Evaluating Design Trade-offs in Customizable Processors

Unmesh D. Bordoloi[1] Huynh Phung Huynh[2] Samarjit Chakraborty[3] Tulika Mitra[2]

[1]Verimag Labs, France

[2]Department of Computer Science, National University of Singapore

[3]Institute for Real-Time Computer Systems, TU Munich, Germany

unmesh.bordoloi@verimag.fr, {huynhph1, tulika}@comp.nus.edu.sg, samarjit@tum.de

ABSTRACT

The short time-to-market window for embedded systems demands automation of design methodologies for customizable processors. Recent research advances in this direction have mostly focused on single criteria optimization, e.g., optimizing performance though custom instructions under pre-defined area constraint. From the designer's perspective, however, it would be more interesting if the conflicting trade-offs among multiple objectives (e.g., performance versus area) are exposed enabling an informed decision making. Unfortunately, identifying the optimal trade-off points turns out to be computationally intractable. In this paper, we present a polynomial-time approximation algorithm to systematically evaluate the design trade-offs. In particular, we explore performance-area trade-offs in the context of multi-tasking real-time embedded applications to be implemented on a customizable processor.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

General Terms

Algorithms, Design, Performance

Keywords

ASIP, Processor customization, Multi-objective design space exploration, Pareto-optimal curve.

1. INTRODUCTION

Instruction-set extensible processors that consist of existing processor cores extended with application specific custom instructions are now very popular in the embedded systems domain. Typically, a custom instruction encapsulates the computation of a frequently executed subgraph of the program's dataflow graph. Custom instructions are implemented as hardwired datapath of the existing processor core and this helps improve the performance of the application. Some examples of commercial customizable processors include Lx, ARC^{TM} core, Xtensa, Stretch S5 among others.

DAC 2009, July 26 - 31, 2009, San Francisco, California, USA.

The past decade has seen a flurry of research activity on automated identification and selection of custom instructions for an application or an application domain. However, most of these design efforts have focused on single-objective optimization, for example, choosing an optimal set of custom instructions either in terms of performance or energy. As the performance/energy improvement offered by custom instructions come at the cost of silicon area, this optimization is typically constrained by a pre-defined silicon area. The designer, on other other hand, can benefit significantly if the automation tools expose all the conflicting trade-offs (e.g., performance versus area) instead of offering a point solution. It is then up to the designer to choose an appropriate trade-off point. More formally, we are interested in generating the *Pareto-optimal curve* in a multi-objective design space (e.g., performance and area) where (a) no point is better than any other point on the curve with respect to both objectives, and (b) no improvement can be made in any objective without trading-off or worsening the other.

Unfortunately, it turns out that computing the Pareto-optimal curve for our design problem is computationally intractable. Therefore, state-of-the-art customization toolchains (such as Tensilica's XPRES compiler [15]) adopt ad-hoc methods that simply compute the best performing design choices at arbitrary silicon area constraints. In this paper, we propose a systematic methodology to explore the performance-area trade-offs in designing customizable processors. We present a polynomial-time approximation algorithm to compute this trade-off. Moreover, as the Pareto curve may potentially contain exponential number of design points, it is impossible to compute this entire set in polynomial time. Hence, our polynomial-time approximation algorithm, by default, has to approximate the (potentially exponential size) set of points on the Pareto curve with only a polynomial number of points. In a typical design cycle of customizable processors, the system designer inspects all the points in the Pareto curve and then selects one, or at most a few implementations. Hence, from a practical perspective, we feel it is more meaningful if the designer is presented with a reasonably few well-distinguishable trade-offs rather than an exponentially large number of solutions, many of which are very similar to each other. Our approximation algorithm is therefore not only attractive in terms of time-complexity, but also returns more meaningful solutions, in terms of the size of the solution set (including the spread/distribution of solutions in this set).

We explore this approximation solution to Pareto curve generation in the context of multi-tasking real-time embed-

ded applications running on customizable processors. Given a multi-tasking application to be implemented on a customizable processor, there are a large number of implementation possibilities with different subsets of custom instructions leading to varying processor utilization versus area trade-offs. We would like to identify *all* schedulable implementations that expose the different performance trade-offs.

