Satisfying Real-Time Constraints with Custom Instructions

Pan Yu
panyu@comp.nus.edu.sg

Tulika Mitra
tulika@comp.nus.edu.sg

School of Computing
National University of Singapore
Republic of Singapore 117543

ABSTRACT

Instruction-set extensible processors allow an existing processor core to be extended with application-specific custom instructions. In this paper, we explore a novel application of instruction-set extensions to meet timing constraints in real-time embedded systems. In order to satisfy real-time constraints, the worst-case execution time (WCET) of a task should be reduced as opposed to its average-case execution time. Unfortunately, existing custom instruction selection techniques based on average-case profile information may not reduce a task's WCET. We first develop an Integer Linear Programming (ILP) formulation to choose optimal instruction-set extensions for reducing the WCET. However, ILP solutions for this problem are often too expensive to compute. Therefore, we also propose an efficient and scalable heuristic that obtains quite close to the optimal results. Experiment results indicate that suitable choice of custom instructions can reduce the WCET of our benchmark programs by as much as 42% (23.5% on an average).

Categories and Subject Descriptors: C.3 [Real-time and embedded systems]

General Terms: Algorithms, Performance, Design.

Keywords: Customizable processors, instruction-set extensions, real-time systems, worst-case execution time.

1. INTRODUCTION

Instruction-set extensible processors allow a processor core to be extended with application-specific custom instructions [1, 8, 9, 18]. These processors have become popular as they strike the right balance between challenging performance requirement and short time-to-market constraints of embedded systems. Custom instructions encapsulate the frequently occurring computation patterns in an application and are implemented as custom functional units (CFU). CFUs improve performance through parallelization and chaining of operations, and also save the fetch, decode, and selection overhead of several base instructions. Thus custom instructions help simple embedded processors achieve considerable performance and energy efficiency.

CODES+ISSS'05, Sept. 19–21, 2005, Jersey City, New Jersey, USA.

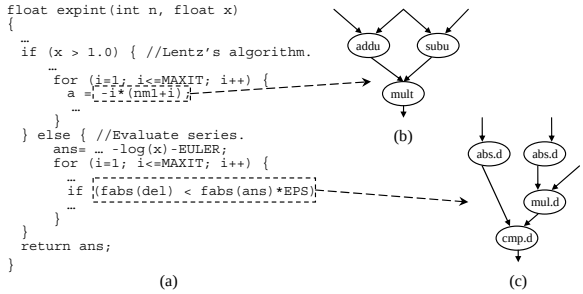


Figure 1: Motivating Example

In this paper, we explore a novel application of instruction-set extensions in the domain of embedded real-time systems. The input to a real-time scheduler is a set of tasks with their corresponding execution time, period, and deadlines. If there does not exist any feasible schedule that meets all the deadlines, then the designer is left with two choices. The first option is to raise the processor's clock frequency (at the cost of increased power consumption) or choose a different higher performance processor. However, it may not always be possible to increase the clock frequency further or change the processor. The second option is to optimize the code so as to reduce the execution time. Again, the current code may already be fully optimized. For these scenarios, we propose using custom instructions to reduce the execution time such that the system can meet hard real-time constraints. The availability of processor cores with programmable logic for CFUs [1, 18] makes this quite a cost-effective solution.

The heart of the problem is then to choose an appropriate set of custom instructions for an application. It is a difficult problem even for non real-time applications. Significant research effort has been invested in developing automated selection techniques. Unfortunately, we cannot apply these techniques out-of-the-box for real-time applications.

The goal of traditional custom instruction selection problem is to reduce the *average case execution time* (ACET) of the application. Therefore, these techniques rely on the execution frequencies of the code fragments through profiling. For real-time tasks, on the other hand, custom instructions should reduce the *worst case execution time* (WCET). The WCET of a task is defined as its maximum execution time for all possible inputs. Let us illustrate the difference with

an example. Figure 1 shows a code fragment that computes the value of the exponential integral. There are two candidate patterns — one from each side of a conditional branch. Let us assume that we can implement only one custom instruction. For reducing the ACET, the pattern selection will depend on the frequency of execution of these two patterns. However, this selection may not be optimal for WCET reduction as the frequently executed pattern may not contribute to the worst-case execution path. For example, if the `else` part of the conditional branch contributes towards the worst case path, then the pattern on the `else` part should be selected for WCET reduction.

