Instruction Selection and Scheduling for DSP Kernels on Custom Architectures

Arslan, Mehmet Ali; Kuchcinski, Krzysztof

Published in:
16TH EUROMICRO CONFERENCE ON DIGITAL SYSTEM DESIGN (DSD 2013)

DOI:
10.1109/DSD.2013.91

2013

Link to publication

Citation for published version (APA):
Arslan, M. A., & Kuchcinski, K. (2013). Instruction Selection and Scheduling for DSP Kernels on Custom Architectures. In 16TH EUROMICRO CONFERENCE ON DIGITAL SYSTEM DESIGN (DSD 2013) (pp. 821-828). IEEE--Institute of Electrical and Electronics Engineers Inc.. DOI: 10.1109/DSD.2013.91

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
Instruction Selection and Scheduling for DSP Kernels on Custom Architectures

Mehmet Ali Arslan and Krzysztof Kuchcinski
Dept. of Computer Science, Lund University, 221 00 Lund, Sweden
{mehmet.ali.arslan, krzysztof.kuchcinski}@cs.lth.se

Abstract—As custom architectures become more and more common for DSP applications, instruction selection and scheduling for such applications and architectures become important topics. In this paper, we explore the effects of defining the problem of finding an optimal instruction selection and scheduling as a constraint satisfaction problem (CSP). We incorporate methods based on sub-graph isomorphism and global constraints designed for scheduling. We experiment using several media applications on a custom architecture, a generic VLIW architecture and a RISC architecture, all three with several cores. Our results show that defining the problem with constraints gives flexibility in modeling, while state-of-the-art constraint solvers enable optimal solutions for large problems, hinting a new method for code generation.

I. INTRODUCTION

New demands on performance and power consumption lead to introduction of special execution platforms that are built specially to fulfill the specific requirements of a given set of applications. The platforms are usually built around a particular network that connects processors and application-specific instruction-set processors (ASIPs). ASIPs have specific instruction sets that support a given class of applications. They also have different architectural solutions for parallel execution. Among others they usually support instructions defining complex functionalities, VLIW-like instructions and SIMD execution model. For example, ePUMA architecture contains a Sleipnir processor [1] that has special instructions that can compute DSP kernel applications, such as DCT or IDCT, using two processor instructions. Moreover, the processor can make use of data parallelism present in an application and execute an instruction in the SIMD mode. Other DSP processors endorse the possibility to execute several operations, such as addition, multiplication and store/load, in long instructions following the VLIW principle.

To use all these features effectively is very difficult, and efficient code generation for these architectures using traditional methods is not simple. A typical solution is to combine standard libraries for kernel functions with C code to make it efficient. This approach makes the code not portable and difficult to maintain when new features or platforms are to be used.

In this paper, we address instruction selection and instruction scheduling for such architectures, while incorporating their custom nature in the process as much as possible. We combine selection and scheduling in a single constraint programming (CP) model, providing the opportunity to achieve high quality solutions that are often optimal for many DSP kernels, as indicated by our experimental results. Moreover, our approach makes it possible to easily define new constraints for new architectures and define related models for instruction selection and scheduling. It also makes it possible to combine different constraints and, for example use complex instructions together with their parallel execution, achieving in this way the SIMD execution principle.

We assume that kernels written in some high-level language, such as C or CAL, are compiled to data-flow graphs (DFGs) [2]. We only consider kernels that can be statically scheduled and do not have feedback edges. Such a DFG graph and an instruction set, defined also in DFG format, are input to our instruction selection and scheduling system. Sub-graph isomorphism is then applied to find out possible use of instructions and then instruction selection and scheduling is performed. Instruction selection and scheduling use methods offered by CP to find partially ordered instructions that implement a given kernel, resulting in the shortest schedule. Our scheduling is not limited to sequential execution and can be used both for VLIW-like processors, SIMD executions and parallel execution on several processors.

In the next section, we introduce constraint programming (CP) that is used in this paper. In section III we discuss different approaches to instruction selection and scheduling and compare them with our approach. Section IV presents our formulation for the problem and the solution method. In section V we present experimental data obtained for different DSP kernels. Finally, in section VI we present conclusions for our work and discuss future development of the current approach.

