators. The heterogeneity in designing Application Specifc MPSoC can be exploited to improve the efficiency of parameters like performance, area and energy consumption for the target application.

Low energy consumption is highly desirable for MPSoCs that are used in portable devices to increase the battery life of the device. Prior research has shown that the use of Dynamic Voltage and Frequency Scaling (DVFS) and customization of the processors are two effective techniques for reduction of energy consumption [3]. This is particularly so for streaming applications, which contain sub-kernels that are executed repeatedly in a pipelined fashion [6]. DVFS results in quadratic reduction in energy with only linear sacrifice of the processor speed. Streaming applications benefit from DVFS by exploiting the slack of non- critical stages to reduce their dynamic energy consumption.

Customization of the processors in an MPSoC is typically achieved through the use of Application Specific Instruction set Processors (ASIPs) [7]. ASIPs can be customized according to the sub-kernels of a streaming application. The addition of custom instructions to an ASIP improves it's energy efficiency for the following reasons: the number of instruction fetches reduces, and the number of register file accesses for data transfer between the instructions reduces [5]. On the other hand, custom instructions can also have a detrimental effect because of the increase in on-chip area, and thus in the leakage energy consumption. Hence, a designer has to carefully choose custom instructions to minimze the energy consumption.

In an ASIP, the memory subsystem (caches) contributes significantly to its total energy consumption [4, 2]. In particular, the memory subsystem is a significant contributor of the on-chip area, and thus of the total leakage energy consumption. Therefore, design of a memory subsystem is also crucial in improving not only the energy efficiency, but also the performance and the area utilization of an ASIP. Existing works on reducing energy consumption in the memory subsystems are [2].

Traditionally, researchers have focussed on reducing energy consumption of ASIP based MPSoCs with the addition of custom instructions or customization of the cache or the use of DVFS. They fail to consider all the three techniques together for maximal reduction in energy consumption. With the customization of the ASIPs and their caches, and the added advantage of DVFS, one can synergistically design an energy- efficient heterogeneous MPSoC for streaming applications, which are the basis of entertainment in portable devices **??**.

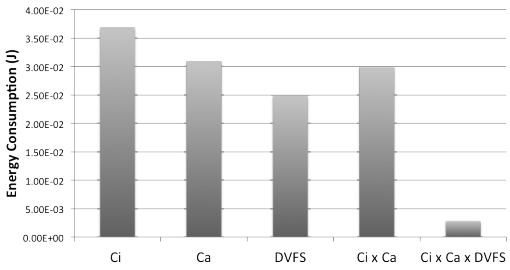

Figure 1: Minimal Energy Consumption for DCT kernel of MP3 encoder

To show the importance of considering all the three techniques together, we analyze a sub-kernel of a streaming application. We chose *discrete cosine transform (DCT)* subkernel from the *MP3* encoder application. The experimental setup is described in detail in Section **??**. Figure **??** plots the minimal energy consumption achievable for each of the following techniques: a) Only custom instructions are added (**Ci**), b) Only cache is customized (**Ca**), c) Only DVFS is used (**DVFS**) d) Custom instructions are added with customization of the cache (**Ci** $\times$ **Ca**) and e) All the three techniques are considered simultaneously (**Ci** $\times$ **Ca** $\times$ **DVFS**). Use of both the custom instructions and customized cache is better than their individual use because addition of custom instructions modifies a kernelâĂŹs behaviour such as code size, data transfers between instructions, memory access pattern, etc. Therefore, a designer should choose an appropriate cache based on the custom instructions to improve the energy efficiency. Next, the use DVFS, in addition to custom instructions and customized cache, results in minimum energy consumption, which is significantly lower than the other techniques. Thus, it is evident that separate use of the three energy reduction techniques may result in a local optima. Therefore, it is desirable to use all the three

techniques together to reach better global optima.