Our contributions: Formally, for any schedulable implementation, let (U, A) denote the corresponding utilization of the base processor and the hardware cost arising from the use of custom instructions. We are then interested in identifying all possible Pareto-optimal solutions $\{(U_1, A_1), \ldots, (U_n, A_n)\}$ that capture the different performance trade-offs [7]. Each (U_i, A_i) in this set has the property that there does not exist any schedulable implementation with a performance vector (U, A) such that $U \leq U_i$ and $A \leq A_i$, with at least one of the inequalities being strict. Further, let S be the set of performance vectors (i.e., (utilization, area) tuples) corresponding to all schedulable implementations. Let P be the set of performance vectors $(U_1, A_1), \ldots, (U_n, A_n)$ corresponding to all the Pareto-optimal solutions. Then for any $(U, A) \in S - P$ there exists a $(U_i, A_i) \in P$ such that $U_i \leq U$ and $A_i \leq A$, with at least one of these inequalities being strict (i.e., the set P contains *all* performance trade-offs). The vectors $(U, A) \in S - P$ are referred to as *dominated solutions*, since they are *dominated* by one or more Pareto-optimal solutions.

In this paper we present a polynomial-time approximation scheme for computing the *utilization-area* Pareto curve. Our proposed solution for computing this Pareto curve involves two distinct stages. First, in the *intra*-task stage, *each* individual task is analyzed. Given the library of possible custom instructions for the task, all possible custom instruction configurations are generated exposing the *workload-area* Pareto curve. In the *inter*-task stage, we consider *all* the tasks in the task-set and their *workload-area* configurations, to generate the processor *utilization-area* Pareto curve P for the overall task set. Our framework for approximately computing the trade-offs extends to both of the above stages.

Related Work: Any custom instruction selection problem starts with the identification of a large number of candidate custom instructions from the program's dataflow graph. [2, 5] have proposed different techniques for this problem. Given such a library of valid custom instructions, various approaches have been proposed to select custom instructions to optimize either the performance or the hardware area. For example, different methods that have been proposed include dynamic programming [1], 0-1 Knapsack [6], greedy heuristic [4, 5], and ILP [11] which maximize the performance under different design constraints, e.g., hardware area. However, the above lines of work were aimed at optimizing a single criteria. In this work, we propose a framework for *multi-objective* optimization to expose the performance versus hardware area trade-offs. Moreover, the above approaches do not take into account the real-time constraints. [9, 18] explore processor customization in the context of real-time systems but only perform single-objective (performance) optimization.

The algorithmic techniques presented in this paper have been motivated by [13]. The result that for *any* Pareto curve and *any* ϵ, there *exists* a polynomial-size ϵ-approximate Pareto curve was shown in [13]. However, for many problems, efficiently (i.e., in polynomial time) *computing* such approximate Pareto curves might not be possible. Our main technical contribution in this paper is to show that such ϵ-approximate Pareto curves can be efficiently computed in the domain of custom instruction selection. An important consequence of this result is a formal basis for custom instruction selection. It shows that in the quest for efficiency, there is no need to resort to ad-hoc techniques – as currently adopted in state-of-the-art customization toolchains – for identifying best-performing custom instruction choices.

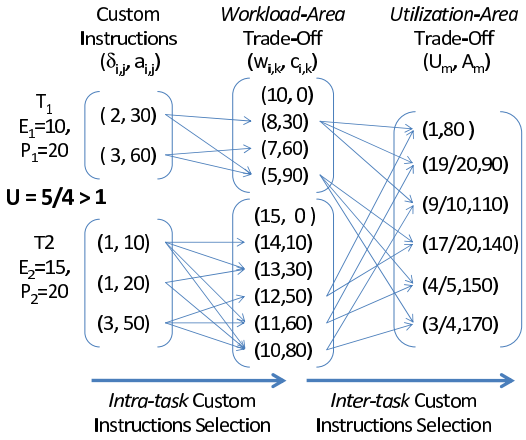


Figure 1: Motivating Example.

Here, it should be noted that in this paper we have taken a classical approximation algorithms standpoint, where the goal is to provide *formal* guarantees on the quality of the results obtained. Our work differs from the existing large body of work on multiobjective optimization [7] that relies on heuristics and randomized search techniques such as evolutionary algorithms.

2. PROBLEM STATEMENT

2.1 Task Model

We use a very general sporadic task model [3] in a preemptive uniprocessor context. Thus, we are interested in the custom instruction selection for a task set $\tau = \{T_1, T_2, \ldots, T_m\}$ consisting of m hard real-time tasks, with the constraint that the task set is schedulable. Any task T_i can get triggered independently of other tasks in τ. Each task T_i generates a sequence of jobs; each job is characterized by the three parameters – the *minimum separation* (P_i) which is the minimum time interval that must elapse before the successive job of the task T_i is triggered, the *deadline* (D_i) by which each job generated by T_i must complete since its release time, and *workload* (E_i) or the worst case execution requirement of any job generated by T_i.

Throughout this paper, we assume the underlying scheduling policy to be the earliest deadline first (EDF). Assuming that for all tasks T_i, $D_i \geq P_i$, the schedulability of the task set τ can be given by the following condition.

