

Processor Modeling and Code Selection for Retargetable Compilation

J. VAN PRAET, D. LANNEER, W. GEURTS, and G. GOOSSENS

Target Compiler Technologies N.V.,

Interleuvenlaan 3, B-3001 Leuven, Belgium

<http://www.retarget.com>

Embedded processors in electronic systems typically are tuned to a few applications. Development of processor specific compilers is prohibitively expensive and as a result such compilers, if existing, yield code of an unacceptable quality. To improve this code quality, we developed a processor model that captures the connectivity, the parallelism and all architectural peculiarities of an embedded processor. We also implemented a retargetable and optimizing compiler working on this model. We present the graph based processor model and we formally define the code generation task, as binding the intermediate representation of an application to this model. We also present a new method for code selection, based on this processor model, that is capable of handling directed acyclic graphs instead of trees.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors—*Code generation; Compilers; Optimization*

General Terms: Algorithms, Design, Compilation

Additional Key Words and Phrases: Retargetable code generation, retargetable compilation, graph, code selection, processor modeling, instruction set graph, embedded systems, system design

1. INTRODUCTION

Over the last decade, there has been an amazing growth in the electronics and semiconductor industry. Consumer electronics, like mobile and personal communication systems, and multi-media, are among the most prominently growing sectors of the electronics industry today [Paulin, Liem, Cornero, Naçabal, and Goossens 1997].

In these highly competitive market segments, designers more and more incorporate processors, called *embedded processors*, in their systems. Programmability offers them cost-effective hardware reuse and the flexibility to support last minute specification changes, or to add new features to the system.

The pressure to reduce the time-to-market, the ever shortening market window, and the complexity of the designs make CAD support a prerequisite to craft these

Copyright ©1999 by the Association for Computing Machinery, Inc.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works, requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept, ACM Inc., 1515 Broadway, New York, NY 10036 USA, fax +1 (212) 869-0481, or permissions@acm.org.

products. For example, system engineers increase their productivity a lot by using a compiler to program embedded processors.

However, surveys indicate that code produced by compilers for embedded processors is almost useless for product development [Paulin, Liem, Cornero, Naçabal, and Goossens 1997; Zivojnović, Velarde, Christian, and Meyr 1994]: the execution time and the size of generated code is several times that of hand-crafted code. Moreover, the cost of developing target-specific compilers is prohibitively high.

The reason is that standard compiler techniques are not well suited for the particular architectures of embedded processors, and that they are not easily *retargetable* to a new target processor. Therefore, new techniques for processor modeling and for compilation are needed.

This paper presents a new approach to code generation, based on a processor model called the instruction set graph (ISG), that efficiently represents the connectivity, parallelism and architectural peculiarities of embedded processors. The model is conveniently specified by a special-purpose processor description language, called nML [Fauth, Van Praet, and Freericks 1995].

In conjunction with the ISG, we also developed a compiler, called CHES, that exploits all knowledge in the model. We thereby allow the code generation techniques to consume more time than in a typical general purpose compiler, arguing that code quality is more important. All our code generation tools are processor independent, as all the required processor related information can be extracted from the ISG. The compiler is retargeted by simply providing it with a new ISG.

This retargetability is a key issue for the success of a new class of application specific instruction set processors (ASIPs) [Paulin, Liem, Cornero, Naçabal, and Goossens 1997; Goossens, Van Praet, Lanneer, Geurts, and Thoen 1996]. These are a hybrid form of custom architectures and standard DSP processors, offering an instruction set and a hardware implementation that are optimized for a small number of applications. Because of the small number of applications, compiler development efforts for ASIPs must be kept minimal.

The data-path architecture of a small ASIP is shown in Figure 1. It consists of an arithmetic-logic unit (ALU) combined with a SHIFT unit, a multiply-accumulate unit (MACC), and two address generation units (AGU). It also has a dual ported data memory DM. Its most parallel instruction format performs an arithmetic operation on the ALU-SHIFT or MACC units, together with one or two parallel memory accesses with address calculations. This is depicted in Table I in a compact way: each bit-string that can be formed from left to right forms a valid instruction. For example, by concatenating the top entry of all columns, one obtains the instruction “0 00 00 00 0 00 000 0 00 00” which adds the values in the registers AX and AY, in parallel loads a value from the memory to AX and updates the memory address, found in Ia[0], by incrementing it with the value in Ma[0].

In Section 2 the instruction-set graph is presented; its use for code generation is discussed in Section 3. Section 4 explains and evaluates the code selection method in detail. Section 5 concludes this paper.

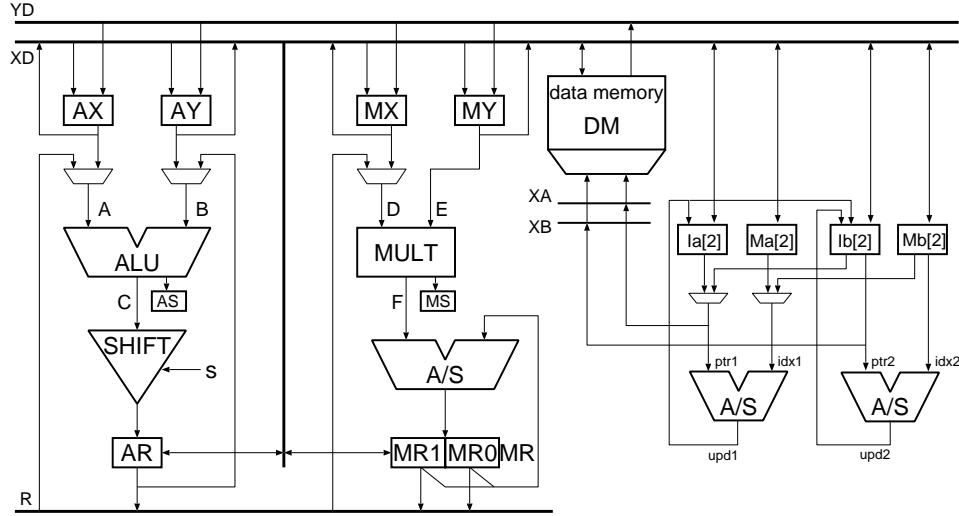


Fig. 1. Data-path of a small example ASIP.

arithmetic operation(s) and parallel load/store(s) with indirect addressing					
	arithmetic operations			single load/store, indirect addressing	
	00	01	10	dst/src	indirect memory address
0	00: + 01: - 10: and 11: or	00: AX 01: AR 10: MR1 11: MR0	0: AY 1: AR	00: load 01: store	0: + 1: - 00: la[0] 01: la[1] 10: lb[0] 11: lb[1] 00: Ma[0] 01: Ma[1] 10: Mb[0] 11: Mb[1]
	0: +; 0: >> 1 1: -; 1: << 1	00: AX 01: AR 10: MR1 11: MR0	0: AY 1: AR		
	00: *, + 01: *, - 10: * 11: sat	00: MX 01: AR 10: MR1 11: MR0	0: MY 1: "1"		
	00: AX 01: AR 10: MR1 11: MR0	00: AX 01: AR 10: MR1 11: MR0	0: MY 1: "1"		
	2 parallel load/stores, indirect addressing				
	10: load 11: store	dst 1	addr. 1	dst/src 2	addr. 2
	00: AX 01: AR 10: MR1 11: MR0	0: + 1: - 0: Ma[0] 1: Ma[1]	0: + 1: - 0: Ma[0] 1: Ma[1]	00: AX 01: AY 10: MX 11: MY	0: + 1: - 0: Mb[0] 1: Mb[1]

Table I. Instruction format with maximal parallelism for the processor of Figure 1.

2. THE INSTRUCTION-SET GRAPH

2.1 A bipartite graph as processor model

The processor model used in CHESS is designed to closely match the intermediate representation of the application. It is called the instruction-set graph (ISG).

Definition 1 (instruction-set graph). The instruction-set graph (ISG) is a directed bipartite graph $G_{ISG} \langle V_{ISG}, E_{ISG} \rangle$ with $V_{ISG} = V_S \cup V_I$, where V_S contains all vertices representing storage elements in the processor and V_I contains all vertices representing its operation-types. The edges in $E_{ISG} \subseteq (V_S \times V_I) \cup (V_I \times V_S)$ represent the connectivity of the processor and model the data-flow from storage elements, through an ISG operation-type, to storage elements.

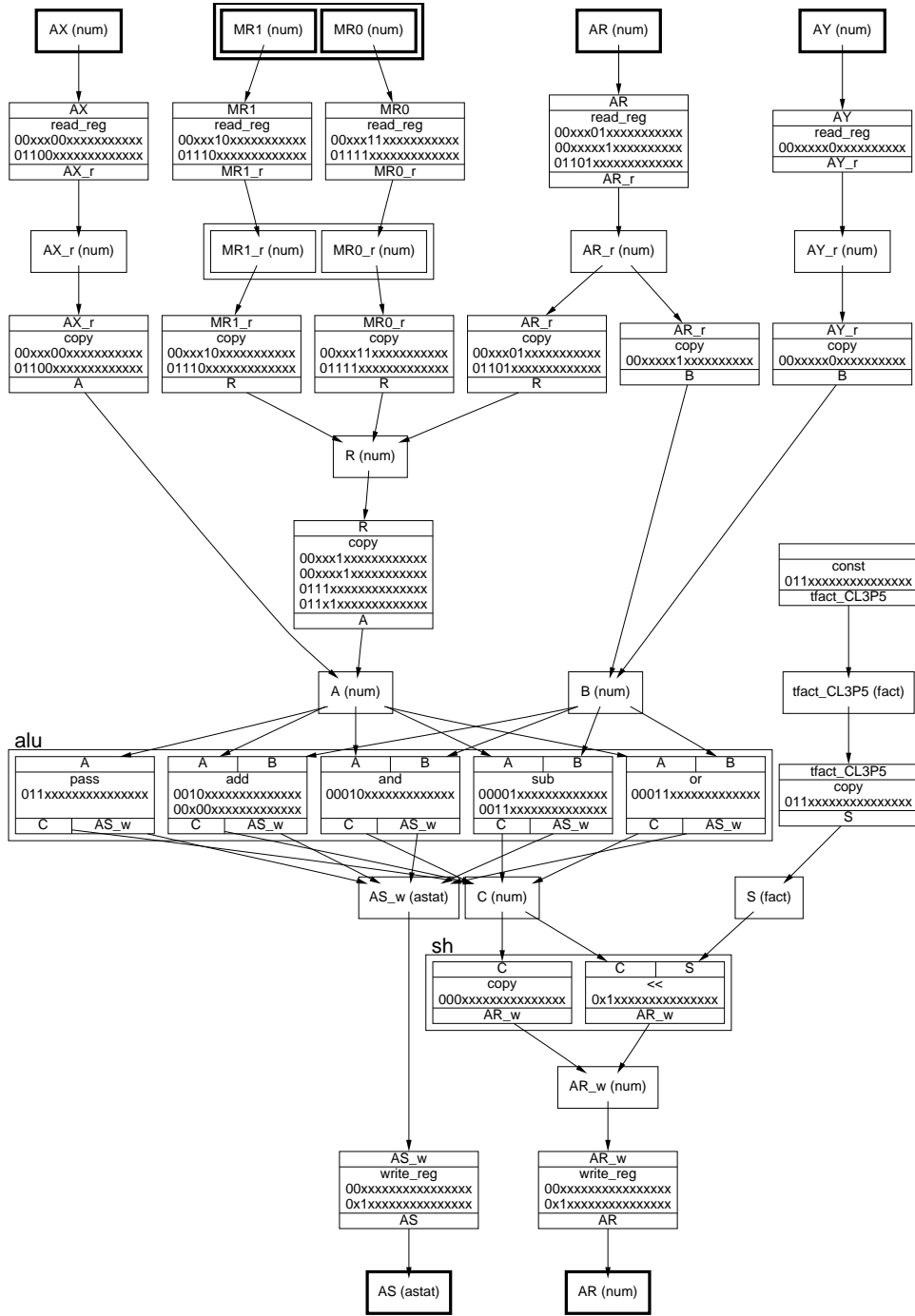


Fig. 2. Partial ISG for the example processor of Figure 1 and Table I.

