Design Space Exploration of Instruction Set Customizable MPSoCs for Multimedia Applications

Unmesh D. Bordoloi[1] Huynh Phung Huynh[2] Tulika Mitra[3] Samarjit Chakraborty[4]
[1]Linköpings Universitet, Sweden, [2]A*STAR Institute of High Performance Computing, Singapore
[3]National University of Singapore, [4]TU Munich, Germany

***Abstract*— Multiprocessor System-on-Chips or MPSoCs in the embedded systems domain are increasingly employing multiple customizable processor cores. Such cores offer higher performance through application-specific instruction-set extensions without sacrificing the flexibility of software solutions. Existing techniques for generating appropriate custom instructions for an application domain are primarily restricted to specializing a single processor with the objective of maximizing performance. In a customizable MPSoC, in contrast, the different processor cores have to be customized in a synergistic fashion to create a heterogeneous MPSoC solution that best suits the application. Moreover, such a platform presents conflicting design tradeoffs between system throughput and on-chip memory/logic capacity. In this paper, we propose a framework to systematically explore the complex design space of customizable MPSoC platforms. In particular, we focus on multimedia streaming applications, as this class of applications constitutes a primary target of MPSoC platforms. We capture the high variability in execution times and the bursty nature of streaming applications through appropriate mathematical models. Thus, our framework can efficiently and accurately evaluate the different customization choices without resorting to expensive system-level simulations. We perform a detailed case study of an MPEG encoder application with our framework. It reveals design points with interesting tradeoffs between silicon area requirement for the custom instructions and the on-chip storage for partially-processed video data, while ensuring that all the design points strictly satisfy required QoS guarantees.**

I. Introduction

Nowadays, there is a tremendous interest in Multiprocessor System-on-Chips (MPSoCs) specifically targeted towards implementing multimedia applications. Designs based on MPSoC platforms are today ubiquitous and range from mobile phones to set-top boxes. Such products are associated with high demands on flexibility, low design costs and stringent time-to-market constraints. On the other hand, they must also satisfy the high performance requirements of the target application domain. In order to strike the right balance between flexibility and performance, MPSoC platforms come with instruction-set extensible processor cores that can extend the base instruction set with special instructions. These instructions capture frequently executed computation patterns of an application. Some examples of commercial instruction-set extensible processors include Lx, ARC^{TM} core, Xtensa and Stretch.

Customizing processor cores in an MPSoC with extensible instruction sets can lead to additional logic gates in the processor's core, but potentially significant savings in on-chip buffer sizes and performance. A designer would be typically interested in identifying how the performance and on-chip buffer requirements change with different choices of custom instructions on an MPSoC platform. However, as each task can be enhanced with multiple extensible instructions, identifying such tradeoffs necessitates effective traversal of a huge search space. Secondly, MPSoCs are increasingly targeted towards multimedia streaming applications that are characterized by high variability in their execution times. As such, conventional methods to select custom instructions fail to return accurate results. In this paper, we will propose techniques to address these issues for streaming applications.

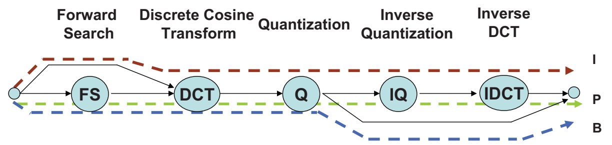


Fig. 1. The task graph of an MPEG-2 encoder.

A. Overview of our Scheme

A streaming application processes an infinite stream of data items or events. The application may be modeled as a task graph where the nodes are abstract representations of the different functions or blocks of code. Figure 1 shows one such task graph implementing an MPEG-2 encoder. The events processed by such an application are *typed* and the processing of the events invokes different paths in the task graphs. The MPEG-2 encoder application for example processes three types of frames to encode video information: *type I*, *type B* and *type P*. Depending on the frame type, the encoder executes different subtasks to compress the frame as illustrated in Figure 1. The *type P* events triggers all the subtasks of the encoder, while the *type I* triggers the tasks DCT, Q, IQ, and IDCT; and *type B* triggers only the tasks FS, DCT, and Q. As a result, each of these three event types demand different amounts of resource for execution. As illustrated in this example, in streaming applications the execution time associated with any event might vary considerably, depending on the type of the event. As a result, the choice of custom instruction will have to consider this variability in execution times with respect to the event types in order to achieve maximum performance gain within given area constraints. Conventional methods ([17], [11]) to select custom instructions ignore this variability.

In this paper, we propose a technique to effectively bound the worst-case resource requirements of the application thereby capturing the variability associated with streaming applications. Towards this, our proposed mechanism exploits the application-specific task execution patterns which are given in the form of a transition system. The transition system describes the possible compositions of any event stream, and our technique takes into account all possible sequences of events that might arrive based on this specification. This allows us to effectively bound the worst-case and best-case execution times, and thereby accurately compute a number of relevant performance metrics.

In our scenario, where we consider multimedia applications running on an MPSoC platform, there are two relevant performance metrics. First, we estimate the on-chip buffer size, which is an important metric for an MPSoC platform. Second, we evaluate the jitter of the processed stream as a measure of QoS because higher burstiness at the output implies a poor quality of the application.

To estimate these performance metrics, we rely on a formal mathematical framework, based on the theory of network calculus [13]. As a part of the input specification, we are given *bounds* on the arrival rate of the event stream. Given (i) these timing properties of the streams in the form of bounds, and (ii) the bounds on resource requirements of the events that we computed from the transition system, our scheme utilizes a series of algebraic equations based on [13] to compute the relevant performance metrics. The details of this framework will be described in Section III.

In the above discussion, we gave a brief overview of our technique which evaluates performance metrics corresponding to a particular set of custom instruction by capturing the variability of the execution times of a streaming application. In practice, each task of an application can be customized using multiple choices, thereby leading to a combinatorially large number of choices in the overall design space. Each of these design points captures a tradeoff between the additional logic for the custom instructions and the performance metrics. In this paper, we also propose a scheme to efficiently search the design space of customization choices to expose the tradeoffs. Instead of resorting to heuristics like evolutionary algorithms [9], which are not guaranteed to return optimal results, we propose a Branch and Bound method with provably optimal solutions.