THEOREM 1. *A set of sporadic tasks τ is schedulable under EDF if and only if $(U = \sum_{i=1}^{N} \frac{E_i}{P_i}) \leq 1$ where U is the processor utilization due to τ [3, 12].*

2.2 Intra-Task Custom Instructions Selection

We now state the intra-task custom instruction selection problem for a task T_i. For the task T_i, let there be n_i custom

instruction candidates. Each of these n_i custom instructions is associated with a certain hardware area. Choosing the jth custom instruction will lower the workload of the task T_i by $\delta_{i,j}$. Equivalently, the new workload will be $E_i - \delta_{i,j}$. Hence, for each task T_i we have a set of choices $S_i = \{(\delta_{i,1}, a_{i,1}), \ldots, (\delta_{i,n_i}, a_{i,n_i})\}$, where $a_{i,j}$ is the hardware cost associated with the jth custom instruction. Our objective is to select a set of custom instructions that would minimize the workload on the base processor, as well as use the minimum amount of hardware area for custom instructions. In other words, our goal is to compute all *workload-area* trade-offs in the form of a Pareto curve $\{(w_{i,1}, c_{i,1}), \ldots, (w_{i,N_i}, c_{i,N_i})\}$, where $w_{i,j}$ is the workload of task T_i accelerated with a particular set of custom instructions and $c_{i,j}$ is the corresponding cost in terms of silicon area.

In Figure 1, we illustrate this problem for two different tasks T_1 and T_2. The task characteristics for T_1 are $\{E_1 = 10, P_1 = 20\}$. T_1 has two entries in its library of custom instructions, and thus, $n_1 = 2$. Following our notation, $\delta_{1,1} = 2$ and $\delta_{1,2} = 3$, and the corresponding hardware areas are $a_{1,1} = 30$ and $a_{1,2} = 60$. The goal is to identify the *workload-area* Pareto curve for the task T_1. For example, in this case the Pareto curve consists of four solutions $\{(10, 0), (8, 30), (7, 60), (5, 90)\}$. Note that the solution with zero area does not use any custom instruction. Therefore, the application workload is not reduced and is the highest amongst all the solutions. On the other hand, the solution $(5, 90)$ contains both the custom instructions and has the smallest workload with the largest area. The task characteristics for T_2 and the custom instruction candidates may be read in the same way as described above for T_1.

2.3 Inter-Task Custom Instructions Selection

For each task T_i, let there be N_i hardware implementation choices $\{(w_{i,1}, c_{i,1}), \ldots, (w_{i,N_i}, c_{i,N_i})\}$ in the *workload-area* Pareto curve that is computed by the intra-task custom instruction selection phase. Each of these N_i choices represent a custom instruction configuration for the task. A task may be chosen to run in one these configurations where it will incur certain a hardware cost $c_{i,j}$ and would lower the execution time of the task on the processor from E_i to $w_{i,j}$.

However, in a real-time embedded system there is not one task but a set of tasks running on a processor. Thus, a designer is interested not in the performance of a single task, but rater the *utilization* of the processor by the entire task-set. The goal in the inter-task custom instruction selection phase is to identify one custom instruction configuration for each task, which would minimize the overall processor utilization and minimize the total hardware cost. Therefore, similar to intra-task customization, we generate a *Pareto curve* containing the *Pareto-optimal* set of *utilization-area* vectors $\{(U_1, A_1), \ldots, (U_n, A_n)\}$ with the trade-off between processor utilization U and hardware area A.

In Figure 1, we showed the intra-task custom instruction selection for two different tasks. For T_1 and T_2 we have 4 and 6 elements in the custom instruction configurations respectively. For example, for the task T_1, we have $\{(w_{1,1} = 10, c_{1,1} = 0), (w_{1,2} = 8, c_{1,2} = 30), (w_{1,3} = 7, c_{1,3} = 60), (w_{1,4} = 5, c_{1,4} = 90)\}$. The goal is to identify the *utilization-area* Pareto curve for the task set $\{T_1, T_2\}$. As shown in Figure 1, in this case the Pareto curve consists of six solutions $\{(1, 80), (\frac{19}{20}, 90), \ldots, (\frac{3}{4}, 170)\}$. Without using any custom instructions, the task set $\{T_1, T_2\}$ is not schedulable with $U = \frac{5}{4} > 1$. But the use of custom instructions speed up task executions, thereby lowering the utilization. This yields six schedulable solutions with conflicting *utilization-area* trade-offs.