Moreover, it is not sufficient to use the execution frequencies corresponding to the WCET path and then employ the traditional custom instruction selection techniques. The WCET path may not remain the same throughout the selection process. Once we have selected some custom instructions to reduce the current WCET path, a new path may become the WCET path. Therefore, custom instruction selection problem for WCET reduction is more challenging compared to ACET reduction.

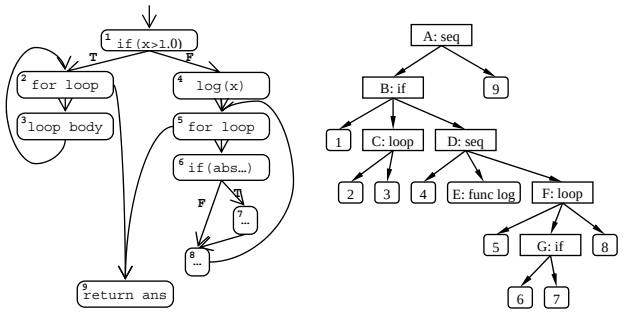
In this paper, we first provide an *Integer Linear Programming* (ILP) formulation of the custom instruction selection problem for real-time tasks. The formulation guarantees optimal reduction of the WCET. However, the ILP formulation is compute intensive and even fails to terminate within reasonable time for large applications. So, we also propose a scalable heuristic algorithm that achieves close to the optimal result. The contributions of this work are the following.

- To the best of our knowledge, ours is the first work that explores the possibility of using custom instructions to meet the timing constraints in real-time systems.
- Custom instructions should reduce the WCET of a real-time task as opposed to ACET. Towards this end, we provide both an optimal solution based on ILP and a heuristic algorithm. Our results indicate that an appropriate choice of custom instructions can reduce WCET by as much as 42% (23.5% on an average).

2. RELATED WORK

The design space exploration problem to select suitable custom instructions for an application consists of two steps. The first step [2, 3, 6, 7, 11, 21] identifies a large set of candidate patterns from the program's dataflow graph and their frequencies via profiling. Given this library of patterns, the second step selects a subset to maximize the average case performance under constraints on number of allowed custom instructions and/or area budget. Various different approaches have been proposed for this step such as dynamic programming [2], branch-and-bound [19], 0-1 Knapsack [7], greedy heuristic [5, 6, 13, 20], and ILP [13, 20]. In contrast, our goal is to improve the worst case performance.

Compiler techniques to reduce the worst case execution time of a program have started to receive attention very recently. Reduction in WCET has been achieved though dual instruction sets [14], reordering of the sequence of compiler optimizations [22], and code positioning [23]. [14, 23] greedily optimize the current WCET path till there is a shift to another path. However, this approach may miss the global optima as it does not consider closely competing paths simultaneously. [22] uses genetic algorithm. In contrast, our ILP formulation ensures the optimality of the solution. Also, our heuristic takes a more global perspective and selects a pattern that reduces the WCET across all the paths.



(a) CFG (b) Syntax tree

Figure 2: CFG and syntax tree corresponding to the code in Figure 1

3. PROBLEM FORMULATION

Given an application, we first use our previously proposed algorithm [21] to exhaustively identify all possible computation patterns in that application. All the generated patterns satisfy certain pre-defined constraints on the maximum number of allowed input and output operands. Let us assume that we have identified N candidate patterns in a program denoted by $C_1 \dots C_N$. A pattern C_i can have n_i different instances occurring in the program denoted by $c_{i.1} \dots c_{i.n_i}$. Let P_i be the performance gain obtained by implementing C_i in hardware as opposed to software. R_i is the amount of area (hardware blocks) required to implement C_i. Suppose we have a constraint on the total number of custom instructions that can be implemented in the architecture, say M ($M < N$). Then our goal is to cover each original instruction in the code with zero/one instance of at most M custom instructions, such that the WCET of the task is minimized. Similarly, we may have a constraint that the total amount of area required by the selected custom instructions should not exceed R.

