A Code-Generator Generator for Multi-Output Instructions

Hanno Scharwaechter, Rainer Leupers, Gerd Ascheid and Heinrich Meyr
Integrated Signal Processing Systems
RWTH Aachen University, Germany

Jonghee M. Youn and Yunheung Paek
Software Optimization & Restructuring Research Group
Seoul National University, Korea

ABSTRACT

We address the problem of instruction selection for Multi-Output Instructions (MOIs), producing more than one result. Such inherently parallel hardware instructions are very common in the area of Application Specific Instruction Set Processors (ASIPs) and Digital Signal Processors (DSPs) which are frequently used in System-on-Chips as programmable cores. In order to provide high-level programmability, and consequently guarantee widespread acceptance, sophisticated compiler support for these programmable cores is of high importance. Since it is not possible to model Multi-Output Instructions as trees in the compiler's Intermediate Representation (IR), traditional approaches for code selection are not sufficient. Extending traditional code-generation approaches for MOI-selection is essentially a graph covering problem, which is known to be NP-complete. We present a new heuristic algorithm incorporated in a retargetable code-generator generator capable of exploiting arbitrary inherently parallel MOIs. We prove the concept by integrating the tool into the LCC compiler which has been targeted towards different Instruction Set Architectures based on the MIPS architecture. Several network applications as well as some DSP benchmarks were compiled and evaluated to obtain results.

Categories and Subject Descriptors

C.3 [**Special Purpose and Application Based Systems**]: [Microprocessor/Microcomputer Systems]

General Terms

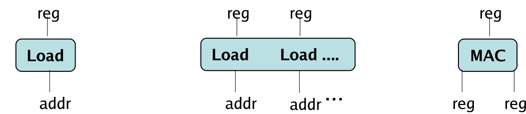
Design

Keywords

Compiler/ Architecture Co-Design, Code-Selection, ASIP, ISS

1. INTRODUCTION

Design and development of embedded processors are usually done under very tight constraints. The processors have to handle high data rates either in *digital signal* or *network protocol* processing, consume as little power as possible and have to be programmable in order to guarantee reusability towards different applications. Therefore, particularly in protocol processing, *Application Specific Instruction Set Processors* (ASIPs) are more and more applied as *Intellectual Property*-blocks in *System-On-Chips*, since they offer a good compromise between efficiency and flexibility. Typically, ASIPs provide a set of special instructions, tailored to the needs of their target applications. The predominant feature of these instructions is inherent parallelism. The instructions comprise several parallely executed operations like ADD, MUL, MAC etc. and produce multiple results at the same time. Consequently they are referred to as *Multi-Output Instructions* (MOIs).



(a) Simple Instruction (b) Multi-Output Instruction (c) Chained Instruction

Figure 1: IR-Patterns for Different Instructions

It is exactly this property which makes the difference between MOIs and other kinds of instructions. Whereas simple instructions (figure 1(a)) and chained instructions (figure 1(c)) can be represented as tree patterns in the IR, MOIs will always have a fanout larger than one (figure1(b)).

In the area of *digital signal processing*, MOIs are already a natural way to increase code performance. Prominent examples are instructions to support access of different memory banks at the same time. For example in Sony pDSP processor, an instruction such as *PLDXY r1, @a, r2, @b* can load variables a and b from memory into registers r1 and r2 simultaneously. These instructions can access memory faster by performing loads and stores in parallel on partitioned memory banks using parallel data and address buses. Designers for embedded DSPs prefer such techniques over more complicated hardware mechanisms. By the encapsulation of parallel operations in hardware instructions, impressive speedups can also be obtained for protocol processing without sacrificing too much flexibility for the implementation of applications [22]. However, the increasing complexity of network protocols makes it prohibitively difficult to implement them in assembly language. Sophisticated compiler support is therefore strongly demanded in order to guarantee fast application development and consequently consumer acceptance for a processor. This implies the problem of handling MOIs by a compiler which is currently not possible, since typical compilers rely on *tree parsing* [1] during the code selection phase. If the target processor architecture contains inherently parallel instructions, the code produced by tree parsing may strongly deviate from the optimum. The reason for this is that the scope of tree parsing is restricted to *Data-flow Trees* (DFTs). Consequently, instructions comprising functionality that exceeds the scope of one DFT cannot be matched by tree parsing, as their associated IR-patterns features a fan-out larger than one. In order to overcome this, the scope has to be extended at least to the size of a basic block which is then represented as a *Data-flow Graph* (DFG). Unfortunately, pattern matching on DFGs is NP-complete in general [24]. Usual approaches to solve this problem rely on *Compiler Known Functions* (CKFs) and *Inline Assembly*. In both cases, the applications have to be modified manually which may lead to overhead for big applications.

