Binding and Scheduling Algorithms for Highly Retargetable Compilation

Masayuki Yamaguchi†‡ ishiura@ise.eng.osaka-u.ac.jp
† Precision Technology Development Center Sharp Corporation Tenri, Nara, 632 Japan Tel: +81-743-65-2531 Fax: +81-743-65-4980
‡ Dept. Information Systems Engineering Osaka University Suita, Osaka, 565 Japan Tel: +81-6-879-7808 Fax: +81-6-875-5902

Abstract—This paper presents new binding and scheduling algorithms for a retargetable compiler which can deal with diverse architectures. Application specific embedded processors often includes a “non-orthogonal” datapath where all the registers are not equally accessible from all the functional units. Non-orthogonal datapath makes a binding task very hard because inadvertent assignment of an operation to a functional unit may rule out all the possible assignments to other operations due to reachability constraints among datapath resources. Scheduling must take register capacity constraints into account in addition to resource constraints. We discuss these problems and propose algorithms to solve them.

I. INTRODUCTION

A retargetable compiler is a useful tool in the design of systems using application specific instruction processors. It enables (1) efficient software development using programming languages, (2) software development in concurrence with hardware design, and (3) easier software transportation from an architecture to newly designed or modified architectures. Motivated by these merits, a lot of researches have been carried out [Mar95, Mar97, Pra96, Lie95, Aka33, Sat94] to enhance retargetability and to improve code quality of such compilers.

Among many aspects of retargetable compilation, the objective of our research is to enhance the retargetability. Embedded processors used in commercial audio and video products often employ so radical architectures customized to their applications that conventional retargetable compilers aiming at simple extension of general purpose processors can not deal with them. Figure 1 shows an example of such an architecture, which includes the following constructs:

- Multiple functional units
  A processor may have multiple functional units. They can be activated simultaneously by horizontal microcodes or VLIW instructions.

- Non-orthogonal datapath structure
  Datapath may contain so called “heterogeneous registers.” Unlike general purpose registers, they are not equally accessible from all functional units.

- Data memories and address generation units
  Besides register files and main memories, medium scale ROMs and RAMs may be included in the datapath. Address generation units may be attached to memories to supply special address patterns efficiently.

CHESS system [Pra96] has presented an approach to a retargetable compilation of such diverse architectures. Unlike conventional methods, it maps DFGs derived from basic blocks of a source program onto a graph representing datapath or instruction set configuration by binding and scheduling just like in high-level synthesis. Binding of the DFG onto a non-orthogonal datapath is reduced to a path search problem.

However, we found that a problem of “resource unreachability” had not been necessarily solved in conventional methods.

By the resource unreachability problem, we refer to a problem where binding of operations does not always suceed due to lack of data transfer paths. For example, in Figure 1, suppose we must read RAM1 using an address obtained by adding two values stored in registers ACC0 and ACC1. The desirable solution is to bind the addition to the adder ADD connected to the address register of RAM1. However, a compiler may assign the addition to the ALU since it takes a lot of clock cycles to carry the data in ACC0 and ACC1 to ADD. In this case, the result of the addition is unreachable to the address port of RAM1, and thus compilation fails.

Absence of the path between units is not the only reason for resource unreachability. If there is a conflict on buses or MUXs in transfer paths of input or output data of an operation, for example, the compilation fails even though there exist transfer paths.

Although the problem of resource unreachability has
not been treated strictly so far, we think it is an essential
problem to be resolved for highly retargetable compila-
tion. In this paper, we propose binding and scheduling
algorithms to solve the problem. We first do binding and
then scheduling. The task of binding consists of opera-
tion binding and data transfer path binding. In the first
phase, each operation node in the DFG is assigned to a
functional unit in the datapath, so that the existence of
the paths between functional units corresponding to the
edges in the DFG are guaranteed (if they exist at all.)
We use BDD-based algorithm to solve this search problem
and enumerate the feasible combination of operation as-
ignment. In the second phase, assignment of data paths to
the edges in DFG is attempted for one of the operation
assignments obtained in the first phase.

In order to handle non-orthogonal datapaths, we also
need to solve difficult issues in the scheduling. We pro-
pose a new scheduling algorithm taking the capacity con-
straints of non-general registers as well as resource con-
nstraints into account.

We first show how we model the datapath architecture
and software in Section 2. The details of binding and
scheduling algorithms are shown in Section 3 and 4, re-
spectively. Section 5 shows experimental results.

II. MODELS AND PROBLEM FORMULATION

A. Modeling of Datapath

Datapath architecture is represented by the combina-
tion of datapath structure (DS), parallel constraints (PC),
and auxiliary information (AI).