Since we need to improve the WCET, the problem formulation is intrinsically related to the method used for estimating the WCET. In this work, we use the *Timing Schema* approach to estimate the WCET of a task.

3.1 WCET Analysis using Timing Schema

Timing schema is an efficient technique to estimate the WCET of a structured program [15]. The structure of the program is represented as a hierarchical syntax tree with basic blocks as leaf nodes and control structures (i.e., sequences, branches, and loops) as interior nodes. The entire program is represented at the root of the syntax tree. Figure 2 shows control flow graph (CFG) [1] and syntax tree corresponding to the code in Figure 1. Function calls are represented by leaf nodes (e.g., node E in Figure 2 (b)). A separate syntax tree is constructed for each function. So the entire program is represented as a *syntax forest*.

WCET of a program is estimated by traversing its hierarchical syntax tree in a bottom-up fashion. First, the execution times of the leaf nodes, i.e., basic blocks are obtained

[1]We build CFG for optimized assembly code. The figure uses source code for illustration purpose only.

(e.g., by counting the number of execution cycles for each basic block). For each interior node V of the syntax tree, this method computes $wcet(V)$ that represents the WCET of the code fragment corresponding to V as a function of the WCETs of its children as follows:

Basic block:	wcet(V) = constant
Sequence:	wcet(V1;V2) = wcet(V1) + wcet(V2)
Branch:	wcet(if V1 then V2 else V3) = wcet(V1) + max(wcet(V2),wcet(V3))
Loop:	wcet(for V1 loop V2) = (n+1) × wcet(V1) + n×wcet(V2)

where the loop iterates at most n times. The WCET of a function is computed at the root node of its syntax tree. The WCET of a program is computed at the root node of its `main` function. There are other sophisticated schemas [16] for capturing infeasible paths, unstructured programs and timing effects due to cache, pipeline. However, this simple timing schema suffices to illustrate the concept.

4. OPTIMAL SOLUTION USING ILP

We can formulate the selection of optimal instruction-set extension for minimizing the WCET as an Integer Linear Programming (ILP) problem. The objective function minimizes the WCET of the root node of the `main` function:

$$minimize : wcet_{\texttt{main}}$$

The first part of the ILP formulation defines $wcet_{\texttt{main}}$ in terms of the WCET of the basic blocks using timing schema. The rules of the timing schema can be easily mapped to a set of linear equations. The second part defines the WCET of the basic blocks in the presence of custom instructions.

$wcet_{\texttt{main}}$ depends on the WCET of its children as discussed in Section 3.1. Let V be a non-leaf node in the syntax tree and let $V_1 \dots V_k$ be its children. If V is a sequence node, then following timing schema, we have $wcet_V = \sum_{i=1}^{k} wcet_{V_i}$. If V is a conditional branch, then it has at most three children corresponding to the condition (V_1), taken (V_2), and non-taken (V_3) paths (if any), respectively. Then,

$$\begin{aligned} wcet_V &\geq wcet_{V_1} + wcet_{V_2} \\ wcet_V &\geq wcet_{V_1} + wcet_{V_3} \end{aligned}$$

If V is a loop node with loop bound n and two children corresponding to the condition ($V1$) and the loop body ($V2$), then $wcet_V = (n+1) \times wcet_{V_1} + n \times wcet_{V_2}$. If node V represents a call to a function `func`, then $wcet_V = wcet_{\texttt{func}}$.

