Data I/O Minimization for Loops on Limited Onchip Memory Processors*

Lei Wang and Santosh Pande**

Compiler Research Lab, PO Box 210030,
Department of Electrical & Computer Engineering and Computer Science
University of Cincinnati, Cincinnati, OH- 45219
E-mail: {leiwang, santosh}@ececs.uc.edu

1 Introduction

Due to significant advances in VLSI technology, "mega-processors" made with a large number of transistors has become a reality. These processors typically provide multiple functional units which allow exploitation of parallelism. In order to cater to the data demands associated with parallelism, the processors provide a limited amount of on-chip memory. The amount of memory provided is quite limited due to higher area and power requirements associated with it. Even though limited, such on-chip memory is a very valuable resource in memory hierarchy. An important use of on-chip memory is to hold the instructions from short loops along with the associated data for very fast computation. Such schemes are very attractive on embedded processors where, due to the presence of dedicated hard-ware on-chip (such as very fast multipliers-shifters etc.) and extremely fast accesses to on-chip data, the computation time of such loops is extremely small meeting almost all real-time demands. Biggest bottleneck to performance in these cases are off-chip accesses and thus, compilers must carefully analyze references to identify good candidates for promotion to on-chip memory. In our earlier work [6], we formulated this problem in terms of 0/1 knapsack and proposed a heuristic solution that gives us good promotion candidates. Our analysis was limited to a single loop nest. When we attempted extending this framework to multiple loop nests (intra-procedurally), we realized that not only it is important to identify good candidates for promotion but a careful restructuring of loops must be undertaken before performing promotion since data i/o of loading and storing values to on-chip memory poses a significant bottleneck.

2 Our Approach

Our analysis begins by undertaking intraprocedural loop fusion assuming unlimited memory. We first calculate the amount of data re-use between the statements after carrying out loop fusion. We then calculate the closeness factor of every pair

---

* Supported in part by NSF through grant no. #EIA 9871345
** Corresponding author
of statements. Closeness factor between two statements quantifies the amount of data reuse per unit memory required. We decorate the program dependence graph (PDG) with this information by inserting undirected edges between the nodes of the PDG that represent statements. We then group the statements greedily under a given loop nest i.e. statements which have higher closeness factor are grouped together over those which have lesser. At every step, when we group statements, we examine if we must include some other statement(s) so as to preserve the dependences. Sometimes we may not be able to include statements due to dependence and memory capacity constraints. In such cases, we adjust the dependence groups to eliminate useless re-use edges. Finally, we iterate to expand the group of statements and when we exceed the capacity of the on-chip memory we halt. We carry out the above steps of grouping and adjusting the dependence groups until there are no more re-use edges. After finding out all possible groupings and then grouping the statements having more CF, we block the groups so as to fit the available memory size.

2.1 Block Size v/s Data I/O

Block size is another major feature to decide when to stop bringing more statements (data elements) into one group. The trade off is that due to more statements (array elements) being brought into the same group, the re-uses will be introduced, but it is possible the existing data reuse utilization may be reduced due to reduction in block size.

Example:

```
for i=1 to 50
    c[i]=a[i-1]+b[i+2]+d[i-1]; //S1
    r[i]=d[i+7]+e[i]+f[i]+g[i]; //S2
    k[i]=a[i+4]+b[i-3];        //S3
```

If we group statement S1 and S3 together only, and on chip memory is big enough all the data elements, then all the data for the inner loop in the following example will be stored in the onchip memory. The total data I/O saved (in the inner loop) of one block will be 26 (13 for a[], 13 for b[]). Total I/O saved is 70.

Output Loops:

```
for i'=1 to 50 step 18     //L1
    for i=i' to min(i'+17, 50)
        c[i]=a[i-1]+b[i+2]+d[i-1]; //S1
        k[i]=a[i+4]+b[i-3];        //S3

for i=1 to 50              //L2
    r[i]=d[i+7]+e[i]+f[i]+g[i]; //S2
```

Obviously, between S2 and S1, there is data reuse (d[]). Once we bring in S2 into the group, the loop will be: