LETTER *Special Section on Reconfigurable Systems*

Optimisations Techniques for the Automatic ISA Customisation Algorithm

Antoine TROUVE[†a)], ***Nonmember and*** **Kazuaki MURAKAMI**[†], ***Member***

SUMMARY This article introduces some improvements to the already proposed custom instruction candidates selection for the automatic ISA customisation problem targeting reconfigurable processors. It introduces new opportunities to prune the search space, and a technique based on dynamic programming to check the independence between groups. The proposed new algorithm yields one order less measured number of convexity checks than the related work for the same inputs and outputs.

key words: *reconfigurable computing, custom instruction generation, optimisation*

1. Introduction

Dynamically reconfigurable processors (DRP) are processors which a part of the instruction set architecture (ISA) can be modified at runtime. By making it possible to adapt the instruction set to the executed application, such processors can potentially be more efficient than their general purpose counter-parts. However, the success of DRPs has been hindered by two challenges: (1) the performance of the hardware lags behind other solutions (2) the problem of finding the best ISA per application is intractable. This article focuses on the second one.

Automatic ISA customisation algorithms (AICA) aim at extracting from a given input applications the custom ISA (made of custom instructions, written CIs) which yields the best performance with respect to some important metrics (decided by the user). The custom ISA can be generated at run-time [1] or at compile time [2]–[5]. Hardware generation enables binary compatibility with existing general purpose processors (GPP). However, the quality of so-generated CIs is lower than with software implementations as designers have to keep the AICA simpler to mitigate the hardware overheads due to the presence on the chip of a hard-wired ACA. This article focuses on AICA implemented at the compiler intermediate representation (IR) level, as a compiler pass [2], [4], [5], [7], [9].

The remainder of this article is organised as follow. The two first sections present the related work and the context regarding the exact solution CI candidate selection for AICA. The next section presents some optimisations to the algorithm, which constitutes the article contribution, before assessing their benefits through experiments.

Manuscript received April 27, 2011.
Manuscript revised August 12, 2011.

[†]The authors are with the Department of Informatics, Kyushu University, Fukuoka-shi, 812–8581 Japan
a) E-mail: trouve@isit.or.jp

DOI: 10.1587/transinf.E95.D.437

2. Related Work

The automatic ISA customisation problem is intractable. In a program P containing $|P|$ instructions, there are $2^{2^{|P|}}$ different possible custom ISA (the number of sub-sets of the set containing all the clusters of instructions). Hence, most of the works in the literature endeavour to mitigate the complexity of the problem. First, many approximate solutions have been proposed. For instance Pozzi et al. [4] use genetic algorithms to iteratively select the best CI candidates. Clark et al. [2] use a greedy algorithm which grows clusters of CIs until some conditions are satisfied. Brisk et al. [3] tries to find common patterns in the data flow graph. While they have prove to be able to generate good CIs under some conditions, those algorithms do behave poorly in the general case. Mei et al. [9] use an original approach based on loop scheduling which yields good results, but only for applications in which such loop analysis can be carried out successfully. On the other hand, some other works strive to solve the problem exactly by smartly pruning the design space. Atasu et al. [5] proposed such algorithm which leverages a topological sorting of the nodes and take into consideration micro-architectural constraints early. More recently, Ahn et al. [7] introduced a method to balance the memory usage and the execution time through dynamic programming techniques and memoisation.

This article aims at introducing further improvement of Atasu et al.'s algorithm using a different approach than Ahn et al. [7] (i.e. memoisation). Especially, it concentrates on the CI candidate selection phase, which aims at parsing all the clusters of nodes of the programs (the *candidate CIs*) which are likely to be included into the custom ISA.

3. Definition of the Problem

3.1 Automatic ISA Customisation

Taking into input a data-flow graph (DFG) generated by the compiler at the IR level, the automatic ISA customisation problem consists in finding which nodes or cluster of nodes (the *groups*, or *cuts*) should become parts of the custom ISA. In the case where the processor does not contain any base ISA (i.e. a general purpose one), the custom ISA should be an exact cover of the program's DFG. A AICA is similar to hardware / software partitioning and can be clustered into two sub problems. First, the *CI candidate selection* explores

all the possible groups inside the program and keeps them into a pool $\mathscr{P}$. Second, the *CI election* selects the subset $P \in 2^{\mathscr{P}}$ of the pool which yields the better results regarding an objective function: this is the custom ISA. In the work by Atasu et al. [5] the CI election phase strives to find only one CI per compilation unit ($|P| = 1$); this is however not mandatory and one can think of extracting more CIs per compilation unit ($|P| \geq 1$). The focus of this work is the CI candidate selection phase. Like Atasu et al. [5] and Ahn et al. [7], we limit the study to groups which do not include any branches; in other words the compilation unit is the basic block. The reason for this limitation is that including branches in CIs raises several challenges which resolution is not the purpose of this article; in particular expanding the compilation unit to more than one basic block may have a severe impact on the measured compilation time.