Now, we define the WCET of the leaf nodes (basic blocks) in the presence of custom instructions. WCET of a basic block depends on the selection of the custom instructions. Let us define binary variables $s_{i.j}$ $(1 \leq i \leq N; 1 \leq j \leq n_i)$ corresponding to each of the custom instruction instances. $s_{i,j}$ is equal to 1 if custom instruction instance $c_{i.j}$ is selected and 0 otherwise. Similarly, we define binary variable S_i $(1 \leq i \leq N)$ to be equal to 1 if custom instruction C_i is selected and 0 otherwise. That is,

$$\begin{aligned} S_i = \quad & 1 \quad \text{if } \sum_{j=1}^{n_i} s_{i.j} > 0 \\ = \quad & 0 \quad \text{otherwise} \end{aligned}$$

The following logically equivalent linear equations can substitute the above non-linear equations.

$$\sum_{j=1}^{n_i} s_{i.j} - U \times S_i \leq 0$$

$$\sum_{j=1}^{n_i} s_{i.j} + 1 - S_i > 0$$

where U is a large constant greater than $max(n_i)$.

Let T_V be the original execution time of a basic block V without any custom instruction. Let $c_{a.b} \dots c_{e.f}$ be the custom instruction instances that can possibly cover instructions of basic block V. Then,

$$wcet_V = T_V - (P_a \times s_{a.b} + \dots + P_e \times s_{e.f})$$

Now, we express the various constraints for this optimization problem. First, a base instruction in the program can be covered by at most one custom instruction instance. If $c_{x.y} \dots c_{w.z}$ cover a base instruction, then $s_{x.y} + \dots + s_{w.z} \leq 1$. Second, if M is the constraint on the maximum number of custom instructions allowed, then $\sum_{i=1}^{N} S_i \leq M$. Similarly, if R is the total area budget for implementing all custom instructions, then $\sum_{i=1}^{N} S_i \times R_i \leq R$.

5. HEURISTIC ALGORITHM

Solving ILP formulation takes prohibitively long time as the number of patterns and their instances increases. Therefore, we also present an efficient and scalable heuristic algorithm for this problem. We first introduce a greedy heuristic algorithm. Subsequently we improve the heuristic to take care of its limitations.

Algorithm 1: Custom Instruction Selection Heuristic

Input: P: all patterns
Output: pat, ins: selected patterns, instances

```
1 m := 0; pat := φ; ins := φ;
2 while m < M do
3     ∀p ∈ P compute profit(p);
4     Let p ∈ P be the pattern with max profit;
5     if profit(p) = 0 then return pat;
6     add p to pat; remove p from P;
7     add selected instances of p to ins;
8     m := m + 1; wcet := wcet − profit(p);
end
```

Algorithm 1 shows the heuristic for selecting custom instructions such that the WCET of the program is minimized. We iteratively select the pattern that reduces the WCET most (defined by the $profit$ function). We continue until either the maximum number of custom instructions allowed in the architecture is reached (M in Algorithm 1) or no further reduction is possible (line 5). The $profit$ of a pattern is defined as the reduction in the program's WCET if the pattern is chosen as custom instruction. Note that the profit function computes the global WCET reduction (considering *all* paths) as opposed to just execution time reduction of the current WCET path. This difference is important if we have two or more closely competing worst case paths. A pattern that reduces the execution time across all these competing paths will be a better choice than the one that only reduces the execution time of the current worst case path.

Notice that the selection of a pattern does not imply selection of *all* its instances (line 7). This is because (1) an instance of the currently selected pattern may overlap with

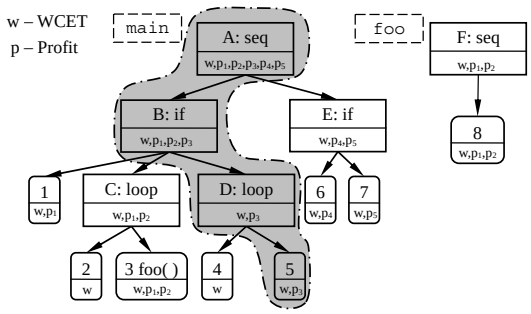


Figure 3: Efficient computation of profit function.