A datapath structure DS consists of a set $R$ of hardware
resources and set $C$ of interconnections. $R$ is divided
into three disjoint subsets $R_F$, $R_R$, $R_S$. $R_F$ is a set of func-
tional units like ALUs and multipliers. For each $r \in R_F$
a set of the operations which $r$ can execute is defined.
ROMs and RAMs are also classified into this category.
$R_R$ is a set of register files which operates in synchro-

ous to the clock. For $r \in R_R$ defined is the capacity of $r$, de-
noted by $\text{capacity}(r)$, indicating the number of data $r$ can

hold. $R_S$ is a set of selectors. A bus receiving $n$ data is
regarded as an $n$-to-1 selector. Each $r$ in $R$ has an output
port and input ports. The output port and the $k$-th input
port of $r$ are denoted by $r.out$ and $r.in_k$, respectively.

Let $P_I$ be the set of all the input ports and $P_O$ be the
set of all the output ports of $R$. Interconnection $C$
is a subset of $P_O \times P_I$ where $(r_{1.out}, r_{2.in_k}) \in C$
indicates the existence of a connection from $r_{1.out}$ to $r_{2.in_k}$.
We say there is a path from output $r_{m.out}$ to input $r_{m.in_k}$
if $r_{m.in_k}$ is reachable from $r_{m.out}$. Immediate data sup-
plied from instruction words or control bits are modeled
as registers in DP.

Parallel constraints PC are constraints posed on the
datapath structure, which limit the simultaneous activi-
ties of the resources and interconnections. Let $x_r$ ($r \in R$)
and $y_c$ ($c \in C$) be Boolean variables where $x_r = 1$ ($y_c = 1$)
iff $r$ ($c$) is activated. Then the parallel constraints are ex-
pressed in the form of a logical expression $PC$ consisting
from $x_r$ and $y_c$. It declares that activities which makes
$PC = 1$ is inhibited. For example, $x_{ALU} \land x_{MULT}$ means
that ALU and MULT can not operate at the same step.

Parallel constraints are used to specify the effect of
instruction formats in an indirect way, as proposed in
the architecture evaluation system [Yam97]. In the case
where parallel constraints are not specified, we can gen-
erate horizontal codes to control the datapath. By grad-
ually adding constraints, we can specify which operations
and data transfers should be grouped to form fields of the
instruction format.

The other information necessary for compilation is
given as auxiliary information AI. It includes spill code in-
formation (which registers can be saved and how the value
transfer from/to the memory is done) and subroutine call
information (how the registers are saved and which regis-
ters work as stack pointers, etc.)

B. Model of Software

We are primarily aiming at C language for a source pro-
gramming language. The program is decomposed into basic
blocks (sequences of operations containing no branches)
each of which is represented in the form of a data flow
graph (DFG).

In order to handle multiple RAMs and ROMs, we cur-
rently do not allow pointers and dynamic array allocation.
We assume that all the array variables are declared global
and the mapping of array variables to ROMs and RAMs
are user defined or predetermined.

III. BINDING

A. Difficulties

Binding involves two problems: (1) binding of opera-
tions, and (2) binding of transfer paths. We explain these
problems using Figure 2 and 3.

(1) Binding of operations
Suppose we have to map the DFG onto the DP in Figure 2, where there is no restriction for $o_1, o_2,$ and $o_7$ and we are given a task of binding $a_4$ and $o_6$ to ALUs and $o_5$ to multipliers. If we assign $o_4$ to ALU2, for example, we can assign $o_5$ to multiplier M1, but there is no ALUs which can receive data both from ALU2 and M1. The only solution is to assign $o_4, o_5,$ and $o_6$ to ALU3, M2, and ALU2, respectively. In this way, reconvergence in DFGs and asymmetry in DPs makes the problem hard to solve.

(2) Binding of transfer paths

Suppose we have to map the DFG onto the DP in Figure 3, where $o_1, o_2, o_3, o_4,$ and $o_6$ are assigned to register file RF. Assignment of operations $o_2, o_6,$ and $o_7$ to ALU1, MULT, ALU2, respectively, guarantees the existence of paths corresponding to the edges in the DFG. Because all the transfer paths exist in the DP, we can say the operation binding is successful in the sense of (1). However, in the case we select the direct connection from ALU1 to ALU2 as a transfer path, the result of multiplication cannot be supplied to ALU2 by way of RF, because of the conflict on BUS1. If parallel constraints are posed on simultaneous behavior of ALU1 and ALU2, direct connection from ALU1 to ALU2 become infeasible.