Additionally, the extremely heterogeneous landscape of application specific architectures demands for highly flexible programming tools. Such architectures are usually developed in an iterative manner during which the architecture is incrementally refined. Obviously, compilers have to be easily retargetable in order to support ar-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CODES+ISSS'07, September 30–October 3, 2007, Salzburg, Austria.
Copyright 2007 ACM 978-1-59593-824-4/07/0009 ...$5.00.

bitrary kinds of instruction sets during the processor development. There exists several code-generator generators (such as Burg [9], Iburg [8], Olive [23]) which produce code-selector descriptions in C. Such code-selector descriptions can be very comfortably integrated into existing compiler backends. Basically, such a code-generator generator accepts a tree grammar as input which models the instruction set of the underlying target architecture in terms of IR-patterns. Based on this grammar, C code for instruction selection is produced. Since this tree grammar only comprises tree patterns, the produced code-selectors lack of capability to exploit Multi-Output Instructions.
In this paper a new tool is presented that extends the concept of tree-based code-selector generators towards a graph-based code-selector generator called Cburg. Cburg extends the concept of the mentioned traditional tools in the way that the grammar is changed and a graph-based matching algorithm is produced such that Multi-Output Instructions can be modeled and matched. Thereby, manual insertion of inline assembly or CKFs into the application is omitted. The concept is proven by integrating Cburg into the LCC compiler [3] which has been targeted for an extended ASIP based on the MIPS architecture [19].
The remainder of this paper is organized as follows: In the following section 2, related work on code selection for embedded processors is mentioned and our contribution is put into the context. Section 3 explains the developed heuristic algorithm to exploit MOIs during the code selection phase. Subsequently, the retargetable implementation of the algorithm is introduced in section 4 and experimental results are presented in section 5. The paper concludes with section 6.

2. RELATED WORK

Due to the limitations of tree parsing, several contributions have been published in the past, dealing with a generalization of tree-based code selection. In [7], optimal graph-based code selection is described for regular data path architectures without instruction level parallelism. However, ASIPs mostly feature irregular *Instruction Set Architectures* (ISAs) comprising parallel instructions. Especially for the domain of signal processing, where it is most natural to have MOIs, further approaches have been developed for irregular architectures. Araujo presents a solution to effectively break up the DFG into expression trees while taking irregular register architectures into account [2]. It is shown in [15] that tree parsing leads to suboptimal results in the presence of MOIs. To overcome this, a new technique based on the splitting of instructions into *register transfer* primitives and recombining these primitives in an optimal manner using integer linear programming is proposed. Liao solves the same problem by augmenting a binate covering formulation [16]. Unfortunately, Liao does not provide results on the applicability of their algorithm. Furthermore, both approaches [15, 16] feature exponential runtime in the worst case based on the size of considered DFGs. This might be disadvantageous for large benchmarks. Liem [17] has proposed a graph-based instruction selection methodology that features a flexible pattern representation style. This style includes data-flow, control-flow and mixed data/control-flow informations. In [6], a code selection methodology is described for complete functions in order to take control flow into account. A *Static Single Assignment* (SSA)-Representation is used as IR and the pattern matching is solved numerically. As the code selection was tested for a *Very Long Instruction Word* (VLIW)-processor, instruction level parallelism in terms of MOIs is not an issue for this approach. The most recent approach to graph-based code selection is presented in [4]. The paper targets code selection algorithms for hardware accelerators based on unate covering. As the authors only consider single connected IR-patterns, Multi-Output Instructions are not in the paper's scope. In contrast, Multi-Output Instructions can also comprise multiple independent patterns on IR-level.
Summarized, the existing approaches either feature exponential runtime on the size of considered DFGs or do not consider inherent instruction parallelism producing multiple results. Our contribution within this paper is the presentation of a new tool called Cburg. Cburg is very similar to well known code-selector generators like Iburg or Olive. The difference between Cburg and other tools is firstly the kind of accepted input grammar and secondly, the produced algorithm to match IR-patterns. In contrast to other tools, Cburg's grammar is not restricted to tree patterns and therefore capable to model arbitrary inherently parallel instruction patterns. Furthermore, Cburg produces C-code for a graph-based instruction selection algorithm based on the received grammar such that the matching scope for pattern matching is not limited to single statement trees. The algorithm's average runtime is linear w.r.t. the size of input DFGs. This approach results in an easily retargetable graph-based code-selector which enables compiler designers to provide quickly new compiler backends for irregular instruction sets.

3. CODE SELECTION ALGORITHM

During code selection each DFT is processed by two phases: the *matcher* and the *cover* phase. The matcher traverses the DFT in bottom-up manner and applies a tree grammar in order to annotate at each visited node rules, according costs and resulting *nonterminals* (NTs) that can be used to match it. The cover algorithm then traverses the DFT in top-down manner and takes advantage of the annotated information for the selection of the rule with minimal costs for each node. For every hardware instruction, *rules* exist in the tree grammar of the form:

$$NT \rightarrow \underbrace{opcode(op_1, \ldots op_n)}_{\text{simple rule}}\{costs\} = \{actions\}. \quad (1)$$