Although the use of custom instructions, customized cache and DVFS achieves better energy efficiency, their combined use significantly increases the complexity of the optimization problem. For instance, consider an application with only four tasks, four different custom instructions per task, four different voltage/frequency levels and four different cache configurations. Then, the total number of points in the design space is more than a billion. Therefore, in this paper, we focus on the problem of selecting custom instructions, cache configuration and voltage/frequency level for sub-kernels of a streaming application, which is executed on an MPSoC. To aid quick and efficient exploration of the design space, first we propose estimation methods to compute execution time and energy consumption of a sub-kernel. Then, we design a novel branch and bound algorithm to efficiently prune and search the design space. In particular, this paper has the following contributions:

1. We propose a novel problem formulation for ASIP systhesis based on energy efficiency.
2. We develop an analytical framework to explore the complex design space involving custom instruction set versions, cache sizes and voltage/ frequency settings simultaneously.
3. We propose a novel branch and bound algorithm to identify the optimal points in the complex design space.
4. Finally, we compare our technique with the exisiting conventional techniques for energy optimization.

2. RELATED WORK

3. PROBLEM FORMULATION

In this paper, we target heterogeneous MPSoCs that consist of customizable Processing Elements (PEs), which can be realized with the use of ASIPs. Each PE has private cache and communicates with other PEs via dedicated communication buffers (for example, FIFO queues). Each PE can be customized by both extending its baseline instruction set architecture and customizing its cache. Additionally, each PE can operate in several discrete voltage and frequency levels. Thus, the heterogeneity in the MPSoC is manifested in terms of custom instructions, custom cache configurations and voltage/frequency levels.

The target application domain comprises of streaming applications, which contains multiple compute-intensive subkernels or tasks. The task graph of an application is a directed acyclic graph, where the tasks are mapped to PEs to enable pipelined execution of the streaming application. Figure 3 illustrates the pipelined execution in MPSoC. For example, two tasks are mapped to the first PE and one task is mapped to the second PE. The PEs form the logical stages of the pipeline. At the end of each stages of the pipeline, a single iteration of a task is completed. The steady state in the given example is reached after the stage one of the second PE. The period of the pipeline is defined as the maximum latency among all the stages (as shown in Figure 3.

Each task can be accelerated with a set of custom instructions. Hence, there are multiple implementations of each task corresponding to differing sets of custom instructions that can be used. Each set of custom instructions for a task is associated with its additional area. Furthermore, each task can be executed with one of the available discrete voltage and frequency levels. The execution time and energy consumption of a task then depends on the cache configuration of the PE on which it is mapped, and the set of custom instructions and voltage/frequency level selected for it. The area of the baseline PE, additional custom instructions and the cache configuration determines the total area of the PE. The area of the MPSoC is then the summation of the area of all the PEs.

Benoit et al. [1] categorizes the policies to map tasks of a task graph on an MPSoC with fixed number of PEs into: one-to-one mapping, where only a single task is mapped to a PE; interval based mapping, where only adjoining tasks are mapped to a PE; and, general mapping, where no restrictions are placed at all. In this paper, we use general mapping which offers greater flexibility, and has the potential to reach better global optima. In summary, our design space exploration should search through: a) the number of PEs, b) mapping of the tasks on the the PEs, c) the cache configurations for each of the PEs, d) the sets of custom instructions for each of the tasks , and e) the voltage/frequency levels for each of the tasks.

The problem can be formally stated as follows: given an acyclic task graph of an application, differing sets of custom instructions for each task, differing discrete voltage/frequency levels for each task, differing cache configurations for a PE, the steady state period constraint and the area constraint, the goal is to minimize the total energy consumption of the MPSoC. Therefore, the following need to be determined:

- The energy-wise optimal number of PEs and mapping of the tasks on them.
- The energy-wise optimal cache configuration for the each of the individual PEs.
- The energy-wise best set of custom instructions and voltage/ frequency level for each task.

Note that the optimization problem described above cannot be solved naively because of its exponential complexity, which results from all the possible combinations of task mapping, cache configurations, sets of custom instructions, and voltage/frequency levels.