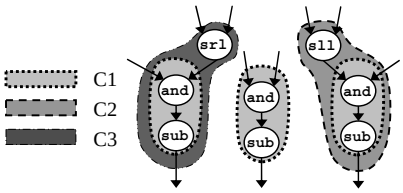


Figure 4: Limitation of the heuristic.

an instance of a previously selected pattern, and (2) two or more instances of the selected pattern may overlap among themselves. We handle the first case by ignoring the instances of the currently chosen pattern that overlap with previously selected patterns. In the second case, we have to choose a subset of the currently selected pattern's instances such that there is no overlap. However, selecting the optimal subset is too compute intensive. Therefore, we again apply a greedy selection process. The overlapping instances of a pattern always belong to a single basic block. The instances are selected according to the order in which they appear inside the basic block. An instance cannot be selected if it conflicts with a previously selected instance. Note that the selection of pattern instances is performed during the profit calculation itself (line 3); otherwise, the profits will be over-estimated.

If we have a constraint on total area instead of number of instructions, then we choose the pattern with the best profit/area ratio (line 4) until we cannot fit any pattern within the remaining area.

5.1 Computing Profits for Patterns

Our algorithm requires to re-compute profits for all the unselected patterns at each iteration. This is because of two reasons. First, the selection of a pattern may make some new paths competing for the worst case and hence the profit values of all the patterns change. Recall that the profit of a pattern is defined as the reduction in the program's WCET if the pattern is chosen as custom instruction. Second, a selected pattern eliminates certain other overlapping pattern instances from further consideration. For example, selection of the pattern $C1$ in Figure 4 implies that the pattern instance of $C2, C3$ cannot be selected in future. The eliminated pattern instances cannot contribute towards reducing the execution time and hence the profit values of the corresponding patterns should be re-computed. A naive computation of the profits requires a bottom-up traversal of the entire syntax tree for each pattern. We avoid this costly computation based on the following optimizations.

1. We can compute profits for all the patterns through a single traversal of the syntax tree.
2. As all the instances of a pattern are typically localized in the program, selection of a pattern requires update of only a small portion of the syntax tree.

During the initialization phase, we compute profit values for all the patterns through a single bottom-up traversal of the syntax tree. We also annotate each node of the syntax tree with (1) the profit values for all the patterns appearing in the corresponding code fragment and (2) the WCET of that code fragment. We first compute the profit values at the leaf nodes (basic block). The computation of profits at an interior node applies rules similar to timing schema for WCET computation except for the branch nodes. Let V be a branch node with C, T, F as the children corresponding to conditional, taken and non-taken paths, respectively. Then profit of a pattern p at the branch node V is defined as

$$profit(p, V) = wcet(V) - (wcet(C) - profit(p, C)) - max\,(wcet(T) - profit(p, T), wcet(F) - profit(p, F))$$

The root node is annotated with all the patterns in the program. As the instances of a pattern are typically localized, number of patterns is quite small for most of the interior nodes, as shown in Figure 3.

Once a pattern is chosen at an iteration, we identify the leaf nodes (basic blocks) in which its selected instances appear. We re-compute the profits for all the unselected overlapping patterns in these leaf nodes. Changes in a leaf node are propagated towards the root of the syntax tree. The only nodes that need to be updated in this phase are the nodes that lie on the path from the root to a modified leaf node. The shaded nodes in Figure 3 gives an example of update after the selection of $C3$.

With this optimization, the complexity of the algorithm is $O(M \times |P| \times D \times A)$ where M is the number of patterns to be selected from a library of $|P|$ patterns, D is the height of the syntax tree and A is the average number of selected instances of a pattern.

5.2 Improving the Heuristic

The greedy heuristic presented in the previous subsection runs pretty fast. Unfortunately, it makes poor choices in the presence of *subsumed* patterns. We call p a subsumed pattern of q if there exists at least one instance of pattern p that is fully covered by an instance of q. We call q the *subsuming* pattern. As the greedy heuristic chooses the pattern with the maximum profit at each iteration, it typically favors subsumed patterns. However, this choice may not be globally optimal as the selection of a subsumed pattern eliminates some of the subsuming pattern instances from further consideration. For example, in Figure 4, the performance gain of custom instructions $C1$, $C2$ and $C3$ are 1, 2 and 2 cycles, respectively. Also, all the instances contribute towards the reduction of WCET. The greedy heuristic will choose $C1$ and all its instances leading to a total profit of 3 cycles. However, the optimal solution in this case is one instance of $C1$, $C2$, and $C3$ each for a total profit of 5 cycles.