B. *Related Work*

In recent years, lot of research has been devoted to custom instruction selection techniques so as to optimize either performance or hardware area. However, most of them are restricted to analyzing single processor systems. Various approaches proposed along this line of work include techniques based on dynamic programming [1], 0-1 Knapsack [8], greedy heuristic [7], and ILP [14].

On the other hand, selecting custom instructions for MPSoCs has still not received sufficient attention, despite MPSoCs being equipped with extensible cores. In [17], a technique is proposed to select appropriate custom instruction configuration for each task in the task graph of the application which is mapped into an MPSoC. Recently, [11] presented a design flow to customize streaming application on heterogeneous pipelined multiprocessor systems. However, both these approaches have certain limitations. First, they do not account for buffers between any two processors, the size of which is one of the main design parameters in MPSoC design, and determines on-chip area. Second, none of the previous research efforts exploited specific characteristics of streaming applications (e.g., bursty arrival patterns and the variability in execution demands) in order to achieve accurate results. In this paper, we present a systematic approach to identify the quality of different design points by capturing the variability

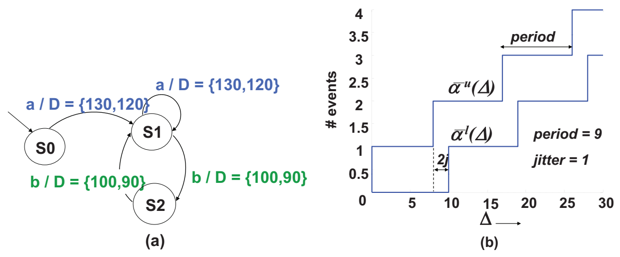


Fig. 2. Specifications of a streaming *application* model: (a) Transition System and (b) Arrival Curve.

in execution times inherent in streaming applications. Finally, unlike [11] or [17], we do not resort to heuristics to accelerate the running times of the proposed algorithms. Our solutions are provably optimal as well as fast because we utilize effective pruning techniques based on a Branch and Bound strategy.

Note that in this work we focus on the custom instruction *selection* problem. This approach is along the lines of research efforts discussed above. Hence, we assume that a library of custom instruction candidates is given. Such a library of custom instructions may be *enumerated* by extracting frequently occurring computation patterns from the data flow graph of the program [6], [16], [20]. Recently, [2], [15], [19] also proposed a method to generate custom instructions by relaxing the constraints on the number of input and output operands. There have also been efforts [16], [4] to combine the two steps of *enumeration* and *selection* in order to generate custom instructions on-the-fly.

C. *Organization of the Paper*

In the next section, we introduce the system model and some necessary notations. In Section III, we explain our framework to evaluate the performance metrics for one design point of a customized MPSoC running a streaming application. This is followed by a discussion on how to efficiently search the design space for good quality designs in Section IV. Some of experimental results that we obtained by applying our technique to an MPEG encoder application are presented in Section V. Finally, we conclude in Section VI, where we outline some directions for future work.

II. System Model

We consider a multimedia streaming application mapped to an MPSoC architecture. Broadly, the model of such a system may be defined in terms of the *application*, the MPSoC *architecture* and the *mapping* of the application to the architecture.

Application: We are concerned with *typed* event streams that may be formally specified in two parts. The **first** part of the stream specification is a transition system $\mathcal{T} = (S, S_0, \Sigma, D, \Psi)$ which captures all possible *sequences of event types* that might occur in the stream. Here, Σ is the finite set of event *types*. S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, and $\Psi \subseteq S \times \Sigma \times S$ is a set of transitions. Henceforth, we denote any transition $\langle s, \sigma, s' \rangle$ in Ψ by $s \xrightarrow{\sigma} s'$. Any sequence of events in the stream can only be generated as follows. The system starts in an initial state, and if $s \xrightarrow{\sigma} s'$ then the system can change its state from s to s' and generate an event of type σ. Finally, any transition Ψ, $s \xrightarrow{\sigma} s'$ is annotated with the worst-case ($WCET$) and best-case execution times ($BCET$) for the event type σ. We denote this as a tuple, $D(\Psi) = \{WCET, BCET\}$. The $WCET$ and $BCET$ for the event type σ will be based on the selection of custom instructions. Each custom configuration of the subtasks triggered by σ will imply a unique set of $WCET$ and $BCET$. We also assume that all subtasks triggered by σ are running on one processor. Thus, here, we have defined $\mathcal{T}$ with respect to a particular instance, i.e., one custom configuration and one processor. Later in Section III, we shall discuss how to instantiate $\mathcal{T}$ for different configurations and different processors.

Such a transition system $\mathcal{T}$ can be used to model constraints on allowable sequences of events. $\mathcal{T}$ can either be determined by analyzing the device or the system that generates the stream, or by analyzing a sufficiently large number of representative input streams. Figure 2(a) shows a toy transition system with $\Sigma = \{a, b\}$. It captures the constraints that (i) the event stream starts with the type a, (ii) the event type a may arrive in bursts and (iii) between two bursts of events of type a, at least two events of type b must arrive.

The **second** part of the stream specification is concerned with its timing properties. Towards this, we are given the functions $\bar{\alpha}^u(\Delta)$ and $\bar{\alpha}^l(\Delta)$, which we will refer to as the arrival curves. $\bar{\alpha}^u(\Delta)$ and $\bar{\alpha}^l(\Delta)$ bound the maximum and minimum number of events that can arrive within any time interval of length Δ. Thus, given any concrete arrival process $R(t)$, which denotes the total number of events that arrive during the time interval $[0, t]$, the inequalities $\bar{\alpha}^l(\Delta) \leq R(t + \Delta) - R(t) \leq \bar{\alpha}^u(\Delta)$ hold true for all $\Delta \geq 0$ and $t \geq 0$. It may also be noted here that this specification is more general than the event models traditionally studied in the real-time systems literature, such as periodic, periodic with jitter or the sporadic event model [3], [5]. Figure 2(b) shows the arrival curve for an event stream where upto 2 events arrive within any time interval of length less than 17 time units. In other words, if we consider any concrete (timed) trace of an arrival process, and slide a "window" of length less than 17 time units along this trace, then for any position of this window at most 2 events will be recorded inside the window. Similarly, if the window is of length 17, then for any position of this window at most 3 events will be recorded inside the window.