In (1), *opcode* designates the rule's operator (MUL, ADD, etc.) which is used as its IR-node name at the same time and $\{op_1 \ldots op_n\}$ represent the operands of the rule. Additionally, rules typically comprise costs and action-sections. As the names imply, costs are computed in the costs-section and the action-section contains C code to emit assembly or lowered IR code. Since this presentation is not sufficient for Multi-Output Instructions, Cburg's grammar extends this concept of rules in the way that rules have the form:

$$NT_1, NT_2, \cdots \rightarrow \underbrace{opcode_1(op_1, ...)}_{\text{split rule}}, \underbrace{opcode_2(op_1, \ldots)}_{\text{split rule}}, \ldots \quad (2)$$

In (2), multiple nonterminals $NT_1, NT_2 \ldots$ are produced by a complex rule. The complex rule comprises several rules as presented in (1) as well as costs- and action-sections which are not shown in (2).
Before starting to explain the algorithm, the terminologies which will be used throughout the remainder of this document are defined. A

- **rule:** represents an instruction pattern in the tree grammar,
- **simple rule:** represents a typical tree pattern like ADD, MUL, MAC,
- **complex rule:** consists of several simple rules,
- **split rule:** is a simple rule that is a member of a complex rule.

As shown in figure 2, the presented algorithm consists of 5 major phases: Split Rule Extraction, Pattern Matching, Split Rule Identification, Candidate Set Selection and Pre-Cover.
First, all complex rules in the tree grammar are analyzed and split up into their split rules (subsection 3.1). These split rules are used to find *candidate nodes* in the IR, representing operations that are part of some MOI. The labeler (subsection 3.1) annotates simple rules and all split rules at each candidate node where they match. After annotating the rules, a *split rule map* is created which contains split rules and related IR-nodes. Using this map, *candidate node sets* are identified for every complex rule, such that a clear picture exists about all possible covering solutions. During the subsequent set selection phase (subsection 3.2), the data flow of the different node

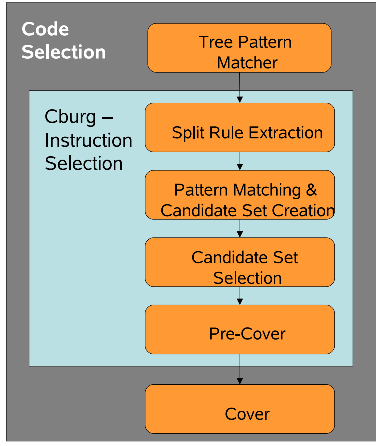


Figure 2: Algorithm Overview

sets is examined in order to eliminate invalid sets. Furthermore, the remaining sets are evaluated by a new cost metric which has been introduced to compare the costs of simple and split rules. In case of overlapping candidate sets, additionally the most valuable sets have to be selected. This decision is mapped to the problem of finding the *Maximum Weighted Independent Set* (MWIS) of a graph. Finally, the resulting candidate sets for all complex rules are selected and checked whether every candidate can be really covered by the associated complex rule (subsection 3.3). At the end, a normal cover algorithm can process the output of the presented heuristic, emitting valid assembly code.

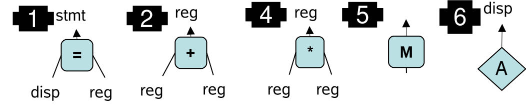
3.1 Labeling & Split Rule Identification

In contrast to normal tree-parsers which annotate at each node for every nonterminal only those rules with minimal costs, the developed labeler annotates additionally all matched split rules at each node regardless of their costs and nonterminals. For this, split rules are extracted from each complex rule and rule numbers are assigned to them which act as identifiers. Figure 3 gives an example about this procedure. In the upper half of figure 3, a tree grammar is given by its simple and complex rules. These rules are annotated at the nodes of the IR tree in the lower half of figure 3. At nodes 5 and 10 both split rules and simple rules are annotated, each of which is producing the same nonterminal. For sake of simplicity, costs and produced nonterminals are not presented within this figure.
After the labeling phase, a *split rule map* is created that stores node–split rule combinations. Succeeding phases can use the information of this map to figure out candidate node sets for every split rule, e.g. node 5 is a candidate for split rule 11.

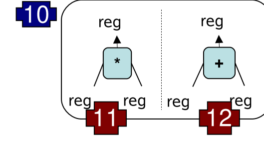
3.2 Candidate Set Selection

During the candidate selection phase the best *candidate node sets* (CS) for all MOIs are identified and evaluated. For this purpose, the information of the split rule map is applied. A candidate set of a MOI contains all *valid combinations* of nodes for this instruction. A valid node combination designates a set of *candidate nodes* each of which is matched by a different split rule of the same complex rule. Since candidate nodes can also be complete tree patterns like MAC, only the *root nodes* are stored inside a candidate set: Root nodes are those nodes which have no successor node inside their pattern. For the example given in figure 3, the only node set for complex rule 10 is $\{5, 10\}$. The number of node sets is further reduced by checking data dependencies in the DFG between candidate nodes and eliminating node sets containing dependent nodes. The remaining node sets are evaluated in order to maximize the benefit of code selection. The set evaluation takes place in relation to the other available rules for a certain node. The basis for this evaluation is obviously the cost metric of the rules. Unfortunately, the typical cost evaluation of tree patterns is also not sufficient for the evaluation of MOIs. Traditional cost metrics for rules concern only *fixed costs* of rules like the number of emitted instructions. However, applying complex rules affects several statement trees and therefore, causes different