We take care of this problem in the improved heuristic shown in Algorithm 2 as follows. Instead of simply se-

Algorithm 2: Improved Custom Instruction Selection Heuristic

`selectPatterns(`in`)`
Input: in: partial selection of patterns, instances, and corresponding wcet
Output: complete selection of patterns, instances, and corresponding wcet

1 **if** $in.m = M$ **then** return in;
2 Let $p \in (P - in.pat)$ be the pattern with max $profit$;
3 **if** $profit(p) = 0$ **then** return in;
4 Let $ins(p)$ be the selected instances of pattern p;
5 $tmp.m := m + 1$; $tmp.wcet := in.wcet - profit(p)$; $tmp.pat := in.pat \cup p$; $tmp.ins := in.ins \cup ins(p)$;
6 $choice1$:= `selectPatterns` (tmp);
7 **if** $subsuming(p) - in.pat = \phi$ **then** return $choice1$;
8 Let $q \in subsuming(p) - in.pat$ be the pattern with max $profit$;
9 **if** $profit(q) = 0$ **then** return $choice1$;
10 Let $ins(q)$ be the selected instances of pattern q;
11 $tmp.m := m + 1$; $tmp.wcet := in.wcet - profit(q)$; $tmp.pat := in.pat \cup q$; $tmp.ins := in.ins \cup ins(q)$;
12 $choice2$:= `selectPatterns` (tmp);
13 **if** $choice1.wcet \leq choice2.wcet$ **then** return $choice1$; **else** return $choice2$;

Program	Source	WCET cycles
adpcm†	SNU suite	3,365,394
blowfish	Mibench	4,847,327
compress†	Gothenburg	56,428
crc†	SNU suite	42,227
djpeg	MediaBench	13,447,397
gsmdec	MediaBench	28,163,930
g721dec	MediaBench	28,420,193
ndes†	FSU suite	47,897
rijndael	Mibench	1,835,219
sha	Mibench	356,061

Table 1: Benchmark Characteristics.

lecting the pattern with the maximum profit (pattern p in Algorithm 2) and eliminating all the subsuming patterns' instances from further consideration, we also make an alternative choice by selecting a subsuming pattern with the maximum profit (pattern q). The search then proceeds corresponding to these two choices separately (lines 6 and 12, respectively). The inputs to the recursive `selectPatterns` function are the patterns and instances selected so far and the corresponding WCET. The function returns with the complete selection of up to M patterns. Finally, the WCET corresponding to the two choices are compared (line 13) and the better one is selected. Experimental results show that this simple modification reduces the WCET by an additional 2%–23% for our benchmarks.

6. EXPERIMENTAL EVALUATION

Table 1 shows the characteristics of the benchmark programs selected from MediaBench [12], MiBench [10] and WCET-specific application suite [17] (marked by †). We use SimpleScalar tool set [4] for the experiments. The programs are compiled using gcc 2.7.2.3 with -O3 optimization. We assume a single-issue in-order base processor core with perfect cache and branch prediction. We assume that loop bounds are provided through manual annotation to compute the WCET of a program. All the experiments are performed on a Pentium4 1.7Ghz PC with 1GB memory.

Program	No. Pat.	No. Inst.	WCET Red. Heur.	WCET Red. Opt.	Time (sec) Heur.	Time (sec) Opt.
adpcm	51	150	9%	9%	0.002	0.02
blowfish	15	276	16%	16%	0.002	0.02
compress	37	92	2%	2%	0.002	0.01
crc	12	23	15%	15%	0.001	0.01
djpeg	64	485	7%	7%	0.017	0.12
gsmdec	158	2312	21%	22%	0.031	0.10
g721dec	73	180	4%	4%	0.006	0.03
ndes	22	77	10%	10%	0.002	0.12
rijndael	49	2520	16%	16%	0.034	1.25
sha	9	40	12%	12%	0.001	0.01

Table 2: WCET Reduction under 5 custom instruction constraint with constrained topology.