Architecture: The streaming application described above runs on an MPSoC platform with P processing elements (PE). The processors are arranged in a pipelined fashion. An input multimedia stream enters PE_i, gets processed by the tasks implemented on this PE, and the processed stream enters PE_{i+1} for further processing. Figure 3 shows such an MPSoC with two PEs onto which the various parts of an abstract application with 5 tasks are mapped. Each PE has an internal buffer, which is a FIFO channel of fixed capacity, and is used to store the incoming stream to be processed. The frequencies of the processors are known to us. Each PE consists of an extensible instruction-set architecture.

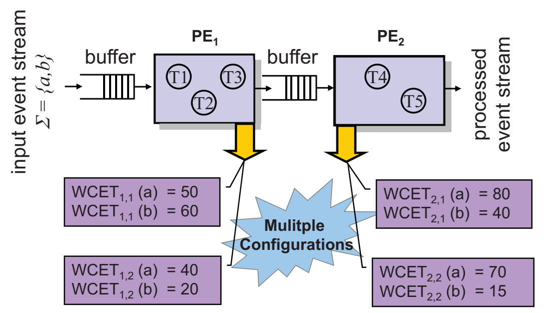


Fig. 3. WCET for events differ for each configuration giving rise to multiple global configurations.

Mapping: We assume that the mapping is given, i.e., we know which subtasks of the given application are running on a particular PE. Clearly, a different mapping would directly influence the choice of custom instructions on the MPSoC. Thus, to find the optimal set of custom instructions, the problem of mapping the application task graph to the different PEs has to be intertwined with the custom instruction selection problem. However, we do not attempt to address this combined problem because the problem of selecting the optimal custom instructions for an MPSoC even for one instance of a mapping is already computationally intensive, as explained below. Our goal here is to address the custom instruction *selection* problem, given a particular mapping and an enumeration of all possible custom instructions for this mapping. An important future work would be to jointly optimize task mapping and custom instruction selection, which will benefit from the techniques proposed here.

Let there be M subtasks in the task graph of the application which are mapped to various PEs on the MPSoC. Let us consider that $\Lambda(PE_i)$ is the set of tasks which are mapped to the PE_i. Figure 3 shows a possible mapping for a task graph with 5 subtasks. Tasks T_1, T_2 and T_3 are mapped to PE_1 while the rest of the tasks are mapped on to PE_2. As each task is running on a customizable processor, we consider that there are n_k custom instruction configurations for the kth task and refer to each of these configurations as $C_{k,1}, C_{k,2}, \ldots, C_{k,n_k}$. However, more than one task is mapped to the same PE giving rise to multiple configurations of the PE. Thus, PE_i can run in any of $|config_i| = \prod_{j \in \Lambda(PE_i)} n_j$ configurations. For example, in Figure 3, T_1, T_2 and T_3 are mapped to PE_1. Assume that T_1 and T_2 has one configuration each, thus $n_1 = n_2 = 1$. Also, assume that T_3 has two custom instruction configurations and thus, $n_3 = 2$. This leads to $n_1 \times n_2 \times n_3 = 2$ configurations.

We would like to note here that workload on PE_i for any event would depend on its configuration. Let us consider the jth configuration of PE_i. We then define worst-case $WCET_{i,j}(a)$, (and the best-case $BCET_{i,j}(a)$) execution requirements of a event type a on PE_i for its configuration j as the summation of worst-case (best-case) running times of the tasks in set S that are triggered by event a. In Figure 3, let us consider tasks T_1, T_2 and T_3 to have worst case execution requirements of 20, 30 and 40 — for their first configuration choices. If a triggers only T_1 and T_2 in PE_1, then $WCET_{1,1}$ (i.e., $WCET$ of a on PE_1 for PE_1's first configuration) is

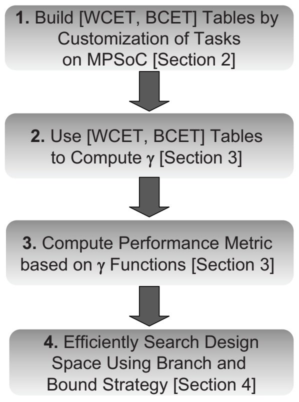

1. Build [WCET, BCET] Tables by Customization of Tasks on MPSoC [Section 2]
2. Use [WCET, BCET] Tables to Compute γ [Section 3]
3. Compute Performance Metric based on γ Functions [Section 3]
4. Efficiently Search Design Space Using Branch and Bound Strategy [Section 4]

Fig. 4. Overview of our proposed scheme.

$20 + 30 = 50$.

A global configuration for the MPSoC is defined by each of its processors being in one its $|config_i|$ configurations. Hence, the total number of possible global configurations will be $|config_1| \times |config_2| \ldots \times |config_P|$, with P processors in the pipeline. Given such a large and complex design space it is not trivial for the designer to identify a configuration that would yield the optimal performance while satisfying hardware area constraints. In what follows, we shall introduce a framework to identify good customization choices in a prohibitively large design space. Towards this, first we shall discuss how to evaluate a single design point in Section III. In Section IV, we shall discuss an effective pruning strategy which can quickly evaluate trade-offs in the entire design space.

III. Performance Analysis

In this section, we first describe how to evaluate the performance metrics for a single processor. This will be followed by a discussion on how this evaluation technique may be utilized for global performance evaluation of the entire MPSoC platform. Towards this, our techniques build upon a general mathematical framework for analyzing real-time systems [5]. We extend this framework to analyze the performance metrics associated with customization choices of MPSoCs.