Figure 2 contains a small part of the ISG for the example processor of Figure 1 and will be explained in the next sections. The small rectangles represent storage elements, the larger ones represent operation-types and are depicted with the binary instruction encodings that enable them.

2.2 ISG operation-types

Definition 2 (ISG operation-type). An *ISG operation-type* is a primitive processor activity. It has a fixed number of *ordered* input arguments and a fixed number of *ordered* output arguments. Each argument is connected through one edge to exactly one storage element.

Most ISG operation-types, for example all arithmetic operation-types, transform values in storage elements into other values in other storage elements. The ISG operation-types are connected as defined in an nML processor description [Fauth, Van Praet, and Freericks 1995].

Figure 2 contains all operation-types on the ALU-SHIFT unit of the example processor of Figure 1.

2.2.1 Enabling conditions. In each instruction, the processor executes a number of ISG operation-types, as specified in the nML processor description. Conversely, a given ISG operation-type can be *enabled* by several instructions; this must be known for code generation.

Definition 3 (enabling condition). The set of instructions E_i that enables an operation-type i in the ISG is called its *enabling condition*, and is returned by the function $\text{enabling} : V_I \rightarrow 2^{\mathbb{B}^w} : i \mapsto E_i \subseteq \mathbb{B}^w$; $2^{\mathbb{B}^w}$ is the power set of \mathbb{B}^w ; \mathbb{B}^w is the set of all binary strings (bit-strings or bit-vectors) of length w , with w being the width of the instruction word.

In Figure 2, the enabling conditions are shown in a binary cubic representation (with x meaning “don’t care”). For example, the enabling conditions of the ALU operation-types mentioned above can be matched to the top left part of Table I.

2.2.2 Encoding restrictions. In the code generation method used by the CHES compiler, ISG operation-types are combined into patterns that match sub-patterns of the CDFG. Such a pattern should be executable by an instruction, consequently it must be free of *encoding conflicts*.

Definition 4 (encoding conflict). Given a subset of ISG operation-types $V_{I_o} \subseteq V_I$, the intersection of the enabling conditions of its elements, $\text{enabling}(V_{I_o}) = \bigcap_{i \in V_{I_o}} \text{enabling}(i)$, is the enabling condition for the set V_{I_o} . The set V_{I_o} has an *encoding conflict* when its enabling condition is empty: $\text{enabling}(V_{I_o}) = \emptyset$.

For example, a multiplexer is modeled as a set of *copy* operation-types having a common output storage element, with all pairs of *copy* operation-types in the set having encoding conflicts. The multiplexer at the left ALU input in Figure 1, corresponds to the two *copy* operation-types having an encoding conflict and writing to transitory A in the middle of Figure 2.

2.2.3 Control operation-types. There are no special provisions for the control unit of a processor in the ISG. The basic idea behind modeling the control unit of

a processor in the ISG is to introduce *abstract control-flow operation-types*. The details of its behavior are not put in the ISG, as they are not needed for code generation. A key element to the conciseness of the ISG is indeed that it structurally represents information needed for code generation, but separates this from the behavioral information needed for simulation.

2.3 Storage

A distinction is made between two kinds of storage elements in the ISG: *static data storage elements* and *transitory data storage elements* [Landskov, Davidson, Shriver, and Mallett 1980].

Definition 5 (static storage). A *static data storage element* is a storage element that can store each of its values during one or several machine cycles, until it is explicitly overwritten. The *capacity* of a static storage element indicates how many values it can contain at the same time.

Definition 6 (transitory storage). A *transitory data storage element*¹ is a storage element that does not permanently store a value, but only *passes* it on, without delay. A transitory storage element has a capacity of *one*.

Static storage consists of memory and controllable registers, respectively denoted in the ISG by the sets V_M and V_R . Buses and wires are examples of transitories, which form the set V_T . Together, the storage elements define a *structural skeleton* of the target processor: $V_S = V_M \cup V_R \cup V_T$.

Each storage element is typed with the specific *data-type* of the values it can carry. In Figure 2, storage elements are depicted as small rectangular boxes, each having a label denoting their data-type (`num`, `astat` or `fact`) between parentheses. The ones at the top and bottom, drawn with thicker lines, are registers, all others are transitories. Remark that, for graphical reasons, static storage elements may be duplicated in a figure of the ISG; this is not done in the actual ISG model.

2.3.1 Hardware conflicts. A *hardware or resource conflict* exists when several operation-types need a common resource during their execution.

The ISG is constructed in such a way that each possible hardware conflict is represented as an access conflict on a transitory: two operation-types have an access conflict (and thus a hardware conflict) when they write to the same transitory during the same cycle.

- Resource conflicts on a functional unit are modeled by means of transitories representing its outputs. For example, in Figure 2 access conflicts on transitory C model resource conflicts on the ALU.
- Also read/write ports of static storage may be modeled as transitories to make the code generator check for *port conflicts*; examples are `AR_r` and `AR_w`, which are respectively the read and write port of `AR`.
- As a last example, consider the result bus `R` in Figure 1, which can be written by several *tristate drivers*. This is modeled by the *copy* operation-types writing to transitory `R`, shown in Figure 2.

¹A transitory storage element will simply be called a *transitory* in the sequel.

The CHES compiler prevents port conflicts, bus conflicts and other resource conflicts by avoiding access conflicts on transitories; it allows at most one operation-type to write to each transitory in each machine cycle.

Hardware conflicts in the ISG are easily checked for:

Definition 7 (hardware conflicts). The function $\mathbf{resources} : V_I \rightarrow 2^{V_T} : i \mapsto \{t \in V_T \mid \exists n \in \mathbb{N} : t = \mathbf{output}(i, n)\}$, returns the set of transitories that are written by the ISG operation-type i , with $\mathbf{output}(i, n)$ returning the n -th output of i .

A subset of ISG operation-types $V_{I_o} \subseteq V_I$ is free of structural hazards if it does not introduce access conflicts on transitories:

$$\forall i, i_j \in V_{I_o} : i_i \neq i_j \Rightarrow \mathbf{resources}(i_i) \cap \mathbf{resources}(i_j) = \emptyset \quad (1)$$

A sequence of ISG operation-types that are connected through transitories and are free of conflicts is called a *valid direct path*.

Definition 8 (valid direct path). An *ISG path* is a sequence of vertices $i_0, s_0, i_1, s_1, \dots, i_p, s_p$ in the ISG with $i_i \in V_I, s_i \in V_S$ and $(i_i, s_i), (s_i, i_{i+1}) \in E_{ISG}$. It is notated as $P[i_0 \rightarrow s_0 \rightarrow i_1 \rightarrow s_1 \rightarrow \dots \rightarrow i_p \rightarrow s_p]_{s_i \in V_S, i_i \in V_I}$.

The predicate $\mathbf{direct-path} : V_I \times V_I \rightarrow \mathbb{B} : (i, i') \mapsto \mathbf{true/false}$ indicates whether there exists a *valid direct path* between ISG operation-types i and i' :

$$\mathbf{direct-path}(i, i') \iff \begin{cases} \exists P[i = i_0 \rightarrow s_0 \rightarrow i_1 \rightarrow s_1 \rightarrow \dots \rightarrow s_{p-1} \rightarrow i_p = i']_{s_i \in V_S, i_i \in V_I} & \text{and} \\ \forall s_i \in P : s_i \in V_T & \text{and} \\ \forall i_i, i_j \in P : i_i \neq i_j \Rightarrow \mathbf{resources}(i_i) \cap \mathbf{resources}(i_j) = \emptyset & \text{and} \\ \bigcap_{i_i \in P} \mathbf{enabling}(i_i) \neq \emptyset & \end{cases} \quad (2)$$

A valid direct path is notated as $P[i_0 \xrightarrow{d} t_0 \xrightarrow{d} \dots \xrightarrow{d} t_{p-1} \xrightarrow{d} i_p]_{t_i \in V_T, i_i \in V_I}$.

A valid ISG can not contain valid cyclic direct paths.

2.4 The operation-type hierarchy

In the CHES compiler, the application that must be compiled is represented as a *control/data-flow graph (CDFG)* [Lanneer 1993], of which the operations are instances of operation-types in a library \mathcal{L} . The operation-types in \mathcal{L} are *hierarchically* organized to represent the different ways in which a CDFG operation can be mapped on the ISG. Figure 3 shows an example of this hierarchy, where four implementations exist for the sub operation-type (a subtraction). Each implementation is a *subtype* of an abstract operation-type, for example, `sub` is a subtype of `functional_opn`; `sub_XY` is a subtype of `sub`. The leaf operation-types in the hierarchy of Figure 3 are ISG operation-types. As a matter of fact, the ISG always forms a plane through the hierarchy of operation-types, and represents those operation-types that can be implemented on the processor.

Each CDFG operation *always* has a unique operation-type, of which it is an instance, in the hierarchy of \mathcal{L} : $\forall o \in V_O, \exists l \in \mathcal{L} : \mathbf{type}(o) = l$.

All choices to implement a CDFG operation o are readily looked up as all descendants of its operation-type $l = \mathbf{type}(o)$ that are contained in the ISG.

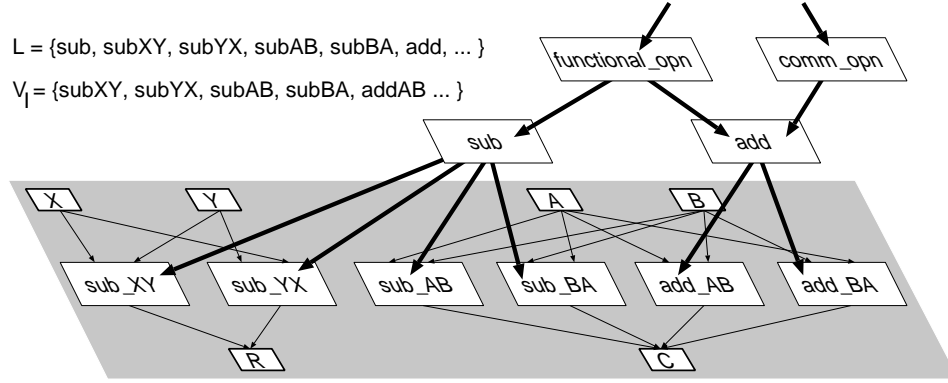


Fig. 3. Example of the type hierarchy in the operation-type library \mathcal{L} . The gray plane represents the ISG.

The complete library \mathcal{L} of operation-types consists of a processor independent part, and a processor dependent part that is derived from an nML description of the target processor. The latter part contains the ISG operation-types.

The operation-type hierarchy can for example be used to group operation-types that have the same behavior and are connected in a different way to the same storage elements. This provides a modeling trick for commutative operation-types: in Figure 3 a CDFG operation of type `add` can then be either mapped on `add_AB` or on `add_BA`, making use of commutativity.

3. CODE GENERATION USING THE ISG

This section discusses how the ISG processor model can be used to generate code for the execution of a given application on a given processor. Abstraction is made of the control structure of the CDFG, as if code generation would proceed basic block by basic block. For simplicity, it is also assumed in this section that each operation-type executes within a single cycle.