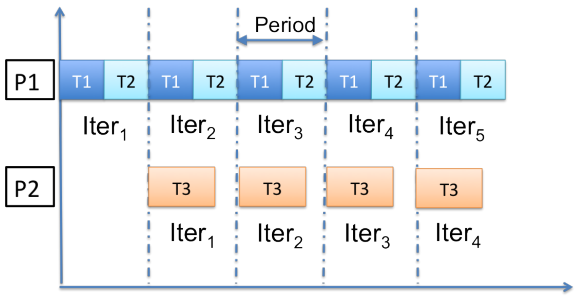

Figure 2: Pipelined Execution

4. PROPOSED FRAMEWORK

In this section, we explain our framework to solve the energy optimization problem described in the last section. The input to the framework is as follows: A streaming application, which is represented as a task graph with N tasks $< T_1, T_2, ..., T_N >$, M_i sets of custom instructions for the task $T_i < CI_{i1}, ..., CI_{iM} >$, V levels of available voltages/frequencies $< vf_1, ..., vf_V >$ and C cache configurations $< Ca_1, ..., Ca_C >$. The designer provides the steady state *period* constraint, P_c, and the *area* constraint, A_c as the input constraints to the framework. Figure 4 shows the flow of the framework. In the profiling stage, we profile the differing sets of custom instructions of the individual tasks on a single PE at differing voltage/frequency levels and with differing cache configurations to obtain latencies and energy consumptions of their single iterations. For instance, for task T_i, $M_i \times V \times C$ simulations are run to capture latency and energy consumption of every implementation of T_i on a single PE. Additionally, we collect the trace of each simulation during the profiling stage. The second stage, named latency- energy estimation, is then used to analytically estimate the latency and energy consumption of differing mappings of the tasks on the PEs using the profiling information and traces generated by the first stage. Lastly, the framework uses a novel branch and bound algorithm based upon the estimation performed by

After Iteration	Cache State (CS)	Compulsory Misses (Comp)	Capacity/ Conflict Misses
1	$\{m_0, m_5, m_6, m_7\}$	$\{m_0, m_1, m_2, m_3\}$	$\{m_5, m_6, m_7\}$
2	$\{m_0, m_5, m_6, m_7\}$	$\boldsymbol{\{m_1, m_2, m_3\}}$	$\{m_5, m_6, m_7\}$
3	$\{m_0, m_5, m_6, m_7\}$	$\boldsymbol{\{m_1, m_2, m_3\}}$	$\{m_5, m_6, m_7\}$

Table 1: Cache across iterations

the second stage to quickly and efficiently prune and search the design space for the optimal design point – the one with minimum energy consumption.

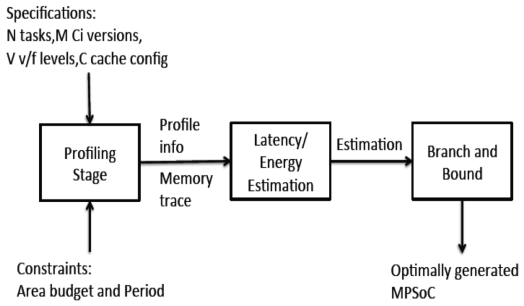

Figure 3: Framework Flow

4.1 Latency-Energy Estimation

As mentioned earlier, we use general task mapping as it allows greater flexibility and efficiency in terms of energy. Generating all the possible task mappings is equivalent to enumerating all the possible set partitions []. For example, an application, consisting of five tasks, can be mapped onto five PEs in 52 different ways (some PEs might not have any task mapped on them). It is not realistic to simulate all the possible mappings of all the tasks when the number of tasks increases. This is further exacerbated by the availability of differing custom instructions, cache configurations and voltage/frequency levels. Therefore, the *profiling stage* simulated only mapping of a task on a single PE, with its differing custom instructions, cache configurations and voltage/frequency levels so as to keep the simulation time minimal and reasonable. Furthermore, the simulations in the *profiling stage* were only done for a single iteration, and hence do not represent the steady state. Now, we propose a technique to estimate the latency and energy consumption of a task under steady state and its different mappings.