Before presenting the details of our performance evaluation framework, we outline the main steps of our technique in Figure 4. Our proposed method starts with computing the custom instruction configurations for each task and the corresponding WCET and BCET values as defined in Section II. This is Step 1 in the Figure 4. This phase is related to the *enumeration* of custom instructions and as discussed in Section I-B we consider this as given. In Step 2, we compute the maximum and minimum processing requirements arising from the incoming events on each PE. Towards this, we define a function γ, the computation of which leverages on $WCET$ and $BCET$ of various event types. Next, in Step 3, we utilize this function γ to compute the performance metrics like the maximum delay, backlog (which is a measure of the maximum buffer requirement) or jitter experienced by any input event stream. Steps 2 and 3 will be the focus of this section and are discussed in detail in the following. Step 4 is the design space exploration strategy, and will be discussed in Section IV.

Formally, let us define the functions $\gamma^u_{i,j}(k)$ and $\gamma^l_{i,j}(k)$ whose argument is an integer k and return the maximum and minimum processing times that may be demanded by *any* sequence of k consecutive events belonging to the input stream. The subscripts i, j refer to the fact that the function $\gamma_{i,j}(k)$ is computed for the jth custom instruction configuration for PE_i on the MPSoC. Thus, j ranges from 1 to $|config_i|$ and encapsulates all possible configurations of tasks that are mapped onto PE_i (see Section II).

A. Computing γ:

We now show how to compute the functions $\gamma^u_{i,j}$ and $\gamma^l_{i,j}$. Towards this, let us first recall the definition of the transition system $\mathcal{T}$, where each transition Ψ from (s_1) to (s_2) represents the processing of an event of the type σ and is annotated with $D(\Psi) = \{WCET, BCET\}$. As an event σ passes through MPSoC its arrival sequence (as defined by the states and transitions of $\mathcal{T}$) at each PE remains same, but it will generate different workload (as defined by the annotation $D(\Psi)$ on the transitions of $\mathcal{T}$) on each PE based on the customization choice and the set of the tasks mapped to that PE. We say that, $\mathcal{T}$ will have to be *instantiated* with the worst-case and best-case execution times associated with each customization choice for each processor. Let $\mathcal{T}_{i,j}$ refer to the instance of $\mathcal{T}$ where we consider the ith processor's jth configuration. Thus, the annotation on each such transition, i.e., the tuple $D_{i,j}(\Psi) = \{WCET_{i,j}(\sigma), BCET_{i,j}(\sigma)\}$, denotes the maximum and minimum processing time of the event σ. We shall illustrate this idea of *instantiation* of $\mathcal{T}$ with the example in Figure 3. Here, we will have 4 instances of $\mathcal{T}$: $\mathcal{T}_{1,1}$ and $\mathcal{T}_{1,2}$ corresponding to the two configurations on PE_1, and $\mathcal{T}_{2,1}$ and $\mathcal{T}_{2,2}$ corresponding to the two configurations on PE_2. Figure 5 shows these 4 instances graphically. As the event stream passes from PE_1 to PE_2, the sequence of arrival of the event types remains the same. Hence, all the 4 instances of $\mathcal{T}$ have the same states and transitions, while differing only on the annotation $D_{i,j}(\Psi)$. For simplicity of exposition, it is assumed that the BCET is always 5 units less than WCET in this example. In the Figure 5, $\gamma_{i,j}$ denotes the γ function for the ith processor and the jth configuration. The $\mathcal{T}_{i,j}$ instance of the transition system is utilized to compute the $\gamma_{i,j}$ function. The computation of any such $\gamma_{i,j}$ function is discussed below. By definition, $\gamma^u_{i,j}(k)$ ($\gamma^l_{i,j}(k)$) is the weight of the maximum-weight (minimum-weight) path of length k in the transition system $\mathcal{T}_{i,j}$. For any k, $\gamma^u_{i,j}(k)$ (or $\gamma^l_{i,j}(k)$) can thus be computed from the single source longest (or shortest) paths of length k for all vertices in $\mathcal{T}_{i,j}$. The single-source longest or shortest path for a given vertex is computed using standard dynamic programming methods. Algorithm 1 shows how to compute $\gamma^u_{i,j}(k)$ and $\gamma^l_{i,j}(k)$ for all integers $1 \le k \le n$, where n is an input to this algorithm. In the rest of this section, where we are concerned with the performance metrics for *one* configuration on *one* processor we shall drop the subscripts for clarity of exposition. Henceforth, $\gamma^u(k)$ ($\gamma^l(k)$) will be used to represent $\gamma^u_{i,j}(k)$ ($\gamma^l_{i,j}(k)$) without any ambiguity.

B. Computing Performance Metrics:

To compute the maximum backlog in a buffer, we first need to define a function β^l, which can be considered as the *pseudoinverse* of the function γ^u that we already defined

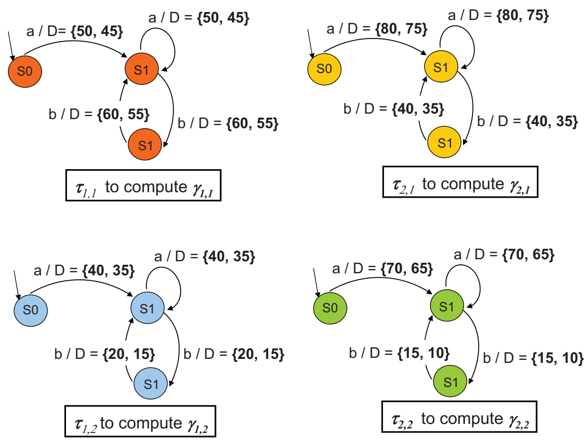


Fig. 5. The transition systems $\mathcal{T}$ for PE_1 and PE_2 corresponding to the two different custom instruction configurations in each.

Algorithm 1 Computing $\gamma(k)$. (Note that subscripts i and j are dropped for ease of exposition.)

