Energy-Aware Synthesis of Application Specific MPSoCs

Thannirmalai Somu Muthukaruppan[1], Haris Javaid[2], Tulika Mitra[1], and Sri Parameswaran[2]

[1]School of Computing, National University of Singapore, Singapore
[2]School of Computer Science and Engineering, University of New South Wales, Sydney, Australia
{tsomu, tulika}@comp.nus.edu.sg, {harisj, sridevan}@cse.unsw.edu.au

***Abstract*—In this paper, we propose a framework for synthesis of application specific MultiProcessor System on Chip (MPSoC) for multimedia applications. Our framework searches for a design with minimum energy consumption under area and period constraints. We simultaneously explore selection of voltage-frequency levels, custom instructions, cache configurations, and task mapping. We propose an optimal algorithm based on *prune* and *search* operations to efficiently search the complex design space. We also present a heuristic based on *map* and *customize* stages to better handle the exponential complexity of the design space, and rapidly find near-optimal solutions. These algorithms are aided by two estimators that can quickly estimate period and energy consumption of a given design point. Experiments reveal that our framework can reduce energy consumption by 37.9% on an average and 57.1% maximum reduction compared to solutions obtained from a combination of existing techniques.**

I. Introduction

MultiProcessor System-on-Chip (MPSoC) architectures have significantly proliferated in portable devices, where they have have to satisfy stringent requirements of the target application(s) and/or the target device. For example, MPSoCs for multimedia applications have to deliver a certain performance to provide reasonable quality of service to the users (performance constraint), must have area smaller than a certain limit due to the size of the portable devices (area constraint), and should have low energy consumption to increase the battery life. Therefore, application specific MPSoCs are deployed in portable devices [29] where an MPSoC is (extremely) customized for a given application under an objective function and various constraints. This paper focuses on customization of MPSoCs for multimedia applications with the objective of minimum energy consumption under performance and area constraints. We explore the following four energy reduction techniques (design parameters):

- Dynamic Voltage and Frequency Scaling (DVFS) allows processors to operate at multiple discrete voltage-frequency (v-f) levels. DVFS is particularly suitable for multimedia applications where the slack of non-critical tasks is exploited by the use of a lower v-f level to reduce the energy consumption without sacrificing the performance [26].
- Customization of processors aims to match the processing elements of an MPSoC to the computational requirements of the tasks at hand. Processor customization involves addition/removal of functional units, hardware accelerators, custom register files, etc. Custom processors are typically realized through the use of Application Specific Instruction set Processors (ASIPs) [12], where custom instructions are added to access the custom hardware. These custom instructions, when carefully designed, can reduce instruction fetches and register file accesses and improve the energy efficiency of a processor [19].
- The cache of a processor contributes significantly to its power consumption [13], in particular static power because it consumes significant amount of on-chip area. Customization of cache according to memory access pattern of a task can significantly reduce energy consumption [10].
- Task mapping allows a designer to map tasks of an application to the processors. Task mapping is done so as to balance the workload across all the processors in an MPSoC, improving their utilization and thus reducing energy consumption of the MPSoC [4].

Given the above design parameters, customization of an MPSoC for a target application becomes an optimization problem where the MPSoC's design space (resulting from the options available for the design parameters) is explored for an optimal solution. While there exist several works in literature that have focused on a subset of the aforementioned design parameters (for example, [12] considered processor customization and task mapping; [13] considered cache customization), these optimization techniques are designed to work efficiently only with the considered set of design parameters. A mere combination of these individual optimization techniques to cover all the aforementioned design parameters is not the most effective solution. In fact, the authors of [12] illustrate that optimization with simultaneous processor customization and task mapping resulted in solutions that are 16% better in performance compared to when processor customization and task mappings were performed independently one after the other.

Motivational Example. We analyze three typical multimedia applications (JPEG encoder, MP3 encoder and H.264 encoder) to observe the sub-optimality in using independent optimization techniques for DVFS, processor customization, cache customization, and task mapping. For each application, we optimized the MPSoC for minimum energy consumption under performance and area constraints, where multiple v-f levels per task, multiple custom instructions per task, multiple cache configurations per processor, and general task mapping were used as the design parameters. Further details of the experimental setup are provided in Section V. Figure 1 plots the minimum energy design point obtained by the "independent" and "integrated" optimization techniques. In the "independent" technique, an optimal solution is sought for each design pa-

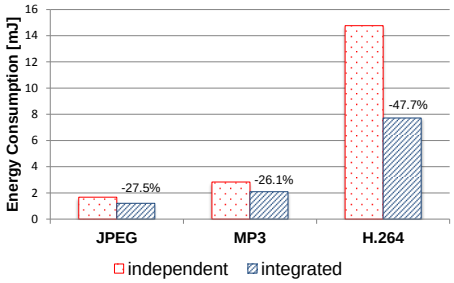

Fig. 1: Comparison of 'independent' and 'integrated' optimization techniques.

rameter one after the other. For example, first an optimal task mapping is selected, then optimal v-f levels for all the tasks are identified given the task mapping, and so on. It is important to note that the sequence of independent optimization techniques affects the optimality of the solution, and thus we exhaustively attempted all possible orders of individual optimizations (for n design parameters, independent optimizations can be performed in $n!$ ways). Therefore, the solution of "independent" optimization technique is the best possible solution from the use of independent individual optimization techniques. The "integrated" optimization technique explores all the design parameters in an integrated and synergistic fashion so as to take into account the complex interplay of DVFS, processor customization, cache customization and task mapping. For example, use of custom instructions for a task modifies its code size and memory access pattern, which in turn affects the customization of the cache for the processor on which this task will be mapped. Thus, the interplay of design parameters must be considered to find a globally optimal solution.

It is evident from Figure 1 that an "integrated" optimization technique has far better potential in reaching the globally optimal solution than the "independent" technique. More importantly, the quality of the solutions from the "independent" technique are significantly inferior even when all the possible ways of combining optimal solutions from individual techniques are exhausted. For example, as shown in Figure 1, the amount of energy saved using "integrated" technique is atleast 26.1%. The advantage of synergistic use of various optimization techniques comes at a price. The complexity of the optimization problem, which depends on the number and types of the design parameters, and the number of options considered for those parameters, increases manifold. In fact, the optimization problem with DVFS, processor customization, cache customization and task mapping is an NP-Hard problem [4]. For a glimpse of the optimization problem's complexity, consider an application with only four tasks, four custom instructions per task, four v-f levels and four cache configurations. Then, the total number of design points is more than a billion. Therefore, a carefully crafted optimization technique that takes into account the interplay of DVFS, processor customization, cache customization and task mapping is required to quickly find globally optimal or near-optimal solutions. Our key contributions are:

- We propose a comprehensive framework for exploration of a complex design space consisting of four design parameters: DVFS, processor customization, cache customization and task mapping.
- As part of the framework, we propose two analytical estimators that use a minimal number of cycle-accurate simulations, and hence speed up design space exploration.
- Additionally, we propose an optimal algorithm and a heuristic to search the complex, exponential design space for optimal or near-optimal solutions.
- Finally, we demonstrate the effectiveness of our framework compared to an optimization technique consisting of existing techniques using real multimedia applications.