The latency and energy consumption of a task depends on the number of cache hits/misses. Cache misses can be broadly classified as compulsory misses, capacity misses and conflict misses. The capacity and conflict misses remain constant across all the iterations of a task. Thus, the capacity and conflict misses can be captured by executing a single iteration of the task, which was done in the profiling stage. In the steady state, the compulsory misses change across iterations which is due to the repeated execution of the task. Consider that the terminology "cache state" denotes the contents of all the cache blocks for a given cache configuration. For the sake of simplicity, a direct mapped cache is assumed in the following example. However, the estimation technique can easily be extended to set- associative caches. The state of a direct mapped cache is a set of n elements, c[0 ... n-1], where c[i] = m if the cache block i holds the memory block m. Let *Comp* be the set representing the blocks that were fetched due to compulsory misses. Let *CS* (obtained from the trace captured during the profiling stage) represent the cache state at the end of an iteration of the task. Since a task is repeatedly executed in the steady state, its compulsory misses will reduce. For example, suppose that the cache has four blocks and the memory access pattern of the task is $\{m_0, m_1, m_2, m_3, m_5, m_6, m_7\}$. Then, $CS_1 = \{m_0, m_5, m_6, m_7\}$. The cache blocks obtained due to compulsory misses are $\{m_0, m_1, m_2, m_3\}$. Thus, in the steady state m_0 will always be a hit in the cache. Therefore, the reduction in number of misses is

$$Nr_{miss} = CS \cap Comp \tag{1}$$

Thus, the steady state latency and energy consumption of the task T_i can be estimated using the following equations:

$$L^{ss}_{T_i} = L^1_{T_i} - (Nr_{miss} * penalty_{ll}) E^{ss}_{T_i} = E^1_{T_i} - (Miss_r * ML) \tag{2}$$

where $L^{ss}_{T_i}$ and $L^1_{T_i}$ is the steady state latency and the single iteration latency of the task T_i respectively and $penalty_{ll}$ is the penalty to access the lower level memory.

$$E^{ss}_{T_i} = E^1_{T_i} - (Nr_{miss} * energy_{ll}) \tag{3}$$

where $E^{ss}_{T_i}$ and $E^1_{T_i}$ is the steady state energy consumption and the single iteration energy consumption of the task T_i respectively and $energy_{ll}$ is the energy required to access the lower level memory. The effects of conflict misses and capacity misses are already captured in $L^1_{T_i}$ and $E^1_{T_i}$ from the profiling information.

Equations 2 and 3 provide the steady state latency and energy consumption of a single task for a given cache configuration. For a single task, the steady state latency and energy can also be computed by simulating the task for more than single iterations. For long running tasks, the simulation time would increase for more iterations. Moreover, our estimation technique results in error less that 1% compared to the actual simulation **??**. When more than one task is mapped on a PE, each task can pollute the cache state of each other. We assume that all the tasks are non preemptible during their execution. This is a valid assumption in streaming applications [] because each task has to process its input before transferring it to the next task. We explain our estimation technique with two tasks T_1 and T_2, but it can easily be extended to any number of tasks mapped on a PE. Let CS_1 and CS_2 be the cache states at the end of first iteration of tasks T_1 and T_2 respectively (obtained from the trace captured during the profiling stage). Similarly, let $Comp_1$ and $Comp_2$ represent the set containing the blocks obtained using compulsory misses. In steady state, the number of misses reduces for a particular task, when the compulsory miss blocks survive through the execution of the other task. The aim is to estimate the number of blocks that endured in the cache. For a particular cache block, we define the operator $\bigodot$ as $m' \bigodot m'' = m''$ which means that the memory block m(double prime) has replaced the memory block m(single prime) in the cache block c.