II. CONSTRAINT PROGRAMMING

In this paper we extensively use constraint satisfaction methods implemented in the constraint programming environment JaCoP [4]. We introduce constraint programming and related constraints used in this work briefly in this section.

A constraint satisfaction problem is defined as a 3-tuple \( \mathcal{I} = (\mathcal{V}, \mathcal{D}, \mathcal{C}) \) where \( \mathcal{V} = \{x_1, x_2, \ldots, x_n\} \) is a set of variables, \( \mathcal{D} = \{D_1, D_2, \ldots, D_n\} \) is a set of finite domains (FD), and \( \mathcal{C} \) is a set of constraints. Finite domain variables (FDV) are defined by their domains, i.e. the values that are possible for them. A finite domain is usually expressed using integers,
for example $x : 1..7$. A constraint $c(x_1, x_2, \ldots, x_n) \in \mathcal{C}$ among variables of $\mathcal{V}$ is a subset of $D_1 \times D_2 \times \ldots \times D_n$ that restricts which combinations of values the variables can simultaneously take. Equations, inequalities and even programs can define a constraint.

A global constraint on the other hand, is a conjunction of several simpler constraints. While semantically redundant, these constraints let the solver exploit the structure of the problem by providing a broader view to it \cite{5}. In this paper we use intensively several global constraints, such as Cumulative, Diff2 and SubGraphMatch.

Cumulative constraint \cite{6} was originally introduced to specify the requirements on task scheduling on a number of resources. It expresses the fact that at any time the total use of these resources for the tasks does not exceed a given limit. It has four parameters: a list of tasks’ starts, a list of tasks’ durations, a list of amount of resources required by each task, and the upper limit of the amount of used resources. All parameters can be either domain variables or integers.

The Diff2 \cite{7} constraint is designed to model non-overlapping rectangles. It takes as an argument a list of 2-dimensional rectangles and assures that for each pair of $i, j$ ($i \neq j$) of 2-dimensional rectangles, there exist at least one dimension $k$ where $i$ is after $j$ or $j$ is after $i$. The 2-dimensional rectangle is defined by a tuple $[O_1, O_2, L_1, L_2]$, where $O_1$ and $L_2$ are respectively called the origin and the length of the 2-dimensional rectangle in $i$-th dimension. The Diff2 constraint is used in this paper for defining constraints both for resource binding and scheduling.

Finally, the graph matching constraint (SubGraphMatch) \cite{8} defines conditions for (sub-)graph isomorphism between target and pattern graphs (the pattern graph can be defined as a set of separate sub-graphs). It has been implemented using a pruning algorithm developed for applications in reconfigurable computing and instruction selection.

A solution to a CSP is an assignment of a value from variable’s domain to every variable, in such a way that all constraints are satisfied. The specific problem to be modeled will determine whether we need just one solution, all solutions or an optimal solution given some cost function defined in terms of the variables.

The solver is built using constraints’ own consistency methods and systematic search procedures. Consistency methods try to remove inconsistent values from the domains in order to reach a set of pruned domains such that their combinations are valid solutions. Each time a value is removed from a FD, all the constraints that contain that variable are revised. Most consistency techniques are not complete and the solver needs to explore the remaining domains for a solution using search, which consists of systematically assigning values from variable domains to the variables.

### III. RELATED WORK

Instruction selection and scheduling for a given processor or multi-processor are complex problems known to be NP-complete. Special attention has been recently given to special architectures that have complex instructions and non-regular instruction sets as well as possible reconfigurability of the processor under run-time. This makes it difficult to use well known compiler infrastructures, such as LLVM \cite{9}.

There are methods used for solving these problems optimally. Mixed integer programming (MIP), constraint programming (CP) or dynamic programming are common methods for mixed constrained version of these problems.

Bednarski \cite{10} explores optimal or highly optimized code generation techniques for in-order issue superscalar processors and various VLIW processors, using dynamic programming and ILP. The dynamic programming method generates all possible solutions and searches for the optimal while shrinking the search space via pruning and compression techniques. Bednarski’s work continues with investigating ILP formulation of the optimal code generation problem, again for VLIW architectures.