3. EVALUATING DESIGN TRADE-OFFS

We shall now present our algorithms for efficiently computing the Pareto curves in the *intra-task* and the *inter-task* custom instruction selection phases. Note that computing the exact Pareto curves in both these cases is computationally intractable. First, such Pareto curves would typically contain an exponential number of trade-off points (which obviously cannot be computed in polynomial time). Second, it may be shown using a reduction from the classical Knapsack problem, that the problem of computing even one point on the Pareto curve is NP-hard. Hence, our algorithms approximate both – the number of points on the Pareto curve, as well as the "coordinates" of these points on the curve.

3.1 Intra-Task Trade-offs

In what follows, we first present a pseudo-polynomial time dynamic programming algorithm (called DP) to compute the *exact* Pareto curve, which is then used to devise an approximation scheme. Let the maximum cost associated with any custom instruction be M, i.e., $M = max_{(j=1,2,\ldots,n_i)} a_{i,j}$, where n_i is the number of custom instruction candidates for the task T_i and $a_{i,j}$ is the hardware cost associated with the jth custom instruction. Let $\Omega_{k,j}$ be the minimum workload that might be achieved by considering only a subset of custom instructions of task T_i from $\{1, 2, \ldots, k\}$ when the cost is exactly j. The algorithm DP first initializes $\Omega_{0,j} = \infty$ for $j = \{1, 2, \ldots, n_i M\}$ and $\Omega_{j,0} = E_i$ for $j = \{0, 1, \ldots, n_i\}$. Note that $n_i M$ is an upper bound on the total hardware cost that might be incurred. After initialization, the DP computes the values of $\Omega_{k,j}$ ($k = 1$ to n_i) by the recursion:

$$\Omega_{k,j} \leftarrow \min\{\Omega_{k-1,j}, \Omega_{k-1,j-a_{i,k}} - \delta_{i,k}\} \quad (1)$$

That is, given an area j, we explore all possible configurations and choose the one that results in minimum workload for custom instructions $1, \ldots, k$.

After running DP, we retain the undominated solutions from amongst the solutions in the final iteration ($k = n_i$) to obtain the exact *workload-area* Pareto curve, $\{(w_{i,1}, c_{i,1}), \ldots, (w_{i,N_i}, c_{i,N_i})\}$, where N_i is the size of the Pareto curve. This algorithm runs in pseudo-polynomial time $O(n_i^2 M)$, and will suffer from long running times. Hence, our goal is to *approximately* compute this curve in *polynomial* time.

Our approximation scheme takes an error parameter ϵ as input and returns an *ϵ-approximate Pareto curve* denoted as ϵ-Pareto curve (or $\mathcal{P}_\epsilon$). Given a Pareto curve $\mathcal{P} = \{(x_1, y_1), \ldots, (x_p, y_p)\}$, an ϵ-approximate Pareto curve $\mathcal{P}_\epsilon$ is defined as *any* set $\mathcal{P}_\epsilon = \{(x'_1, y'_1), \ldots, (x'_q, y'_q)\}$ such that for any $(x_i, y_i) \in \mathcal{P}$, there exists a $(x'_j, y'_j) \in \mathcal{P}_\epsilon$ for which $x'_j \leq (1+\epsilon)x_i$ and $y'_j \leq (1+\epsilon)y_i$. In other words, corresponding to any point on the Pareto curve $\mathcal{P}$, there exists a point on $\mathcal{P}_\epsilon$, each of whose coordinates are at most ϵ distance away from the corresponding coordinates of the point on $\mathcal{P}$.

Papadimitriou and Yannakakis [13] has shown that for any multi-objective optimization problem and any ϵ, there *exists* a polynomial-sized ϵ-approximate Pareto curve $\mathcal{P}_\epsilon$. Further, [13] showed that a necessary and sufficient condition for computing such a $\mathcal{P}_\epsilon$ in polynomial time is the existence of a polynomial-time algorithm for solving, what was referred to as the *GAP problem*. In what follows, we state the version

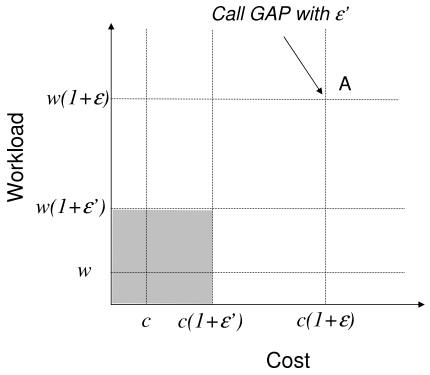


Figure 2: Solving the GAP problem for the corner point A will either return a dominating solution or declare that there is no solution in the shaded area.

of the GAP problem that arises in our setting and show that it can be solved in polynomial time. Finally, we outline our scheme to compute the approximate Pareto curves using the polynomial time GAP subroutine.