Simple Rules:



Complex Rule:



Labeled IR Trees:

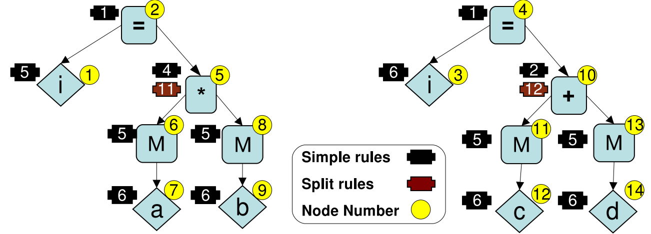


Figure 3: Labeling

costs in different statement trees at the same time which depend strongly on the ongoing matching situation. Such costs can be described as *dynamic* or *opportunity costs* which are orthogonal to the fixed costs of simple rules.

3.2.1 Cost Computation for Complex Rules

If an IR-node can be matched by a simple rule ($rule_{smpl}$) and a split rule ($rule_{split}$) which are reduced to the same nonterminal, Cburg compares the costs of the simple rule and the costs of the split rule to determine the best covering solution for this node. The cost computation of a complex rule ($rule_{cplx}$), consists of two parts: *Saved Costs* and *Duplication Costs*:

Saved Costs.
$\mathbf{C_{saved}(CS)}$ designates the **difference of fixed costs** between a covering solution with simple rules and a solution with a complex rule for a certain candidate set:

$$C_{saved}(CS) = (\sum_{nodes \in CS} C_{fix}(rule_{smpl})) - C_{fix}(rule_{cplx})$$

where $\mathbf{C_{fix}}$ describes the **fixed costs** of arbitrary rules (simple/complex), producing the same nonterminal.

Duplication Costs.
$\mathbf{C_{dup}(CS)}$ are produced by the appearance of *Common Subexpressions* (CSEs) inside of a node pattern in the IR that can be covered by a split rule. *Split rule patterns* (SRP) can express arbitrary tree patterns like MAC or other chained instructions, consisting of several sub-nodes. The set of sub-nodes can be separated into the *root node* (R) and the *child nodes* (K). In contrast to the root node whose result represents the result of the SRP, every child node has exactly one successor inside the pattern of the split rule. If a CSE is covered by a child node of an SRP its result is not available anymore for successor nodes outside the SRP since chained instructions compute only one result. Therefore, the node has to be duplicated in order to compute the result and maintain the validity of the produced assembly code. Accordingly, we define

$$\begin{aligned} K(CS) &= \{node \mid node \in \{\text{SRP}\} \wedge \text{node} \neq \text{R}\} \\ CSE(CS) &= \{node \in K(CS) \mid \text{ node is a CSE}\} \\ C_{dup}(CS) &= \sum_{node \in CSE(CS)} (\sum_{fan_{out}(node)} C_{fix}(rule_{smpl})) \end{aligned}$$

Here, fan_{out} denotes the number of outgoing edges of a $node \in K(CS)$ emanating the SRP. For each of these edges, $C_{fix}(rule_{smpl})$ represents the costs of the according simple rule that produces the required nonterminal for the successor node.

Opportunity Costs .
$\mathbf{C_{opp}(CS)}$ of a candidate set are the costs that denominate the **overall cost-difference** between the application of a complex rule and an alternative covering by a set of simple rules for the nodes in the CS:

$$C_{opp}(CS) = C_{saved}(CS) - C_{dup}(CS).$$

Split Rule Costs.
Finally, the costs of the split rule are calculated by the average opportunity costs of the complex rule's candidate set:

$$C(rule_{split}) = C_{fix}(rule_{split}) - C_{average_opp}(CS)$$

where $C_{fix}(rule_{split})$ are the fixed costs of the split rule. Furthermore, the average opportunity costs $C_{average_opp}(CS)$ of a candidate set CS are computed as:

$$C_{average_opp}(CS) = C_{opp}(CS)/num_{split}(CS),$$

where $num_{split}(CS)$ is the number of split rules of a candidate set.

