Dependency Driven Partitioning Objects Generation for Hardware/software Partitioning

Shengtian Sang, Xiaoming Li, Yizheng Ye
Microelectronics Center
Harbin Institute of Technology, Harbin 150001
Email: stsang@hit.edu.cn

Abstract—Hardware/software partitioning is a key issue in the design of embedded systems where performance, chip area and/or power dissipation constraints have to be met. The granularity for automatic partitioning has great impact on the run-time of the partitioning and the quality of the final implementation. In this paper we present a new approach that generates partitioning objects using the dependency of the operations in the system. By clustering the dependent operations into one object, the number of the objects and the coupling between them are lowered. Compared to the fine-grain partitioning, our approach, which is called Dependency Driven Partitioning, prunes a great portion of invalid solution space during partitioning objects formation, so as to expedite the partitioning without loss of the quality. Experiments with simulated annealing optimization show a faster convergence than that of fine-granularity with comparable partitioning quality.

I. INTRODUCTION

Hardware/Software partitioning is a challenging problem in today's system design since complete systems can be integrated onto one single chip. The system designer faces the problem whether to implement the system parts as hardware or software. Which decision is finally taken depends on a variety of design constraints/goals like performance, hardware effort, power dissipation, etc. Therefore, the process of hardware/software partitioning is a complex optimization problem.

The automatic partitioning usually starts from the high-level system specification (in C, C++). First, the system is divided into small parts, namely partitioning objects. Then these parts are moved from software to hardware or vice versa in order to find the best hardware/software tradeoff. Because the solution space is very large, optimization algorithms are applied in which the cost, performance and power metrics are used to search the optimal partition. The granularity during partitioning has great impact on the run-time of the partitioning and the quality of the final solution. A coarse-grain granularity usually has a small solution space, while it may hide the optimization potential, especially for the cost metric, which is critical for small embedded applications. On the contrary, the fine granularity will keep all the potential open. However, due to the communication overhead of fine-grain partitioning objects, a great number of unreasonable solutions need to be pruned during the partitioning, which leads to long computing time of the optimization algorithm.

In this paper we present a new approach that generates partitioning objects using the dependency of the operations. Unlike the previous methods, the clusters of dependent operations are used to form the partitioning objects, which have strong cohesion and weak coupling. Experiments show that dependency driven partitioning can reduce the search space and achieve satisfactory partition quality.

The outline of this paper is as follows: In Section II some hardware/software partitioning methods, especially the method of partitioning objects formation, are given. Section III describes the Dependency Driven Partitioning(DDP) algorithm in detail. In section IV the results that confirm the usefulness of the approach are demonstrated. Finally, a conclusion is given.

II. RELATED WORK AND MOTIVATION

Automated approaches to hardware/software partitioning have emerged since the early 1990s. The task of the partitioning is to find a design implementation that fulfills all the specification requirements at a minimum cost. Approaches using different optimization algorithms, such as simulated annealing [1][2], integer programming [3] and Genetic Algorithm [4], were proposed. All these approaches have a common characteristic that their partitioning granularity is fixed. The granularity determines the size of system parts that can be either implemented as software program or hardware. The granularities ranged from the fine-grain operations (instructions) and basic blocks to coarse-grain functions or processes. Henkel et al [5][6][7] researched on the problem of granularities and presented the run-time dynamically determined granularity approaches.

All the approaches reported so far use solely structural information to derive the partitioning objects. The fine granularity, such as instruction level, will increase the run-time of the partitioning process with a great number of invalid solutions. While in the coarse-grain partitioning, the partitioning objects, such as functions, are affected by the layout of the source codes, which is determined by the programming styles.