Input: Transition system $\mathcal{T} = (S, S_0, \Sigma, D, \Psi)$, function $pred(s)$ which returns all predecessors of the state $s \in S$, and an integer n;

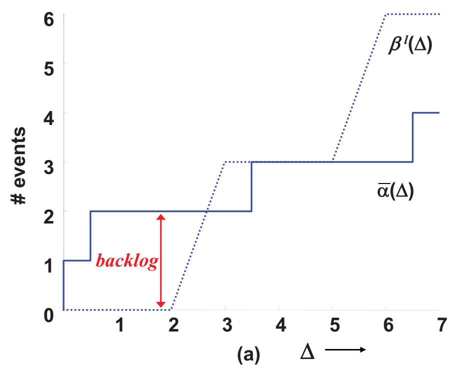
Output: $\gamma^u(k)$ and $\gamma^l(k)$ for all $1 \leq k \leq n$;

```
1:  $w_s^u(k) \leftarrow -\infty, w_s^l(k) \leftarrow -\infty$ for all $s \in S$, $1 \leq k \leq n$;
2:  $w_s^u(0) \leftarrow 0, w_s^l(0) \leftarrow 0$ for all $s \in S$;
3:  for $k = 1$ to $n$ do
4:    for $\forall\ s \in S$ do
5:      if $|pred(s)| > 0$ then
6:        $w_s^u(k) \leftarrow \max_{p \in pred(s)}\{w_p^u(k-1) + WCET(p \rightarrow s)\}$
7:        $w_s^l(k) \leftarrow \min_{p \in pred(s)}\{w_p^l(k-1) + BCET(p \rightarrow s)\}$
8:      end if
9:    end for
10:   $\gamma^u(k) \leftarrow \max_{s \in S}\{w_s^u(k)\}$
11:   $\gamma^l(k) \leftarrow \min_{s \in S}\{w_s^l(k)\}$
12: end for
```

above. We define, $\beta^l(\Delta) = \inf_{k \geq 0}\{k : \gamma^u(k) \geq \Delta\}$. Hence, $\beta^l(\Delta)$ returns the minimum number of events that can generate a processing requirement of Δ. In other words, *at least* $\beta^l(\Delta)$ events from the stream are guaranteed to be processed within a time interval of length Δ. Within this time interval, at most $\bar{\alpha}^u(\Delta)$ events might arrive. Hence, the backlog generated within this interval is $\bar{\alpha}^u(\Delta) - \beta^l(\Delta)$. Therefore, the maximum or worst-case *backlog* is given by: $backlog = \sup_{\Delta \geq 0}\{\bar{\alpha}^u(\Delta) - \beta^l(\Delta)\}$ Intuitively, *backlog* can be interpreted as the maximum vertical distance between the curves $\bar{\alpha}(\Delta)$ and $\beta^l(\Delta)$ (see Figure 6). Due to space restrictions, we will omit a discussion on computing *delay*, which involves similar analysis.

C. Extending the Analysis to other PEs:

Above, we presented a framework to analyze one PE, which lies at the beginning of the path of an input stream. However, the analysis may be extended in a compositional fashion to other PEs. The event stream generated by a PE_i will trigger PE_{i+1} in the pipeline. Towards this, let us denote using $\bar{\alpha}^{u'}(\Delta)$ and $\bar{\alpha}^{l'}(\Delta)$, the maximum and minimum number of processed events respectively, that can possibly be seen at the output of PE_i within any time interval of length Δ. $\bar{\alpha}'(\Delta)$ is therefore exactly of the same form as $\bar{\alpha}(\Delta)$ which bounds an input stream. It may be shown that



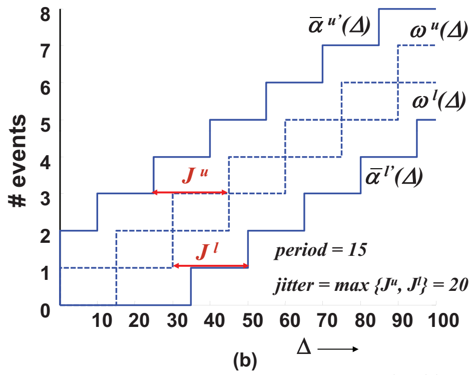


Fig. 6. Computing the the maximum number of (a) *backlogged* events and (b) the *jitter* experienced by an event stream.

$$\begin{aligned}\bar{\alpha}^{l'}(\Delta) &= \min\{\inf_{0 \leq \mu \leq \Delta}\{\sup_{\lambda > 0}\{\bar{\alpha}^l(\mu + \lambda) - \beta^u(\lambda)\} \\ &+ \beta^l(\Delta - \mu)\}, \beta^l(\Delta)\} \\ \bar{\alpha}^{u'}(\Delta) &= \min\{\sup_{\lambda > 0}\{\inf_{0 \leq \mu < \lambda + \Delta}\{\bar{\alpha}^u(\mu) + \beta^u(\lambda + \Delta - \mu)\} \\ &- \beta^l(\lambda)\}, \beta^u(\Delta)\}\end{aligned}$$

In the above equation, $\beta^u(\Delta) = \sup_{k \geq 0}\{k : \gamma^l(k) \geq \Delta\}$. Hence, $\beta^u(\Delta)$ returns the maximum number of events that can generate a processing requirement of Δ. In other words, *at most* $\beta^u(\Delta)$ events from the stream may be processed within a time interval of length Δ. Further details and proof maybe found in [5].