The reduction in the number of misses for T_1,

$$Nr_{miss,T_1} = (CS_1 \bigodot CS_2) \cap Comp_1 \tag{4}$$

Similarly for T_2,

$$Nr_{miss,T_2} = (CS_2 \bigodot CS_2) \cap Comp_1 \tag{5}$$

Therefore, the steady state execution time and energy consumption,

$$E_{ss,T_1,T_2} = E_{T_1} + ET_2 - ((Nr_{miss,T_1} + Nr_{miss,T_2}) * penalty_{ll}) \tag{6}$$

$$En_{ss,T_1,T_2} = En_{T_1} + En_{T_2} - ((Nr_{miss,T_1} + Nr_{miss,T_2}) * energy_{ll}) \tag{7}$$

Our estimation technique allows estimation of the steady state latency and energy consumption of any number of tasks on a PE without the need for simulation of all the possible mappings of those tasks, as long as the latency and energy consumption of the first iteration is known. We verify the accuracy of our estimation technique in Section **??**.

4.2 Branch and Bound Algorithm

Given N tasks, let the total number of different mappings possible be B_N, which is equal to the N^{th} Bell number **??**. Let the average number of customized configurations for task T_i be M. Each different versions can be run at V different voltage/frequency levels. Let C be the number of cache configurations supported. Thus, an axhaustive search will go through all the design points, where the worst-case complexity is exponential and can be written as $O(B_N * M^N * V^N * C^{B_N})$.

The algorithm proposed here is based ona branch and bound technique so that the optimal design point can be searched quickly. Although the worst-case complexity of the branch and bound technique used in our algorithm is the same as the exhaustive search; however, it is able to prune a large part of the design space by exploiting the constraints. Thus, our algorithm completes much faster than the exhaustive search (see Section **??**). The algorithm is used to efficiently search through the complex design space tree. For the sake of simplicity, Figure 4.2 shows only a part of the design space tree.

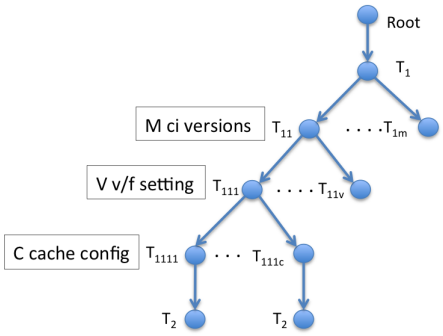

Figure 4: Example of Branching

The pseudo code for our branch and bound algorithm is shown in Algorithm 1. Let A_c and P_c be the area and period constraints provided as an input. In each iteration, we map a task to one of the existing PEs (in lines 14-28) or to a new PE (in lines 30-45), and select a set of custom instructions and voltage/frequency level for it. For a task T_i, the algorithm enumerates all of its possible implementations which are the combinations of sets of custom instructions, cache configurations and voltage/frequency levels. And each of the customized versions are further branched into multiple versions corresponding to different voltage/ frequency settings. For each of these versions, it could be mapped to one of the already existing PEs or a new PE. And each PE can further be tailored based on the various cache configuration supported. This explains our branching technique in searching the entire design space tree. In lines 19 and 35, the proceduce *calculate_params* updates the current estimation of area, period and energy. In Figure 4.2, the number of existing PE for the first task T_1 is null.

At level i of the search tree, we have a partial solution explaining the choice of configurations chosen for tasks $T_1, T_2...T_i$, the voltage/frequency settings and the cache configuration for each of the PEs already mapped. Let the period and area in the partial solution space be *period* and *area* respectively. With this partial solution, we can prune the subtree based on the violation of any of the following constraints:

1. Pruning based on area constraint: By mapping only the software only version of the remaining tasks in the existing PEs, the total area occupied on-chip is still equal to *area*. This is because the software only version of the tasks does not occupy any extra area. If this *area* violates the area constraint, it is safe to prune the entire subtree. For example,