Program	No. Pat.	No. Inst.	WCET Red. Heur.	WCET Red. Opt.	Time (sec) Heur.	Time (sec) Opt.
adpcm	101	258	14%	14%	0.005	0.04
blowfish	56	1221	39%	39%	0.012	11.1
compress	141	248	6%	6%	0.003	0.02
crc	24	39	17%	17%	0.001	0.01
djpeg	226	1056	11%	11%	0.028	0.30
gsmdec	796	6782	26%	26%	0.064	0.28
g721dec	220	392	11%	11%	0.010	0.05
ndes	77	182	17%	18%	0.003	0.03
rijndael	156	9032	39%	39%	0.096	943
sha	47	148	31%	31%	0.002	0.04

Table 3: WCET Reduction under 5 custom instruction constraint with relaxed topology.

Given a binary executable of an application, we first exhaustively enumerate all possible patterns and their instances under certain pre-defined constraints on the maximum number of input and output operands using our previously proposed algorithm [21]. The hardware latency of a custom instruction is approximated as the summation of the latencies of the original operations along the critical path of its dataflow graph. Latency values for the original operations from the base ISA are obtained through Synopsys synthesis tool with a popular cell library. Finally, execution cycles of a custom instruction is computed by normalizing its latency (rounded up to an integer) against that of a multiply-accumulate (MAC) operation, which we assume takes exactly one cycle. We do not include floating-point operations, memory accesses, and branches in custom instructions as they introduce non-deterministic behavior.

We use ILOG CPLEX, which is a leading commercial linear programming solver, to obtain the optimal solutions. `lp_solve`, a popular public domain linear programming solver, fails to terminate within reasonable time for most problems. We compute WCET reduction as follows:

$$Reduction = \frac{Original\ WCET - Reduced\ WCET}{Original\ WCET} \times 100\%$$

We perform the custom instruction selection under a variety of scenarios in order to stress our heuristic algorithm. The number of patterns and instances has direct impact on the time required to solve the optimization problem. We control the number of patterns generated for a benchmark by imposing different constraints on the number of input and output operands allowed for a pattern. Our pattern generation phase only emits the patterns that satisfy the operand constraints. First, we consider a *constrained topology* that

Program	WCET Red.		Time (sec)	
	Heur.	Opt.	Heur.	Opt.
adpcm	16%	16%	0.02	0.04
blowfish	42%	42%	0.04	2.11
compress	7%	7%	0.01	0.01
crc	20%	20%	0.01	0.01
djpeg	13%	13%	0.12	0.38
gsmdec	28%	28%	0.32	0.39
g721dec	13%	13%	0.08	0.15
ndes	19%	20%	0.01	0.03
rijndael	40%	40%	0.11	120
sha	37%	37%	0.01	0.04

Table 4: WCET Reduction under 10 custom instruction constraint with relaxed topology.

Program	WCET Red.		Time (sec)	
	Heur.	Opt.	Heur.	Opt.
adpcm	12%	12%	0.02	0.05
blowfish	41%	42%	0.05	2.70
compress	7%	7%	0.02	0.02
crc	20%	20%	0.01	0.01
djpeg	13%	13%	0.10	8.10
gsmdec	25%	26%	0.25	2.60
g721dec	12%	12%	0.03	0.18
ndes	18%	19%	0.02	0.30
rijndael	40%	40%	0.48	295
sha	37%	37%	0.03	0.02

Table 5: WCET Reduction under resource constraint (area of 20 adders) with relaxed topology.

allows at most 2 register inputs, 1 immediate input and 1 register output for each custom instruction. This is realistic for most modern processors without major impact on their ISA format and micro-architecture. Second, we consider a more aggressive *relaxed topology* that allows at most 4 inputs (either register or immediate value) and 2 outputs. The relaxed topology results in significantly more number of patterns (*No. Pat.*) and instances (*No. Inst.*) compared to the constrained topology.