Each basic block is then given as a data-flow graph (DFG) which, similar to the ISG, takes the form of a bipartite graph $G_{DFG}\langle V_{DFG}, E_{DFG} \rangle$, where $V_{DFG} = V_O \cup V_V$ with V_O representing CDFG operations and V_V representing the values they produce and consume. The edges in $E_{DFG} \subseteq (V_O \times V_V) \cup (V_V \times V_O)$ represent the data-flow. Code generation then consists in finding a mapping of $G_{DFG}\langle V_{DFG}, E_{DFG} \rangle$ onto $G_{ISG}\langle V_{ISG}, E_{ISG} \rangle$ with the values in V_V mapped on storage elements in V_S and the CDFG operations of V_O mapped on ISG operation-types in V_I .

3.1 Code generation phases

The code generation task is split in consecutive phases, similar to the code generation phases found in classical compilers. However, the algorithms that are used for these phases in the CHES compiler are quite different. Below, the tasks are briefly described in the context of CHES.

First, during the *code selection phase*, it is decided which values will be bound to transitories. When a value is bound to a transitory, this implies that the head and

tail CDFG operations of the corresponding dependency will be executed during the same cycle. Therefore their operation-types may not be conflicting; otherwise the value can only be bound to a static storage element, either a register or a memory,² which implies that the conflicting operation-types are scheduled in different cycles. Conflict free CDFG operations that will be executed in the same cycle, because of the binding of their data dependencies, are grouped and each group is called a *bundle*. A subtask of the code selection phase in CHES is to *refine* CDFG operations, that is, to make them instances of ISG operation-types.

After the code selection phase, the *register allocation phase* binds the remaining values to static storage elements and completes the CDFG with the necessary data transfers. Finally, during the *scheduling phase*, the bundles are bound to time.

The binding of a CDFG to the ISG is further detailed below in a formal way. Sections 3.5, 3.6, and 3.7 then relate this binding to the actual code generation tasks. Code selection is the topic of Section 4.

3.2 Refinement

During code generation, CDFG operations are repeatedly replaced by instances of children of their operation-types. Eventually, they become instances of ISG operation-types, so they can be executed by the processor. This is called *refinement*.

Definition 9 (valid refinement - valid mapping). A CDFG operation $r \in V_{O_R}$ is a *valid refinement* for a CDFG operation $o \in V_O$ with $\text{type}(o) \in \mathcal{L}$ if and only if

$$\left\{ \begin{array}{l} \text{type}(r) = i \in V_I \wedge i \text{ is a subtype of } \text{type}(o) \quad \text{and} \\ \forall n \in \mathbb{N} : \text{data-type}(\text{input}(o, n)) = \text{data-type}(\text{input}(i, n)) \quad \text{and} \\ \forall n \in \mathbb{N} : \text{data-type}(\text{output}(o, n)) = \text{data-type}(\text{output}(i, n)) \end{array} \right. \quad (3)$$

with the functions $\text{input}(o, n)$ and $\text{output}(o, n)$ returning the n -th input value, respectively the n -th output value, of a CDFG operation o ; with $\text{input}(i, n)$ and $\text{output}(i, n)$ returning the appropriate storage elements for ISG operation-type i ; the function data-type returns the data-type of either a value or a storage element.

V_O is notated as V_{O_R} when it contains refined CDFG operations. The function $\text{refinement} : V_O \rightarrow V_{O_R} : o \mapsto r$ defines a valid refinement for each CDFG operation.

The ISG operation-type $i = \text{type}(r) = \text{type}(\text{refinement}(o))$ in equation (3) is a *valid mapping* for the CDFG operation $o \in V_O$. The function $\text{mapping} : V_O \rightarrow V_I : o \mapsto i$ defines a valid mapping for each CDFG operation.

Figure 4 illustrates the functions type , mapping , and refinement for a small CDFG and its two valid mappings on an ISG. Each of these valid mappings has its own mapping and refinement functions, although this is not visible in Figure 4.

For a given CDFG and a given ISG, many different mapping (refinement) functions will usually exist. However, many of them might violate data-dependency or register-capacity constraints, for example, and must be discarded. Still, many mappings may remain that do not violate any constraint. It is the task of the code

²The choice of binding a value to either a register or a memory location is taken by the register allocation phase, see below.

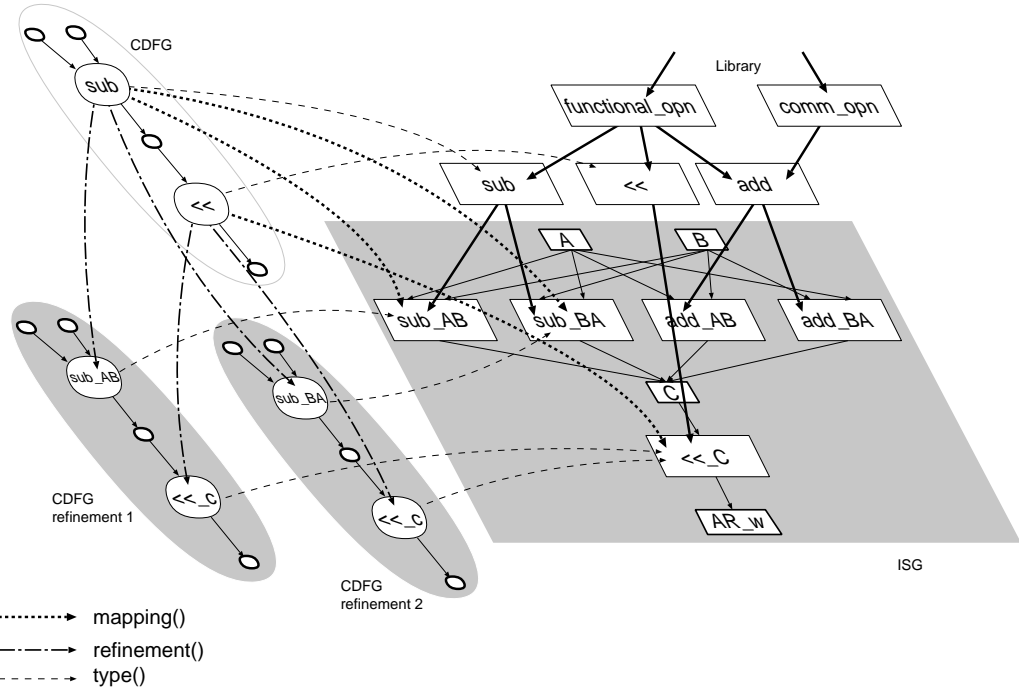


Fig. 4. A small CDFG with two refinements and their relations to the library \mathcal{L} and the ISG.

generator to define a **mapping** function which does not violate any constraint and *minimizes the final cycle count* for the generated code.

3.3 Binding data dependencies

Consider a data dependency between two CDFG operations o_1 and o_2 that are refined, for a given **refinement** function, as r_1 and r_2 , with corresponding value $v_1 \in V_V$. Assume that the refined CDFG operations are bound by the code generator to the ISG operation-types $i_1 = \text{type}(r_1)$ and $i_2 = \text{type}(r_2)$. Because of the data dependency, $\exists n, n' \in \mathbb{N} : \text{output}(o_1, n) = \text{input}(o_2, n') = v_1$. The code generator has different alternatives to bind this data dependency to the ISG, as illustrated in Figure 5 and explained below.

3.3.1 Direct data dependencies. Figure 5(a) shows the binding of a data dependency where $\text{output}(i_1, n) = \text{input}(i_2, n') = t$ with $t \in V_T$. Value v_1 is then bound to the transitory t , denoted $\text{carrier}(v_1) = t$.

However, in the more general case with $\text{output}(i_1, n) = t_0 \neq \text{input}(i_2, n') = t_p \wedge t_0, t_p \in V_T$, the code generator has to add a **move** operation m to the CDFG, as shown in Figure 5(b). Operation m moves its input value v_1 along a path in the ISG, from $\text{carrier}(v_1) = t_0$ to $\text{carrier}(v_2) = t_p$, with v_2 being its output value.

Generally, a **move** operation is implemented by ISG operation-types in the set $V_I^{\text{move}} \subseteq V_I$. V_I^{move} contains several types of data-transfer operation-types, among which **copy** operation-types to copy values between transitories.

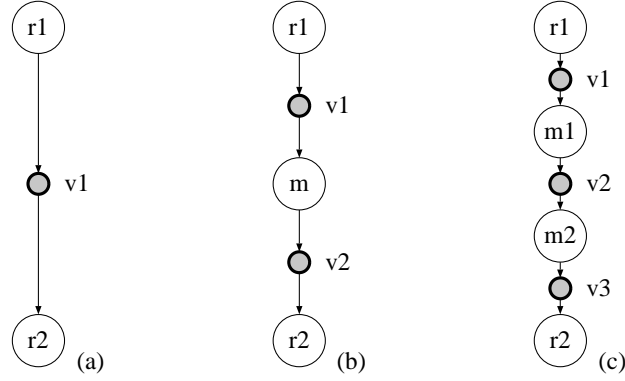


Fig. 5. Data dependencies in the CDFG: (a) direct data dependency; (b) direct data dependency with `move` operation; (c) allocated data dependency.

Definition 10 (delivery of a value). A value is delivered, from the storage element where it is produced to a storage element where it is consumed, by a move operation m . The function $\text{delivery} : V_{O_M} \rightarrow 2^{V_I^{\text{move}}} : m \mapsto I_m$ returns the set I_m of ISG operation-types that are selected by the code generator to implement a particular move operation m . $V_{O_M} \subset V_{O_R}$ is the set of move operations in the CDFG.

For the data dependency of Figure 5(b), the move operation m will be implemented by a series of copy operation-types. This concept is formalized in the following definition of a *direct data dependency*:

Definition 11 (direct data dependency — direct coupling).

A data dependency between two refined CDFG operations $r_1, r_2 \in V_{O_R}$ is *direct* if it is implemented in the ISG as a valid direct path containing copy operations:

$$\begin{aligned}
 & \exists n, n' \in \mathbb{N}, \exists m \in V_{O_M} : \text{output}(r_1, n) = v_1 = \text{input}(m) \\
 & \wedge \text{output}(m) = v_2 = \text{input}(r_2, n') \\
 & \wedge \text{type}(r_1) = i_1 \wedge \text{type}(r_2) = i_2 \wedge i_1, i_2 \in V_I \\
 & \wedge \exists P[i_1 \xrightarrow{d} t_0 \xrightarrow{d} c_1 \xrightarrow{d} t_1 \xrightarrow{d} \dots \xrightarrow{d} c_p \xrightarrow{d} t_p \xrightarrow{d} i_2]_{t_i \in V_T, c_i \in \text{delivery}(m) \subseteq V_I^{\text{move}}} \\
 & \wedge \text{output}(i_1, n) = t_0 \wedge \text{input}(i_2, n') = t_p
 \end{aligned} \tag{4}$$

A *direct data dependency* between two refined CDFG operations $r_i, r_j \in V_{O_R}$ is notated by means of a superscript d above a dependency arrow, in a CDFG operation path: $P[r_i \overset{d}{\rightsquigarrow} r_j]_{r_i, r_j \in V_{O_R}}$.