2. Pruning based on period constraint: By mapping the version of the remaining tasks that has the minimal steady state execution time, the lower bound of the new period can be estimated to be the $max(period_i, max(E_{i+1}, ..$ If the new period violates the period contraint, it is safe to prune the entire subtree. For example,

3. Pruning based on Lower Bound Cost (LBC): The LBC is defined as the lowest possible energy consumption estimated at the level i. For each of the task, we identify the versions of the task that consume the minimal energy. Then, the LBC can be estimated by summing the minimal energy version of the remaining tasks $T_{i+1}, T_{i+2}...T_N$. For example, consider two tasks T_i, T_j with the lowest steady state energy consumption as $En_{ss,i}, En_{ss,j}$ respectively.

$$LBC = En_{ss,i} + En_{ss,j} \tag{8}$$

The estimated LBC indirectly represents that the tasks are mapped to a dedicated PE. If the more than one task is mapped to a single PE, the energy consumed is definitely greater than or equal to the sum of the steady state energy consumption of the individual tasks. If the tasks T_i, T_j were to be mapped in a single PE, then

$$LBC \leq En_{ss,i,j} \tag{9}$$

From Equation 9, it is evident LBC estimated is indeed the lowest possible energy acheivable in the subtree. One can easily prune a subtree, if the existing best solution is smaller than the estimated LBC for the energy.

5. EXPERIMENTAL SETUP

We use a commercial tool set Xtensa RD-2011.2 from Tensilica Inc., **??**. Our base processor is Xtensa LX2 extensible cores and its configurations are summarized in Table 2. The tool set includes C/C++ compiler that can compile C/C++ code for the target LX2 processor. The target application is simulated in the Instruction Set Simulator (ISS) that is included in the toolset. We use the Xtensa PRocessor Extension Synthesis (XPRES) compiler provided by the toolset to generate the various customized versions of the tasks. The newly generated task versions consist of any combination of fused operations, FLIX instructions, vector operations and specialized operations.

In our experiments, we use five different instruction cache configurations ranging from 1 KB to 16 KB. Table 2 shows the area consumed by various cache configurations. We modify only instruction cache, but our technique can easily be extended to data cache too. We use Xt-Xenergy tool included in the toolset to measure both the dynamic and leakage energy consumptions. We use five different frequency levels ranging from 1.5 Ghz to 533 Mhz. We always set the voltage that supports the frequency being set **??**. We perform all our simulations at the maximum frequency and voltage. We estimate the power consumption at the frequency level i using the following equation,

$$P_i = \frac{P_{max} * (V_i)^2 * F_i}{(V_{max})^2 * F_{max}} \tag{10}$$

We use popular streaming applications like MP3 encoder and JPEG encoder. Figure 5 shows the various compute intensive kernels in the application.

6. RESULTS

We verify the estimation of the steady state execution time and energy consumption from the stage *latency/energy estimation* in our framework. Table 6 summarizes the accuracy of our estimation for the streaming applications JPEG and MP3. We varied the number of tasks mapped on a PE and compared our estimations with the actual simulation.

Benchmark	Experiment Number	Hierarchical Approach			Ci x Ca x DVFS		
		Maximum Improvement (%)	Minimum Improvement (%)	Average Improvement (%)	Maximum Improvement (%)	Minimum Improvement (%)	Average Improvement (%)
JPEG	1	29.51	1.27	24.35	32.31	3.27	26.52
	2	29.04	2.23	26.31	29.04	0.532	25.12
MP3	1	41.78	3.27	43.02	49.01	10.49	44.17
	2	43.03	6.78	39.08	47.30	12.50	43.97

Table 3: Experimental results in comparison with Ci+Ca

Frequency	1.5 Ghz
Pipeline Depth	5
Process Techology	65nm GP
Core Area	82806 gates or 0.451 mm^2
Max Instruction Width	8 bytes
Data Cache size	32 KB
Instruction Cache Size 16K, 8K, 4K, 2K, 1K	0.452 mm^2, 0.275 mm^2, 0.174 mm^2, 0.124 mm^2, 0.103 mm^2
I-Cache & D-cache line size	16 bytes
I-Cache & D-cache Assoc	Direct mapped
PIF Interface	32 bytes
Other Features	Boolean Registers, MUL32 and MAC16 instructions Functional unit clock gating, Flip-Flop register files

Table 2: Baseline Processor Configuration.