3.2 CI Candidate Selection

CI candidate selection boils down to visiting all the possible sub-graphs of a program's data flow graph (DFG). As the compilation unit is a basic block, the considered DFGs do not contain any cycle. As introduced by [5], CI candidate selection consists in building groups by adding nodes one by one. This operation is called *growing*. For a program P, it defines initially the set $Grow_1 = \{\{u\} \mid u \in P\}$; then iterates on this set to define the $Grow_i$ ($i > 1$) through the following recursion:

$$Grow_i = \{G \cup \{u\} \mid G \in Grow_{i-1}, u \notin G\} \quad (1)$$

At each iteration, the algorithm determines among the elements of $Grow_i$ which would be suitable to be candidate CI. Those are to be the inputs of the CI election phase downstream (not considered in this article).

In the worse case for a program P, $|Grow_i| = \binom{i}{|P|}$, and there are $\sum_{j=1}^{i} \binom{j}{|P|}$ candidate CIs [†]. However, as suggested by other authors [5], [8], the search space can be pruned regarding micro-architectural and convexity constraints. The former consists in removing groups which can not be mapped on the target hardware. Convexity is an intrinsic mandatory property for a CI to be mapped on a datapath. A group G is convex if $\forall$ nodes $u, v \in G$, $\forall$ path $\mathcal{P}$ between u and v, $\forall\, w \in \mathcal{P}$, $w \in G$. Atasu et al. [5] mentions that convexity can be checked quickly but omits to develop further. Ahn et al. [7] introduces an algorithm which makes it possible to check the convexity in $O(1)$ by maintaining for each growing group a set of *watcher inputs*, which negatively impacts the memory usage of the algorithm exponentially in the worse case. The stance of this paper is that convexity checks are expensive (wether in execution time or memory usage), and that reducing their number is an important challenge when designing a CI candidate selection algorithm.

Moreover, the same authors further noticed that it is not mandatory to consider all the $G \cup \{u\}$ ($G \in Grow_i$ and $u \in P$). By defining a topological order on the nodes of P (noted $\prec$) so that $\forall u, v \in P$, $u \prec v$ if and only if there is a path from u to v, and by considering for a group G only the elements of $\{u \mid \forall g \in G,\ u \prec g\}$, it is possible to mitigate the number of groups to be considered by the algorithm. By doing so, Eq. (1) can be re-written as follow:

$$Grow_i = \{G \cup \{u\} \mid G \in Grow_{i-1}, u \prec G\} \quad (2)$$

This article uses this algorithm as a baseline and introduces a method which aims at reducing the number of convexity checks it performs by noticing that most of the $G \cup \{u\}$, $u \prec G$ are not convex (cf. Eq. (2)).

4. Improvements to the CI Candidate Selection

4.1 Reducing the Number of Groups to Visit

For two nodes $u, v \in P$, let us write $u \rightarrow v$ if there is an edge in the DFG from u to v; $u \rightsquigarrow g$ if there exists a path from u to v in the DFG. Moreover, let us define $P_{u \rightsquigarrow v}$ the set of all such paths. For a group G, let us also write $\mathscr{C}(G)$ to express the convexity of G. Let us further define the notion of parallelism between groups and nodes:

Definition 1. *Let F and G be two groups on the program P. We say that F and G are **parallels** if there are no data dependencies between both. The notation $F \parallel G$ will be used to express this property.*

Using those vocabulary and notations, the following property can be demonstrated:

Theorem 1. *During the growing phase of the group G, for all node g' so that $g' \prec G$,*

$$\mathscr{C}(G \cup \{g\}) \Rightarrow \begin{cases} g \rightarrow G \text{ or} \\ g \parallel G \end{cases}$$

The relation is an equivalence in the second case.

Proof. Let us demonstrate the first part of the theorem. Let G be a group on the program P, and let us consider $g \in P$, a node visited during group gathering to grow $G \in Grow_{|G|}$ into $G' = G \cup \{g\}$ ($g \prec G$) so that $\mathscr{C}(G')$. First, let us consider the case where $P_{g \rightsquigarrow G} \neq \emptyset$; there exists a path ϕ in $P_{g \rightsquigarrow G}$. By definition, $\forall p \in \phi \backslash g,\ p \in G$. Hence, $g \rightarrow G$ (1). In the case where $P_{g \rightsquigarrow G} = \emptyset$, there are no path from g to G. Moreover, as $g \prec G$, there are no path from G to g neither (2). By (1) and (2), the first property is demonstrated.