Two refined CDFG operations are *directly coupled* to each other if they are related by a direct data dependency. This is a symmetric relation. The predicate $\text{direct} : V_{O_R} \times V_{O_R} \rightarrow \mathbb{B} : (r_i, r_j) \mapsto \text{true/false}$ indicates if two refined CDFG operations

r_i and r_j are *directly coupled* or not:

$$\text{direct}(r_i, r_j) \iff \begin{cases} P[r_i \overset{d}{\rightsquigarrow} r_j]_{r_i, r_j \in V_{O_R}} & \text{or} \\ P[r_j \overset{d}{\rightsquigarrow} r_i]_{r_j, r_i \in V_{O_R}} \end{cases} \quad (5)$$

Because transitories have zero delay, two CDFG operations having a direct data dependency will be executed during the same cycle. The decision to implement a data dependency as a direct data dependency or not is taken during the *code selection phase*.

3.3.2 Allocated data dependencies. In Figure 5(c), an *allocated data dependency* is shown. Such a data dependency is bound to a path in the ISG that includes a static storage element. It also holds for an allocated data dependency that $\text{output}(i_1, n) = t_0 \neq \text{input}(i_2, n') = t_p \wedge t_0, t_p \in V_T$, but now *two* move operations m_1 and m_2 are added to the CDFG by the code generator. Operation m_1 moves v_1 from $\text{carrier}(v_1) = t_0$ to $\text{carrier}(v_2) \in V_R \cup V_M$ and m_2 moves v_2 to $\text{carrier}(v_3) = t_p$. Because v_2 is bound to a static storage element, $\text{delivery}(m_1)$ will now contain a **write** or a **store** operation-type and analogously $\text{delivery}(m_2)$ will contain a **read** or a **load** operation-type. V_I^{move} indeed also contains **read** and **write** operation-types to access register(file)s, and **load** and **store** operation-types to access memories.

Definition 12 (allocated data dependency).

A *data dependency* between two refined CDFG operations $r_1, r_2 \in V_{O_R}$ is *allocated* if it is implemented by an ISG path P that is *not* direct:

with $\text{type}(r_1) = i_1 \wedge \text{type}(r_2) = i_2$, $\exists P[i_1 \rightarrow t_0 \rightarrow i_1^m \rightarrow s_1 \rightarrow \dots \rightarrow i_p^m \rightarrow t_p \rightarrow i_2]_{t_0, t_p \in V_T; s_i \in V_S; i_i^m \in V_I^{\text{move}}, \exists s_i \in P : s_i \notin V_T}$.

An *allocated data dependency* between two refined CDFG operations $r_i, r_j \in V_{O_R}$ is notated by means of the superscript a : $P[r_i \overset{a}{\rightsquigarrow} r_j]_{r_i, r_j \in V_{O_R}}$.

The symmetric relation “*allocated*”, between two refined CDFG operations holds if there exists an allocated data dependency between them. The predicate $\text{allocated} : V_{O_R} \times V_{O_R} \rightarrow \mathbb{B} : (r_i, r_j) \mapsto \text{true/false}$ evaluates this relation:

$$\text{allocated}(r_i, r_j) \iff \begin{cases} P[r_i \overset{a}{\rightsquigarrow} r_j]_{r_i, r_j \in V_{O_R}} & \text{or} \\ P[r_j \overset{a}{\rightsquigarrow} r_i]_{r_j, r_i \in V_{O_R}} \end{cases} \quad (6)$$

The *register allocation* or *data routing* tool binds allocated data dependencies to static storage, either to registers or to memory. Some allocated data dependencies must be bound to a series of two or more static storage elements before the destination carrier can be reached from the source carrier. In these cases a chain of three or more **move** operations must be inserted in the CDFG, of which all but those at the extremities are between static storage elements. The formalism above can be extended in a straightforward way to these cases.

3.4 Bundles and patterns

Based on the relation **direct** of Definition 11 an *equivalence relation coupled* can be defined on a *refined* CDFG, for a given **refinement** function.

Definition 13 (coupled CDFG operations). Two refined CDFG operations $r_i, r_j \in V_{O_R}$ are *coupled* if there exists a sequence of pairwise directly coupled

refined CDFG operations of which r_i and r_j are at the extremities. The predicate $\text{coupled} : V_{O_R} \times V_{O_R} \rightarrow \mathbb{B} : (r_i, r_j) \mapsto \text{true/false}$ indicates if two CDFG operations r_i and r_j are *coupled* or not, and is defined recursively:

$$\text{coupled}(r_i, r_j) \iff \begin{cases} r_i = r_j & \text{or} \\ \text{direct}(r_i, r_j) & \text{or} \\ \exists r_k \in V_{O_R} : \text{coupled}(r_i, r_k) \wedge \text{coupled}(r_k, r_j) \end{cases} \quad (7)$$

The relation coupled is reflexive, symmetric and transitive, and is thus an equivalence relation.

Definition 14 (bundles and refined bundles).

A *refined bundle* R is a set of refined CDFG operations that is an equivalence class defined by the relation coupled for a given **refinement** function. The set of all refined bundles is denoted by \mathcal{R} .

$$\forall R \in \mathcal{R} : R \subseteq V_{O_R} \wedge \forall r_i, r_j \in R : \text{coupled}(r_i, r_j) \quad (8)$$

A *bundle* B is a set of CDFG operations that *can* be refined to form a refined bundle. The set of all bundles is denoted by \mathcal{B} . The set of all refined bundles that are refinements of B is denoted by \mathcal{R}_B .

$$\forall B \in \mathcal{B}, \exists R \in \mathcal{R} : \forall o \in B, \exists r \in R : \text{refinement}(o) = r \quad (9)$$

Each bundle may be associated with a set of **refinement** functions, namely those of its refined bundles. In Figure 4 the bundle $\{\text{sub}, \ll\}$ exists because of the refined bundles $\{\text{sub_AB}, \ll_C\}$ and $\{\text{sub_BA}, \ll_C\}$.

The following theorem will be used in Section 4 to construct (refined) bundles:

THEOREM 1 (CONSTRUCTION OF BUNDLES). *Two refined operations that have a direct data dependency belong to the same bundle and two operations having an allocated data dependency must be in different bundles:*

$$\forall r_i, r_j \in V_{O_R}, r_i \in R_i, r_j \in R_j : \text{direct}(r_i, r_j) \Rightarrow R_i = R_j \quad (10)$$

$$\forall r_i, r_j \in V_{O_R}, r_i \in R_i, r_j \in R_j : \text{allocated}(r_i, r_j) \Rightarrow R_i \neq R_j \quad (11)$$

Theorem 2 defines the *timing* for operations in a bundle, which will lead to a number of constraints for the construction of *valid (refined) bundles*.

THEOREM 2 (TIMING IN A BUNDLE). *If, in an ISG, all transitories have zero delay and each ISG operation-type executes in a single cycle, then all operations in a bundle B must be executed in one and the same cycle.*

3.4.1 Correctness constraints of bundles. Because operations in a bundle will be scheduled such that they execute in the same cycle (Theorem 2), they are not allowed to have encoding conflicts nor hardware conflicts with each other. The code selection tool *prevents conflicts* by changing a direct data dependency into an allocated data dependency when required, thereby *splitting the bundle*.

In the remainder of this section, definitions will be introduced to eventually define formal correctness constraints for valid bundles. First some functions that were defined for ISG operation-types, are overloaded for CDFG operations.

Definition 15 (enabling condition of a refined CDFG operation).

The function $\text{enabling} : V_{O_R} \setminus V_{O_M} \rightarrow 2^{\mathbb{B}^w} : r \mapsto E_r$ returns the set of instructions E_r that enable the refined operation r .

$$\text{enabling}(r) = \text{enabling}(\text{type}(r)) \quad (12)$$

The function $\text{enabling} : V_{O_M} \rightarrow 2^{\mathbb{B}^w} : m \mapsto E_m$ returns the set of instructions E_m that enable the move operation m .

$$\text{enabling}(m) = \bigcap_{c \in \text{delivery}(m)} \text{enabling}(c) \quad (13)$$

Definition 16 (resources of a refined CDFG operation).

The function $\text{resources} : V_{O_R} \setminus V_{O_M} \rightarrow 2^{V_T} : r \mapsto T_r$ returns the set of transitories T_r to which the results of the refined CDFG operation r are written.

$$\text{resources}(r) = \{t \in V_T \mid \exists n \in \mathbb{N} : t = \text{output}(r, n)\} \quad (14)$$

The function $\text{resources} : V_{O_M} \rightarrow 2^{V_T} : m \mapsto T_m$ returns the set of transitories T_m to which the results of the move operation m are written.

$$\text{resources}(m) = \{t \in V_T \mid \exists c \in \text{delivery}(m) : t = \text{output}(c)\} \quad (15)$$

THEOREM 3 (CONFLICT FREE REFINED BUNDLE). *A refined bundle R is conflict free if and only if it is free of encoding conflicts and of hardware conflicts, as respectively defined by equation (16) and equation (17):*

$$\text{enabling}(R) = \bigcap_{r \in R} \text{enabling}(r) \neq \emptyset \quad (16)$$

$$\forall r_i, r_j \in R : r_i \neq r_j \Rightarrow \text{resources}(r_i) \cap \text{resources}(r_j) = \emptyset \quad (17)$$

Equation (16) defines the enabling condition of a refined bundle; the resources of a refined bundle are defined similarly:

$$\text{resources}(R) = \bigcup_{r \in R} \text{resources}(r) \quad (18)$$

Definition 17 (validity of bundles and refined bundles). A refined bundle that is free of conflicts is a *valid refined bundle*. A bundle is *valid* if its operations can be refined to form a valid refined bundle. The sets \mathcal{B} , \mathcal{R} and \mathcal{R}_B (Definition 14) are from here on restricted to contain only valid bundles and valid refined bundles.

3.4.2 Conflict constraints between refined bundles. Given the fact that all operations in a bundle are executed in the same cycle (Theorem 2) and given equations (16) and (18), the scheduler can use a conflict model between refined bundles. The elements of a set of bundles $\alpha \subseteq \mathcal{R}$ are not conflicting with each other, if the following two constraints hold:

$$\forall R_i, R_j \in \alpha : R_i \neq R_j \Rightarrow \text{resources}(R_i) \cap \text{resources}(R_j) = \emptyset \quad (19)$$

$$\bigcap_{R \in \alpha} \text{enabling}(R) \neq \emptyset \quad (20)$$

3.4.3 *Patterns.* A (refined) bundle is a *subset* of CDFG operations. Sometimes it will be needed to refer to a *subgraph* of the CDFG, which corresponds to a bundle, or to a subgraph of the ISG. These subgraphs are respectively called *CDFG-patterns* and *ISG-patterns*.

3.5 Code selection

As said above, in the CHES code generator the objects in the CDFG are bound to objects in the ISG in subsequent phases. In a first phase, called code selection, it is decided which data dependencies will become *direct data dependencies*. In other words, the operations in the CDFG are *partitioned* into bundles.

This phase matches CDFG-patterns onto ISG-patterns. The CDFG operations are refined according to equation (3) and direct data dependencies are bound, as shown in Figure 5(a) and (b), and according to Definition 11. The code selection tool may add some *move* operations to the CDFG for this. The allocated data dependencies are not yet bound, but it is verified that each input and output of a refined bundle can access static storage. It is made certain that the bundles are valid, according to Definition 17.

The selection of a valid refined bundle to implement each bundle is delayed to the subsequent task of register allocation, see Section 4.4.1.

3.6 Register allocation

In the register allocation phase, which is called *data routing* in the CHES compiler, the (allocated) data dependencies between bundles are bound [Lanneer 1993]. This means that for each bundle a refined bundle is chosen and *move* operations are added at their inputs and outputs to access static storage, while satisfying constraints (10) to (17) to ensure the bundle's validity.