II. Related Work

There is a plethora of work on optimization of application specific MPSoCs, where researchers have considered different objective functions, constraints and design parameters. We report the most relevant works categorized according to the four design parameters described in Section I.

With regards to DVFS, the authors of [2], [7] used it to balance workload across processors connected in a pipeline, in order to reduce their energy consumption. They proposed feedback controllers to monitor the occupancy levels of buffers in the pipeline, and either increased or decreased the v-f level of a processor accordingly. Chen et al. [9] also considered a pipeline of processors with the availability of DVFS; however, they minimized the energy consumption of the system under an end-to-end application deadline using quadratic programming. For processor customization, Bonzini et al. [6] studied the effects on energy consumption and performance due to addition of custom instructions in an ASIP. They built an estimation model for a simplescalar-like processor to quickly evaluate different custom instructions. In [5], the authors characterized the energy benefits of extending the baseline instruction set architecture of an FPGA based soft processor. Lin et al. [19] targeted multiobjective optimization of an ASIP where custom instructions are added considering area and energy consumption. They used mixed integer linear programming (MILP) for an optimal solution and a simulated annealing based heuristic for a near-optimal solution.

Using cache customization, the authors of [10], [13] explored the design space of a cache (cache size, line size, associativity) to select a cache configuration with minimum energy consumption. The authors proposed a heuristic to quickly search through complex design space of cache configurations for a near-optimal solution. Rawlins et al. [22] targeted runtime adaptation of L1 data cache to minimize energy consumption of a heterogeneous MPSoC architecture.

Jung et al. [18] customized an MPSoC, where custom instructions and different v-f levels were used for the ASIPs in the system. They employed MILP to find the design point with minimum dynamic energy consumption under an area constraint. Ruggiero et al. [23] considered an MPSoC with variable number of processors and DVFS. They used a design space exploration algorithm to determine the optimal number of processors and v-f levels for a given application to minimize the MPSoC's power consumption under quality of service constraints. The authors of [3] considered resource allocation and voltage selection problem in an MPSoC. They minimized MPSoC's energy consumption with the use of integer programming and constraint programming. Lu et al. [20] considered the problem of task mapping/scheduling and DVFS in homogeneous MPSoCs. They proposed a processor utilization based algorithm for task mapping and exploited the slacks available in periodic tasks to minimize energy consumption. Sun et al. [12] proposed an iterative algorithm to select custom instructions for ASIPs in an MPSoC along with the mapping and scheduling of tasks to maximally improve performance under an area constraint. A dynamic programming based algorithm was introduced in [8] to find optimal mapping of tasks on ASIPs of an MPSoC under a period constraint, where custom instructions for ASIPs and interval-based mapping were considered. The works in [15], [16], [25] considered a pipeline of ASIPs for multimedia applications. They maximized performance improvement per unit area [25] or minimized area under performance constraints [15], [16] while exploring custom instructions and cache configurations. Pruning algorithms, heuristics, and integer linear programming based approaches were proposed in these works.

It is clear that none of the above works considered combined use of DVFS, processor customization, cache customization, and task mapping, which has the potential to save significant amounts of energy (see Figure 1). To the best of our knowledge, our work is the first to use these techniques together for energy minimization under performance and area constraints in application specific MPSoCs for multimedia.

III. Problem Formulation

MPSoC architecure. In this paper, we target application specific MPSoCs that consist of customizable processors, which can be realized with the use of ASIPs. As shown in Figure 2, each processor has a private cache and local memory, and communicates with other processors via dedicated communication buffers (for example, FIFO queues). Each processor can be customized by both extending its baseline instruction

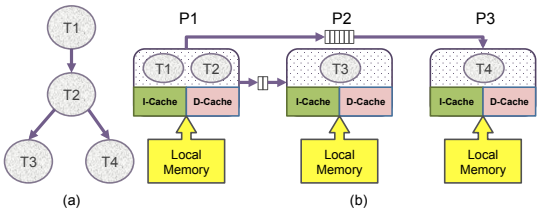

Fig. 2: (a)Task graph (b)MPSoC architecture.

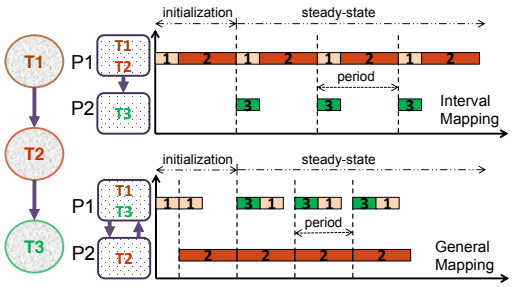

Fig. 3: Different task mappings on an MPSoC.

set architecture (with the addition of custom instructions) and customizing its cache (size, line size, etc.). Additionally, each processor can operate at several discrete v-f levels.

Application model. The target application domain comprises of multimedia applications, which contain compute-intensive sub-kernels or tasks that are executed repeatedly. We represent these applications as directed acyclic graphs[1], where vertices represent tasks and edges represent communication between the tasks. The tasks are mapped to the processors, and then buffers are instantiated only between those processors whose mapped tasks need to communicate data. Benoit et al. [4] categorized mapping of a task graph on an MPSoC with fixed number of processors into: one-to-one mapping, where only a single task is mapped to a processor; interval based mapping, where only adjoining tasks are mapped to a processor; and, general mapping, where no restrictions are placed at all. The type of task mapping determines the placement of the communication buffers between the processors in an MPSoC. In this paper, we use general mapping because it offers greater flexibility and has the potential to reach a better solution (explained later). Once the tasks are mapped, the MPSoC executes those tasks in the form of a virtual pipeline because multimedia applications inherently benefit from a pipelined execution [24].