The $\bar{\alpha}'$ functions defined above, along with the $\gamma_{i+1,j}$ functions for the next PE (PE_{i+1}), can be now utilized in the framework described in this section to evaluate various performance metrics. In this way, the entire MPSoC platform may be analyzed. From the outgoing $\bar{\alpha}'$ curves at the final PE in the MPSoC we can analyze timing properties like burstiness and jitter. Towards this, let ω^u and ω^l be the upper and lower curves corresponding to an event stream with period Θ, where Θ is the period of the input arrival curves. We then compute $J^u = min\{J_0 : \omega^u(\Delta + J_0) \geq \bar{\alpha}^{u'}\}$ and $J^l = min\{J_0 : \omega^l(\Delta - J_0) \leq \bar{\alpha}^{l'}\}$. The jitter of the outgoing stream is $J = max\{J^u, J^l\}$. Intuitively, *jitter* is the deviation of the stream from its periodicity. For example, in Figure 6(b), $J = J^l = J^u = 20$. For algorithms to compute the jitter of more complex arrival curves and when the periodicity is not known, please refer to [12].

To summarize, in this section we discussed the performance evaluation of an MPSoC for a one custom instruction configuration. In particular, techniques were discussed to estimate the worst-case backlog for an event stream being processed on a PE. The summation of the backlogs at all the PEs gives us an estimate of the total on-chip buffer requirement on the MPSoC. Second, we discussed how to measure the jitter of the processed stream. The jitter or burstiness is a relevant quality-of-service metric for multimedia applications.

IV. Design Space Exploration

In the previous sections, we explained how to analyze various performance metrics for an MPSoC platform for *one* custom instruction configuration. In this section, we will be concerned with exploring the performance metrics associated with all possible configurations to identify the solutions that satisfy given QoS guarantees. Recall that the computation of the function γ (which computes the worst and best-case execution requirement of k events) lies at the heart of our analysis engine. It should be clear that the function γ will have to be recomputed for each configuration because each unique custom instruction configuration for a task implies different execution requirements of the task. Computation of γ is based on a traversal of the transition system $\mathcal{T}$. Thus, in any iteration of a design space exploration process, the first step is to annotate the transition system with $D_{i,j}(\Psi) = \{WCET_{i,j}(\sigma), BCET_{i,j}(\sigma)\}$ for this configuration. Note that the above procedure is an exhaustive search process iterating over all design points. In order to improve the high running times associated with such a framework, we next propose a fast search strategy.

A. Branch and Bound

Since an exhaustive exploration of all possible design points can turn out to be prohibitively huge, we propose a Branch and Bound ($B\&B$) algorithm to select appropriate custom instruction configuration for each task. We choose a $B\&B$ strategy because of two reasons. The first reason being that it returns the optimal solution. Other optimization strategies like evolutionary algorithms [9] or tabu search [10], being heuristics, cannot give optimality guarantees on their results. Second, we designed effective pruning techniques that lead to short running times of our $B\&B$ search strategy. These pruning techniques are described in the following, along with the description of our $B\&B$ algorithm.

The goal of our design space exploration, i.e., the $B\&B$ algorithm, is to find the configuration where (i) the jitter at the output is minimized and where (ii) the area constraints on custom instructions are satisfied. We have chosen jitter as an optimization criterion because for multimedia applications, higher burstiness at the output results in poor quality. The $B\&B$ algorithm was designed to effectively exploit specific characteristics of the design space for quickly identifying optimal solutions. This is validated by our experiments

Algorithm 2 Branch and Bound Strategy for Custom Instructions Selection

```
Input: Tasks$\mathrm{T_1 \dots T_M}$ with configurations; Area constraint:
    AREA; Mapping: $\Lambda$
Output: Minimum jitter;
1:  J $\leftarrow$ 0; optimalSoln $\leftarrow \emptyset$; A $\leftarrow$ AREA;
2:  MinJitter $\leftarrow$ compute_original_jitter();
3:      /* $\mathrm{T_1}$ is the first task */
4:  search($\mathrm{T_1}$, J, A, $\emptyset$, $\Lambda$)
5:  return J;
6:  end;
7:  Function search($\mathrm{T_k}$, J, A, Soln, $\Lambda$)
8:  for each $\mathrm{C_{k,l}}$ of $\mathrm{T_k}$in increasing order of execution time
    do
9:    if (area($\mathrm{C_{k,l}}$) $\leq$ A) then
10:     partialSoln $\leftarrow$ Soln$\cup C_{k,l}$ ; A $\leftarrow$ A $-$ area($\mathrm{C_{k,l}}$);
11:     if (is_the_last_task_on_processor($\mathrm{T_k}$, $\Lambda$)) then
12:       J $\leftarrow$ J + compute_jitter(i);
13:     end if
14:     if (is_the_last_task_on_the_last_processor($\mathrm{T_k}$, $\Lambda$))
        then
15:       if (J<MinJitter) then
16:         MinJitter $\leftarrow$ J; optimalSoln $\leftarrow$ partialSoln
17:         continue;
18:       end if
19:     end if
20:     if (bound(partialSoln)<MinJitter) then
21:       search(next_task($\mathrm{T_k}$), J, A, partialSoln,$\Lambda$)
22:     end if
23:   end if
24: end for
```

as only 2295 out of a total 55926 of design points were enumerated. The pseudocode of our proposed strategy is shown in Algorithm 2 and is explained below.

Algorithm Description: The $B\&B$ algorithm defines a tree structure to represent the search space. Each level k in the $B\&B$ search tree corresponds to the choice of a configuration for the task T_k. Thus, each node at level k corresponds to a partial solution with the configurations about the tasks T_1 up to T_k. If T_k is the last task to be considered by the $B\&B$ on processor i (lines 11-12), the jitter of the partial solution is increased by the jitter of processor i using $compute\_jitter(i)$. Whenever we reach a leaf node of the search tree, that is the last task on the last processor (lines 14-19), we have a complete solution with selected configurations for each task. During the traversal of the search tree, the minimum jitter achieved so far at any leaf node is kept as $MinJitter$. If the jitter J of the current complete solution is less $MinJitter$, we update $MinJitter$ as well as the optimal solution computed so far (see line 16).