3.2.2 Benefit Optimization

After the candidate validation, the candidate sets cannot be further reduced and the remaining nodes have to be covered by the according complex rules. Nevertheless, it might happen that intersections between candidate sets occur. This is the case when a node is a candidate for several MOIs and consequently is listed in several candidate sets. We call such nodes *shared nodes*. Since nodes can only be covered by one rule, the covering decision for shared nodes has to ensure that the benefit in terms of costs is maximized.
The problem of finding an optimal solution for the covering of shared nodes can be reduced to the problem of finding a *Maximum Weighted Independent Set* (MWIS) in a weighted undirected graph $G = (V, E, W)$ without loops and multiple edges. In G, V designates the set of vertices, E the set of edges and W is the vertex weighting function.
An independent set in a graph is a collection of vertices that are mutually non-adjacent. The problem of finding an independent set of maximum cardinality is one of the fundamental combinatorial problems and known to be NP-complete even when all nodes have uniform weights [10]. Due to this, we applied a heuristic called *GWMIN2* [21]. Generally, GWMIN2 belongs to the class of *minimum-degree greedy* algorithms that construct an independent set by selecting some vertex of minimum degree, removing it and its neighbors from the graph and iterating on the remaining graph until it is empty. Such algorithms run in linear time in the number of vertices and edges. GWMIN2 selects in each iteration a vertex v, such that

$$\frac{W(v)}{\sum_{w \in N_G^+(v)} W(w)}, \forall u \in V \tag{3}$$

is maximized. In [21] it is proven that the resulting independent set has at least a weight of

$$\sum_{v \in V} \frac{W(v)^2}{\sum_{u \in N_G^+(v)} W(u)}.$$

The notation $N_G(v)$ designates the neighborhood of a vertex v in G and $N_G^+(v)$ the set $\{v\} \cup N_G(v)$.

Application of MWISP for Benefit Maximization.
For the benefit maximization in the presence of overlapping MOIs, a graph $G = (V, E, W)$ is constructed. In G, every vertex $v \in V$ represents a complex rule and the associated weight $W(v)$ is equal to its benefit. Basically, the benefit of a MOI is computed as the negated sum over all costs of comprised split rules: $(-1)\sum C_{rule_{split}}(CS)$. In between two vertices of G an edge exists, if and only if the associated MOIs have one or more candidate IR-nodes in common. The algorithm now simply selects those non-adjacent vertices with the highest weight (benefit) in a greedy manner and eliminates them including their edges form the graph G.

3.3 Pre-Cover Phase

In the last phase of the algorithm, the node selection has to be evaluated and pre-covered before the original cover phase starts, since split rules do not necessarily offer minimal costs for every producible nonterminal at a candidate node. Consequently, due to different nonterminal requirements of subsequent IR-nodes, a candidate node might not be covered by a split rule although the split rule has minimal costs regarding its produced nonterminal. In this case, it must be ensured that all other nodes of the same candidate set are also not covered by their split rules. This is achieved by *pre-covering* the IR. During this, the cover phase of a tree pattern matcher is simulated and in case a candidate node is not covered by a split rule, all nodes of the according candidate set are re-matched by simple rules.

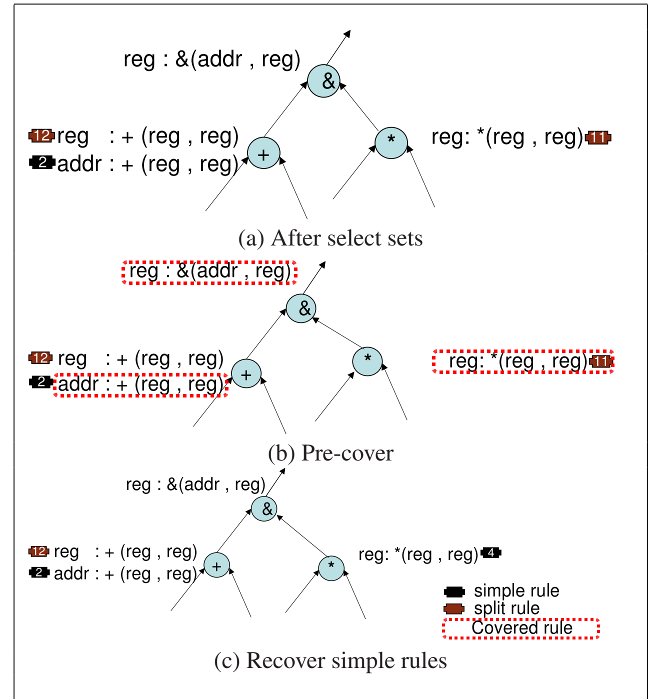


Figure 4: Pre-Covering Nodes

Figure 4(a) shows an example of such a situation. It presents a set of IR-trees which are already labeled, and also a selected set of candidate nodes which are marked by their associated split rules 11 and 12 of figure 3. In Figure 4(b), split rule 12 is not used for covering, since the succeeding rule consumes a nonterminal that is cheaper produced by simple rule 2. However split rule 11 is used for covering at the same time. To solve this antagonism, the split rules are eliminated and all candidate nodes are re-matched by simple rules as presented in figure 4(c). Now, the IR-trees can be traversed by the cover phase and the assembly code is emitted.