Since we think the two problems are also too hard to solve in a single step, we decided to employ a two-phase binding strategy. In the first phase, we search possible solutions for operation binding, and in the second phase, we try path binding to consummate the binding task.

B. Binding of Operations

A problem of operation binding may be well explained using "binding space graph (BSG)" for a given DFG and a DP. The Figure 4 shows the binding graph for the DFG and the DP in Figure 2. BSG is also a directed acyclic graph. The nodes of BSG are grouped into subsets $A_1, \ldots, A_7$ where $A_i$ consists of nodes representing possible assignments of operation $o_i$. The edges in BSG indicates the existence of paths between bound operation resources. Our goal is to find a combination of assignments which provides all the necessary paths to the edges. A subgraph depicted by bold lines indicates a solution.

Formally, BSG for a DFG $(O, E)$ and a DP $(R, C)$ is a directed acyclic graph $(A, T)$. The node set $A$ comprises of $n = |O|$ disjoint subsets $A = A_1 \cup A_2 \cup \cdots \cup A_n$ where $A_i$ is associated with operation $o_i \in O$. If $\{r_{f_1}, r_{f_2}, \ldots, r_{f_m}\}$ is a set of functional resources which can execute operation $o_i$, then $A_i = \{a_{i,f_1}, a_{i,f_2}, \ldots, a_{i,f_m}\}$ where $a_{i,f_j}$ represents that $o_i$ is bound to $r_{f_j}$. Edge $(a_{p,f_j}, a_{i,f_j})$ in $T$ exists iff $(o_p, o_i) \in E$ is the $k$-th incoming edge of $o_i$ and there is a path from $r_{f_j.out}$ to $r_{f_j.in_k}$.

The operation binding problem is formulated as to find an embedding of a DFG $(O, E)$ into the BSG $(A, T)$. Namely, operation binding is a pair of mappings $\alpha = (\alpha_o, \alpha_e)$ where $\alpha_o : O \rightarrow A$ and $\alpha_e : E \rightarrow T$ satisfying $\alpha_o(o_i) \in A_i$ and $\alpha_e((o_i, o_j)) = (\alpha_o(o_i), \alpha_e(o_j))$.

We have developed a BDD-based algorithm for solving this problem. We introduce Boolean variable $x_{i,f_j}$ for each node $a_{i,k_i}$. Variable $x_{i,f_j}$ becomes 1 iff $\alpha_o(a_i) = a_{i,f_j}$. Then we can set up the following two conditions:

1. Since only one element of $A_i = \{a_{i,f_1}, a_{i,f_2}, \ldots, a_{i,f_m}\}$ is chosen as $\alpha_o(o_i)$, only one of $x_{i,f_1}, x_{i,f_2}, \ldots, x_{i,f_m}$ becomes 1. Thus we have,

$$\text{cond}_1(A_i) = \left( \sum_{j=1}^{m} x_{i,f_j} = 1 \right). \quad (1)$$

2. \begin{equation}
\text{cond}_1 = \bigwedge_{A_i \in A} \text{cond}_1(A_i). \quad (2)
\end{equation}
Here \( \Sigma \) stands for arithmetic summation.

2. In order that \( \alpha_{i,f} \) be chosen as \( \alpha_0(\alpha_i) \), all the incoming edges to \( \alpha_i \) must be mapped onto some edge \((a, \alpha_{i,f}) \in T\) where \( a \) is also chosen as \( \alpha_0(\alpha) \) for some \( \alpha \in \mathcal{O} \). Let \( K_i \) be the number of incoming edges to \( \alpha_i \) and \( T_{i,f,k} \) be the set of the incoming edges to \( \alpha_{i,f} \) that are associated with the \( k \)-th incoming edges to \( \alpha_i \). Let \( x_{p(c)} \) be the Boolean variable of the source node of edge \( e \in T \). Then the condition is formulated as:

\[
cond_2(\alpha_{i,f}) = \bigvee_{k=1}^{K_i} \bigwedge_{c \in T_{i,f,k}} x_{p(c)}. \tag{3}
\]

\[
cond_2 = \bigwedge_{a_{i,f} \in A} cond_2(\alpha_{i,f}). \tag{4}
\]

The operator \( \rightarrow \) means implication and \( x \rightarrow y \) is equivalent to \( \neg x \lor y \).

The necessary and sufficient condition for the existence of the solution is written as

\[
cond = cond_1 \land cond_2. \tag{5}
\]

The satisfiability of this condition is computed by constructing a BDD for \( cond \). The resulting BDD represents the set of all feasible operation bindings.

C. Binding of Transfer Paths

In this phase, we assign for each edge in the DFG to a path in the DP. We must consider feasibility of solution with respect to path assignment. The feasibility is easily checked by constructing “BDFG (bound DFG),” decompose it into “simultaneous components,” and check resource constraints, parallel constraints, and register constraints for each component.