Table 2 and Table 3 show the WCET reduction if we can implement at most 5 custom instructions under constrained and relaxed topology, respectively. We observe that custom instructions can indeed reduce the WCET of a program significantly and make it easier for a real-time task to meet its deadline. Even with constrained topology and a limit of only 5 custom instructions, we can still achieve up to 22% reduction in worst case execution time. Allowing relaxed topology obtains further reduction of WCET.

We also note that our improved heuristic (*Heur.* in the Tables) obtains close to the optimal results at a fraction of the ILP (*Opt.*) solving time. A comparison of the solution time (*Time* column) in Table 2 and Table 3 shows that the heuristic is quite scalable as we increase the problem size; but ILP is not. ILP solution time increases from 1.25 sec to 943 sec for the `rijndael` benchmark as we increase the number of patterns. In fact, with even more relaxed topology constraint, there are a few cases that *CPLEX ILP solver cannot solve even after 24 hours*. The heuristics only takes a few seconds and the result produced is better than the intermediate result returned by CPLEX after 24 hours.

Table 4 shows the effectiveness and scalability of the heuristic algorithm under increased number of allowed custom instructions. As expected, allowing more custom instructions reduces the WCET further. Table 5 shows that our heuristic algorithm is equally effective if we have resource constraint (area corresponding to 20 adders) instead of constraint on the total number of custom instructions.

7. CONCLUSION

Instruction-set extensions open the opportunities for program acceleration. In this work, we have exploited this opportunity to satisfy timing constraints for real-time tasks. Our methodology is tailored towards improving the worst case execution time as opposed to average case execution time. In the future, we would like to extend this methodology for multiple real-time tasks mapped to one processor.

8. ACKNOWLEDGMENTS

This work was partially supported by NUS research grants R252-000-088-112, R252-000-171-112 and A*STAR Project 022/106/0043.

9. REFERENCES

[1] Altera. Nios embedded processor, 2003.
[2] M. Arnold and H. Corporaal. Designing domain-specific processors. In *CODES*, 2001.
[3] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *DAC*, 2003.
[4] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2), 2002.
[5] N. Cheung, S. Parameswaran, and J. Henkel. Inside: Instruction selection/identification & design exploration for extensible processors. In *ICCAD*, 2002.
[6] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. In *MICRO*, 2003.
[7] J. Cong et al. Application-specific instruction generation for configurable processor architectures. In *FPGA*, 2004.
[8] P. Faraboschi et al. Lx: A technology platform for customizable VLIW embedded processing. In *ISCA*, 2000.
[9] R. E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2), 2000.
[10] M. R. Guthaus et al. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE Annual Workshop on Workload Characterization*, 2001.
[11] R. Kastner et al. Instruction generation for hybrid reconfigurable systems. *ACM TODAES*, 7(4), 2002.
[12] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO*, 1997.
[13] J. Lee, K. Choi, and N. Dutt. Efficient instruction encoding for automatic instruction set design of configurable asips. In *ICCAD*, 2002.
[14] S. Lee et al. A flexible tradeoff between code size and WCET using a dual instruction set processor. In *SCOPES*, 2004.
[15] C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on source-level timing schema. *IEEE Computer*, 24(5), 1991.
[16] G. Pospischil et al. Developing real-time tasks with predictable timing. *IEEE Software*, 9(5), 1992.
[17] F. Stappert. WCET benchmarks. Available from `http://www.c-lab.de/home/en/download.html`.
[18] Stretch. S5000 software-configurable processors, 2004.
[19] F. Sun, S. Ravi, A. Raghunathan, and N.K.Jha. Custom-instruction synthesis for extensible-processor platforms. *IEEE TCAD*, 23(2), 2004.
[20] P. Yu and T. Mitra. Characterizing embedded applications for instruction-set extensible processors. In *DAC*, 2004.
[21] P. Yu and T. Mitra. Scalable custom instruction identification for instruction-set extensible processors. In *CASES*, 2004.
[22] W. Zhao et al. Tuning the WCET of embedded applications. In *RTAS*, 2004.
[23] W. Zhao et al. WCET code positioning. In *RTSS*, 2004.