In addition to the above constraints, data routing must also ensure that the capacity of each of the storage elements (Definition 5) is never exceeded. This can be guaranteed by *spilling* some values from register to memory, or by partly fixing the execution order between some bundles. The latter ensures that values assigned to the same storage element are not *alive* at the same moment. To spill a register to memory and to reload it, ISG-paths visiting more than one static storage must be considered. If such a path is chosen, new bundles that only consist of a single *move* operation are inserted in the CDFG.

3.7 Scheduling

After register allocation all operations and values are bound to the ISG, but they must still be bound to time. During the scheduling (or *compaction*) phase, the operations are packed in instructions, with the constraint that operations in the same bundle must be executed by the same instruction. Of course, different bundles may be scheduled in parallel, in the same instruction. As the objective is to minimize cycle count, this should be done as much as possible, while satisfying the conflict constraints (19) and (20).

An operation can be executed by each instruction in its enabling condition, so for each group of operations that must execute in the same cycle, an instruction is selected from the intersection of their respective enabling conditions.

4. CODE SELECTION

Informally, code selection is the task of partitioning the (C)DFG — a directed acyclic graph (DAG), henceforth called the *subject DAG* — into DAG patterns that can each be implemented by a single instruction. Code selection consists of two subtasks:

- (1) *Matching* template patterns to the subject DAG. This means finding subgraphs of the subject DAG that are *isomorphic* with the template DAGs, preserving the order of the incident edges at each operation vertex. This DAG matching problem is NP-complete, because the graph isomorphism problem is NP-complete [Garey and Johnson 1979]. The matched patterns in the subject DAG may overlap with each other, or be contained in other matched patterns.
- (2) *Covering* the subject DAG with matched patterns. A set of matched template DAGs must be selected that cover all vertices of the subject DAG, optimizing a given cost function. The DAG covering problem is also NP-complete.

Since both its subtasks are NP-complete, code selection is NP-complete. A special case of the code selection problem occurs when the intermediate representation is a sequence of *expression trees* that must be covered by *template trees*. Code selection then becomes an instance of the tree pattern matching and tree covering (or tree parsing) problems, for which an optimal solution can be found in linear time. Therefore the majority of compilers use heuristics to split the subject DAG into expression trees and perform code selection based on tree pattern matching and covering, with template trees modeling the instruction-set [Aho, Sethi, and Ullman 1986; Wilhelm and Maurer 1995; Fraser and Hanson 1995].

4.1 Matching by bundling

In CHESS, no template patterns are given; instead, the DAG patterns are built on the fly, while they are matched, by a technique called *bundling*.

This technique looks up *all* valid bundles of the CDFG, but does not take any decision; the bundles may still overlap, or be contained in each other.

The bundles that partition the CDFG are selected in the covering subtask by a branch-and-bound algorithm, as described in Section 4.2, .

4.1.1 Restrictions for bundle candidates. Below some relations are defined between CDFG operations, based on definitions given in Section 3. Their aim is to determine bounds on the search space of bundling. These bounds will also allow to partition the subject DAG in subgraphs that can be covered separately.

Each CDFG operation is annotated with its valid mappings (Definition 9).

$$\forall o_i \in V_O : M_{o_i} = \{i_{i,k} \in V_I \mid \exists \text{mapping}_k : V_O \rightarrow V_I : o_i \mapsto i_{i,k}\} \quad (21)$$

Each data dependency in the CDFG is checked to see if it can be made a direct data dependency (Definition 11) by choosing an appropriate *refinement* function.

Definition 18. The predicate $\widetilde{\text{direct}} : V_O \times V_O \rightarrow \mathbb{B} : (o_i, o_j) \mapsto \text{true/false}$ indicates whether two CDFG operations o_i and o_j have a data dependency that

can be implemented as a direct data dependency or not:

$$\widetilde{\text{direct}}(o_i, o_j) \iff \exists \text{refinement}_k : V_O \rightarrow V_{O_R} : \begin{cases} \text{refinement}_k(o_i) = r_{i,k} & \text{and} \\ \text{refinement}_k(o_j) = r_{j,k} & \text{and} \\ \text{direct}(r_{i,k}, r_{j,k}) \end{cases} \quad (22)$$

Remark that $\widetilde{\text{direct}}$ generalizes the predicate direct (Definition 11) to apply for non-refined CDFG operations, by calculating direct for all their valid refinements. Based on $\widetilde{\text{direct}}$, each operation is annotated with a *candidate set* for bundling:

$$\forall o_i \in V_O : C_{o_i} = \{o_j \in V_O \mid \widetilde{\text{direct}}(o_i, o_j)\} \quad (23)$$

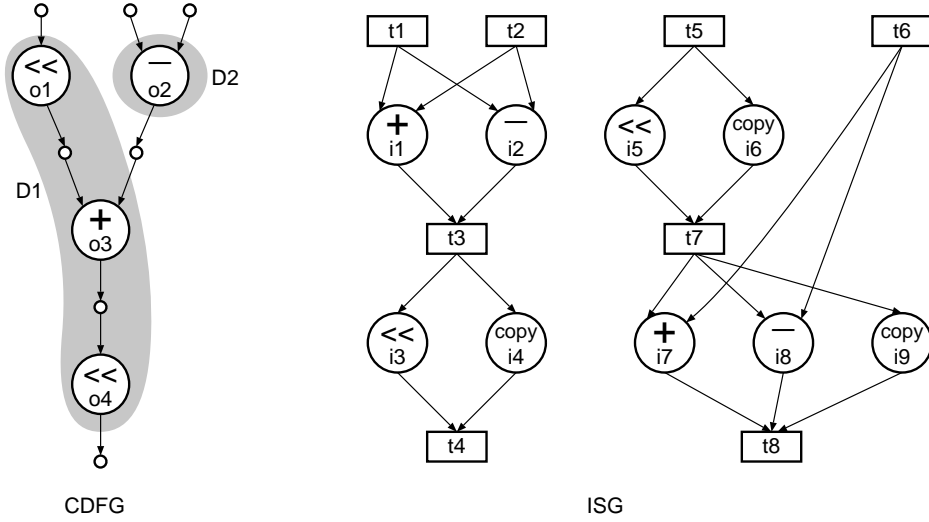


Fig. 6. An example CDFG and an ISG on which it can be mapped in several ways.

Figure 6 shows an ISG and a CDFG that can be mapped on it (all storage elements in this ISG, named t_1, \dots, t_8 , are transitories³). The mapping possibilities of the CDFG operations are $M_{o_1} = \{i_3, i_5\}$, $M_{o_2} = \{i_2, i_8\}$, $M_{o_3} = \{i_1, i_7\}$, and $M_{o_4} = \{i_3, i_5\}$. The relation $\widetilde{\text{direct}}$ evaluates as **true** for the data dependency (o_1, o_3) : mapping $\{(o_1, i_5), (o_3, i_7)\}$ makes it a **direct** dependency because of the direct ISG path $P[i_5 \xrightarrow{d} i_7]$; $\widetilde{\text{direct}}(o_3, o_4)$ is also **true**, because of mapping $\{(o_3, i_1), (o_4, i_3)\}$ and the direct ISG path $P[i_1 \xrightarrow{d} i_3]$; $\widetilde{\text{direct}}(o_2, o_3)$ evaluates to **false**. For Figure 6 the candidate sets are $C_{o_1} = \{o_3\}$, $C_{o_2} = \emptyset$, $C_{o_3} = \{o_1, o_4\}$, and $C_{o_4} = \{o_3\}$.

³ t_1, t_2, t_5 , and t_6 can be written with values from a static storage element; values on t_4 and t_8 can be written to a static storage element.

In order to set bounds to the search space for valid bundles, the relation $\widetilde{\text{coupled}}$ is defined based on $\widetilde{\text{direct}}$, much like coupled was defined based on direct in Definition 13:

Definition 19 (direct components). The predicate $\widetilde{\text{coupled}} : V_O \times V_O \rightarrow \mathbb{B} : (o_i, o_j) \mapsto \text{true/false}$ is true for two CDFG operations $o_i, o_j \in V_O$, if there exists a sequence of CDFG operations for which the predicate $\widetilde{\text{direct}}$ holds between each pair of adjacent operations and for which o_i and o_j are at the extremities. This is recursively defined as:

$$\widetilde{\text{coupled}}(o_i, o_j) \iff \begin{cases} o_i = o_j & \text{or} \\ \widetilde{\text{direct}}(o_i, o_j) & \text{or} \\ \exists o_k \in V_O : \widetilde{\text{coupled}}(o_i, o_k) \wedge \widetilde{\text{coupled}}(o_k, o_j) \end{cases} \quad (24)$$

The relation $\widetilde{\text{coupled}}$ is an equivalence relation (reflexive, symmetric, transitive) that partitions the set of CDFG operations V_O into equivalence classes, which will be called *direct connected components* or in short *direct components*. The set of all *direct components* in V_O is denoted as \mathcal{D} .

The *direct components* can be computed incrementally by iterating once over all operations and over their candidate sets, as suggested by the combination of equations (23) and (24). For Figure 6, $\mathcal{D} = \{D_1, D_2\}$ with $D_1 = \{o_1, o_3, o_4\}$ and $D_2 = \{o_2\}$. In practice, the *direct components* only contain a few operations, as can be seen from the results in Section 4.5.

The following theorem allows to divide the code selection problem into subproblems.

THEOREM 4. *Each bundle is a subset of a direct component :*

$$\forall B \in \mathcal{B}, \exists D \in \mathcal{D} : B \subseteq D$$

PROOF. The proof follows directly from the fact that *direct components* are defined by the relation $\widetilde{\text{direct}}$, which is a generalization of the relation direct that defines bundles. \square

Consequently, all valid bundles in a CDFG can be found by searching valid bundles in each of its *direct components* separately.

Theorem 4 is also useful to reduce the complexity of the covering subtask of code selection, because it will allow to cover the *direct components* separately. This is formulated by Theorem 5 in Section 4.2.

The valid bundles for the CDFG of Figure 6 are — taken into account the accessibility of static storage, and other correctness constraints (Definition 17) — $\{o_2\}$, with mapping $\{(o_2, i_2)\}$ or $\{(o_2, i_8)\}$; $\{o_3\}$, with mapping $\{(o_3, i_1)\}$ or $\{(o_3, i_7)\}$; $\{o_1\}$, with mapping $\{(o_1, i_5)\}$; $\{o_4\}$, with mapping $\{(o_4, i_5)\}$; $\{o_1, o_3\}$, with mapping $\{(o_1, i_5), (o_3, i_7)\}$; and $\{o_3, o_4\}$, with mapping $\{(o_3, i_1), (o_4, i_3)\}$.

All these bundles are subsets of the *direct component* D_1 , except for $\{o_2\}$ which is equal to D_2 ; D_1 itself is not a valid bundle. Note that once the CDFG is partitioned into bundles, several bundles may still have more than one valid mapping.

4.1.2 *Generation of valid bundles.* Bundles are constructed by defining intermediate results representing “different stages of validity”: a bundle candidate, a valid bundle candidate and a valid bundle.

- A *bundle candidate* is a group of CDFG operations which are all pairwise coupled. The algorithm starts with trivial bundle candidates, each consisting of just one operation. A new bundle candidate is constructed by extending an existing valid bundle candidate (see below) with an operation taken from the candidate set of an operation in the existing bundle candidate. Recalling equation (23), it should be clear that this step uses the relation direct and thus only constructs bundle candidates that are subsets of a direct component, in accordance with Theorem 4.
- A *valid bundle candidate* is a bundle candidate that has at least one refinement that can be part of a valid refined bundle. For that refinement, all internal dependencies must be direct, and it must consist of operations that have no conflict with each other; thus constraints (10), (16) and (17) must hold.
- A *valid bundle* has at least one refinement that is a valid refined bundle according to Definition 17; thus, in addition to being a valid bundle candidate, constraint (11) must hold, which requires that its data dependencies with external operations can be allocated to static storage.