3.4 Complexity Analysis

The basis of the code selection algorithm introduced in section 3, relies on a depth-first traversal of a $DFG = (V, E)$ in $O(V + E)$ time in the labelling phase. Additional cost computations in order to maximize the benefit of code selection are applied at each node. These computations can be split into two parts: the computation of C_{saved} and C_{opp}. Whereas the former can be computed linearly

dependent on the number of candidate nodes for all MOIs, the latter computation takes

$$O(\sum_{\substack{CS\subseteq V,\\ V\in DFG}} \sum_{CSE\subset CS} |Adj(CSE(CS))|)$$

time. In worst case, this might evolve to an exponential runtime, if every node $v \in V$ is a CSE and at the same time covered by a split rule. Furthermore, in case of overlapping MOI patterns – the MWISP is solved by a greedy heuristic that runs in linear time, dependent on the amount of overlapped MOIs. The final pre-cover phase visits all nodes in every candidate set once. Thereby its complexity can be expressed as

$$O(\sum_{\substack{CS\subseteq V,\\ V\in DFG}} |CS|)$$

which also equals a linear runtime. Overall, the worst case runtime is exponential on the number of CSEs covered by split rules, but it is linear on the size of considered DFGs.

4. RETARGETABLE CODE SELECTION WITHIN CBURG

Apart from efficient code selection in the presence of MOIs, retargetable compilation is the second aspect that this paper targets. Retargetable compilation for high-level languages like C/C++ is one of the major challenges in ASIP architecture exploration. Nonetheless, iterative architecture exploration approaches demand for very flexible software development tools, in order to explore many architecture design alternatives within short time. In this context, many design platforms including retargetable compilers have been developed in the past [11, 12, 20, 14]. As presented in figure 5(a), design flow iterations comprise usually the compilation, simulation and profiling of C/C++ applications for a certain virtual architecture prototype. Based on the profiling results, bottlenecks are identified, the instruction pipeline is fine-tuned and customized instructions are added to stepwise improve the architecture's efficiency. The new instructions are declared to the compiler in order to evaluate their benefit for the targeted applications during the next compilation-simulation cycle. Whereas simple instructions can be declared within the tree grammar of the code selector, MOIs are typically implemented as CKFs or inline assembly as they cannot be targeted by the compiler. This implies the manual modification of the application and the compiler, respectively which leads to high overhead for large applications. Since usually only the bottlenecks itself are implemented through MOIs, further utilization of the developed MOIs stays unexplored and reusability towards different applications cannot be ensured.

To support a maximum of different compilation frameworks and to overcome the compiler-related limitations within architecture exploration, the described code selection algorithm has been incorporated into a retargetable code-generator generator called Cburg.

Cburg is based on Olive [23] which takes a configuration file as input and produces a set of data structures and code-generation functions for a certain target ISA. However, in contrast to Olive, Cburg's code selection algorithm works on DFGs rather on DFTs as described in section 3.

Cburg's configuration file is very similar to Olive. The file contains the description of the target ISA in terms of IR-patterns as well as a set of functions, necessary to access the compiler's IR. The IR-patterns represent the grammar that is used to identify patterns in the compiler's IR and map them to adequate assembly language or lowered IR. Rules inside this grammar have the form of both, rule 2 and rule 1 (see section 3). The generated data structures and functions provide the complete methodology based on the described algorithm.

Compiler designers can use these to comfortably implement a code selection algorithm for arbitrary target machines with MOIs. Thereby it is possible to declare every kind of hardware instruction to the compiler and the necessity to manually modify source applications or the compiler itself is omitted (figure 5(b)).

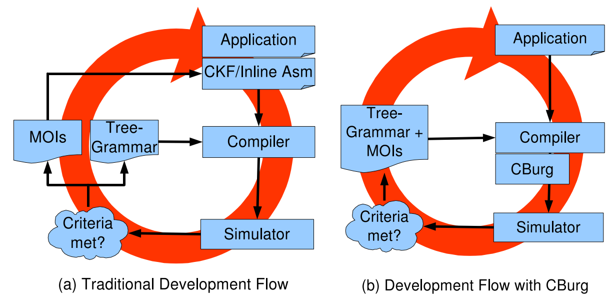


Figure 5: Traditional vs. New Design Flow

5. EXPERIMENTAL RESULTS

In order to evaluate the quality of the proposed code selection methodology to facilitate a more efficient instruction set design, Cburg has been integrated into the *Little C Compiler* (LCC) [3]. As the target architecture, the MIPS architecture [19] has been used. Based on the MIPS ISA, new MOIs have been developed. The nomenclature of each MOI mirrors through the concatenation of instruction names, the parallel operations inside the MOI. For example, an instruction "lwlw" describes the simultaneous execution of two "lw" which is a simple load in the MIPS ISA. The benchmark suite comprises typical symmetric encryption algorithms as the *Data Encryption Standard* (3DES), and the *Advanced Encryption Standard* (AES) [5]. and an internet protocol stack that comprises an IPv6-layer including authentication, encryption as well as an Ethernet layer. Additionally, an *Adaptive Differential Pulse Code Modulation* (ADPCM) DSP-application taken from the DSP-Stone benchmark suite [18] has been examined. To develop new MOIs, all benchmarks have been profiled with a fine-grained profiler [13] to identify execution hot spots and promising candidate instructions. Several MOIs have been developed giving special attention to the symmetric encryption algorithms, i.e. AES and 3DES. Since symmetric encryption is one of the major bottlenecks of IPv6 processing [**?**], it was also expected that such MOIs will affect the overall performance of the protocol stack, too. From the profiling results, it turned out that shift (sll/srl), xor and load (lw) operations are most frequently used in the encryption algorithms. Consequently, the developed MOIs are based on these instructions. First, all MOIs have been applied separately. Later 3 and more MOIs have been combined. Table 1 provides an overview of the obtained experimental results. The best results have been achieved with the MOIs "lwsll" ($+16.96\%$ speedup/-13.99% code size) for 3DES and "lwxor" ($+12.83\%$ speedup/ -9.96% code size) for AES, respectively. Overall performance improvements of $+24.07\%$ (3DES), $+21.76\%$ (AES) and $+17.21\%$ (IP stack) were possible. Obviously, the MOIs did not lead to notable improvements for the ADPCM benchmark, since its operator usage significantly differs from those of encryption and protocol processing.