The second property stances that additionally, $g \parallel G \Rightarrow (C)(G \cup \{g\})$. This is trivial as in this case $P_{g \rightsquigarrow G} = \emptyset$ by definition. □

For the sake of the further explanations, let us define the following sets for a group G: $\vec{G} = \{u \mid u \rightarrow v, u \notin G, v \in G\}$ and $\hat{G} = \{u \mid u \parallel G\}$.

[†]Which grows exponentially with i and $|P|$ according to Lovasz's bound

Using those definitions and the properties demonstrated in this section, it is now possible to re-write the definition of $Grow_i$ (previously defined in Eq. (2)) as bellow:

$$Grow_i = \{G \cup \{u\} \mid G \in Grow_{i-1},\ u \in \vec{G} \cup \hat{G},\ u \prec G\} \quad (3)$$

This article proposes to leverage this property to reduce the average number of convexity checks performed by the CI candidate selection. It considers $\vec{G}$ and $\hat{G}$ instead of P as a whole to grow groups, and checks the convexity of the so-obtained groups only in the first case.

4.2 Fast Calculation of $\hat{G}$

For a group G to grow, the previous section put into light two sets, $\vec{G}$ and $\hat{G}$, useful to reduce the number of convexity checks performed by the algorithm. $\vec{G}$ corresponds to the set of the inputs of G and is straightforward to calculate. This is not the case for $\hat{G}$. This section further introduces a strategy based on dynamic programming to decide the ownership of a nodes to this set.

Let us first define the sets $Out(G)$ and $In(G)$ to designate the input and output nodes of a group G as bellow:

$$Out(G) = \{u \in G \mid \exists\, v \notin G,\ v \rightarrow u\}$$
$$In(G) = \{u \in G \mid \exists\, v \notin G,\ u \rightarrow v\}\ (\ = \vec{G}\ )$$

Those sets can be parsed in constant time if we consider that $\exists\, M_{in}, M_{out} \in \mathbb{N},\ |In(G)| < M_{in}, |Out(G)| < M_{out}$ for all groups G an a program P. Additionally, let us define for a program P the closure of a node $u \in P$ (already mentioned in Sect. 3.2) as the set $\bar{u} = \{v \mid u \rightsquigarrow v\}$.

With those definitions, the following property can be demonstrated:

Theorem 2. *For a group G and a node $u \in P$, $u \prec G$*

$$G \parallel u \Leftrightarrow In(G) \cap \bar{u} = \emptyset \Leftrightarrow \forall v \in In(G),\ v \notin \bar{u}$$

Proof. Let us notice first that $(In(G) \cap \bar{u} = \emptyset) \Leftrightarrow (\forall v \in In(G),\ v \notin \bar{u})$. With that said, let us demonstrate the first equivalence. To this end, let us notice that there is no path from G to u because $u \prec G$. Then, let us consider u, $G \parallel u$. Furthermore, there is no path from u to G neither by definition of $\parallel$, i.e. G is not in $\bar{u}$. This proves $\Rightarrow$. Let us consider $\exists\, v \in G \cap \bar{u}$, this implies that there exists a path from u to v and $v \in G$ so there is a path from u to G and they are not parallel. This proves $\Leftarrow$ ad absurdum. □

Hence, the pre-computation of the closure of all nodes of P makes it possible to decide $G \parallel u$ in f operations in average, if we consider a data structure which makes it possible to check the ownership to a set (e.g. $\bar{u}$) in $O(1)$ operations, and if f is the average value of $|In(G)|$.

Algorithm 1: The new growing function

```
Input: N node of the program (filtered and sorted), G the group
       to grow
Output: The grown groups
1  NewGrowing ← ∅;
2  for n ∈ G⃗, n ≺ G do
3  |   F ← G ∪ {n};
4  |   if 𝒞(F) then
5  |   |   NewGrowing ← NewGrowing ∪ {F};
6  |   end if
7  end for
8  for n ∈ N, n ≺ G⃗ do
9  |   if ∀g ∈ In(G), g ∉ n̄ then
10 |   |   F ← G ∪ {n};
11 |   |   NewGrowing ← NewGrowing ∪ {F};
12 |   end if
13 end for
14 → NewGrowing;
```

4.3 Consequence on the Growing Function

The new growing function for CI candidate selection which takes advantage of the theorems introduced in the previous sections is presented in Alg. 1. It consists of two loops. The first loop (l. 2 to 7) considers opportunities to grow the group G by parsing $\vec{G}$. It contains one convexity check at line 4.