The power of our $B\&B$ algorithm comes from the following two effective pruning strategies for the design space. At any non-leaf node m in the search tree, we compute a lower bound, $bound(m)$, on the minimum possible jitter at any leaf node in the subtree rooted at m. This lower bound is computed

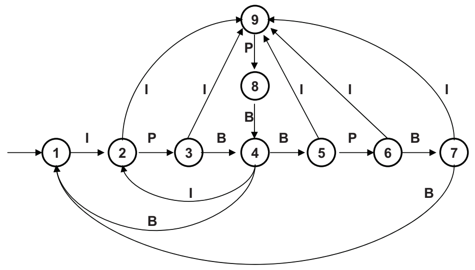


Fig. 7. The transition system $\mathcal{T}$ specifying the arrival sequence of the frame types I, B and P for an MPEG-2 encoder.

by summing up the jitter due to the tasks that have been enhanced with custom instructions with respect to the mapping Λ (see Algorithm 2) and the minimum jitter possible from the remaining tasks (which is the jitter when enhanced with the custom instruction configuration such that $wcet - bcet$ is minimum among all event types). If $bound(m) > MinJitter$ (line 20), then the search space corresponding to the subtree rooted at m can be pruned. Second, we have the constraint on hardware area used for custom instructions in MPSoC. Due to the fact that one processing element may require more custom instructions than the others, it will have larger area for custom instructions. Therefore, the hardware area for custom instructions is allocated among processing elements in the MPSoC while performing design space exploration. If the remaining unallocated hardware area for custom instructions is less than the required hardware area for the custom instructions of a particular configuration of a task at any node, then the subtree rooted at the corresponding node is pruned (line 9).

V. Case Study

In the following, we discuss the results obtained by applying our technique on an MPEG-2 encoder; the subtasks of the encoder are shown in Figure 1. The arrival sequence of these frame types were specified by the transition system shown in Figure 7. The decoder is run on an MPSoC architecture with two pipelined PEs, with PE_1 running at 100MHz frequency and PE_2 running at 50MHz. We assume that the FS task is running on the first PE and the rest of the tasks are running on the second PE. The task FS has signifacntly higher execution requirement compared to the rest of the tasks, and hence, it is reasonable to assume that PE_1 will run at a higher frequency than PE_2. In our setup, the encoder has to encode 30 frames per second and its output is a 64×64 pixel encoded clip.

A. Experimental Set-up

The entire framework has been implemented in C. All the experiments were conducted on a Linux machine running on a 8-core Xeon(R) 3.0 GHz processor. We generated different custom instruction configurations for each task in the MPEG-2 encoder using Tensilica tool [18]. The number of configurations for the tasks varied from 5 to 13 thus creating a large design space ($55,926$ design points in total). In order to obtain $WCET$ and $BCET$ estimates, we used a simulation/measurement-based method by inserting timing counters to the MPEG-2 encoder code at suitable points.

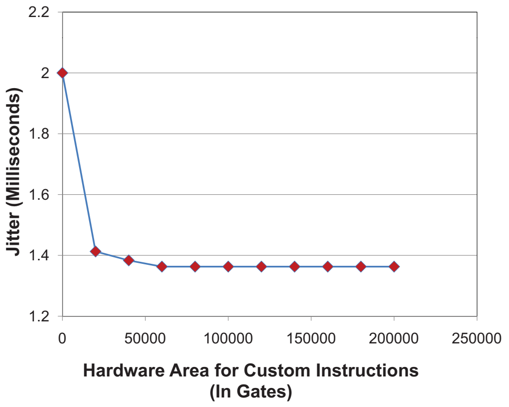


Fig. 8. Tradeoffs between custom instruction area and output jitter.

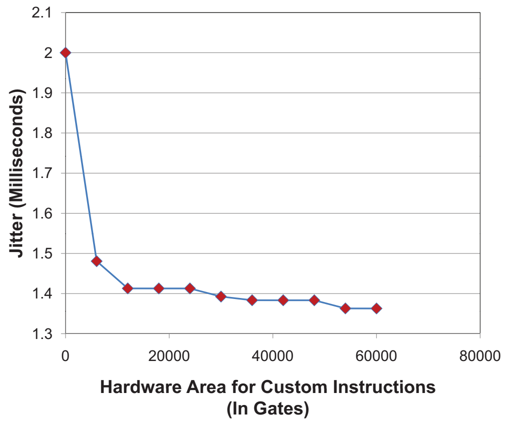


Fig. 9. Zoomed-in chart of Figure 8 showing the tradeoffs between custom instruction area and jitter for upto 60K gates.

Let the maximum possible custom instruction area be $MaxArea$. For our experiments, we iteratively called the $B\&B$ algorithm with the area constraint varying from $0.1 \times MaxArea$ to $MaxArea$ with an interval/step size of $0.1 \times MaxArea$. This allowed us to evaluate the trade-offs at various points in the design space. At each of these intervals (i.e., design points) our tool found a system with the minimum possible jitter and associated on-chip buffer size. The results from these experiments are discussed below.

B. Discussion

Figure 8 shows the relation between output jitter and custom instruction area. Note that as the hardware area increases, the jitter starts to decrease. This is because with the use of custom instructions the value $WCET - BCET$ starts to decrease. In fact, the jitter decreases up to 32% with an area of $0.3 \times MaxArea$ (i.e., 60,000 gates). However, after $0.4 \times MaxArea$ (i.e., 80,000 gates), the output jitter does not decrease any further. This can be expected because jitter is dependent on the *difference* between the $WCET$ and $BCET$ and not on $WCET$ or $BCET$ independently. This result is interesting because our tool reveals the *sweetspot* ($0.3 \times MaxArea$) after which adding more custom instructions will not minimize jitter. To confirm our findings, we ran another set of experiments where the constraint on hardware area for custom instructions was varied from 0 to $0.3 \times MaxArea$

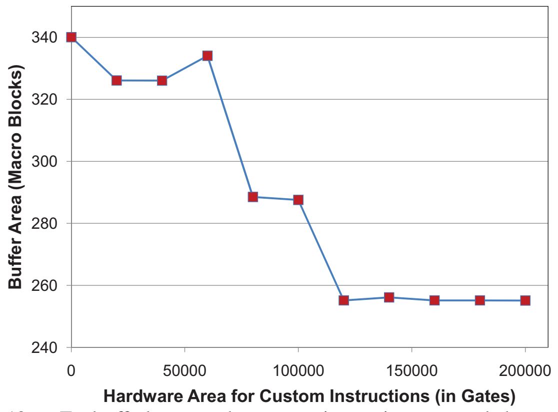


Fig. 10. Tradeoffs between the custom instruction area and the on-chip buffer.