6. CONCLUSION

The presented code-generator generator Cburg, offers a new methodology to exploit MOIs during architecture explorations. Instead of handcoding error prone inline assembly or CKFs, system designers can model all developed instructions in one grammar file which is fed into the compiler's code-selector. As a consequence, compiled applications comprise automatically all customized instructions without any manual modification. Thereby the benefit of newly added MOIs can be faster evaluated. Furthermore, the evaluation is not restricted to isolated code fragments. Instead, the whole application is examined at once which leads to much more accurate results regarding usability and achieved code quality. Shorter design cycles and time-to-market periods are the consequence.

The general problem of code selection for MOIs relies on the fact, that at each IR-node, locally, a decision has to be made which affects the global result of the pattern matching process. The described formulas of subsection 3.2.1 provide Cburg with a powerful cost metric that is able to evaluate the quality of local decisions for the global result by taking so called opportunity costs of complex

Results of Benchmarks								
	3DES (603 lines of C code)		AES (881 lines of C code)		IP Stack (3906 lines of C code)		ADPCM (743 lines of C code)	
complex instructions	Speedup	Code Size	Speedup	Code Size	Speedup	Code Size	Speedup	Code Size
srlsll	+13.19%	-10.98%	+1.03%	-1.17%	+9.14%	-12.34%	+0.10%	-0.15%
srlsrl	+3.77%	-3.35%	+0.00%	-0.08%	+3.74%	-4.88%	+0.04%	-0.05%
sllsll	+7.64%	-6.58%	+0.69%	-0.64%	+5.09%	-6.70%	+0.04%	-0.05%
lwlw	+7.64%	-6.66%	+8.60%	-7.58%	+2.65%	-4.97%	+3.40%	-3.62%
lwsll	+16.96%	-13.99%	+1.03%	-1.00%	+6.22%	-8.10%	+0.29%	-0.50%
lwxor	+7.54%	-6.22%	+12.83%	-9.96%	+4.10%	-5.71%	+0.00%	-0.00%
xorsll	+7.54%	-6.22%	+0.69%	-0.86%	+3.75%	-4.90%	+0.00%	-0.00%
xorsrl	+7.54%	-6.22%	+8.25%	-6.63%	+3.72%	-4.85%	+0.00%	-0.00%
xorsll, xorsrl	+7.54%	-6.22%	+8.25%	-6.83%	+3.75%	-4.90%	+0.00%	-0.00%
lwsll, lwxor	+16.96%	-13.99%	+13.07%	-10.71%	+6.59%	-9.01%	+0.29%	-0.50%
srlsrl, sllsll	+7.54%	-9.89%	+1.03%	-0.89%	+8.83%	-12.44%	+0.08%	-0.10%
srlsrl, sllsll, srlsll	+13.29%	-11.42%	+1.38%	-1.47%	+9.34%	-12.69%	+0.14%	-0.20%
srlsrl, sllsll, srlsll, xorsll, xorsrl	+17.06%	-14.53%	+8.60%	-7.35%	+11.73%	-15.79%	+0.14%	-0.20%
srlsrl, sllsll, srlsll, lwsll, lwxor	+24.06%	-20.74%	+14.10%	-11.68%	+14.68%	-20.03%	+0.33%	-0.55%
srlsrl, sllsll, srlsll, lwlw	+20.83%	-17.88%	+13.41%	-11.60%	+11.98%	-17.66%	+2.57%	-2.23%
srlsrl, sllsll, srlsll, xorsll, xorsrl, lwsll, lwxor, lwlw, mvmv	+24.07%	-21.18%	+21.67%	-18.35%	+17.12%	-21.89%	+3.62%	-3.97%

Table 1: Overview of Experimental Results

rules into account. It is the basis for the process of code selection within Cburg, because only by the computation of opportunity costs, it is possible to make fair matching decisions between simple and complex rules.
In future, Cburg will be integrated into more Compilers to verify its reusability towards different IRs and backends. Furthermore, the hardware effort that is associated with each MOI has to be examined in order to evaluate the obtained speedup of the executables properly. And finally, an interesting open issue is the exploitation of the most valuable set of MOIs to support a certain set of applications. At this state, the presented sets of MOIs have been manually figured out, based on dynamic results obtained at runtime. In contrast to this, static analyses of the compiler could be used to find a solution for the "best" MOI sets..