The constraint programming (CP) approach, developed during last years for different purposes, is one of the methods used for solving these problems optimally. Beek and Wilken \cite{11} use CP to optimally schedule basic block instructions on single-issue RISC processors. They address arbitrary latencies for instructions. This work is particularly relevant to tour work since they also use the MediaBench \cite{12} benchmark applications. An important difference though is that we consider several RISC processing units running in parallel, while they schedule the basic blocks only on a single processing unit.

The work in \cite{13} use CP based register allocation and scheduling. They use LLVM for the compiler front-end and assume that instruction selection has already been done yielding a representation of the input program in SSA (static single assignment). They perform register allocation by using Diff2 constraints and rectangles defining define-use times for variables. On top of register allocation, they present a decomposition-based code generation technique where they locally schedule the instructions in each basic block and optimize execute cycles based on an estimate block execution frequency. The approach presented here concentrates on instruction selection and scheduling but can be combined with the methods proposed in \cite{13} and similar methods proposed for operator based architectures in \cite{14}.

Our previous work used CP for instruction identification, selection and scheduling for reconfigurable processor extensions \cite{8}. It uses sub-graph isomorphism for instruction identification and selection as well as CP-based scheduling. In the current work we assume an instruction set but we extend our previous approach by addressing new architectures. In particular, we consider very complex instructions, VLIW processors as well as multicore RISC’s. Previously only RISC processors with an accelerator \cite{5} or operator based reconfigurable architectures \cite{14} were considered.

### IV. OUR APPROACH

#### A. Inputs and Assumptions

The input to our system is an application that is compiled to a DFG together with the DFGs of available instructions.
We consider three architectures with different granularity for instructions: Sleipnir, VLIW and RISC.

The sleipnir architecture [1] can run special instructions for computing DSP kernel applications. One such instruction that comprises 24 operations is depicted in Fig. 1. In this work we consider Loeffler’s algorithm for IDCT [15] on Sleipnir, which takes two instructions to complete IDCT with 8 words and 8 constants as input. We also handle several Sleipnir processing units running in parallel.

For representing VLIW architectures, we use a generic model where an instruction comprises a set of basic operations: one add or subtraction; one multiplication and one load or store operation. For RISC architectures, our model consists of several cores running in parallel, where each core can run one single operation (add, subtract, multiply, load, store) at a time.

We will use the application and instruction graphs depicted in Fig. 2 as a running example in the rest of this paper. The figure depicts an application with two multiplications and four additions. Each operation is represented by a node with an identification label, comprising the type of the operation and an identity number (referred to as node id in the following). The instructions are also depicted in the same format, although without the identification labels. Instruction two and three are one-operation instructions while instruction one comprises a multiplication followed by an addition.

In this simple example, we assume that there is only one processing unit that can run all three instructions, but only one at a time. Each instruction is assumed to take one cycle to execute (duration $\Delta t = 1$).

### B. Instruction Matching

Each instruction can be present in several sub-graphs of the application graph (i.e. sub-graph isomorphic to the application graph) as shown in figures 3 and 4. We call each of these sub-graphs a *match*. The problem is to select the matches that give a full cover and shortest schedule. Since each match needs an identity (so that they can be referred to later on), we number them as illustrated in the figures 3 and 4. To find all possible matches of the instructions on the application graph, we record all different matches of the instruction graphs on the application graph. A match is identified if it is sub-graph isomorphic to the application graph. In our case, a graph $h$ is sub-graph isomorphic to a target graph $g$ if each node and edge in $h$ matches a node and edge in a sub-graph of $g$. We introduce this as a set of constraint satisfaction problems (CSP) and find all matches for each instruction. For each instruction $h$, we record its matches in the set $matches_h$.