Benchmark	Number of tasks mapped	Execution Time		Energy Consumption	
		Max error (%)	Avg error (%)	Max error (%)	Avg error (%)
JPEG	1				
	2				
	3				
	4				
	5				
MP3	1				
	2				
	3				
	4				
	5				

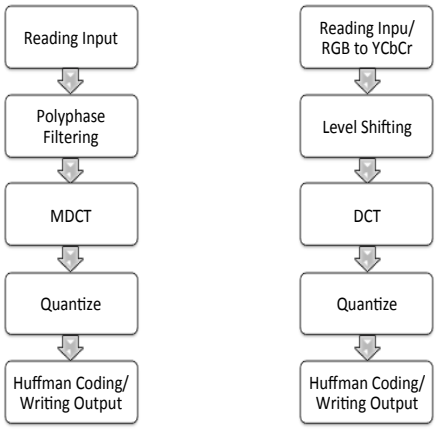

Figure 5: MP3 and JPEG

We compare the optimal points ASIP configuration obtained by the following techniques:

1. **CixCaxDVFS**: We use our branch and bound technique to find the optimal energy efficient ASIP configuration while modifying custom instruction set, cache configurations and voltage/frequency setting.
2. **CixCa+DVFS**: We call this technique as **Hierarchical Approach**. At the first stage, we optimize for the energy by only modifying custom instruction set and cache configurations. With the ASIP configuration obtained from the first stage, we apply different voltage/frequency setting to further reduce the energy. This technique can be derived by adding DVFS to the technique mentioned in **??**.
3. **Ci+Ca**: Energy optimization by modifying only custom instruction set and cache configurations.

All the optimal points selected respects the input area and period constraints provided. As mentioned before, our design space consists of more than billion points along the axes of period, area and energy. Hence, it is not practical to plot all the points in a graph. Furthermore, we have two input constraints in terms of area and period. To get representative inputs for comparison, we perform two experiments per benchmark. In each of the experiments, we employ Latin hypercube sampling to generate 500 different tuples consisting of the area and period constraint. Using our branch and bound technique, we determine the lower bound of the period and the upper bound of the area by giving *infinity* as an input constraint for both area and period for the target application. Similarly, the upper bound for the period is determined by summing the execution time of the software only versions of each task at the lowest possible cache configuration. The lowest bound for the area is equivalent to the size of a single PE and smallest cache configuration supported. With these bounds, latin hypercube sampling **??** efficiently covers the entire design space.