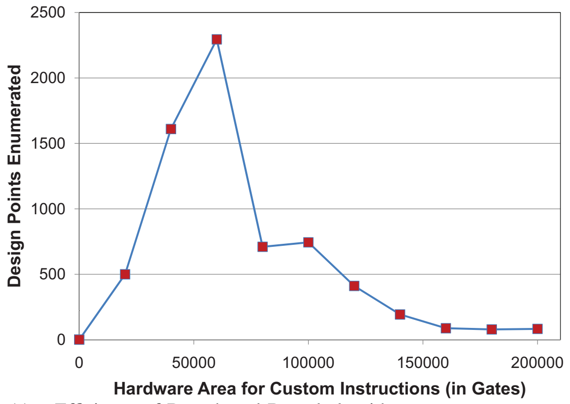


Fig. 11. Efficiency of Branch and Bound algorithm.

(i.e., $60,000$ gates) in the steps of $6,000$ gates. The zoomed-in results shown in Figure 9 validate our observation.

While the jitter is minimized with the custom instruction area being $0.3 \times MaxArea$, the buffer required to maintain that value of jitter can be further reduced if more hardware area is used for custom instructions. Figure 10 shows the area utilized by custom instructions corresponding to various buffer requirements. The reduction in on-chip buffer requirement is expected because the performance of the processors increase with increased use of custom instructions, which in turn decreases the number of backlogged events at the processors. However, note that the decrease is not monotonic. This is because the total on-chip buffer depends on the computation capability of both PE_1 and PE_2 in a complex fashion. For example, custom instructions might enhance the performance of PE_1 thus decreasing the buffer requirement before PE_1. However, this enhanced performance of PE_1 might now increase the number of backlogged events in front of PE_2 by a large amount leading to an overall increase of on-chip buffer requirement.

C. Running Times

Finally, we illustrate the efficiency of our proposed $B\&B$ strategy in Figure 11. This figure shows the relation between custom instruction area and the number of design points that were explored by the $B\&B$ algorithm. The total number of points (which is the total number of nodes in $B\&B$ search tree) were $55,926$. However, as shown in Figure 11, the maximum number of points to be evaluated in any of our invocations is only $2,295$ which clearly manifests the power of our $B\&B$ algorithm. For each iteration of our $B\&B$, the running time varied from 1 to 6 minutes, clearly outperforming the exhaustive design space exploration techniques which took over 12 hours to complete.

VI. Concluding Remarks

In this paper we have proposed a technique to select optimal custom instruction configurations for multiple PEs on an MP-SoC platform. Our method accurately characterizes the high variability associated with the execution times of multimedia streaming applications. We demonstrated the utility of our proposed framework by analyzing a real-life case study. In future, we plan to design search strategies with optimization objectives other than jitter (e.g., delay, power etc.). In this paper, we assumed that the designer has taken appropriate decisions regarding the mapping of the tasks to PEs. It will also be worthwhile to explore the influence of such decisions on the customization choices.

References

[1] M. Arnold and H. Corporaal. Designing domain-specific processors. In *CODES*, 2001.
[2] K. Atasu, O. Mencer, W. Luk, C. Ozturan, and G. Dundar. Fast custom instruction identification by convex subgraph enumeration. In *ASAP*, 2008.
[3] S. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems*, 24(1):93–128, 2003.
[4] P. Biswas, S. Banerjee, N. Dutt, L. Pozzi, and P. Ienne. ISEGEN: An iterative improvement-based ISE generation technique for fast customization of processors. In *IEEE TVLSI*, 2006.
[5] S. Chakraborty, S. Künzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *DATE*, 2003.
[6] N. Cheung, S. Parameswaran, and J. Henkel. INSIDE: INstruction Selection/Identification & Design Exploration for extensible processors. In *ICCAD*, 2002.
[7] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. In *MICRO*, 2003.
[8] J. Cong, Y. Fan, G. Han, and Z. Zhang. Application-specific instruction generation for configurable processor architectures. In *FPGA*, 2004.
[9] K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, 2001.
[10] F. Glover and F. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.
[11] H. Javaid and S. Parameswaran. A design flow for application specific heterogeneous pipelined multiprocessor systems. In *DAC*, 2009.
[12] S. Künzli, A. Hamann, R. Ernst, and L. Thiele. Combined approach to system level performance analysis of embedded systems. In *CODES*, 2007.
[13] J.-Y. Le Boudec and P. Thiran. *Network Calculus - A Theory of Deterministic Queuing Systems for the Internet*. LNCS 2050, Springer, 2001.
[14] J. Lee, K. Choi, and N. Dutt. Efficient instruction encoding for automatic instruction set design of configurable ASIPs. In *ICCAD*, 2002.
[15] T. Li, Z. Sun, W. Jigang, and X. Lu. Fast enumeration of maximal valid subgraphs for custom-instruction identification. In *CASES*, 2009.
[16] L. Pozzi, K. Atasu, and P. Ienne. Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE TCAD*, 25(7), July 2006.
[17] F. Sun, S. Ravi, A. Raghunathan, and N.K.Jha. Application-specific heterogeneous multiprocessor synthesis using extensible processors. *IEEE TCAD*, 2006.
[18] Tensilica - XPRES Compiler - Optimized Hardware Directly from C. www.tensilica.com/products/devtools/hw_dev/xpres/.
[19] A. K. Verma, P. Brisk, and P. Ienne. Rethinking custom ISE identification: A new processor-agnostic method. In *CASES*, 2007.
[20] P. Yu and T. Mitra. Scalable custom instructions identification for instruction-set extensible processors. In *CASES*, 2004.