We propose a new method to derive the partitioning objects with the dependency information of the operations, which captures the inherent relationships. In the instruction level parallel computing domain, Satish Narayanasamy et al[8] divided the instructions into dependency chains to be executed efficiently on aggressively clustered data paths. The dependency chains with strong cohesion and weak coupling reduce the inter-cluster communications and simplify the
main()
{
    int c, nl, nw, nc, inword;
inword = 0;
nl = nw = nc = 0;
c = getchar();
while(c != EOF)
{
    nc = nc + 1;
    if (c == 'n')
        nl = nl + 1;
    if (c == ' ' || c == 't')
inword = 0;
    else if (inword == 0)
    {
        inword = 1;
        nw = nw + 1;
    }
c = getchar();
}
printf("%d\n", nl);
printf("%d\n", nw);
printf("%d\n", nc);
}

(a) Original program

int c, nl;
nl = 0;
c = getchar();
while(c != EOF)
{
    nc = nc + 1;
    if (c == 'n')
        nl = nl + 1;
    if (c == ' ')
        c = getchar();
}
printf("%d\n", nc);

(b) Operations related to nc

(a) Original program

int c, nl;
nl = 0;
c = getchar();
while(c != EOF)
{
    nc = nc + 1;
    if (c == 'n')
        nl = nl + 1;
}
printf("%d\n", nc);

(b) Operations related to nc

Software and lower the power. Work in [9] used dependency
to decompose a task into time-critical and non-critical parts,
which can facilitate the scheduling of the real time system.

III. DEPENDENCY DRIVEN PARTITIONING OBJECTS
GENERATION

Most programs or procedures can be decomposed into
several parts, each of which contains the dependent operations
dealing with one sub-task of the original program. Fig. 1
shows the decomposition of a program, which calculates the
number of characters, words and lines of a continuous text
stream. (a) is the original program, and (b), (c) are the subset
of operations counting the characters and lines. Such clusters
of dependent operations are called dependency chains (DCs).
The partitioning objects in the proposed approach are based
on DCs.

Our approach is targeted for the structured program with
procedures in high-level languages (e.g. C). The program is
denoted with language independent control flow graph (CFG).
For a CFG $G = \{V, E\}$, $V$ is the set of nodes corresponding
to operations of the program. $E$ is the set of the edges.
e_{ij}(v_i, v_j) \in E$, if node $v_j$ is dependent on $v_i$.

Definition 1: Tuple $p(n, va)$ is an inspection point of a
CFG $G(V, E)$, where $n$ is a node of $G$ and $va$ is a variable
of the program corresponding to $G$.

Definition 2: Graph $DC_{p(n, va)} = \{V', E'\}$ is the
dependency chain of CFG $G(V, E)$ with respect to inspection
point $p(n, va)$, where $V' \subseteq V$ and $E'$ is the set of all nodes
that have effect on the value of $va$ at node $n$. $e_{i,j}(v_i, v_j) \in E'$
denotes the dependency between $v_i, v_j \in V'$.

A. Sequential nodes

We begin with the sequential data computing nodes without
control structure.

Definition 3: The variable set $SRC(n)$ containing all the
source operands of the computing node $n$ is called the
source set of $n$. The variables set $DST(n)$ containing all the object
operands of the computing node $n$ is called the destination set
of $n$.

For node $n_i$, a predecessor of node $n_j$, the set $DST(n_i)$ and
the inspection point jointly determine if node $n_i$ is included
in the DC.

\[
\text{DC}(n_j, va) = \begin{cases} 
\text{DC}(n_j, va), & \text{if } va \notin \text{DST}(n_i) \\
\{n_i\} \cup \left( \bigcup_{x \in \text{SRC}(n_i)} \text{DC}(n_i, vx) \right) & \text{if } va \in \text{DST}(n_i)
\end{cases}
\]

(1)

Node \(n_i\) is included in a DC for inspection point \(p(n_j, va)\) if node \(n_i\) assigns a value to variable \(va\). So following the rule, the DC can be generated backward recursively from the inspection point. Fig. 2 shows the algorithm.

B. Control structures

Control nodes in the CFG are derived from compound control statements of the high-level language, such as if, while and switch in C. To maintain the correct logic of the program, related control nodes must be included in the DC.

**Definition 4:** Node \(n\) is the dominated by node \(n_D\), if \(n_D\) is in all paths from entry node to \(n\) in the CFG. The set of all the nodes that dominate \(n\) is dominance set of node \(n\), denoted \(\text{DOM}(n)\).