Figure 3 illustrates mapping of a task graph to a two processor MPSoC using interval and general mappings. In interval mapping, tasks $T1$ and $T2$ are mapped to the first processor while $T3$ is mapped to the second processor. The execution of the processors is similar to a virtual pipeline with two stages. During an iteration of the pipeline, all the tasks mapped on a processor are executed once. The period of the virtual pipeline is equal to the maximum latency from all of its stages in the steady-state, as marked in Figure 3. In general mapping, tasks $T1$ and $T3$ are mapped to the first processor and $T2$ is mapped to the second processor to better balance the workload. In this case, the period is determined by $P2$ which is smaller compared to the period from interval mapping. The price is paid in terms of a longer "initialization" period; however, this is done only once at the start of the application. Note that the "initialization" schedule (for example, execution of $T1$ twice before the execution of $T3$) and "steady-state" schedule (for example, execution of $T3$ followed by $T1$) for any general mapping can be produced using software pipelining [4].

Problem Statement. In the MPSoC architecture and application model described above, each processor has a number of cache configurations available for it. Each task can be accelerated with a set of custom instructions, and thus each task has multiple implementations corresponding to different sets of custom instructions that can be used for it. Each set of custom instructions for a task has an additional area cost. Additionally, each task can be executed at one of the available v-f levels. The latency and energy consumption of a task then depends on the cache configuration of the processor on which it is mapped, and the set of custom instructions and v-f level selected for it. The area of the baseline processor, additional custom instructions (from all the tasks mapped on the processor) and the cache configuration determine the total area of the custom processor. The area of the MPSoC is then the summation of the area of all the processors and the communication buffers. Likewise, energy consumption of the MPSoC is the addition of the energy consumption of the processors (including custom instructions, caches and local memories) and the communication buffers. Putting it all together, the optimization problem can be formally stated as follows: *Given an application task graph, several discrete v-f levels for each task, different sets of custom instructions for each task, different cache configurations for each processor, a steady state period constraint, and an area constraint, the goal is to minimize the total energy consumption of the MPSoC under the provided constraints.* To solve this optimization problem, one needs to search the resulting design space for: (1) the optimal number of processors and mapping of the tasks on them, (2) optimal cache configuration for each of the individual processors, and (3) optimal set of custom instructions and v-f level for each of the tasks. It is important to note that our optimization problem cannot be solved naively because of its exponential complexity that results from all the possible combinations of v-f levels, sets of custom instructions, cache configurations and task mappings.

[1]Cyclic graphs are converted to acyclic graphs by graph unfolding [30].

IV. Proposed Framework

We propose a framework, shown in Figure 4, to solve the optimization problem described in the last section. Our framework integrates three components. The profiler component uses a cycle-accurate simulator to produce profiling information for all the application tasks. Next, the profiling information is exploited by the estimation component to estimate the steady-state latency and energy consumption of the application tasks. Finally, the design space exploration component searches for an optimal or near optimal design point. The following paragraphs explain these components in more detail.

A. Profiler

The input to the profiler consists of the following:

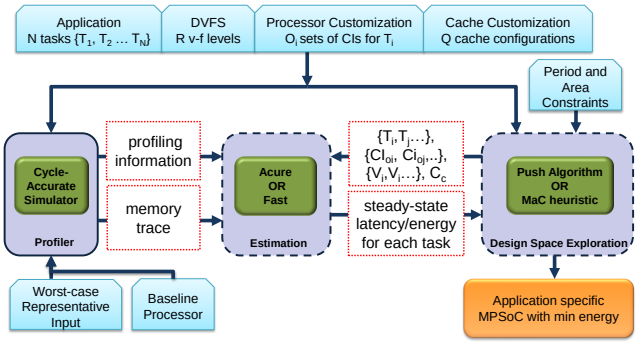

Fig. 4: Framework Overview.

- A multimedia application and its task graph, represented as $\{T_1, T_2, ..., T_N\}$.
- Baseline processor and input data representative of the worst-case.
- R v-f levels for each task, represented as $\{V_1, ..., V_R\}$.
- O_i sets of custom instructions for task T_i, represented as $\{CI_{i1}, ..., CI_{iO_i}\}$. For a task T_i, CI_{i1} refers to the use of only baseline processor without any custom instructions (zero additional area).
- Q cache configurations for the processors, represented as $\{C_1, ..., C_Q\}$.

The profiler uses a cycle-accurate simulator to profile all the possible implementations of a task, where an implementation refers to a combination of a set of custom instructions and a cache configuration with the highest v-f level in the baseline processor. For example, for task T_i, $O_i \times Q$ simulations, all at the highest v-f level, are run to capture its latency, power consumption and memory trace for all combinations of sets of custom instructions and cache configurations. During these simulations, input data representative of the worst-case (provided by the designer) is used so that the MPSoC can deliver the required performance at all times when deployed. Estimation of the steady-state latency and power consumption for a task at a v-f level other than the highest level is done in the estimation component of our framework. It is important to note that simulation of all possible task mappings with different v-f levels, sets of custom instructions and cache configurations is not practically feasible due to exponential nature of the design space. Therefore, our profiler uses a minimal number of simulations so as to keep simulation time low.

B. Latency and Energy Estimation

Two estimators are proposed in this section to estimate steady-state latency and energy consumption of a number of tasks mapped on a baseline processor with their corresponding sets of custom instructions and v-f levels, and a cache configuration.

1) Accurate (Acure) Estimator: **Single task.** For a task with a given set of custom instructions and a cache configuration, we estimate its first iteration's latency (L_v) and energy consumption (E_v) at a v-f level v using:

$$\begin{aligned} L_v &= \frac{L_h \times F_h}{F_v} \\ P_v &= \frac{P_h \times (V_v)^2 \times F_v}{(V_h)^2 \times F_h} \\ E_v &= P_v \times L_v \end{aligned}$$