Algorithm 1 Branch and Bound algorithm

```
1: tasks = {T_1, T_2...T_N};
2: map = [];
3: currA = 0;                                   ▷ current area
4: currP = 0;                                   ▷ current period
5: currE = 0;                                   ▷ current energy
6: existingPEs = {};    ▷ The set containing existing PEs
7:
8: BnB(tasks, A_c, P_c, map, existingPEs);
9:
10: procedure BnB(tasks, A_c, P_c, map, existingPEs)
11:
12:     prune(tasks, A_c);        ▷ prunning based on the area
13:     prune(tasks, P_c);    ▷ prunning based on the period
14:     prune(tasks, best_solution);    ▷ prunning based on
    the LBC
15:
16:     if tasks ≠ null then
17:         \* mapping to one of the existing PEs *\
18:         T = remove a task from tasks;
19:
20:         for j = 1 to M  do                  ▷ CIS versions
21:             for k = 1 to V  do              ▷ V/F settings
22:                 for  each PE in ex_pe  do
23:                     Add T_ijk to map{PE};
24:                     (currA, currP, currE)=cal_params(map(PE));
25:
26:                     if period ≤ P_c && area ≤ A_c then
27:                         BnB(tasks, A_c, P_c, map, existingPEs);
28:                     else
29:                         update  (currA, currP, currE)  to
    previous value;
30:                         Remove T_ijk from map{PE};
31:                     end if
32:                 end for
33:             end for
34:         end for
35:
36:         \* mapping to a new PE *\
37:         for j = 1 to M  do                  ▷ CIS versions
38:             for k = 1 to V  do              ▷ V/F settings
39:                 for l = 1 to C  do      ▷ C Cache Configs
40:                     Add T_ijk to map{PE_l};
41:                     Add PE_l to existingPEs ;
42:                     (currA, currP, currE)=cal_params(map(PE_l));
43:
44:                     if period ≤ P_c && area ≤ A_c then
45:                         BnB(tasks, A_c, P_c, map, existingPEs);
46:                     else
47:                         update (currA, currP, currEn) to
    previous value;
48:                         Remove T_ijk from mapPE_l;
49:                         Remove PE_l from existingPEs;
50:                     end if
51:                 end for
52:             end for
53:         end for
54:
55:     else
56:         \* We have a solution *\
57:         Update the best solution
58:     end if
59:
60: end procedure
```

We compare the results of the **CixCaxDVFS** and **CixCa+DVFS** with that of **Ci+Ca**. It is evident from the Table 3, both the techniques significantly outperforms **Ci+Ca**. On an average, **CixCaxDVFS** performs better than **CixCa+DVFS**. This is because, the hierarchical nature of the technique **CixCa+DVFS** could result in the local optimum in comparison to the global optimum acheivable by **CixCaxDVFS**. Figure 6 and **??** shows the optimal energy acheivable for four different area and period constraint. In case of the hierarchical approach, we observed that there exists input constraints that was not satisfied to determine the optimal point. Thus, suggesting to consider all the three techniques (**Ci, Ca, DVFS** together.

7. CONCLUSION

8. REFERENCES

[1] Anne Benoit and Yves Robert. Mapping pipeline skeletons onto heterogeneous platforms. In Yong Shi, Geert van Albada, Jack Dongarra, and Peter Sloot, editors, *Computational Science âĂŞ ICCS 2007*, volume 4487 of *Lecture Notes in Computer Science*, pages 591–598. Springer Berlin / Heidelberg, 2007.

[2] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. *SIGARCH Comput. Archit. News*, 28(2):83–94, May 2000.

[3] Seungrok Jung, Jungsoo Kim, Sangkwon Na, and Chong-Min Kyung. Energy-aware instruction-set customization for real-time embedded multiprocessor systems. In *Proceedings of the 14th ACM/IEEE international symposium on Low power electronics and design*, ISLPED '09, pages 335–338, New York, NY, USA, 2009. ACM.

[4] Uming Ko, Poras T. Balsara, and Ashwini K. Nanda. Energy optimization of multi-level processor cache architectures. In *Proceedings of the 1995 international symposium on Low power design*, ISLPED '95, pages 45–49, New York, NY, USA, 1995. ACM.

[5] Hai Lin and Yunsi Fei. Exploring custom instruction synthesis for application-specific instruction set processors with multiple design objectives. In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, ISLPED '10, pages 141–146, New York, NY, USA, 2010. ACM.

[6] Seng Lin Shee, Andrea Erdos, and Sri Parameswaran. Heterogeneous multiprocessor implementations for jpeg:: a case study. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '06, pages 217–222, New York, NY, USA, 2006. ACM.

[7] Fei Sun, N.K. Jha, S. Ravi, and A. Raghunathan. Synthesis of application-specific heterogeneous multiprocessor architectures using extensible processors. In *VLSI Design, 2005. 18th International Conference on*, pages 551 – 556, jan. 2005.