**Input:** Application graph $g$, set of instruction graphs $I$

**for all** $h \in I$ **do**

$matches_h = subGraph\_CSP(g,h)$

**end for**

In the above, $subGraph\_CSP(g,h)$ refers to the method that generates a CSP for the sub-graph isomorphism problem for application graph $g$ and instruction graph $h$, and returns all possible solutions to it. Each node in the instruction graph gets a finite domain variable (FDV) that is assigned to the id of the node in the application graph it matches to, i.e. when an isomorphic sub-graph is found in $g$ for $h$.

### C. Instruction Selection and Scheduling

After finding all possible matches, we can continue with selecting the matches that will actually be used and schedule them to get the shortest schedule. Match selection and scheduling are not independent from each other. Some match selections may lead to shorter schedules, while shorter schedules may enforce a particular match selection. Therefore, in a CSP context, solving these two dependent problems simultaneously is likely to result in a better pruning in both match selection and scheduling domains, hence a smaller search space.

Target platforms with different properties (e.g. number of cores, instructions available) require a tailored constraint model, while there is a set of constraints that are generic for instruction selection and scheduling problem, regardless of the target platform. We first describe the generics, then continue with problem specific ones. In the following mathematical notations, $i$ denotes the instruction index and $m$ denotes the
match index for the respective instruction, and they quantify over all instructions and their matches.

1) Generic Constraints: While selecting instruction matches, no node in the application graph should be left uncovered. To keep track of which node is covered by which match, we use a vector of FDVs named nodeMatch, where nodeMatch[n] refers to the match number that node with id n from the application graph is covered by. The initial state of nodeMatch for our running example is depicted in Fig. 5. Note that the domain for these variables are same as the domain of match ids. Therefore any solution to the CSP with the nodeMatch variables assigned to one value, means that the graph is fully covered.

The start times of the nodes within a selected instruction match should be equal since they are to be run as one instruction. In constraint (1) (and the following), matches denotes a three dimensional vector that keeps the relation between the nodes of the matches of the instructions, and the nodes of the application graph. As mentioned before, an instruction graph can have multiple matches on different subgraphs of the application graph. With this information in mind, matches_{i,m,p} points to the id of the application graph node, that node p of match m of instruction i matches to. To illustrate this, matches_1 = [[4, 6], [3, 6], [3, 5]] for our example, which corresponds to the matches of the first instruction (see Fig. 6).

The matrix named sel on the other hand, keeps boolean domain variables for each instruction match, denoting whether or not they are selected to cover the application graph (see constraints (5) and (6) for more details). Together with matches, sel lets us reason about entire matches rather than particular nodes. Using these two vectors, constraint (1) states that if two nodes can be covered by the same instruction match and if the respective match is selected, then their start times will be equal.

\[ \forall p,q \in \text{nodes} \mid p \neq q, p,q \in \text{matches}_{i,m} : 
\quad \text{sel}_{i,m} \Rightarrow \text{start}_p = \text{start}_q \] (1)

Constraint (2) sets the durations of nodes that have output edges to other instructions (namely, the output nodes) to \( \Delta t \) (the duration of the respective instruction), forcing the destination of the output edge to wait until this instruction is finished (together with constraints (3, 4)). Duration of a non-output node is set to 0 since it actually does not cause any wait when scheduling. Constraints (5, 6) on the other hand, impose the precedence constraints caused by any edge \((p,q)\) from node \(p\) to node \(q\) in the application graph. Nodes that are not in the same match are handled by constraint (5) while constraint (6) handles the nodes in the same match. Note that constraint (6) ignores precedence between the nodes in a selected match.

To understand why, consider Fig. 6 where instruction_1 has a multiplication with two outgoing edges: one to the subtraction in the same instruction, another one as an output. The only possible matching for this example is depicted in Fig. 7 therefore to reach a full cover, these matches have to be selected (i.e. sel_1,1 = sel_2,1 = 1). If each instruction has duration \(\Delta t\), the shortest schedule on a processing unit that runs one instruction at a time will be 2 \(\Delta t\). By constraint (2), having an outgoing edge outside the instruction, both duration_1 and duration_3 will be \(\Delta t\). If the precedence between nodes in a selected match is not neglected as in constraint (5), node_2 will have to wait for node_1 to finish, adding an additional \(\Delta t\) duration to the whole match_1. Since node_2 is dependent on the output of node_3, this extra duration will result in a schedule with length 3 \(\Delta t\) instead of 2 \(\Delta t\). By means of constraint (6), we ignore the dependency between node_1 and node_3 and eliminate this problem.