The second loop (l. 8 to 13) looks up for parallel nodes; to this end it parses $\hat{G}$ by first considering all the nodes $u \prec \vec{G}$ and keeping only those which closure does not intersect with $In(G)$ as suggested by theorem 2. No convexity check is performed in this loop (theorem 1).

5. Experimental Results

The previous section shown that by pre-computing the closure of all the nodes of a program P and taking advantage of theorem 1, one can potentially sigificatly reduce the execution time of the AICA first proposed by Atasu et al. [5]. This section shows experimental results led on the top of the COINS compiler infrastructure [†] which confirms those theoretical results.

Results are shown in Fig. 1 for the programs in MiBench [††]. The horizontal axis represents all the basic blocks in the programs. *Atasu* expresses the number of convexity checks carried out by the algorithm presented in [5]. The series *Convexity* plots the number of convexity checks performed by the algorithm 1. Our algorithm performs 50.12 times less convexity checks than the algorithm from Atasu et al. if we consider the arithmetic mean of the values. This ratio however hides great disparities: if we consider the geometric mean it drops down to 2.85. Indeed the optimisations presented in this paper are more efficient on large basic blocks on which the opportunities to grow groups using parallel nodes are statistically more important. However, most

[†] www.coins-project.org

[††] Online: http://www.eecs.umich.edu/mibench/

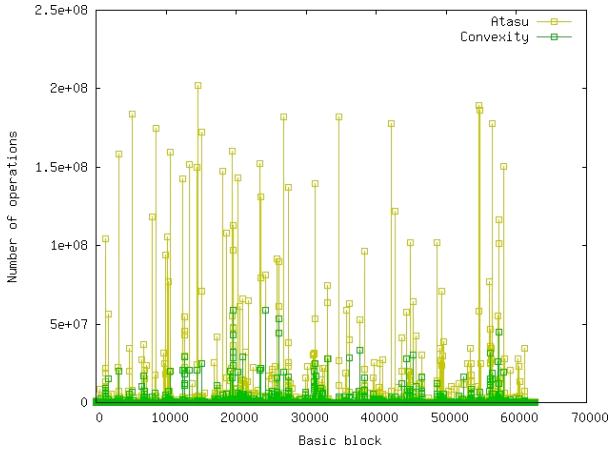

Fig. 1 Nodes visited and convexity check with our method.

basic blocks are small ones (less than 20 operations).

6. Conclusion

This article introduces some enhancement for the automatic ISA customisation algorithm targeting reconfigurable processors. It shows that by dividing the exploration space into two parts during the growing phase of the groups, it is possible to significantly reduce the workload of the algorithm in average: around 50 times less with respects to the number of convexity checks.

References

[1] A.C.S. Beck, M.B. Rutzig, G. Gaydadjiev, and L. Carro, "Transparent reconfigurable acceleration for heterogeneous embedded applications," DATE, pp.1208–1213, 2008.
[2] N. Clark, H. Zhong, and S. Mahlke, "Processor acceleration through automated instruction set customization," MICRO36, 2003.
[3] P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzedeh, "Instruction generation and regularity extraction for reconfigurable processors," CASES, Oct. 2002.
[4] L. Pozzi, K. Atasu, and P. Ienne, "Exact and approximate algorithms for the extension of embedded processors instruction sets," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.25, pp.1209–1229, 2006.
[5] K. Atasu, L. Pozzi, and P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," DAC, pp.256–261, June 2003.
[6] H. Noori, F. Mehdipour, K. Inoue, and K. Murakami, "A reconfigurable functional unit with conditional execution for multi-exit custom instructions," IEICE Trans. Electron., vol.E91-C, no.4, pp.497–508, April 2008.
[7] J. Ahn, I. Lee, and K. Choi, "A polynomial-time custom instruction identification algorithm based on dynamic programming," ASP-DAC, pp.573–578, Jan. 2011.
[8] P. Biswas, N. Dutt, P. Ienne, and L. Pozzi, "Automatic identification of application-specific functional units with architecturally visible storage," DATE, 2006.
[9] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Dresc: A retargetable compiler for coarse-grained reconfigurable architectures," Field-Programmable Technology, pp.166–173, Dec. 2002.