If one of its inputs or outputs is *non-valid*, that is, it can not be routed to a static storage element, a valid bundle candidate can not be a valid bundle, because the corresponding data dependency can not be an allocated one. To become a valid bundle, the valid bundle candidate must then be augmented with an operation, such that the corresponding data dependency becomes direct.

4.2 Covering

For the covering subtask of code selection, a subject DAG is given, together with patterns that are matched to it. In general, each operation can be covered by one or more bundles and each bundle covers one or more operations. The problem is now to cover the subject DAG with patterns, such that the generated code for that cover of the subject DAG will have *minimal cycle count*.

4.2.1 Problem definition

4.2.1.1 *Cost function.* In the assumption that each bundle is executed in a single cycle, one could expect that minimization of the number of bundles leads to optimal code, with minimal cycle count. However, this is not completely true.

- Bundles can often be *executed in parallel*, reducing the total amount of cycles. Consider for example two different covers of the same subject DAG: the first one consisting of three bundles that each must be executed in a separate cycle; the second consisting of four bundles that can be combined two by two, thus resulting in a cycle count of two.
- During the matching subtask, it is verified that inputs and outputs of each bundle can be routed to a static storage. However, if no static storage element exists that is *directly* accessible by both, the bundle producing a value and the bundle consuming it, additional move bundles are needed between two static storage

elements. This costs extra cycles if the move bundles can not be executed in parallel to other bundles. Again, another cover with slightly more bundles might lead to better code.

Nevertheless, the solution presented here neglects the effects mentioned above. It is assumed, like in most code selection approaches, that a cover leading to optimal code may be found with a purely *additive cost function* [Ferdinand, Seidl, and Wilhelm 1994; Wilhelm and Maurer 1995], where each bundle has a non-negative constant cost. The cost of a cover \mathcal{C} is then computed as the accumulated cost of all selected bundles and code selection must find a cover that minimizes this cost:

$$\text{cost}(\mathcal{C}) = \sum_{B_j \in \mathcal{C}} \text{cost}(B_j) \quad (25)$$

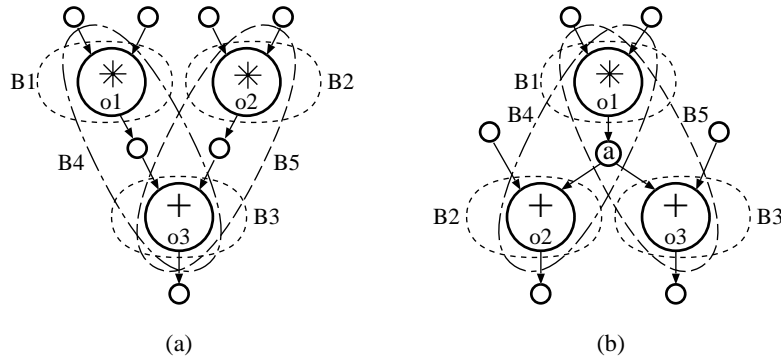


Fig. 7. Two CDFGs with overlapping bundles.

4.2.1.2 *Graph covering.* Figure 7 shows two small example CDFGs, each with 3 operations and with 3 bundles that each cover one of the operations and 2 bundles that each cover 2 operations. None of the bundles have internal output values that can be routed to static storage.

When covering a subject DAG its data dependencies must be preserved. This requirement may prohibit the selection of overlapping bundles in some cases, while in other cases a cover with overlapping bundles leads to the best solution:

- In Figure 7(a), the set $\{B_4, B_5\}$ is not a legal cover of the subject DAG, because the internal value of B_4 is then an *internal output value* that is needed by o_3 of B_5 , and this value can not be routed to static storage; similarly, the internal value of B_5 is needed by o_3 of B_4 . However, by discarding covers with overlapping bundles, the cover $\{B_4, B_5\}$ will not be selected.
- In Figure 7(b), the internal value a of B_4 (and of B_5) can not be routed to static storage, and is thus not available for B_3 (or respectively for B_2). This makes $\{B_2, B_5\}$ and $\{B_3, B_4\}$ illegal as DAG cover. Discarding $\{B_4, B_5\}$ because of the overlap would lead to the inferior solution $\{B_1, B_2, B_3\}$ obtained by tree pattern matching. Even worse, if bundles B_2 and B_3 are not valid — which is quite possible on a DSP — $\{B_4, B_5\}$ is the only valid cover.

Selecting overlapping bundles without need has as a consequence that operations are unnecessarily executed multiple times. This may lead to higher power consumption or, indirectly, because of storage requirements for input and output values, to a higher cycle count.

Problem 1 (minimum graph cover). Given a collection \mathcal{B} of bundles that induce patterns or subgraphs of the graph $G_{CDFG}(V_{CDFG}, E_{CDFG})$, the minimum graph cover problem is the search for a minimal set of patterns that cover G_{CDFG} . When the bundles have a different cost, the corresponding covering problem is *weighted* and is the search for a set of (induced) patterns that cover G_{CDFG} with minimal accumulated cost. In both cases, the resulting cover is denoted as \mathcal{C} and must have the same semantics as G_{CDFG} .

4.3 A branch-and-bound solution

4.3.1 *Basic strategy.* Each operation o_i is annotated with the set of all bundles by which it can be covered: $\mathcal{B}_{o_i} = \{B_j \in \mathcal{B} | o_i \in B_j\}$.

There are operations o_i that can be covered by only *one* bundle. Such a bundle $B_j \in \mathcal{B}_{o_i} : |\mathcal{B}_{o_i}| = 1$ is *essential* in the cover, because o_i would otherwise not be executed by the generated code. Essential bundles are always selected immediately.

After the essential bundles are selected, a *search tree* is constructed where each node represents a *partial* cover of the subject DAG. Figure 8 represents the search tree built when covering the CDFG of Figure 7(b).

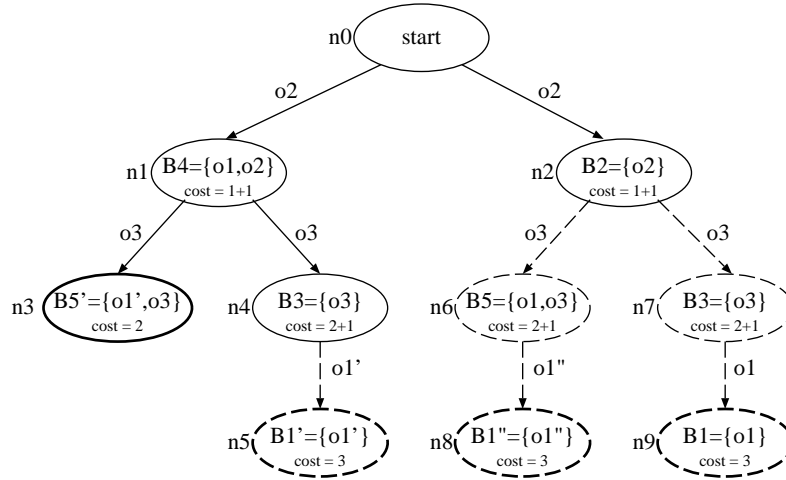


Fig. 8. Search tree to find the best cover for Figure 7(b); all dashed nodes are pruned by the branch-and-bound search (see below).

In general, at each level in the search tree the branches from a node to its children represent different choices to cover an operation o_i . Each branch models the selection of a $B_j \in \mathcal{B}_{o_i}$. Consequently, at each level a bundle B_j is added to the partial cover and $|B_j|$ operations are covered.

In a complete search tree, each leaf node represents a complete cover. For Figure 8 four different complete covers can thus be distinguished: $\{B'_5, B_4\}$, $\{B'_1, B_3, B_4\}$, $\{B''_1, B_5, B_2\}$, and $\{B_1, B_3, B_2\}$. Other complete covers are discarded because they would contain unnecessarily overlapping bundles.

4.3.2 Overlapping bundles. Cases in which overlapping bundles might be needed to find an optimal solution, are handled by transforming the subject DAG. The transformation is illustrated in Figure 9 for the example of Figure 7(b), when bundle B_4 is selected. In general, when a branch in the search tree selects a bundle B_j

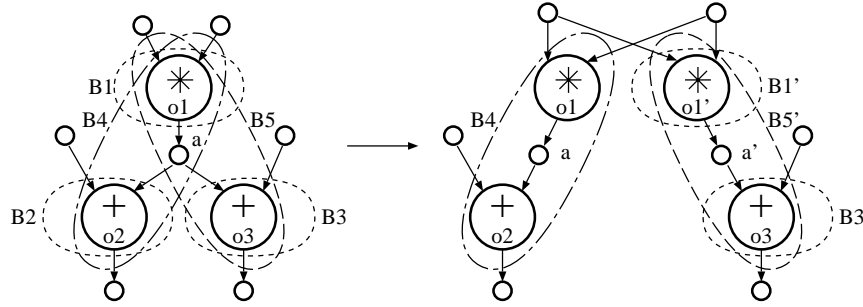


Fig. 9. Duplication of operation o_1 when bundle B_4 is selected.

with a *non*-valid internal output value, the operations that are needed to generate this internal output value in the bundle are duplicated, together with the values they produce. External uses of the internal output value receive as new operand the duplicate of that value. Each operation o'_i that is a duplicate of an operation o_i is annotated with a set $\mathcal{B}_{o'_i}$ containing the bundles by which it can be covered. The bundles $B'_j \in \mathcal{B}_{o'_i}$ are computed from the $B_j \in \mathcal{B}_{o_i}$.

Due to the transformation of the subject DAG described above, the DAG cover never contains overlapping bundles. The choices to cover an operation o_i are *exclusive*; which reduces the size of the search tree. For example, when the children of node n_4 are created in Figure 8, the second possibility to cover o'_1 — by B'_5 — is not considered because B'_5 overlaps with B_3 , so B'_1 becomes essential to cover o'_1 .

The result of Problem 1 will thus be a partition of V_O , in which the partition classes are the bundles $B_j \in \mathcal{C}$. Note however that V_O may have been modified because of duplicated operations.

4.3.3 Narrowing the search tree. Each level in the search tree examines the bundles $B_j \in \mathcal{B}_{o_i}$ that cover an operation o_i . Although a valid search tree may be constructed for every order in which operations o_i and bundles B_j are selected, it is advantageous to consider the operations o_i in the order of increasing $|\mathcal{B}_{o_i}|$, to keep the search tree narrow and thus small.

4.3.4 Pruning the search tree: branch-and-bound search. An *exhaustive search* for the best cover encompasses the construction of the complete search tree, as in Figure 8, and comparing the cost of its leaves, that is, of all complete covers.

However, by constructing the search tree during the search, only part of the tree must be constructed. The search tree can indeed be *pruned* a lot by a *branch-and-bound search* strategy [Papadimitriou and Steiglitz 1982]. In this strategy, a lower bound is calculated for the cost of each node in the search tree. If this lower bound is not smaller than the cost of the current best complete solution (upper bound), the node is pruned.

The amount of pruning is largely determined by the quality of the first complete solution — found by the *branching heuristic* — and by the accuracy of the *lower bound* calculation.

For the search tree of Figure 8, all dashed nodes can be pruned, meaning that only nodes n_0 to n_4 are actually constructed.