Fig. 6: Example with an instruction that has an output node that has an outgoing edge to another node in the instruction.

Fig. 7: Only possible matching for the example in fig. 6.

\[ \forall p \in \text{nodes} \mid p \in \text{matches}_{i,m} : 
\quad (\text{sel}_{i,m} \land p \in \text{outputs}_{i,m}) \Rightarrow \text{duration}_p = \Delta t \] 
\[ \land 
\quad (\text{sel}_{i,m} \land p \notin \text{outputs}_{i,m}) \Rightarrow \text{duration}_p = 0 \] (2)

\[ \forall (p, q) \in \text{edges} \mid p, q \in \text{matches}_{i,m} : 
\quad \neg \text{sel}_{i,m} \Rightarrow \text{start}_p + \text{duration}_p \leq \text{start}_q \] (3)

\[ \forall (p, q) \in \text{edges} \mid p, q \notin \text{matches}_{i,m} : 
\quad \text{start}_p + \text{duration}_p \leq \text{start}_q \] (4)

nodeMatch and sel variables are logically bound to each other since nodeMatch_i denotes which match covers node n in the application graph, while sel_i,m denotes if the match indexed as i,m is selected or not. So, logically, if a match k is selected, all the nodes that can be covered by this match should
have their `nodeMatch` variable set to `k` (denoted in constraint (5)). Similarly, if `k` is not selected, none of the nodes that can be covered by this match can have `k` as their `nodeMatch` variable (denoted in constraint (6)). In mathematical notation, constraint (5) and constraint (6) describes this logic. Note that `nodeMatch` keeps match numbers for practical reasons, therefore we need a flattened intermediate representation of the `matches` matrix, which is denoted as `∀k match_k = matches_{i,m}`. Both constraints are implemented using the global `Count` constraint and reification to reach an effective implementation.

\[
\forall k \mid \text{match}_k = \text{matches}_{i,m} : \\
\bigwedge_{p(\text{match}_k)} \text{nodeMatch}_p = k \iff \text{sel}_{i,m} \\
\bigwedge_{p(\text{match}_k)} \text{nodeMatch}_p \neq k \iff \neg\text{sel}_{i,m}
\]  

A valid schedule does not exceed the resource limit at any time. In our case, the resource limit is the number of processing units available in the given architecture. For this purpose CP offers a well-studied global constraint named `Cumulative`, that is used commonly for task scheduling problems [6] (see II).

Matches in our problem are task candidates, i.e. if they are selected, they are to be scheduled with respect to previously mentioned constraints, plus the cumulative constraint. The start times of the nodes in the same selected match are bound together by constraint (7). Thus, scheduling one representative node for each selected match will also mean scheduling the whole application, given that the application is covered by the selected matches, (5) and (6). Therefore, we pick the first node (any node would do) in every `match_m` as the representative and add its start time to the list of task starts (`mStarts` in constraint (7)) for the cumulative constraint. To consider only the selected matches in scheduling, we multiply `sel_{i,m}` with the duration of the instruction (that is `\Delta t` in this paper) that `match_m` belongs to, and save this as the duration for `match_m` (`mDurations` in constraint (7)). Non-selected matches, which will eventually get zero duration, will have no effect on the schedule. Since every instruction is run on only one processing unit, and processing units run one instruction at a time, resource need for each match is 1 (a vector of `ones` in constraint (7)). Finally, resource limit that is not to be exceeded is number of processing units available (`nProcs` [7]).

\[
\text{Cumulative}(m\text{Starts}, m\text{Durations}, \text{ones}, n\text{Procs})
\]  

2) Architecture/Problem Specific Constraints: Different architectures have different properties. In our problem definition, they may have different number of processing units running in parallel; and the set of possible instructions and the interaction between those instructions can be different. The number of processing units is simply parameterized. Properties involving instruction sets on the other hand, pose a harder problem that requires more attention.