where L_h and P_h are the latency and power consumption at the highest v-f level, captured in the profiler component. Given the first iteration's latency and energy consumption at a certain v-f level, the steady-state latency and energy consumption at the same v-f level depends primarily upon the cache configuration. During repeated execution of a task, some of the cache misses from the first iteration may become hits in subsequent iterations due to reuse. We define "local miss" and "global miss" to distinguish between those cache misses. Let M be the sequence of memory requests accessed in an iteration of a task. Let m be a memory request in M and let sm be the cache set that m maps to. If M is simulated in isolation starting with an empty cache and the first reference to m results in a cache miss, then m is classified as: a) *Local miss* if there are less than N unique references to cache set sm in M before m, where N is the associativity of the cache and b) *Global miss* if m is not a local miss. If m is a global miss or hit, it is not affected by the cache state at the start of an iteration of M, that is, it behaves the same way in every iteration. However, if m is a local miss, then it may hit or miss in subsequent iterations depending on the cache state at the start of an iteration of M. Intuitively, local misses are the first n cold misses to each cache set that may benefit from reuse later. For an example, assume a direct mapped cache with four sets, $c[0...3]$. Additionally, assume that the memory request pattern of a task is $\{m_0, m_1, m_2, m_3, m_5, m_6, m_7\}$, where $\{m_0\}$ maps to $c[0]$, $\{m_1, m_5\}$ map to $c[1]$, $\{m_2, m_6\}$ map to $c[2]$ and $\{m_3, m_7\}$ map to $c[3]$. Let LM^1 be the set representing memory requests that resulted in local misses during the first iteration of the task, and CS^1 be the set representing cache state at the end of the first iteration. Then, $LM^1 = \{m_0, m_1, m_2, m_3\}$ ($\{m_5, m_6, m_7\}$ are global misses) and $CS^1 = \{m_0, m_5, m_6, m_7\}$. In the steady-state, m_0 from LM^1 will always be a hit while $\{m_1, m_2, m_3\}$ will always result in local misses. Table I illustrates local misses, global misses and cache states across different iterations for the running example.

Iter.	Cache State (CS)	Local Misses (LM)	Global 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 I: Cache state across iterations of a task.

In summary, the steady-state latency will be less than or equal to the first iteration latency because the number of local misses might reduce. The reduction in the number of local misses is $LM_r = \left|CS^1 \cap LM^1\right|$. The steady-state latency (L^{ss}) and energy consumption (E^{ss}) is then estimated using the following equations:

$$\begin{aligned} L^{ss} &= L^1 - (LM_r \times ML) \\ E^{ss} &= E^1 - (LM_r \times ME) \end{aligned}$$

where ML and ME refer to lower-level memory latency and energy per access. L^1 and E^1 refer to the first iteration's latency and energy consumption of a task, including its communication latency and energy respectively. Since the global misses remain constant across iterations and have already been captured in L^1 and E^1, they do not affect steady-state latency and energy consumption. For estimation, both CS^1 and LM^1 are computed by processing the memory trace captured in the profiler component. Note that the steady-state latency and energy consumption of a task can also be computed by simulating it for multiple iterations in the profiler. However, for long running tasks, the simulation time for multiple iterations might be significant. Our estimation technique had errors of less that 1% compared to cycle-accurate simulations of multiple iterations (see Section VI), and hence we did not simulate multiple iterations of a task in the profiler.

Multiple tasks. Now we extend our estimation technique to multiple tasks. We assume that all the tasks are non-preemptible which is a valid assumption [25] for multimedia applications because each task has to process its input data before sending it to the next task. When more than one task is mapped to a processor, then each task can pollute the cache state of other tasks. For the sake of simplicity, we explain our estimation technique with two tasks T_1 and T_2; however it can easily be extended to any number of tasks. Let CS_1^1 and CS_2^1 be the cache states at the end of the first iteration of tasks T_1 and T_2 respectively, and LM_1^1 and LM_2^1 be the sets containing local misses during the first iteration. In steady state, the number of misses reduces for a particular task when its locally missed memory requests survive through the execution of the other task. For a particular cache set sm, we define the operator $\bigodot$ as $m' \bigodot m'' = m''$, if m'' is not null or else $m' \bigodot m'' = m'$. This means that the memory request m'' (when $m'' \neq null$) has replaced m' in the cache set sm. Then, the reduction in the number of local misses for T_1 and T_2 is:

$$\begin{aligned} LM_{r,T_1} &= \left|(CS_1^1 \bigodot CS_2^1) \cap LM_1^1\right| \\ LM_{r,T_2} &= \left|(CS_2^1 \bigodot CS_1^1) \cap LM_2^1\right| \end{aligned}$$

Therefore, the steady-state latency and energy consumption of the two tasks are:

$$L^{ss}_{T_1,T_2} = L^1_{T_1} + L^1_{T_2} - ((LM_{r,T_1} + LM_{r,T_2}) \times ML)$$
$$E^{ss}_{T_1,T_2} = E^1_{T_1} + E^1_{T_2} - ((LM_{r,T_1} + LM_{r,T_2}) \times ME)$$

If two communicating tasks are mapped to the same processor, then they do not need to communicate through a communication buffer. We capture the amount of data transferred (in words) and the latency per word during the first iteration of a task in the profiler component. Given this information, we estimate the communication latency of a task by multiplying the latency per word with the amount of data transferred. A similar approach is used for estimation of the communication energy. Once the communication latencies and energies of the two tasks are available, we subtract them from $L^{ss}_{T_1,T_2}$ and $E^{ss}_{T_1,T_2}$ to account for the saving in communication latency and energy from their mapping on the same processor.

It is clear that our estimation technique allows to calculate the steady-state latency and energy consumption of any number and order of tasks from latency, energy consumption and memory trace of first iterations of the individual tasks. Therefore, we do not simulate all the possible mappings of tasks in the profiler component, which reduces simulation time significantly.

2) Fast Estimator: The computational complexity of estimating steady-state latency and energy consumption in *Acure* estimator depends upon the number of tasks and the size of their memory traces. When the number of complex tasks mapped on a processor increases, *Acure* estimator might become slow for rapid design space exploration. Therefore, in *Fast* estimator, we trade-off the time spent in processing of memory traces (to compute LM^1_i and CS^1_i for a task T_i) with the estimation accuracy.

Single task. Like *Acure* estimator, first of all, the latency and energy consumption of first iteration is estimated at the given v-f level. Afterwards, rather than analyzing the memory trace, we use the first iteration's latency and energy consumption as the steady-state latency and energy consumption of a task.

Multiple tasks. The steady-state latency and energy consumption of two tasks, T_1 and T_2, is computed by adding the steady-state latency and energy consumption of the individual tasks. The communication latency and energy are accounted for in a similar fashion to the *Acure* estimator. The accuracy of *Fast* estimator depends upon the cache behavior. If the reduction in local misses across iterations of a single task or across multiple tasks is significant, then the error will be high.

C. Design Space Exploration

1) Prune and Search (Push) Algorithm: The *Push* algorithm uses two basic operations "prune" and "search" to quickly push itself through the complex design space towards the optimal design point. The "prune" operation prunes certain parts of the design space based on constraints, while the "search" operation finds a partial solution in a subset of the design space. These partial solutions are combined successively to reach the globally optimal design point. Theoretically, the worst-case complexity of the Push algorithm is exponential; practically, it is able to prune a large part of the complex design space by exploiting the constraints.