3.1.1 The GAP Problem

For a two-dimensional multiobjective optimization problem, the GAP problem can be stated as follows: Given a vector $b = (b_1, b_2)$, either return a solution vector which dominates b, or report that there is no solution better than b by at least a factor of $1+\epsilon$ in both dimensions. In our setting, the objective is to minimize the workload of a task T_i, $W(S) = E_i - \sum_{j=1}^{n_i} x_{i,j}\delta_{i,j}$ and the cost $C(S) = \sum_{j=1}^{n_i} x_{i,j}a_{i,j}$, where $x_{i,j}$ is a boolean variable which is true if the jth custom instruction is chosen for the solution S. Hence, the corresponding GAP problem can be stated as follows.

Problem Statement: Given a cost c, workload w and an $\epsilon \geq 0$, either return a solution S such that $C(S) \leq c$ and $W(S) \leq w$, or else declare that there is no solution S such that $C(S) \leq \frac{c}{1+\epsilon}$ and $W(S) \leq \frac{w}{1+\epsilon}$.

Solving the GAP Problem: We now present a polynomial-time algorithm to solve this GAP problem. It involves the following two steps:

Step 1: *Transforming Costs*
Let $r = \lceil \frac{n_i}{\epsilon} \rceil$. Modify each cost $a_{i,j}$ of task T_i to $a'_{i,j}$ such that $a'_{i,j} = \lceil \frac{a_{i,j}r}{c} \rceil$. Now, consider the problem of determining whether there exists a solution with the modified costs such that $C'(S) \leq r$. Let us call this problem GAP$'$. We shall show that solving GAP is equivalent to solving GAP$'$. Towards this, we enumerate two properties below. These properties can be easily proved with algebraic equations.

(a) If a solution with the transformed costs satisfies $C'(S) \leq r$, then $C(S) \leq c$.
(b) If a solution satisfies $C(S) \leq \frac{c}{1+\epsilon}$, then $C'(S) \leq r$.

From property (a), we know that if this problem returns an affirmative answer then the GAP problem would also return a dominating solution. On the other hand, if GAP$'$ returns a negative answer then property (b) leads to the conclusion that there is no solution with cost $\leq c/(1+\epsilon)$. Hence, from the above properties we can infer that solving GAP$'$ is equivalent to solving the original GAP problem.

Step 2: *Solving GAP$'$*
We present a dynamic programming algorithm to solve the GAP$'$ problem. This algorithm can be constructed with the following adjustments to Algorithm DP.

Algorithm 1 Approximating the Pareto curve.

1: Partition the range of costs from 1 to n_iM geometrically with a ratio $1+\epsilon' = (1+\epsilon)^{1/2}$, thus dividing the cost space into $O(log_{1+\epsilon}n_iM)$ coordinates.
2: For each coordinate b, call *Algorithm DP* with transformed costs $a'_{i,j} = \lceil \frac{a_{i,j}r}{b} \rceil$, where $r = \lceil \frac{n_i}{\epsilon} \rceil$.
3: For each run of Step 2, find the solution with the minimum utilization.
4: Retain all the undominated solutions from the solutions found in Step 3. This will represent a ϵ-Pareto curve.

1. Run Algorithm DP with the modified costs $a'_{i,j}$.
2. Instead of iterating over all the cost values up to n_iM, iterate only up to a cost value of r, where $r = \lceil \frac{n_i}{\epsilon} \rceil$.
3. Finally, if the minimum value in the final array computed by Algorithm DP is such that it is $\leq w$, then return the solution otherwise declare that there is no solution.

Computing each row of the table built by this dynamic programming algorithm requires $O(r)$ running time. Hence, this algorithm runs in time $O(n_i^2/\epsilon)$.

The above polynomial time subroutine for solving the GAP problem proves the existence of a *fully polynomial-time approximation scheme* (FPTAS) for computing the approximate workload-area Pareto curve $\mathcal{P}_\epsilon$ which is polynomial in the input size and in $1/\epsilon$. This is because the following FPTAS can be devised using the algorithm for solving GAP. First, geometrically partition the objective space along all dimensions with a ratio $1+\epsilon'$, where $\epsilon' = (1+\epsilon)^{1/2} - 1$. For each corner point of this grid, call the GAP routine (i.e., the algorithm for solving GAP) with the parameter ϵ', and keep all the undominated solutions (see Figure 2 for an illustration of this procedure). This implies that for each rectangle which contains a solution in the exact Pareto curve, there will also be a solution within the same rectangle which belongs to $\mathcal{P}_\epsilon$. The *distance* between these two solutions can be bounded using the dimensions of the rectangle. Hence, for every solution s in the Pareto curve, there exists a solution q in $\mathcal{P}_\epsilon$ such that $\frac{q}{(1+\epsilon)} \leq s$. Moreover, because the number of rectangles is polynomially bounded, it follows that the number of points in $\mathcal{P}_\epsilon$ will also be a polynomial.