Sleipnir architecture, for example, has two instructions, named `idct8pbw` and `idct8pbw`, that combined together, implement IDCT with 8 inputs. As seen in figures [1] and [8], both instructions have two disjoint connected components. When matching such instructions on big application graphs, e.g. comprising 8 parallel 8-input IDCTs (see section [7]), disjoint connected components in the same instruction result in a combinatorial explosion of matches.

An illustrative example is depicted in figure [9]. The only instruction in the example includes two disjoint connected components (both components have only one node for simplicity). The application on the other hand, is a parallel repetition of the instruction. We use the labels to count the instances of the instruction where `n` represents the total number of instances. For `n = 2`, the matches for `instruction_1` would look like in figure [10]. `match_1` and `match_2` can be considered as side effects of the fact that `instruction_1` has disjoint connected components. If the instruction included an edge connecting the disjoint components, the total number of matches would be 2 instead of 4 (providing that the connecting edges are replicated in the application graph too). If we generalize the problem over an instruction with `p` disjoint connected components, and the application as `n` parallel replications of this instruction, the number of matches becomes `n^p`, instead of `n`, which happens when the instruction consists of one connected graph. To cope with this combinatorial explosion, we divide the disjoint components in the instruction graphs and represent them as individual instructions. This way, the total number of matches are reduced from `n^p` to `2n`. However, we still have to ensure that the instructions that we divided into two are actually merged in the schedule, i.e. instruction `i` is divided into `i_a` and `i_b`, each `i_a` has to be run at the same time with a corresponding `i_b`. Another side effect is that, the number of processing units that are modeled has to be doubled, in order to represent the half of the Sleipnir core that runs `i_a` and the other half that runs `i_b`. Otherwise these half instructions will be sequentialized, which breaks the atomicity of instructions.

The second instruction for implementing IDCT named `idct8pbw`, depicted on figure [8] comprises two disjoint con-
two parts required a modeling style that doubles the number of available processing units, in order to simulate a Sleipnir core’s capability to run the whole original instruction at once. Thus, two processing units (after doubling) actually represent one core. Based on this assumption, we change the \( nProcs \) in constraint (7) to \( nProcs \times k \), where \( k \) is the number of divisions of the instructions (in Sleipnir case \( k = 2 \)) as shown in constraint (12).

\[
\text{Cumulative}(mStarts, mDurations, ones, nProcs \times k) \tag{12}
\]

We do not explicitly define a constraint for assuring that each half instruction match has a corresponding second half, since the combination of the constraints (8), (9), (10) and (11) together with the minimization of the schedule length (see IV-E) results in a schedule where each match grouping mentioned above are run simultaneously. Note that this is only valid when the original non-divided instructions (i.e. \( \text{idct8p}fw \) and \( \text{idct8pbw} \)) can cover the application graph. In other cases additional constraints are defined.

The fact that \( \text{idct8pbw} \) is divided into two isomorphic graphs creates problems that are not addressed by the constraints mentioned above. As they are parts of originally different instructions, \( \text{idct8pbw} \) and any one half of \( \text{idct8p}fw \) (\( \text{idct8p}fw_a \) or \( \text{idct8p}fw_b \)) can not be merged together i.e. run on the same core simultaneously. A limiting constraint such as constraint (13) can not be used in this case, since it would render constraint (9) useless. This is based on the fact that \( nStarts_r \) and \( mDurations_r \) would include \( mStarts_h \) and \( mDurations_h \) already, and limit them with \( nProcs \) instead of \( nProcs \times 2 \), which eventually prohibits them to run on the same core simultaneously. To solve this, we need to involve resource allocation constraints instead of using only scheduling constraints that disregard specific resource allocation, such as \text{Cumulative}. We describe this further in section V.D.

\[
\text{Cumulative}(mStarts, mDurations, ones, nProcs) \tag{13}
\]

So far, we only considered the Sleipnir architecture for architecture/problem specific constraints. A similar modeling style is used to solve the problem for VLIW. Although this time, we focus on the groups of matches that can not be run simultaneously on the same core i.e. only one of the matches in the group can be run on a core:\{\( add, sub \),\{\( mul \), \{ld, str\}. Again, we generate respective lists for start times and durations as in the previous algorithm and limit the maximum simultaneous execution to \( nProcs \). So, for each group in the list above, we have one corresponding constraint (10) with respective lists for start times and durations generated with the given algorithm. On the other hand we change the constraint (7) to constraint (12) with \( k = 3 \) since an instruction is divided into three in our VLIW model (see section IV-A).