BDFG is an augmentation of a DFG. Differences are:

1. Each operation node in the BDFG is labeled by the hardware resource in DP to which the node in a DFG is mapped.
2. The BDFG contains nodes corresponding to registers and selector resources that forms the path to which each edge in the DFG is mapped. Figure 5 shows some examples of BDFGs derived from the DFG and the DP in Figure 3 using the operation binding obtained in Figure 4.

Simultaneous components are connected components in BDFG that are separated by the nodes labeled by register resources. For notational convenience, a register labeled node belongs to every component it disconnects. Simultaneous components are indicated by broken lines in Figure 5.

We only have to check the following constraints for every simultaneous component to see if an binding of functional units and paths are feasible or not:

**resource constraints:** If a simultaneous component contains the same functional resources or selector resources, the binding is not feasible.

---

**register constraints:** For any register resource \( r \), if the number of writes to \( r \) exceeds the capacity of \( r \), or the number of writes/reads to \( r \) exceeds the limitation of simultaneous writes/reads of \( r \), the binding is not feasible.

**parallel constraints:** If parallel constraints are violated in a simultaneous component, the binding is not feasible.

Now our goal is to find a mapping of edges in BDFG to paths which results in a feasible binding. We employ the following straightforward procedure for this search.

1. Choose one operation binding \( \alpha_0 \) from the set of solutions obtained in the previous phase.
2. Based on \( \alpha_0 \), assign one of the shortest paths to each edge in \( E_D \).
3. Check the constraints for each simultaneous component. If OK then the feasible binding is found.
4. If NG, then mark the resources and the edges that are associated with the violation. For each marked edge, search a path in the DP not going through the marked resources, and replace the original path by it.
5. If possible replacements are exhausted, then try another operation binding.

---

IV. Scheduling

The task of scheduling is to assign each node in the BDFG to a control step so as to minimize the number of control steps under various constraints.

The task of scheduling is basically the same as that in high level synthesis and is solved by a simple list scheduling algorithm. The constraints specific to our problem are:

1. All the operations in a simultaneous component should be scheduled in the same control step.
2. The number of values stored in each register must not exceed its capacity (a register capacity constraint).

The first constraint is incorporated by extending the unit of scheduling from an operation to a simultaneous component. In order to handle the second constraints, we refine the “ready condition” so that the status of the registers to store the result of scheduled operation is considered.
Fig. 6. Example of spill code insertion.

A. Scheduling Algorithm

A simultaneous component is said to be input-ready if all the preceding operations are scheduled. A simultaneous component is said to be output-ready if

\[ \#\text{write}(r) - \#\text{free}(r) \leq \text{capacity}(r) - \text{used}(r) \]

holds for all the output registers of the component, where \#\text{write}(r) is the number of writes from the component, \#\text{free}(r) is the number of data whose life time end by the execution of the component, and \text{used}(r) is the number of data that are currently stored in \( r \).

Using the above two terms, our scheduling algorithm is written as follows.

(i) Start with \( cs = 0 \). Reset a queue.

(1) If a simultaneous component \( c \) is both input-ready and output-ready, then put \( c \) into the queue.

(2) Repeat (2.1) and (2.2) while the queue is not empty.

(2.1) Increase \( cs \), clear the marks of the resources.

(2.2) Repeat (2.1.1) and (2.1.2) while the queue is not empty.

(2.1.1) Take a component \( c \) out of the queue whose ALAP value is the largest, and check if \( c \) does not violate constraints in the \( cs \).

(2.1.2) If OK, mark \( c \) as scheduled. Also mark all the resources \( c \) uses. If unscheduled components newly become input-ready and output-ready, then put them into the queue.

If the algorithm finishes with no unscheduled component, then we have a final solution.

B. Spill Code Insertion

Unfortunately, there are cases where scheduling finishes with unscheduled components. This situation is caused by components that are input-ready but not output-ready. This means that the scheduling failed because of the deadlock of register resources due to insufficient register capacity.

In this case, spill codes are inserted. We assume AI indicates which registers are saved in which memories and how data transfers are carried out, so code insertion is done by just transforming the BDFG. Figure 6 shows an example. When the capacity of register \( R \) in Figure 6 is insufficient, for example, read/write operation of \( R \) is substituted by the sequence of writing the data into memory \( M \) and reading the data back at a later step.