Algorithm 1 summarizes the above steps to compute the ϵ-approximate *workload-area* Pareto curve in some more detail. Note, that in step 1 of Algorithm 1 we partition only the area space (and not both workload and area space). This is because if a point (w, c) dominates the corner (w_1, c_1) and $w_1 < w_2$, then (w, c) definitely dominates (w_2, c_2) . In steps 2 and 3, we scale the costs, run Algorithm DP for every co-ordinate in the partitioned cost space and retain the minimum workload at each co-ordinate. The runtime complexity of this algorithm is $O(\frac{n_i^2}{\epsilon} log_{1+\epsilon}n_iM)$.

3.2 Inter-Task Trade-offs

The existence of an inter-task approximation scheme to compute the *utilization-area* Pareto curve may be argued in the same fashion as for the intra-task approximation scheme described above. This scheme takes the set of pareto-optimal solutions $\mathcal{P}_i$ for each task T_i as input (as shown in the previous section), and generates the set of global design trade-offs $\bar{\mathcal{P}}$ for the entire task set. Each global design configuration $S \in \bar{\mathcal{P}}$ contains contains exactly one solution from each $\mathcal{P}_i$ (for each task T_i). If a brute-force approach is

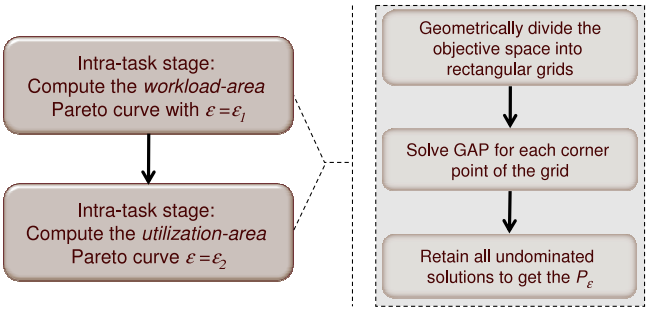


Figure 3: The overall two-stage approximation scheme.

used to examine all possible global design configurations, then $|\mathcal{P}_1| \times \ldots \times |\mathcal{P}_m|$ solutions will have to be examined, where $|\mathcal{P}_i|$ denotes the number of solutions in the set $\mathcal{P}_i$. Hence, the number of solutions grow exponentially with the number of tasks m in the task set, and moreover, not all of these solutions would be optimal (i.e., they would be *dominated* by some Pareto-optimal solution). Our goal is to instead efficiently (but approximately) compute just the Pareto-optimal global design configurations.

Broadly, the procedure follows the same steps as described in Algorithm 1. Due to space constraints, we shall not work out all the details of this stage. However, note that the main difference is the core dynamic programming recursion that is invoked in Step 2 (Algorithm 1), which we present below. Let $U_{i,j}$ be the minimum utilization that might be achieved by considering only a subset of tasks from $\{1, 2, \ldots, i\}$ when the cost is exactly j. Then, $U_{i,j}$ is defined recursively as below, where $\{w_{i,k}, c_{i,k}\} \in \mathcal{P}_i$, (workload-area Pareto curve), and E_i and P_i denote the *execution* time and the *minimum separation* of the task T_i.

$$U_{i,j} \leftarrow \min \left\{ \begin{array}{l} U_{i-1,j}, U_{i-1,j-c_{i,1}} - (E_i - w_{i,1})/P_i \\ \vdots \\ U_{i-1,j}, U_{i-1,j-c_{i,N_i}} - (E_i - w_{i,N_i})/P_i \end{array} \right\} \quad (2)$$

We summarize the overall two-stage approximation scheme in Figure 3. There are two distinct stages: (i) the intra-task stage to compute the *workload-area* Pareto curve for each task, and (ii) the *inter-task* stage which generates the *utilization-area* Pareto curve for the entire task set. The scheme for approximating the Pareto curve follows the three main steps shown on the right hand of Figure 3. Note that at each stage, the approximation scheme takes as an input an error parameter ϵ (chosen by the designer) and returns an ϵ-*approximate* Pareto curve. These parameters might be different for the two stages.