7. ACKNOWLEDGEMENT

This work was partially supported by the IT R&D program of MIC/IITA[2007- S001-01, Components/Module technology for Ubiquitous Terminals], the ITRC(Inofrmation Technology Research Center) support program supervised by the IITA(Institute of Information Technology Assessment) (IITA-2005-C1090- 0502-0031), Nano IP/SoC promotion group of Seoul R&BD Program in 2007, Ministry of Information and Communication, Korea (A1100-0501-0004), and Korea Ministry of Science and Technology (M103BY010004-05B2501-00411).

8. REFERENCES

[1] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code Generation Using Tree Pattern Matching and Dynamic Programing. *ACM Transactions on Programming Languages and Systems.*, 11(4):491–516, Oct. 1989.

[2] G. Araujo, S. Malik, and M. Lee. Using Register Transfer Paths in Code Generation for Heterogeneous Memory Register Architectures . In *Proc. of the Design Automation Conference (DAC)*, pages 591–596, June 1996.

[3] C. Fraser and D. Hanson. *A Retargetable C Compiler : Design and Implementation*. Benjamin/Cummings Publishing Co., 1994.

[4] N. Clark, A. Hormiri, S. Mahlke, and S. Yehia. Scalable Subgraph Mapping for Acyclic Computation Accelerators. In *Proc. of the Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, Oct. 2006.

[5] C. Devine. http://xyssl.org, 2007.

[6] E. Eckstein, O. Koening, and B. Scholz. Code Instruction Selection Based on SSA Graphs. In *Proc. of the Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 49–65, Oct. 2003.

[7] M. A. Ertl. Optimal Code Selection in DAGs . In *"Principles of Programming Languages (POPL '99)"*, 1999.

[8] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering Efficient Code Generators Using Tree Matching and Dynamic Programming. Technical Report TR-386-92, 1992.

[9] C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG–fast optimal instruction selection and tree parsing. Technical Report CS-TR-1991-1066, 1991.

[10] M. R. Garey and D. S. Johnson. *A Guide to the Theory of NP-Completeness*. W. H. Freeman & CO, New York, 1990. ISBN:0716710455.

[11] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 1999.

[12] A. Hoffmann, H. Meyr, and R. Leupers. *Architecture Exploration for Embedded Processors With Lisa*. Kluwer Academic Publishers, Jan. 2003. ISBN 1-4020-7338-0.

[13] K. Karuri, M. A. A. Faruque, S. Kraemer, R. Leupers, G. Ascheid., and H. Meyr. Fine-grained Application Source Code Profiling for ASIP Design . In *Proc. of the Design Automation Conference (DAC)*, pages 329–334, 2005.

[14] K. Keutzer and H. M. et. al. *Building ASIPs: The Mescal Methodology*. Springer, June 2005. ISBN: 0-387-26057-9.

[15] R. Leupers and P. Marwedel. Instruction Selection for Embedded DSPs with Complex Instructions. In *Proc. of the European Conference on Design Automation (EDAC)*, Sept 1996.

[16] S. Liao, S. Devadas, K. Keutzer, and S. Tjiang. Instruction Selection Using Binate Covering for Code Size Optimization. In *Proc. of the Int. Conf. on Computer Aided Design (ICCAD)*, pages 393–399, 1995.

[17] C. Liem, T. May, and P. Paulin. Instruction-set Matching and Selection for DSP and ASIP Code Generation. In *Proc. of the European Design and Test Conference (ED & TC)*, pages 31–37, 1994.

[18] M. Willems and V. "Živojnović. DSP-Compiler: Product Quality for Control-Dominated Applications? In *Proc. of the Int. Conf. on Signal Processing Applications and Technology (ICSPAT)*, Oct. 1996.

[19] MIPS technologies Inc. *MIPS 4Kc Processor Core Datasheet*, Jun. 2000.

[20] S. Kobayashi, Y. Takeuchi, A. Kitajima, M. Imai. Compiler Generation in PEAS-III: an ASIP Development System. In *Workshop on Software and Compilers for Embedded Processors (SCOPES)*, 2001.

[21] S. Sakai, M.Togasaki, and K. Yamazaki. A Note on Greedy Algorithms for the Maximum Weighted Independent Set Problem. *Discrete Applied Mathematics*, 126:313–322, 2003.

[22] H. Scharwaechter, D. Kammler, A. Wieferink, M. Hohenauer, J. Zeng, K. Karuri, R. Leupers, G. Ascheid, and H. Meyr. ASIP Architecture Exploration for Efficient IPSec Encryption: A Case Study. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(2), Mai 2007.

[23] S. W. K. Tjiang. An Olive Twig. Technical report, Synopsys Inc., 1993.

[24] A. Todd. Least-Cost Instruction Selection in DAGs is NP-Complete . In *"http://research.microsoft.com/toddpro/papers/proof.htm"*, Feb. 2007.