Depending on the subgraphs inserted, BDFG should be rescheduled. This is done in the following way:

1. Decrease the control step by the necessary count (the number of registers in the longest paths in the spill code subgraph).
2. Mark scheduled simultaneous components as unscheduled that have larger control step numbers than the decreased one.
3. Resume scheduling. This is done by start the algorithm from the decreased control step instead of \( cs = 0 \).

The only difference in rescheduling is that the highest priority is given to the inserted subgraph.

V. EXPERIMENTAL RESULTS

We implemented a core part of retargetable compiler based on the proposed algorithms. It shares several peripheral softwares with the architecture evaluation system [Yam87]. All inputs are captured via a GUI. From these inputs, the compiler achieves binding and scheduling, so as to generate a sequence of control signals (horizontal code) for each basic block.

We compiled some programs onto the datapath structure shown in Figure 1.

The first example SUM (Figure 7 (a)) is a program which summates array data using incremental accessing of data memory. We used a loop expansion option to generate the DFG shown in Figure 7 (b). The node labels show the names of the bound resources. The additions for the address increments and displacements are bound to INC and ADD which are indirectly connected to the address port of RAM1, whereas the additions for the data summation are bound to ALU. Figure 7 (c) is the result of the compilation, where on (\( * \)) or off (\( \sim \)) status of control signals are listed. Thus, the additions are well bound to these operators according to their connectivities. The compilation of this example took 0.5 seconds on Sparc Station 20.

The second example is FFT (Figure 8), a fraction of a FFT program, to which no feasible binding exists. Since there is no path from MULT to SHIFT in the datapath, there is no way of mapping the sequence of a multiplication (\( * \)) and a shift (\( \ll \)) operation onto the datapath. The compiler proved the deficiency of the datapath in 0.7 seconds. We tried recompilation after adding a connection from MUXA to DBUS. This time, there was a solution and compilation finished in 0.8 seconds.

Table I is a summary of experiments on other programs such as ELLIP (elliptical filter), EDGE (edge detection of image data), and so on. We used the modified version of datapath in the previous experiment. "\#lines," shows the number of lines in each program. "\#DFG" and "\#BSG"
(a) C program of SUM.

```c
sum = 0; num = 10;
for (base = 0, k = 0; k < num; k++) {
    sum += RAM1[base + k];
}
RAM1[base + k] = sum;
```

(b) DFG and binding result.

```
fig(n, k) -> int n, k;
int a, b, c, d, e, f;
int coeffs[i];
coeffs[0] = 512;
coeffs[1] = n + 512;
sin = int(coeffs * k);
coefs = RAM1[coeffs + k];
a = (cos * RAM1[1]) >> 4;
b = (sin * RAM1[1]) >> 4;
c = (cos * RAM1[3]) >> 4;
d = (sin * RAM1[1]) >> 4;
e = a - b;
f = e - d;
RAM1[0] = RAM1[0] + e;
RAM1[1] = e - RAM1[1];
RAM1[3] = f - RAM1[3];
```

Fig. 8. C program of FFT.

(b) Result of Compilation.

Table I: Experimental Result of the Compilation

<table>
<thead>
<tr>
<th>Program</th>
<th>#Lines</th>
<th>#DFG</th>
<th>#BSG</th>
<th>CPU(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUM</td>
<td>22</td>
<td>43</td>
<td>74</td>
<td>0.7</td>
</tr>
<tr>
<td>FFT</td>
<td>21</td>
<td>38</td>
<td>58</td>
<td>0.5</td>
</tr>
<tr>
<td>ELLIP</td>
<td>45</td>
<td>110</td>
<td>134</td>
<td>0.5</td>
</tr>
<tr>
<td>EDGE</td>
<td>84</td>
<td>107</td>
<td>995</td>
<td>1.1</td>
</tr>
</tbody>
</table>

References


Fig. 7. Experimental Results of SUM.

shows the total number of DFG nodes and BSG nodes, respectively. In each case, we could generate a code correctly. “CPU” show the CPU time (on Sparc Station 20) for generating codes.

VI. CONCLUSION

We have presented a problem of resource unreachable in binding and scheduling for highly retargetable compilation, and have proposed new binding and scheduling algorithms to solve it.

There are a lot of problems remain unresolved. Currently, we are trying to improve the spill code insertion and considering the modeling methodology to convert an instruction set specification into a datapath structure and parallel constraints and to derive code sequence from the result of binding and scheduling.

ACKNOWLEDGEMENT

The authors would like to thank Prof. Isao Shirakawa of Osaka University for his advice on this research. We would like to thank Mr. Tetsusaburo Yamamoto, Mr. Yasuaki Hattori, Dr. Takao Ono of Osaka University and to Mr. Tetsuya Fujimoto of Sharp Corporation for their discussion and constructive comments.