4.3.5 Branching heuristic. A commonly used heuristic in branch-and-bound algorithms is to branch each time to the leaf node with the lowest lower bound in the current search tree, independently of when the node was created.

Another, more aggressive approach is to construct the search tree strictly in a depth first manner: only the *lastly created children* are considered when choosing a node to create children for. Other leaf nodes are only considered when no branches from the lastly created children are possible anymore. In this way a first complete cover is constructed by a *greedy search*, rapidly yielding an as-good-as-possible first upper bound.

The branching heuristic currently used in our code selection tool considers first the search node with the smallest lower bound *within the lastly created child nodes*; ties are broken by considering the largest bundles first, as they cover the most operations and can thus be expected to be in the minimal cover.

Experience showed that this heuristic almost never fails, so that the greedy search rapidly yields a first complete cover that is often also an optimal one. This was for example always the case for the examples reported in Section 4.5. Consequently, the greedy search can even be used *on its own* as a *fast covering heuristic*, stopping the search when the first complete solution is found.

The search tree in Figure 8 is constructed in the order shown by the node numbering, by applying the branching heuristic as described above.

4.3.6 Lower bound estimation. A lower bound estimate C_L must be calculated for *each* leaf node, estimating the cost of the complete covers that are represented by its descendants in the search tree. A first lower bound, although not a very accurate one, is given by the accumulated cost of the already selected bundles in the corresponding partial cover, and is denoted C_{part} .

It can be improved by adding the minimum cost to *cover a maximum independent set* of operations that are not yet covered. The last estimate is borrowed from minimum set covering problems in two level logic minimization [Rudell 1989; Coudert and Madre 1995]. Another, less accurate but computationally less expensive lower bound is described in [Van Praet 1997].

Given a maximum set X of operations that can be covered independently, that is, there exists no bundle that covers more than one operation in X , the lower bound

is calculated as follows:

$$C_{LM} = C_{\text{part}} + \sum_{o_i \in X} \text{minimum}_{B_j \in \mathcal{B}_{o_i}}(\text{cost}(B_j)) \quad (26)$$

Because the operations in the set X are independent, the costs of the cheapest bundles covering the operations in X may indeed simply be added, while it is guaranteed that equation (26) yields a lower bound.

To find a maximum independent set of non-covered operations, a graph $G_{\text{ind}}(V_{O_{NC}}, E_{\text{ind}})$ is constructed with as vertices all operations $o_i \in V_{O_{NC}}$. If o_i and o_j are compatible, that is, they are not independent because they can be covered by the same bundle ($\mathcal{B}_{o_i} \cap \mathcal{B}_{o_j} \neq \emptyset$), an edge (o_i, o_j) is put in E_{ind} . Finding a *maximum* independent set in a graph is however an NP-complete problem [Garey and Johnson 1979], so heuristics must be used to efficiently calculate this lower bound. These heuristics calculate a *maximal* independent set X , using a greedy approach, as for example the one described in [Coudert 1996].

For the search tree of Figure 8, the maximum independent set of the start node n_0 is $\{o_2, o_3\}$, as can be seen from the left part of Figure 9; for node n_1 it is either $\{o'_1\}$ or $\{o_3\}$, see the right part of Figure 9.

The lower bound of equation (26) is very good for small covering problems with sparse covering matrices [Coudert 1996]. This is the case for code selection.

4.3.7 “Divide and conquer”. The DAG covering problem of this section is NP-complete, so it is important to have means to keep it small. The subject DAG of Problem 1 can be divided into subgraphs because of Theorem 4.

THEOREM 5 (PARTITIONING THE SUBJECT DAG). *If the graph covering problem (Problem 1) must minimize a purely additive cost function, being the accumulated cost of the selected bundles where each bundle B_j has a constant cost given by $\text{cost}(B_j)$, then each subgraph induced by a direct component may be covered separately. The union of the subgraph covers is a minimal cost cover for the subject DAG.*

PROOF. If $o_i \in D_k \in \mathcal{D}$ and $o_j \in D_l \in \mathcal{D}$ then $\forall B_m \in \mathcal{B}_{o_i} : B_m \subseteq D_k$ and $\forall B_n \in \mathcal{B}_{o_j} : B_n \subseteq D_l$ (Theorem 4). Because $\widetilde{\text{coupled}}$ is an equivalence relation D_k and D_l are partition classes: $D_k \neq D_l \Rightarrow D_k \cap D_l = \emptyset$. Thus $\mathcal{B}_{o_i} \cap \mathcal{B}_{o_j} = \emptyset$ and o_i and o_j can be covered independently. \square

4.4 Phase coupling

To obtain good quality code, the code generation phases may not be executed independently [Vegdahl 1982]. Therefore, some mechanisms are foreseen in CHES to couple the different code generation phases.

4.4.1 Late binding. A solution of Problem 1 defines a cover \mathcal{C} , which partitions V_O . Each partition class is a bundle $B_i \in \mathcal{C}$ which may still be implemented by several refined bundles $R_{i,j} \in \mathcal{R}_{B_i}$. Recall from Section 3 that if a function refinement_k is applied to V_O , V_O is changed in V_{O_R} , containing the refined operations. This corresponds to choosing a refinement $R_{i,j} \in \mathcal{R}_{B_i}$ for each bundle B_i . V_{O_R} is partitioned in refined bundles because of the relation coupled for the given refinement.

The *actual choice* of refined bundles is *delayed* to the register allocation phase (data routing) because for each bundle B_i different refinements $R_{i,j}$ may need the data in different storage locations. This influences the necessary data moves and hence the cycle count.

The results of code selection must thus be back-annotated to the CDFG, in a way that allows register allocation to efficiently select a refinement for each bundle. For this purpose, a preprocessing step is carried out to reduce the complexity of the data routing task. Both the CDFG and the ISG representations are taken to a *higher abstraction level*. The idea is to use the library \mathcal{L} to represent the *bundle* refinement possibilities, just as is explained for the *operation* refinement possibilities in Section 3.2. Each bundle $B_i \in \mathcal{C}$ induces a pattern in the CDFG that will be replaced by a *bundle-operation* b_i . Each of these operations b_i may then be replaced during data routing by a *refined bundle-operation* $\rho_{i,j}$.

4.4.2 Common operands. When a value is used by several operations, it is often advantageous to schedule those operations close to each other, even within the same cycle, to reduce the live-range of the value. Therefore many DSPs even provide special instructions for parallel execution of operations that share a common operand. A classical example of this is the “*indirect load with post-modification*”

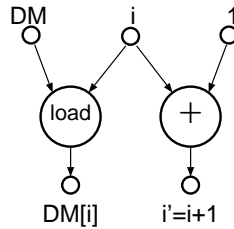


Fig. 10. DAG pattern of an indirect load with post-modification.

instruction, found in many processors. Figure 10 shows the DAG pattern that is found in the ISG for this instruction: DM represents the contents of the data memory, the value at address i must be loaded, and the address i must be modified for the next load. When both the load and the modification are performed in the same cycle, the value i must not be kept any longer; in a load with post-modification instruction, the “current” address is therefore usually overwritten with the modified address value.

To reduce the complexity for the other code generation phases, patterns of operations that consume a common operand and can be executed by one instruction are also looked up, matched, and covered by the code selection phase. This decision is then propagated to the next phases.

To find these patterns the predicate `direct` of Definition 11 is modified to also return `true` when two operations connected by a common operand may be part of the same bundle. Then the behavior of the predicates `coupled` (Definition 13), `direct` (Definition 18), and `coupled` (Definition 19) is modified accordingly, and bundles will also be formed by operations that share a common operand.

4.5 Evaluation

Although the code selection phase must solve two NP-complete problems — DAG matching (while deriving the patterns from the ISG) and DAG covering — its run times are quite acceptable, in the order of a few seconds for functions of a few hundreds of operations. This is mainly because the problem size remains small: the bundles typically contain only a few operations.

Table II contains results of performing code selection for some typical DSP functions on the Analog Devices ADSP-2111 processor [AD 1990]. The first three functions are taken from the ADPCM application [CCITT 1990]. The four last functions are borrowed from the voice coder/decoder described in the GSM standard [ETSI 1994].

Code selection has been performed for each of the functions in two cases: once without considering common operands, and once where operations with common operands are bundled when possible. The results for the latter case are put within parentheses. Remark that considering common operands leads to significantly better results for the “ST filt.” and “RPE dec.” applications, as can be observed from the columns \mathcal{C} and $\overline{|B|}_{B \in \mathcal{C}}$: these algorithms work on arrays residing in memory, so that many “load/store with post-modification” patterns are present.

Application	$ V_O $	$ \mathcal{B} $	$ \mathcal{C} $	$\overline{ B }_{B \in \mathcal{C}}$	$ B _{\max}$	CPU time
fmult	49	51 (51)	30 (30)	1.63 (1.63)	3 (3)	1.57 (1.58)
μ law2lin	28	27 (27)	17 (17)	1.65 (1.65)	3 (3)	0.85 (0.86)
lin2 μ law	28	28 (28)	18 (18)	1.56 (1.56)	3 (3)	0.87 (0.86)
ST filt.	107	107 (131)	83 (63)	1.30 (1.70)	3 (3)	1.46 (1.73)
LTP quant.	143	132 (140)	83 (77)	1.72 (1.86)	4 (4)	2.96 (3.16)
RPE dec.	57	64 (76)	40 (30)	1.42 (1.87)	4 (4)	1.31 (1.46)
Thresh. ad.	257	206 (208)	138 (136)	1.86 (1.89)	3 (3)	3.34 (3.43)

LEGEND

$ V_O $	number of functional operations in CDFG
$ \mathcal{B} $	number of bundles in \mathcal{B} (all valid bundles found for CDFG)
$ \mathcal{C} $	number of bundles in cover \mathcal{C}
$\overline{ B }_{B \in \mathcal{C}}$	average number of operations per bundle in the cover
$ B _{\max}$	maximal number of operations per bundle in the cover
CPU time	total CPU time for code selection, in seconds (on a HP-B132L workstation) (parentheses indicate results where operations with common operands are also bundled)

Table II. Code selection results for some functions from ADPCM and GSM

Because of the branch-and-bound approach, the code selection results in Table II are optimal with respect to the cost function used: in these experiments each bundle has unity cost, and covers are found with a minimal number of bundles.

Note that for the ADSP-2111 the number of operations in a bundle is rather low. For ASIPs, with more application specific data-paths, typically larger bundles are generated, containing 2 to 3 operations on the average.