D. Resource Allocation

Series of \text{Cumulative} constraints guarantee only the existence of a valid schedule given the number of processing units available, but without giving any information regarding the resource allocation. The problem arises from the fact that the scheduling constraints, e.g. \text{Cumulative} constraints, do not
not constrain the resource allocation variables for each match, which we denote as \( m_{\text{Resources}} \). These variables represent the resource that the selected match is assigned to.

To eliminate any invalid resource bindings, we use the \( \text{Diff2} \) global constraint which is described in section II as [49x678]Diff2 [49x702]resource that the selected match is assigned to. which we denote as [49x654]when the match is not selected, as discussed when describing [49x726]not constrain the resource allocation variables for each match

**Diff2** ([MINTAB][49x678],[MINTAB][49x702],m_{\text{Resources}},m_{\text{Durations}},\text{ones})

(14)

Individual matches are assigned to the processing unit that is specific for the instruction of the respective match, whenever possible. For example, considering VLIW, we model each core as divided into three processing units (pu), where pu1 runs add/sub matches, pu2 runs mul matches and finally pu3 runs ld/st matches. Resource variable for an add match on the other hand can get a value from \( \{1,k+1,2k+1,\ldots,k(P-1)+1\} \) where \( k \) is the number of divisions of a core and \( P \) is the total number of cores available. The constraint embodying this idea is given below. Similar constraints for the other two instructions are used for the VLIW architecture.

\[
m \in \text{matches}_{\text{add/sub}} : \\
\text{resource}_m \mod k = 1
\]  

(15)

In case of Sleipnir, matches of idct8fpf\( w_a \) are allocated to the first halves of the cores (\( \text{resource}_a \in \{1,3,5,\ldots,2(P-1)+1\} \)) and idct8fpf\( w_b \) (\( \text{resource}_b \in \{2,4,\ldots,2(P)\} \)) matches to the second. However a valid schedule with a valid resource allocation is still not guaranteed because the halves of idct8fpbw are isomorphic, as discussed in the previous section. It is not possible to allocate these halves to a particular half of the core either, since they can actually be run on both. Instead, for each match of idct8fpf\( w_a \) and idct8fpf\( w_b \), we prohibit having a match of idct8fpbw\( h \) that is selected and starts at the time at the same core (see constraint (16)). Note that since \( \text{matches}_{\text{idct8fpf}w_a} \) are assigned to the first half of the core and \( \text{matches}_{\text{idct8fpf}w_b} \) are assigned to the second half; we prohibit \( \text{resource}_a = \text{resource}_b - 1 \) for \( \text{matches}_{\text{idct8fpf}w_a} \) and \( \text{resource}_b = \text{resource}_a + 1 \) for \( \text{matches}_{\text{idct8fpf}w_b} \). Allocation to the same processing unit at the same time is already prohibited by constraint (14).

\[
\exists h \in \text{matches}_b : a \in \text{matches}_{\text{idct8fpf}w_a} : \\
\text{sel}_b = \text{sel}_a \wedge \text{start}_b = \text{start}_a \wedge \text{resource}_b - 1 = \text{resource}_a
\]  

(16)

E. Search Space Heuristics

The problem definition includes finding a minimal schedule makespan. The completion of a node is defined as start\( n \) + duration\( n \) and minimizing the maximum of this completion for all nodes gives the minimum schedule.

As most consistency techniques are not complete, the constraint solver needs to search for possible solutions, commonly by picking a variable that has not been assigned to a value yet, and setting it to a remaining value in its domain. As long as the constraints are correct, any variable selection method or heuristic leads to a valid solution, eventually. However, depending on the problem size, the search space may grow exponentially and lead to very long search times. In order to decrease the search time, we need to devise a variable selection and search strategy.