Algorithm 1 shows the pseudo code of the *Push* algorithm. For ease of understanding, consider that the design space is represented as a tree, which is shown in Figure 5. The parameters of the design space are summarized in the table. For the sake of simplicity, we do not show all the nodes in the design space tree. Note that L_{1111} represents the latency of the task T_1 with custom instruction set CI_1, v-f level V_1 and cache configuration C_1 (a similar notation is used for energy as well). The annotations on edges illustrate the options of the design parameters. Each level i of the tree corresponds to a call of the *Push* procedure, where the algorithm has a partial solution for tasks $T_1, T_2, ..., T_{i-1}$ (their corresponding sets of custom instructions and v-f levels, stored in *map[]*), and the processors (with their corresponding cache configurations, stored in *eProcs*) that have already been mapped with those tasks. Let period, area and energy consumption of the partial solution be *currP*, *currA* and *currE* respectively (stored in *currMetrics*). With this partial solution at level i, the algorithm prunes the subtrees based on constraints (lines 7-9) which are explained later. Note that the *areaPruning*, *periodPruning* and *energyPruning*

Algorithm 1: Push Algorithm

```
1  tasks = {T_1, T_2...T_N};
2  eProcs = {} ;                    // existing processors
3  map[] = {} ;    // map[P] contains tasks mapped
   on P
4  currMetrics = {currA = 0, currP = 0, currE = 0};
5  bestSol = {};
6  PUSH(tasks, eProcs, map, A_c, P_c)
7    if areaPruning(tasks, A_c) then return;
8    if periodPruning(tasks, P_c) then  return;
9    if energyPruning(tasks, bestSol) then return;
10   if tasks ≠ null then
11       T_i ← task i from tasks;
         // map to an existing processor
12       for each P in eProcs do
13           for o = 1 to O_i do // custom instructions
14               for v = 1 to R do // v-f levels
15                   map[P] ← T_i with CI_io and V_v;
16                   currMetrics = metrics(eProcs, map);
17                   if currP ≤ P_c and currA ≤ A_c then
18                       PUSH(tasks, eProcs, map, A_c, P_c);
19                   else
20                       restore currMetrics previous value;
21                       remove T_i from map[P];
         // map to a new processor
22       for c = 1 to Q do // cache configurations
23           for o = 1 to O_i do // custom instructions
24               for v = 1 to R do // v-f levels
25                   nP = new processor with C_c;
26                   eProcs ← nP;
27                   map[nP] ← T_i with CI_io and V_v;
28                   currMetrics = metrics(eProcs, map);
29                   if currP ≤ P_c and currA ≤ A_c then
30                       PUSH(tasks, eProcs, map, A_c, P_c);
31                   else
32                       restore currMetrics previous value;
33                       remove T_i from map[P];
34                       remove nP from eProcs;
35   if tasks ≠ null then
36       return failure;
37   else
38       update bestSol if required; return;
```

functions return true when the subtrees are pruned. If the pruning is unsuccessful, then the algorithm maps task T_i either to one of the existing processors (lines 13-22) or a new processor (lines 24-36) ensuring the area and period constraints are met, and then moves on to the next task by calling the Push procedure. Here, the algorithm uses the *metrics* function (lines 17, 30) to calculate the area, period and energy consumption of the new mapping using either the *Acure* or *Fast* estimator from Section IV-B. This process is repeated until all the tasks have been mapped or no more tasks can be mapped given the area and period constraints (lines 37-38). Mapping of all the tasks means a new solution is found, which is used to update the best solution seen so far (stored in *bestSol*, line 40) if the new solution's energy consumption is better than the best solution.

For example, at level 1 in Figure 5, the current node indicates that CI_{12} set of custom instructions and V_1 v-f level have been selected for task T_1 which is mapped to a processor with C_1 cache configuration. The algorithm reaches the current node only after traversing the entire left subtree for the task T_1. The metrics for partial solution at level 1 are in *currP*, *currA* and *currE*, while *bestE* is the energy consumption of the best solution seen so far. From the current node, task T_2 can be mapped either to the existing processor (left subtree, edge annotated as e) or a new processor (right subtree, edge annotated as n). The algorithm can prune the subtrees based on the following observations:

Area constraint. If all the remaining tasks are mapped to existing processors without any custom instructions (use of baseline processor only), then the total area will still be equal to *currA* because no

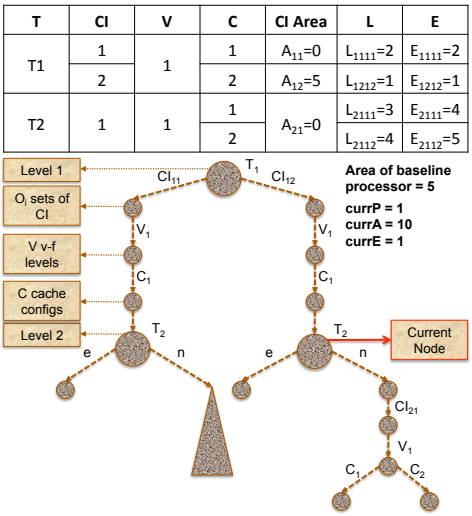

Fig. 5: Illustration of Push algorithm.

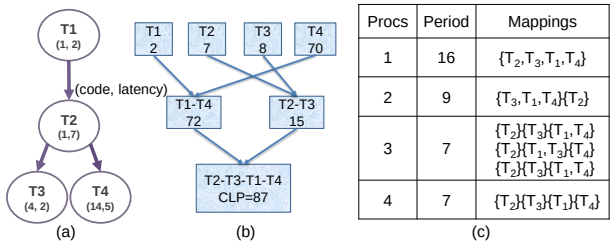

Fig. 6: Illustration of map stage: (a)Task graph (b)Task sequencing (c)Different task mappings.

additional area will be used for the unmapped tasks. If *currA* violates the area constraint A_c, then it is safe to prune the subtree. In the running example, the entire subtree will not be pruned for $A_c \geq 10$.

Period constraint. A lower bound on the period can be estimated by mapping remaining tasks to separate processors and using their lowest latency implementations (i.e L_i^{min}), that is, $max(currP, L_i^{min}, ..., L_N^{min}))$. In the running example, *currP* = 1 and $L_2^{min} = L_{2111} = 3$, so lower bound on period equals max(1, 3) = 3. If $P_c = 2$, then the entire subtree will be pruned.