If a node is not under control of any compound statement, its domination set is \(\phi\). If \(\text{DOM}(n)\) has many members, \(n\) is in nested control structures. Taking control structures into account, for node \(n_i\), a predecessor of node \(n_j\), if \(va \in \text{DST}(n_i)\) then

\[
\text{DC}(n_j, va) = \{n_i\} \cup \left( \bigcup_{x \in \text{SRC}(n_i)} \text{DC}(n_i, vx) \right) \cup \left( \bigcup_{y \in \text{SRC}(k) \cap \text{DOM}(n_i)} \text{DC}(k, vy) \right)
\]

(2)

Now it is obvious that the rule in last subsection is just the special case of this rule when \(\text{DOM}(n_i) = \phi\).

C. Procedures

A system is usually composed of many modules and involved with procedures or functions. The algorithm presented above works well with intra-procedure DC generation. The DCs would always be no larger than functions if they cannot stretch across functions, and most of all, partitioning objects based on such DCs are still dominated by the layout of the functions in the program. When a procedure is called, the effect of a procedure-calling node is similar as that of a simple computing node, i.e. it operates on some source operands and writes the results to destination operands. So the concepts of source set and destination set are still applicable. However, the calculation of \(\text{SRC}(n)\) and \(\text{DST}(n)\) is somehow more complicated. The \(\text{DST}(n)\) includes output parameters, the global variables changed by the function and the variable receiving the return value of function. The \(\text{SRC}(n)\) includes input parameters and the global variables referenced by the function. The point is that not all the variables in \(\text{SRC}(n)\) are effective source variables for a given function invoking, and the effective source variables can be determined only if the DC in the function has been computed. The DC construction at a function invoking node is divided into three steps. For function invoking node \(n_i\), a predecessor of node \(n_j\), \(F\) is the function called at \(n_i\):

(i) Computing the \(\text{DST}(n_i)\) according to \(n_i\) and \(F\);
(ii) Computing the DC in \(F\):

\[
\text{DC}(n_j, va) = \bigcup_{x \in \text{SRC}(n_j) \cap \text{DST}(n_i)} \text{DC}(n_j, vx);
\]

(iii) Computing the \(\text{SRC}(n_i)\) according to \(\text{DC}(n_j, va)\).

D. Partition objects based on DCs

From Definition 2 we know that \(\text{DC}(n, va)\) is dependent on two parameters—variable \(va\) and node \(n\). For hardware/software partitioning, we want to decompose the system without location of the inspection point or with implicit \(n\). According to the algorithm above, it is reasonable to construct DC from the end node of the CFG. However starting from the last node is insufficient to get all nodes involving the variable. Fig. 3 shows a case. \(\text{DC}(6, t)\) includes statement 1, 2, 5 and 6, while \(\text{DC}(4, t)\) includes 1, 2, 3 and 4. Such case arises because of the output operations. Let \(\text{Output}(G, va)\) be the set of nodes in CFG \(G\) that output variable \(va\), let \(l\) be the end node of \(G\), and let \(N = \text{Output}(G, va) \cup \{l\}\), the partitioning object with respect to variable \(va\) is

\[
\text{O}_{(va)} = \bigcup_{n \in N} \text{DC}(n, va).
\]

Let \(V_p\) be the set of all the variables of program of CFG \(G\), the set of partitioning objects of the program is

\[
\text{O}_p = \{\text{O}_{(v)} | v \in V_p\}.
\]

IV. EXPERIMENTAL RESULTS

Experiments have been conducted in order to show the effectiveness of our hardware/software partitioning approach with a dependency driven granularity, and the results were compared to that of fine-grain and coarse-grain granularity partitioning. The experiment inputs come from a real industrial project—a VoIP codec. The codec is conformed to ITU-T G.729 standard and the implementation is based on the ITU-T reference source code. The top three time-consuming functions are identified by profiling and these functions consume more than 56% of the total run-time. Each of the
TABLE I
CHARACTERS OF THE EXPERIMENT INPUTS