Instead of putting all the variables in the same pot, we make a sequential series of variables that we select from, to direct the search. The vector nodeMatch\( matches \), that comprises one representative nodeMatch variable for each match, is the first group of variables. Therefore, the first decision is made by selecting matches. After selecting which matches to use to cover the application graph, we assign values to representative start time variables for each match. And finally, we assign values to the resource allocation variables. Note that by using only the representative variables for each match instead of variables for each node, we shrink the search space considerably.

This sequential way of selecting variables resulted in reasonable search times as seen in section V.

V. Experiments

In our experiments, we considered several application graphs on different architectures. Our main interest in the Sleipnir architecture [11] is the special instructions devised for IDCT and DCT. Therefore, we used an application graph for an implementation of Loeffler’s IDCT [15]. To experiment with different problem sizes and number of Sleipnir cores available, we consider IDCTs sizing upto 8x8, i.e. 8 parallel copies of single IDCT, and upto 4 Sleipnir cores (only for 8x8 IDCT).

For VLIW and RISC architectures, we conducted a series of experiments with some of the benchmark graphs from ExpressDFG [12] which provides dataflow graphs for benchmark applications from MediaBench [16] (among others). Single IDCT used for Sleipnir experiments is also included. To evaluate the scalability of our method, we have also scheduled these applications on several cores for both architectures.

Table I shows the results for experiments on VLIW architecture. The name of the benchmark and the characteristics of the application graph is summarized in the first column, including number of nodes (|V|), number of edges (|E|) and length of the critical path. Together with these, we show the number of FDV’s and constraints generated for the CSP model. We experiment also with the available number of cores. Not all of the experiments for VLIW resulted in a solution that is proved to be optimal. This means that a solution is found, but the solver timed-out before finishing the search for a better solution. The time-out is set as 5 minutes. For the cases that reached the time-out, we recorded the last solution, and ran the solver again with this solution as the lower bound to get the "runtime" (see the column "runtime" for the rows with "no" for the "Proved Optimal" column). Otherwise, "runtime" represents the runtime of the solver. The applications named "Cosine1" and "MESA Smooth Triangle" resulted in solutions that are not proven optimal. The main reason for this is the fact that both application graphs consist of disjoint connected components. Since there is no precedence
between these components, they incur symmetrical solutions, leading to an expanded search space. The case for IDCT is more peculiar. The reason behind the solver’s ability to prove optimality for 2 cores but not 4 cores is subject to further investigation.

Table II shows the results for RISC architecture, while Sleipnir architecture with IDCT has a dedicated table, where we present the results of the experiments with graphs of different sizes and with different numbers of available cores (only for 8x8 IDCT). For both tables all the results were proven optimal.

We picked benchmarks with varying number of nodes and their ratio to critical path, together with different graph characteristics such as number of connected components. These variances are crude measures for the robustness of the model. They may also help identify the characteristics that hinder the solver performance, as in VLIW case.

Our results show that, overall, the approach is one worth investigating further. For VLIW and RISC, maximum runtime for reaching a solution and (if possible) proving it optimal is around two seconds, while the 8x8 IDCT (i.e. the largest graph in our experiments) takes around 7 seconds for Sleipnir.

Number of nodes are between 34 and 197 for the applications for RISC and VLIW, and between 96 and 768 for Sleipnir.

VI. CONCLUSIONS AND FUTURE WORK

We presented a CSP version of the instruction selection and optimal scheduling problem. Our results show that it is a technique worth investigating further. The flexibility in CP lets plugging architecture or problem specific constraints in and out of the model very easily, letting us use the same core model for three essentially different architectures. Identifying characteristics for application and instruction graphs that hinder proving optimality of a solution is an open question we plan to investigate further. As a continuation, we plan to expand our method to include memory and register allocation This will let us generate code and possibly integrate our method in a larger system.

Acknowledgments This work has been supported by the Swedish Foundation for Strategic Research (SSF) as part of the High Performance Embedded Computing project (HiPEC).

REFERENCES