Lowest possible energy consumption (LPE). For all the unmapped tasks, *LPE* is estimated as the summation of their minimum energy consumptions, less their communication energies. That is, *LPE* refers to the scenario where minimum energy implementations of all the tasks are used with no energy spent in data communication. If $(LPE + currE)$ is greater than the energy consumption of the best solution seen so far, then the entire subtree can be pruned as the partial solution is already worse than the best solution. In the running example, if the best solution's energy is 4, then the entire subtree will be pruned because $LPE + currE = 5$.

2) Map and Customize (MaC) Heuristic: To better handle the exponential complexity, we propose a two stage algorithm consisting of the "map" and "customize" stages. In the "map" stage, candidate task mappings are produced considering a homogeneous MPSoC. In the "customize" stage, already produced task mappings are used to customize the MPSoC with the selection of custom instructions, v-f levels and cache configurations. One can think of the "map" stage as application-level balancer and the "customize" stage as system-level balancer, which work in synergy to find a near-optimal solution.

Map stage. In this stage, a homogeneous MPSoC with variable number of processors is considered. The input to this stage consists of $tasks = \{T_1, T_2...T_N\}$, and their code sizes and latencies on a baseline processor with smallest cache configuration, lowest v-f level and without any custom instructions. The goal is to generate a set of task mappings that will possibly lead to a globally optimal solution.

Ideally, we would like to combine tasks that will complement each other in terms of both the latency and the energy consumption. Let us categorize tasks based upon their latencies as *short* and *long* tasks. Likewise, we categorize tasks based upon their code size as *small* and *big* tasks. Our intuition is that a *small* task well complements a *big* task in terms of the cache configuration, while a *short* task well complements a *long* task in terms of latency. Thus, we propose to combine *small-short* tasks with *big-long*, and *small-long* with *big-short* tasks, because they will result in a complementary effect in their combined latency and energy consumption. If $code_i$ and L_i is the code size and latency of a task T_i respectively, then we define code-latency product for a set of tasks $\{T_i, T_j, ...\}$ as $CLP(i, j, ...) = \sum_{x=\{i,j,...\}} code_x \times L_x$. If the tasks are sorted in ascending order according to CLP metric, then the *smallest-shortest* task will set the lower bound while the *biggest-longest* task will set the upper bound. All the other tasks will be dispersed in between these bounds. We use the CLP metric to obtain a task sequence $\Gamma(N)$ where tasks with complementary characteristics are adjacent to each other. Given N tasks, we consider them as N subsequences and compute the CLP metric for each of them, followed by sorting them in ascending order according to CLP. Then, we combine the i-th subsequence with $(N+1-i)$-th subsequence, that is, combining subsequences with complementary characteristics. After the first run, a total of $\lceil \frac{N}{2} \rceil$ subsequences are obtained. We repeat the above process till only one subsequence is left, which is the final sequence of tasks with complementary characteristics. An example of task sequencing for four tasks is shown in Figure 6. The annotation in each node of the task graph on the left-hand side is $(code, latency)$, while the right-hand side illustrates the number of task subsequences and their corresponding CLP metrics for each run. The final sequence $\Gamma(N) = \{T_2, T_3, T_1, T_4\}$ where $\{T_2, T_3\}$ subsequence represents a *small-long* and *big-short* combination, while $\{T_1, T_4\}$ represents a *small-short* and *big-long* combination.

After obtaining the task sequence using the CLP metric, we proceed to enumerate different mappings of the task sequence considering variable number of processors as follows:

$$map_i = \forall T_j \mapsto \{P_1, P_2, ..., P_i\} : 1 \leq j \leq N, \quad i \leq N$$

In essence, map_i represents mapping of all the tasks onto i number of processors. We model the enumeration of task mappings as a chains-on-chains problem [21], where the aim is to map j tasks on i processor such that the mapping is load balanced, that is, the period is minimized. Our intuition is that a balanced mapping at this stage will possibly lead to better customization in the later stage. Although there exist several polynomial-time algorithms for solving the chains-on-chains problem [21], we use a dynamic programming based solution from [4]. Figure 6(c) illustrates different task mappings for the final task sequence of Figure 6(b) with their optimal periods.

Customize stage. The algorithm for customization of the MPSoC for different task mappings is shown in Algorithm 2. For a task mapping (stored in map_i), some area from the total available area (stored in *sArea*) is allocated to each processor (stored in *A[P]*) proportional to its period (lines 9-10). This is based on the intuition that a processor with higher period may have to use complex custom instructions and bigger cache configuration to reduce its period. Given the allocated area for a processor, we employ a modified version of the *Push* algorithm, *PushM* (line 12), to find the optimal set of custom instructions and v-f levels for all the tasks of the given processor, and its optimal cache configuration. The *PushM* algorithm uses lines 13–22 of the *Push* algorithm, that is, an optimal solution is searched for the given (existing) processor only, ignoring the addition of new processors. The custom processor returned by the *PushM* algorithm is added to the best solution for the current task mapping (stored in *bestSol[i]*), while the area of the custom processor is subtracted from the total available area (line 16). This process is repeated until all the processors have satisfied the period constraint (line 11-16) or all the processors currently in *vProcs* could not satisfy the period constraint (lines 17-18). Finally, the algorithm returns the task mapping and customized MPSoC with minimum energy consumption from all of the input task mappings (line 22).

A working example of the algorithm is shown in Figure 7 for one of the task mappings from Figure 6. The first column reports the run of the algorithm while the rest of the columns report the area

Algorithm 2: Customize MPSoC

```
1  maps = {map_1, map_2, ..., map_N};
2  bestSol[] = {} ;                    // bestSol[i] for map_i
3  while maps ≠ {} do
4      vProcs[] ← all the processors from map_i;
5      tasks[] ← tasks mapped to processors in vProcs;
6      sArea = A_c;
7      while vProcs ≠ {} do
           // allocate area proportional to
           period
8          for each P in vProcs do
9              A[P] ← proportion of sArea using P's period;
10         for each P in vProcs do
11             r = PushM(tasks[P], vProcs[P], {}, A[P], P_c);
12             if r ≠ failure then
13                 update bestSol[i][P] ;  // solution for
                   P
14                 remove P from vProcs;
15                 sArea −= area returned by PushM();
16         if all P in vProcs failed then
17             break;
18     if vProcs ≠ {} then
19         bestSol[i] ← failure;
20     remove map_i from maps;
21 return minimum energy solution from bestSol;
```

allocated to each processor and the total available area. For example, in the first run, $P1$ and $P2$ are allocated an area of 8.75 and 2.5 respectively from total available area of 20. During the first run, the *PushM* algorithm succeeds for $P2$ and fails for $P1$ and $P3$. Thus, the area of the custom processor for $P2$ (1.5) is subtracted from the total available area, which is redistributed among $P1$ (9.25) and $P3$ (9.25) for the next runs. In the second and third runs, the *PushM* algorithm successfully customizes $P3$ and $P1$ under the allocated area and period constraint.