<table>
<thead>
<tr>
<th>Function</th>
<th>( N_{op} )</th>
<th>( N_f )</th>
<th>( N_{DC} )</th>
<th>( Max )</th>
<th>( Avg )</th>
</tr>
</thead>
<tbody>
<tr>
<td>D4i40_17</td>
<td>872</td>
<td>12</td>
<td>37</td>
<td>277</td>
<td>71.7</td>
</tr>
<tr>
<td>Lag_max</td>
<td>187</td>
<td>6</td>
<td>14</td>
<td>79</td>
<td>33.2</td>
</tr>
<tr>
<td>Norm_Corr</td>
<td>291</td>
<td>14</td>
<td>21</td>
<td>133</td>
<td>41.6</td>
</tr>
</tbody>
</table>

TABLE II
EXPERIMENT RESULTS

<table>
<thead>
<tr>
<th>Function</th>
<th>Granularity</th>
<th>Run-time(s)</th>
<th>Cost(ngate)</th>
</tr>
</thead>
<tbody>
<tr>
<td>D4i40_17</td>
<td>Operation</td>
<td>31</td>
<td>33.0</td>
</tr>
<tr>
<td></td>
<td>Function</td>
<td>1.7</td>
<td>47.3</td>
</tr>
<tr>
<td></td>
<td>DC</td>
<td>5.6</td>
<td>29.6</td>
</tr>
<tr>
<td>Lag_max</td>
<td>Operation</td>
<td>7.1</td>
<td>14.3</td>
</tr>
<tr>
<td></td>
<td>Function</td>
<td>1.1</td>
<td>27.4</td>
</tr>
<tr>
<td></td>
<td>DC</td>
<td>1.9</td>
<td>15.6</td>
</tr>
<tr>
<td>Norm_Corr</td>
<td>Operation</td>
<td>9.7</td>
<td>17.0</td>
</tr>
<tr>
<td></td>
<td>Function</td>
<td>2.7</td>
<td>27.1</td>
</tr>
<tr>
<td></td>
<td>DC</td>
<td>4.2</td>
<td>19.7</td>
</tr>
</tbody>
</table>

three functions together with its supporting functions is used as an independent input for partitioning.

The CFGs are produced from C-code with MachSUIF[10]. The DC based partitioning objects are generated using the algorithm described in Section III. Table I shows the characters of the three experiment inputs. \( N_{op} \) and \( N_f \) are the number of operations, and functions of the input respectively. \( N_{DC} \) is the number of DCs based partitioning objects; \( Max \) and \( Avg \) are the maximal and average number of operations of the object. The granularity is determined by the dependency between operations and the actual objects have the sizes range from 3 to more than 270 operations.

Partitioning is performed using simulated annealing algorithm similar to that used in [7]. Table II gives the partitioning results using operation, function and DC based partition objects to meet the constraint of 5 times speedup over full software implementations. The run time of the annealing algorithm and the hardware cost (gate equivalents) are compared. The function based coarse-grain partitioning has shortest run time while the hardware costs are higher under the same constraints. The operation based partition approach gets the most cost efficient partition with longest run time of the optimization. Compared to the fine-grain approach, our DC based method achieves the comparable partition with the speed is 2-5 times faster, and for the first experiment, it gets even better results. All the experiments were carried out on a Linux workstation with an Intel Pentium III processor at 550MHz, 256MB RAM.

V. Conclusion

We have presented a new hardware/software partitioning approach that uses dependency based partitioning objects. By clustering the dependent operation into one object, the number of the objects and the coupling between them are lowered. Compared to the fine-grain partitioning, our approach prunes a great portion of invalid solution space during partitioning objects formation, so we can expedite the partitioning without loss of the quality. Experiments with simulated annealing show a 2 to 5 times faster convergence with equal or even better partition qualities compared with the operation based fine-grain partitioning.

Due to the weak correlation between the dependency based partitioning objects, they should have the potential to simplify the scheduling and improve parallelism. The future work will concentrate on this point.

REFERENCES