4. EXPERIMENTAL EVALUATION

In this section we report some of the experimental results obtained by running our approximation algorithm on a set of well-known benchmarks. We compare the running times of the optimal algorithm against our approximation scheme, and also illustrate the difference in the sizes of P_ϵ (the approximate Pareto curves) and the exact Pareto curve.

Experimental setup: We use five WCET benchmarks [14] *(compress, jfdctint, ndes, edn, adpcm)*, two benchmarks *(aes, sha)* from MiBench [8], three benchmarks *(g721 encoder, djpeg, cjpeg)* from MediaBench [10] and one *(ispell)* Trimaran benchmark [16] for our experiments.

Given the C code of an application, we use the Trimaran compilation framework to generate optimized intermediate code, as well as profiling information such as basic block execution frequencies. We then construct the data flow graph for each basic block and enumerate all possible custom instructions with at most 4 input operands and 2 output operands [17]. We use Synopsys design tools with 0.18 micron CMOS cell libraries to synthesize and estimate latency/area of custom instructions. Execution cycles of a custom instruction is its latency normalized against a Multiply-Accumulate (MAC) operation, which has 1 cycle latency in the processor running at 120MHz. Hardware area is represented in terms of the number of adders. Further, we assume a single-issue in-order base processor core with a perfect cache. The workload is measured in number of processor cycles required to execute each task.

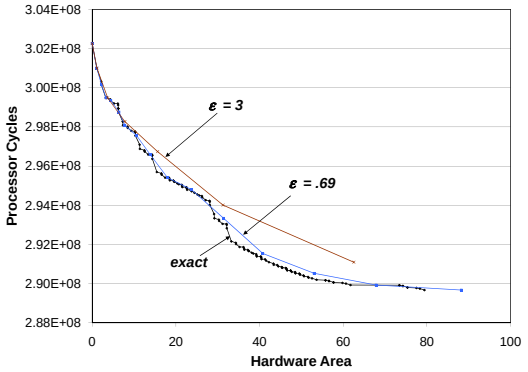
Task Set	Benchmarks
1	cjpeg, adpcm, aes, compress, rijndael ispell
2	djpeg, g721decode, cjpeg, ispell, adpcm jfdctint, aes
3	cjpeg, ispell, edn, sha, g721decode, djpeg compress, ndes
4	adpcm, rijndael, cjpeg, ispell, sha ndes, djpeg, compress, edn
5	aes, djpeg, g721decode, rijndael, jfdctint cjpeg, edn, ispell, sha, ndes

Table 1: Composition of the task sets.

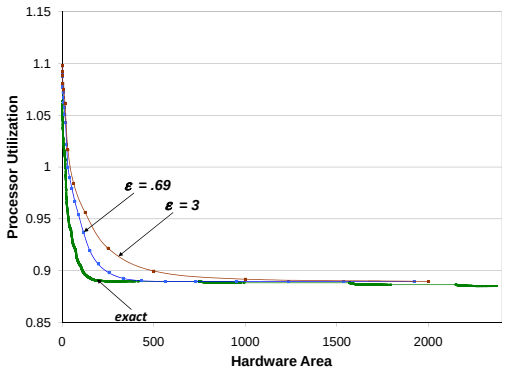
We create five task sets (see Table 1) with the number of tasks in each set varying from 6–10. We chose a total utilization for the task set (without any custom instructions) and then select individual minimum separation period for each task (P_i) to achieve the corresponding utilization. We also chose five different utilization factors $U = 0.80$, 1.00, 1.05, 1.08 and 1.10. When $U = 0.8$ or 1.0, a task set is schedulable without using any custom instructions. In these cases, we are interested in finding out by how much we can reduce the utilization through custom instructions and what are the hardware trade-offs. For $U > 1.0$, a task set is not schedulable on its own. Here, the goal is to find schedulable solutions by using custom instructions as well as expose the performance-area trade-offs. All CPU times reported below are measured on a desktop with 3.0 GHz Pentium 4 CPU and 1 GB RAM.

Running times: Table 2 shows the running time speedups resulting from our approximation scheme, compared to computing the exact Pareto curve for three different values of ϵ, for each of the five task sets. Computing the exact Pareto curve for task sets 1–5 require 139.78 sec, 514.20 sec, 622.32 sec, 747.17 sec, and 711.52 sec, respectively. Even for small values of ϵ ($\epsilon = 0.44$) our approximation algorithm runs about three orders of magnitude faster than the exact algorithm. For larger values of ϵ (e.g., $\epsilon = 3$), the speedups are even more significant (note that ϵ need not be ≤ 1). The reason behind choosing the values 0.21, 0.44, and 0.69 for ϵ is as follows. Our approximation algorithm involves the computation of $(1 + \epsilon)^{1/2}$. This value might turn out to be an irrational number if ϵ is not carefully chosen. Hence, to avoid any possible rounding-off errors in our implementation, the above values were chosen for ϵ.