Table III contains some figures about the covering subtask. The subtable on top shows that a lot of operations can be implemented by only one pattern on the ADSP-2111; this is also the case for many other general-purpose DSPs. The corresponding bundles are thus *essential* in the selected cover, reducing the number of choices to

Application	$ V_O $	$ \mathcal{B} $	$ \mathcal{C} $	#essent. B	# essent. o
fmult	49	51 (51)	30 (30)	19 (19)	23 (23)
μ law2lin	28	27 (27)	17 (17)	12 (12)	15 (15)
lin2 μ law	28	28 (28)	18 (18)	13 (13)	15 (15)
ST filt.	107	107 (131)	83 (63)	75 (31)	84 (74)
LTP quant.	143	132 (140)	83 (77)	62 (52)	83 (39)
RPE dec.	57	64 (76)	40 (30)	30 (12)	31 (13)
Thresh. ad.	257	206 (208)	138 (136)	104 (100)	170 (133)

Covering **direct** components separately:

Application	$ \mathcal{D} $	$ D _{\max}$	# nodes	CPU time
fmult	10 (10)	3 (3)	32 (32)	0.01 (0.03)
μ law2lin	5 (5)	2 (2)	16 (16)	0.02 (0.03)
lin2 μ law	5 (5)	3 (3)	16 (16)	0.01 (0.03)
ST filt.	8 (28)	2 (3)	24 (84)	0.03 (0.09)
LTP quant.	19 (23)	3 (3)	60 (74)	0.08 (0.07)
RPE dec.	9 (17)	3 (3)	34 (60)	0.05 (0.05)
Thresh. ad.	34 (36)	3 (3)	111 (117)	0.14 (0.13)

Covering without splitting in **direct** components:

Application	#BB	$ \text{BB} _{\max}$	# nodes	CPU time
fmult	5	31	27 (27)	0.13 (0.11)
μ law2lin	3	24	14 (14)	0.06 (0.07)
lin2 μ law	6	17	17 (17)	0.04 (0.04)
ST filt.	17	16	33 (77)	0.09 (0.28)
LTP quant.	28	19	69 (79)	0.16 (0.16)
RPE dec.	9	21	34 (52)	0.12 (0.20)
Thresh. ad.	60	15	137 (141)	0.22 (0.23)

LEGEND

- $|V_O|$ number of matched operations in CDFG
- $|\mathcal{B}|$ number of bundles in \mathcal{B} (all valid bundles found for CDFG)
- $|\mathcal{C}|$ number of bundles in cover \mathcal{C}
- # essent. B number of essential bundles
- # essent. o number of operations covered by essential bundles
- $|\mathcal{D}|$ number of **direct** components in \mathcal{D}
- $|D|_{\max}$ maximal number of operations in a **direct** component
- # nodes total number of search nodes actually constructed
- CPU time CPU time for covering, in seconds (on a HP-B132L workstation)
- # BB number of basic blocks in CDFG
- $|\text{BB}|_{\max}$ maximal number of operations in a basic block
- (parentheses indicate results where operations with common operands are also bundled)

Table III. Covering complexity for some functions from ADPCM and GSM

be made. The results shown in this subtable are obtained when covering **direct** components separately and also when covering basic blocks as a whole; this is in accordance with Theorem 5.

The subtable in the middle indicates the complexity of the covering subtask when **direct** components are covered separately. The size of each covering problem is limited by the maximal size of the **direct** components $|D|_{\max}$. For other functions on other processors — for example on ASIPs with more specialized data-paths — $|D|_{\max}$ may be higher, typically up to 20. The subtable at the bottom contains

figures for the case where basic blocks are not split before covering. Then the size of each covering problem is limited to the maximal size of a basic block.

When comparing both subtables, a rather surprising result is that the number of search nodes that are actually constructed is almost the same in both cases. This is because the lower bound C_{LM} (see Section 4.3) is very accurate: in the experiments presented in this section, C_{LM} almost always yields the *exact* minimal cover cost. In each covering problem only the search nodes needed to arrive at a first complete solution must then be constructed, which are the ones needed for the greedy search.

Would the lower bound be less accurate, the division of a basic block in **direct** components is more important. This is illustrated in Table IV, using C_{part} as lower bound. The same table also contains the number of search nodes in the complete search tree (for an exhaustive search), which is much larger when covering basic blocks than when covering **direct** components.

Application	exh. (BB)	C_{part} (BB)	exh. (D)	C_{part} (D)
fmult	875	149	36	32
μ law2lin	113	41	18	16
lin2 μ law	62	28	18	16
ST filt.	719	229	92	92
LTP quant.	186	101	88	78
RPE dec.	542	161	82	72
Thresh. ad.	268	167	131	117

LEGEND

- exh. (BB) # search nodes in exhaustive search, covering basic blocks
- C_{part} (BB) # search nodes with lower bound C_{part} , covering basic blocks
- exh. (D) # search nodes in exhaustive search, covering **direct** components
- C_{part} (D) # search nodes with lower bound C_{part} , covering **direct** components
(operations with common operands are bundled)

Table IV. Number of search nodes for exhaustive search and for bad lower bound; comparison between splitting in **direct** components or not.

The main advantage of covering the subject DAG as a whole, instead of splitting it into subject trees and cover it with tree patterns by tree parsing [Wilhelm and Maurer 1995; Fraser, Hanson, and Proebsting 1993], is that parts of shared subexpressions may be duplicated when this decreases the cost (Figure 9). This duplication is decided upon during the covering sub-phase, on a case-by-case basis. Another advantage of our approach is that the patterns may be DAGs, which is sometimes needed to be able to accurately model the complex instructions executed by embedded processors.

Liao, Devadas, Keutzer, and Tjiang [1995] also describe a code selection approach that allows to cover a subject DAG by DAG patterns. However, in his approach the template patterns are not derived from a processor model, but must be given separately. Also, they formulate the DAG covering problem as a binate covering problem to be solved by a standard package, which only applies to additive cost functions. By explicitly executing the branch-and-bound search by an algorithm that knows about DAG covering, it is also possible to favor solutions in a subtle way; for example to choose within the solutions with minimal cost the one which

also minimizes overlap between bundles. It also allows to use non-additive cost functions in the future.

5. CONCLUSION

The instruction-set graph (ISG) models a target processor for the retargetable compiler `CHES` and the instruction-set simulator generator `CHECKERS` [Van Praet, Lanneer, Geurts, and Goossens 1999; Van Praet, Lanneer, Geurts, and Goossens 1998].

The ISG replaces the pattern base and additional processor views needed in other compilers. Instead, it explicitly models the processor connectivity and it captures the processor parallelism so that both encoding conflicts and hardware conflicts are easily checked for. This makes the ISG well suited as a model to perform register allocation for processors with heterogeneous register structures.

In the back-end of the `CHES` compiler, the consecutive code generation phases use the ISG to incrementally define patterns of conflict-free operations that eventually form complete instructions. Thereby, some binding decisions are delayed from one phase to another.

Architectural peculiarities of DSPs, such as for example residually controlled operations, conditionally executable operations and other special control-flow instructions, also directly fit in the ISG model. Different data-types may be associated with one storage element, in order to allow `CHES` to insert reversible data-type conversions in the code, while guaranteeing bit-trueness [Van Praet, Lanneer, Geurts, Goossens, and De Man 1996].

During the matching sub-phase of code selection, called bundling, DAG patterns are incrementally constructed while they are matched to the subject DAG. The validity of the DAG patterns is determined using the ISG processor model.

The covering sub-phase of code selection has been implemented by a customized branch-and-bound algorithm. Because DAG covering is NP hard, it is shown that the subject DAG may be split into so-called `direct` components that are covered independently. This still yields an optimal cover for the complete graph in the case of additive cost functions.

ACKNOWLEDGMENTS

Part of the research described in this paper was performed by the authors when they were with the micro-electronics center IMEC (Leuven, Belgium) of which Target Compiler Technologies is a spin-off company.

References

- AD. 1990. *ADSP-2111, User's Manual*. Analog Devices.
- AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers : Principles, Techniques, and Tools*. Addison-Wesley Publishing Company.
- CCITT. 1990. *General Aspects of Digital Transmission Systems; Terminal Equipments*. Recommendation G.726 ed. The International Telegraph and Telephone Consultative Committee.
- COUDERT, O. 1996. On solving covering problems. In *Proceedings of the 33rd Design Automation Conference* (June 1996). 197–202.
- COUDERT, O. AND MADRE, J.-C. 1995. New ideas for solving covering problems. In *Proceedings of the 32nd Design Automation Conference* (June 1995). 641–646.

- ETSI. 1994. *European Digital Cellular Telecommunications System (Phase 2); Full rate speech transcoding*. ETSI spec. GSM 06.10, GSM 06.32 ed. The International Telegraph and Telephone Consultative Committee.
- FAUTH, A., VAN PRAET, J., AND FREERICKS, M. 1995. Describing instruction set processors using nml. In *Proceedings of the European Design and Test Conference (EDTC) (1995)*. 503–507.
- FERDINAND, C., SEIDL, H., AND WILHELM, R. 1994. Tree automata for code selection. *Acta Informatica* 31, 8, 741–760.
- FRASER, C. W. AND HANSON, D. R. 1995. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley.
- FRASER, C. W., HANSON, D. R., AND PROEBSTING, T. A. 1993. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems* 1, 3 (Sept.), 213–226.
- GAREY, M. R. AND JOHNSON, D. S. 1979. *Computers and Intractability: A guide to the Theory of NP-completeness*. Freeman and Co.
- GOOSSENS, G., VAN PRAET, J., LANNEER, D., GEURTS, W., AND THOEN, F. 1996. Programmable chips in consumer electronics and telecommunications. In *Hardware/Software Co-Design*, G. De Micheli and M. Sami, Eds., *NATO ASI Series E: Applied Sciences* vol. 310, 135–164. Kluwer Academic Publishers.
- LANDSKOV, D., DAVIDSON, S., SHRIVER, B., AND MALLETT, P. 1980. Local microcode compaction techniques. *ACM Computing Surveys* 12, 3 (Sept.), 261–294.
- LANNEER, D. 1993. *Design Models and Data-Path Mapping for Signal Processing Architectures*. Ph. D. thesis, K.U.Leuven (IMEC); Leuven, Belgium.
- LIAO, S., DEVADAS, S., KEUTZER, K., AND TJANG, S. 1995. Instruction selection using binate covering for code size optimization. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD) (Nov. 1995)*. 393–399.
- PAPADIMITRIOU, C. H. AND STEIGLITZ, K. 1982. *Combinatorial Optimization — Algorithms and Complexity*. Prentice-Hall.
- PAULIN, P. G., LIEM, C., CORNERO, M., NAÇABAL, F., AND GOOSSENS, G. 1997. Embedded software in real-time signal processing systems: Application and architecture trends. *Proceedings of the IEEE* 85, 3 (Mar.), 419–435.
- RUDELL, R. L. 1989. *Logic Synthesis for VLSI Design*. Ph. D. thesis, University of California, Berkeley. Memorandum No. UCB/ERL M89/49.
- VAN PRAET, J. 1997. *Processor Modelling and Code Generation Techniques for Retargetable Compilation*. Ph. D. thesis, K.U. Leuven (IMEC), Leuven, Belgium.
- VAN PRAET, J., LANNEER, D., GEURTS, W., AND GOOSSENS, G. 1998. Method of generating code for programmable processor, code generator and application thereof. European Patent and U.S. Patent No. 5,854,929.
- VAN PRAET, J., LANNEER, D., GEURTS, W., AND GOOSSENS, G. 1999. A method for processor modeling in code generation and instruction set simulation. U.S. Patent No. 5,918,035.
- VAN PRAET, J., LANNEER, D., GEURTS, W., GOOSSENS, G., AND DE MAN, H. 1996. Modelling hardware-specific data-types for simulation and compilation in hw/sw co-design. In *Proceedings of the Workshop on Synthesis and System Integration of Mixed Technologies (SASIMI) (Fukuoka, Japan, Nov. 1996)*. 255–262.
- VEGDAHL, S. R. 1982. Phase coupling and constant generation in an optimizing microcode compiler. *Proceedings of 15th Annual Workshop on Microprogramming (Micro-15), Sigmicro Newsletter* 13, 125–133.
- WILHELM, R. AND MAURER, D. 1995. *Compiler Design*. International Computer Science Series. Addison-Wesley Publishing Company.
- ZIVOJNOVIĆ, V., VELARDE, J. M., CHRISTIAN, S., AND MEYR, H. 1994. DSPSTONE: A DSP-oriented benchmarking methodology. In *Proceedings of the International Conference on Signal Processing Applications and Technology (ICSPAT) (Dallas, Texas, Oct. 1994)*. 715–720.