V. Experimental Setup

We used a commercial environment from Tensilica [1] to realize application specific MPSoCs. We used Xtensa LX2 processors and accompanying toolset RD-2011.2 which includes Xtensa ISS cycle-accurate simulator, XTMP multiprocessor simulation environment, and XPRES compiler. For each application task, we used XPRES compiler to generate different sets of custom instructions, which consist of any combination of FLIX, fused, vector and specialized instructions. At least, five sets of custom instructions were generated per application task. We used five different instruction cache configurations by changing cache sizes from 1 KB to 16 KB. Although we only tested our framework with instruction cache configurations, a designer can easily apply it to data cache configurations. For each processor, we used five different frequency levels ranging from 533 MHz to 1.5 Ghz with their corresponding voltage levels. The Xt-Xenergy tool from Tensilica is used to compute the energy consumption of a processor at the highest v-f level, including its caches and local memory for a given 90nm technology. The area of the processor and its caches and local memories is also obtained from Tensilica toolset. For communication buffers, we estimated their area and energy consumption using CACTI [27].

Task Mapping: {T_2} {T_3} {T_1,T_4} A_c = 20 P_c = 5

Run	P1 (period = 7)		P2 (period = 2)		P3 (period = 7)		sArea (20)
	A[P1]	PushM	A[P2]	PushM	A[P3]	PushM	
1	8.75	--	2.5	1.5	8.75	--	18.5
2	9.25	--	--	1.5	9.25	9.0	9.5
3	9.5	9.4	--	1.5	--	9.0	0.1

Fig. 7: Illustration of customize stage.

Mapped Tasks	Latency Error [%]		Energy Error [%]	
	Acure	Fast	Acure	Fast
1	0.54	1.72	0.69	2.92
2	0.61	1.99	0.82	3.10
4	0.75	4.82	0.97	6.65
8	0.86	8.31	1.04	9.98
16	0.91	11.20	1.22	13.52
20	1.07	13.64	1.29	15.71

TABLE II: Maximum error in the Acure and Fast estimators.

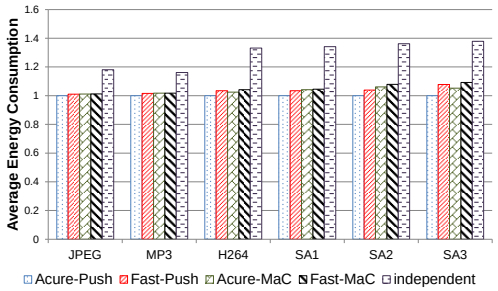

Fig. 8: Comparison of different optimization techniques, normalized to Acure-Push.

For evaluation, we used both real multimedia and synthetic applications. We partitioned the multimedia applications into their tasks as mentioned in [8] for JPEG encoder (5 tasks) and MP3 encoder (5 tasks), and in [17] for H.264 encoder (7 tasks). For synthetic applications, we generated task graphs using TGFF [11], and used kernels from Mibench [14] and StreamIt [28] as tasks in those task graphs. We chose ten kernels from Mibench and StreamIt where a reasonable trade-off in performance and area was observed for different custom instructions and cache configurations. We created three synthetic applications: SA1 with 10 tasks, SA2 with 15 tasks and SA3 with 20 tasks to evaluate the scalability of the proposed framework. Given the above setup, the design space of each application contained *at least a billion design points*. All the experiments were conducted on an Intel Xeon 2.53 Ghz processor with 16 GB memory.

VI. Results

Table II summarizes the error observed in computation of the steady-state latency and energy using the *Acure* and *Fast* estimators, compared to cycle-accurate simulation. The table reports the *maximum error* observed, from amongst all the applications, when different number of tasks are mapped on a processor with different sets of custom instructions and cache configurations. The errors observed in *Acure* estimator are very low and remain constant across the number of tasks. On the other hand, the errors in the *Fast* estimator increase with the number of tasks, reaching to a maximum of 15.71%. This is because the cache behavior is disrupted when a greater number of tasks are mapped to the same processor. Thus, the *Acure* estimator will better guide the design space exploration algorithms than the *Fast* estimator. In our framework, the *Acure* and *Fast* estimators can either be combined with the *Push* algorithm or *MaC* heuristic, which results in four possible optimization techniques. Additionally, we also use the "independent" optimization technique described in Section I, where optimal solutions are sought for in individual optimization problems, and all the possible ways of combining individual optimal solutions are exhausted. Since the "independent" technique can be constructed from existing techniques, we use it as the state-of-the-art for comparison purposes. Note that the use of *Acure* estimator with the *Push* algorithm will yield the most optimal solution from amongst all the five optimization techniques. Since our design spaces contain at least billion points, it is not practical to apply all the possible period and area constraints. We used Latin Hypercube Sampling to generate 50 uniformly distributed tuples for each application, where each tuple represents a combination of area and period constraints.

Figure 8 plots the average energy consumptions of the solutions obtained from various optimization techniques, normalized to the energy consumption of the solution from *Acure-Push* technique under

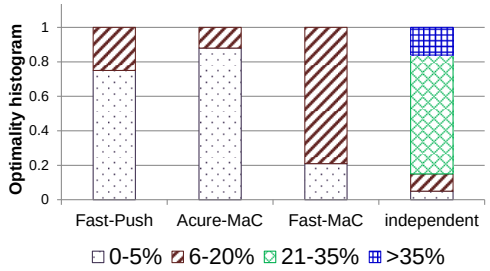

Fig. 9: Error distribution in different optimization techniques for SA3 application.

Apps.	# Tasks	Acure Push	Fast Push	Acure MaC	Fast MaC	Ind.
JPEG	5	910	770	4	3	75
MP3	5	1181	890	12	9	75
H.264	7	1492	1028	84	17	492
SA1	10	3662	2621	299	87	991
SA2	15	5021	3267	450	230	1982
SA3	20	7732	4919	785	540	3811

TABLE III: Exploration time (in secs) of optimization techniques.