Pareto curve size: The *workload-area* Pareto curve (the output of intra-task stage) and the *utilization-area* Pareto curve (the output of intra-task stage) typically contain an exponential number of points. The approximation algorithm generates a polynomial-sized approximate Pareto curve $\mathcal{P}_\epsilon$. We now compare the sizes of the exact Pareto curve and $\mathcal{P}_\epsilon$.



(a) *workload-area* Pareto curve for *g721decode*



(b) *utilization-area* Pareto curve for task set 1

Figure 4: The exact and approximate Pareto curves for $\epsilon = 0.69, 3$.

Task Sets:	1	2	3	4	5
$\epsilon = 0.21$	643	1075	1037	990	729
$\epsilon = 0.44$	3248	5918	5712	5457	3933
$\epsilon = 0.69$	7106	14587	14389	13922	10208
$\epsilon = 3.0$	29615	72525	89285	69054	77508

Table 2: Speedup obtained from our approximation scheme for the task sets $1-5$.

For the intra-task results, Figure 4(a) shows the exact Pareto curve and the P_ϵ generated by our algorithm for the *g721decode* benchmark. For the inter-task case, we show the results for task set 1 in Figure 4(b). For clarity, we have only plotted $\mathcal{P}_\epsilon$ for $\epsilon = 0.69$ and 3. Note that (i) the number of points in P_ϵ decrease as ϵ increases, and (ii) the gap between the exact and approximate curves widen with larger values of ϵ, implying that the relative error indeed increases. We would like to report that even for small values of ϵ (e.g., $\epsilon = 0.21$), P_ϵ contains almost 97% fewer points compared to the exact Pareto curve. Similar trends were seen for all the other benchmarks and task sets, which are not shown here due to space constraints.

Benefits of approximation: Although the running times associated with constructing the exact Pareto curve might seem to be small (10–12 mins), in an interactive design process where the designer repeatedly makes changes and generates new Pareto curves, this might hamper designer productivity. A tool which generates these curves faster (e.g., using our proposed approximation algorithms) would be more usable. Secondly, exact Pareto curves return too many (similar) design trade-offs. Approximate Pareto curves return less, well spread out trade-offs, which might be more manageable for the designer.

5. CONCLUDING REMARKS

In this paper, we proposed a framework for evaluating trade-offs in custom instruction selection for instruction set customizable processors. This framework consists of two stages – in the first custom instruction configurations representing different trade-offs are chosen for each task, and in the second, different configurations from each task are chosen to derive system-level trade-offs. There are a couple of directions in which this framework can be further refined, e.g., by modeling isomorphic instructions across tasks or by accounting for more general scheduling policies for task sets (currently only EDF has been modeled).

6. REFERENCES

[1] M. Arnold and H. Corporaal. Designing domain-specific processors. In *CODES*, 2001.

[2] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *DAC*, 2003.

[3] S. Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *RTSS*, 1990.

[4] N. Cheung, S. Parameswaran, and J. Henkel. INSIDE: INstruction Selection/Identification & Design Exploration for extensible processors. In *ICCAD*, 2002.

[5] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. In *MICRO*, 2003.

[6] J. Cong, Y. Fan, G. Han, and Z. Zhang. Application-specific instruction generation for configurable processor architectures. In *FPGA*, 2004.

[7] K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, 2001.

[8] M. R. Guthaus et al. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE Annual Workshop on Workload Characterization*, 2001.

[9] H. P. Huynh and T. Mitra. Instruction-set customization for real-time systems. In *DATE*, 2007.

[10] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO*, 1997.

[11] J. Lee, K. Choi, and N. Dutt. Efficient instruction encoding for automatic instruction set design of configurable ASIPs. In *ICCAD*, 2002.

[12] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[13] C. H. Papadimitriou and M. Yannakakis. On the approximability of trade-offs and optimal access of web sources. In *FOCS*, 2000.

[14] F. Stappert. WCET benchmarks. http://www.c-lab.de/home/en/download.html.

[15] Tensilica - XPRES Compiler - Optimized Hardware Directly from C. www.tensilica.com/products/devtools/hw_dev/xpres/.

[16] Trimaran: An infrastructure for research in backend compilation and architecture exploration. http://www.trimaran.org.

[17] P. Yu and T. Mitra. Scalable custom instructions identification for instruction-set extensible processors. In *CASES*, 2004.

[18] P. Yu and T. Mitra. Satisfying real-time constraints with custom instructions. In *CODES+ISSS*, 2005.