50 different constraints. From amongst all the applications, on average, *Fast-Push*, *Acure-MaC* and *Fast-MaC* found 8%, 6% and 9% increase in energy consumption compared to *Acure-Push* respectively. It is noteworthy that the *Acure-MaC* outperformed *Fast-Push* for H.264 and SA3 applications even though the *Push* algorithm is optimal. This is due to higher estimation errors in the *Fast* estimator compared to the *Acure* estimator for H.264 and SA3 applications, which misguided the *Push* algorithm. Out of all the five optimization techniques, the "independent" technique performs the worst; on average, it resulted in up to 37.9% increase in energy consumption. More importantly, our most sub-optimal heuristic, *Fast-MaC*, improved the energy consumption of the solutions by up to 76.25% on average when compared to the "independent" technique. The maximum increase in energy consumption from various optimization techniques compared to *Acure-Push* yielded similar findings. At maximum, the "independent" technique resulted in solutions that were up to 57.1% higher in energy consumption, while *Fast-Push*, *Acure-MaC* and *Fast-MaC* resulted in solutions that were 16%, 13% and 19% higher in energy consumption respectively. Figure 9 plots the distribution of energy difference between the solutions obtained from those technqiues with the one obtained from Acure-Push for SA3 application. It is evident that 85% of the solutions from the independent technique have more than 20% increase in energy with respect to Acure-Push, while all the solutions from Fast-Push, Acure-MaC and Fast-MaC have less than 20% increase in energy.

Table III reports the exploration time of all the five optimization techniques in seconds. For each application, these exploration times are calculated by taking an average of the total time spent in finding solutions for all the 50 constraints. It is evident that the optimization techniques with the *Fast* estimator are faster than the *Acure* estimator. More importantly, the *MaC* heuristic is at least 9 times (SA3 application), faster than the *Push* algorithm. It is also noteworthy that the "independent" technique's exploration time is at least 7 times more than our fastest heuristic *Fast-MaC* even with 76.25% sub-optimal solutions.

VII. Conclusion

We have proposed a framework to synthesize an energy-aware application specific MPSoC. We synergistically explore the complex interplay of DVFS, custom instructions, cache configurations and task mapping. Our framework uses two analytical estimators, the *Push* algorithm for optimal solutions and *MaC* heuristic for near-optimal solutions. The experimental results show that the *MaC* heuristic is at least 14 times faster than the *Push* algorithm with average errors of up to 9%. Also, our *MaC* heuristic reduces energy consumption by up to 76.25% on average with 7 times lower exploration time compared to the "independent" optimization technique.

Acknowledgment

This work was partially supported by Singapore Ministry of Education Academic Research Fund Tier 2 MOE2009-T2-1-033, MOE2012-T2-1-115.

References

[1] Xtensa processor, Tensilica inc. http://www.tensilica.com.

[2] Alimonda et al. A feedback-based approach to dvfs in data-flow applications. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 2009.

[3] Benini et al. Allocation, scheduling and voltage scaling on energy aware MPSoCs. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. 2006.

[4] Benoit et al. Mapping pipeline skeletons onto heterogeneous platforms. In *ICCS 2007*.

[5] Biswas et al. Performance and energy benefits of instruction set extensions in an FPGA soft core. In *VLSID*, 2006.

[6] Bonzini et al. A study of energy saving in customizable processors. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*. 2007.

[7] Carta et al. A control theoretic approach to energy-efficient pipelined computation in MPSoCs. *ACM Trans. Embedded Comput. Syst.*, 2007.

[8] Chen et al. Customized MPSoC synthesis for task sequence. In *SASP*, 2011.

[9] Chen et al. Energy optimization with worst-case deadline guarantee for pipelined multiprocessor systems. In *DATE*, 2013.

[10] Chuanjun et al. Cache configuration exploration on prototyping platforms. In *Rapid Systems Prototyping, 2003. Proceedings. 14th IEEE International Workshop on*, 2003.

[11] Dick et al. Tgff: task graphs for free. In *Proceedings of the 6th international workshop on Hardware/software codesign*, 1998.

[12] Fei et al. Synthesis of application-specific heterogeneous multiprocessor architectures using extensible processors. In *VLSI Design, 2005.*

[13] Gordon et al. Automatic tuning of two-level caches to embedded applications. DATE, 2004.

[14] Guthaus et al. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization. WWC-4. IEEE International Workshop on*, 2001.

[15] Javaid et al. Optimal synthesis of latency and throughput constrained pipelined MPSoCs targeting streaming applications. In *CODES+ISSS*, 2010.

[16] Javaid et al. Rapid design space exploration of application specific heterogeneous pipelined multiprocessor systems. *IEEE TCAD*, 2010.

[17] Javaid et al. Low-power adaptive pipelined MPSoCs for multimedia: an h.264 video encoder case study. In *DAC*, 2011.

[18] Jung et al. Energy-aware instruction-set customization for real-time embedded multiprocessor systems. ISLPED, 2009.

[19] Lin et al. Exploring custom instruction synthesis for application-specific instruction set processors with multiple design objectives. ISLPED, 2010.

[20] Lu et al. Scheduling and mapping of periodic tasks on multi-core embedded systems with energy harvesting. In *IGCC*, 2011.

[21] Pınar et al. Fast optimal load balancing algorithms for 1d partitioning. *Journal of Parallel and Distributed Computing*, 2004.

[22] Rawlins et al. An application classification guided cache tuning heuristic for multi-core architectures. In *ASP-DAC*, 2012.

[23] Ruggiero et al. Application-specific power-aware workload allocation for voltage scalable mpsoc platforms. In *ICCD*, 2005.

[24] Shee et al. Heterogeneous multiprocessor implementations for jpeg:: a case study. CODES+ISSS, 2006.

[25] Shee et al. Design methodology for pipelined heterogeneous multiprocessor system. DAC, 2007.

[26] Shin et al. Power optimization of real-time embedded systems on variable speed processors. In *ICCAD*, 2000.

[27] Tarjan et al. Cacti 4.0. *HP laboratories, Technical report*, 2006.

[28] Thies et al. Streamit: A language for streaming applications. In *Compiler Construction*. Springer, 2002.

[29] Wolf et al. Multiprocessor system-on-chip (MPSoC) technology. *IEEE TCAD*, 2008.

[30] Yang et al. Heuristic algorithms for scheduling iterative task computations on distributed memory machines. *Parallel and Distributed Systems, IEEE Transactions on*, 1997.