International Journal of Computer Applications (0975 – 8887)
Volume 49– No.21, July 2012

Multiple Output Complex Instruction Matching Algorithm for Extensible Processors

Puneet Goyal
Associate Professor
CS, Graphic Era University Dehradun, Uttarakhand, India

Narayan Chaturvedi
Assistant Professor
CS, Graphic Era University Dehradun, Uttarakhand, India

ABSTRACT

In order to meet the increasing challenges concerning the performance and power demands of embedded applications, a processor is now embedded with the Application-specific functional units. Customized Functional Units both as hardware and the corresponding instructions are embedded to the base processor in order to improve the computational efficiency for a target application. During this process of generating the complex instructions and also for the code generation on this extended processor, one of the critical challenges for the compiler is to automatically perform fast and efficient instruction matching and selection. In this project, we developed a novel and efficient algorithm for matching the multiple-output complex Functional Units (FU's). We will also illustrate that the assumption, which is the basis of the most of the current covering methodologies, may not always hold true. Current covering algorithms, generally aim to find the optimal cover within each basic block that minimizes the number of selected matches. Fewer matches translate to fewer operations for the schedule, and it is expected that the increased scheduling freedom leads to better (shorter) schedule. We provide some examples showing that this assumption need not necessarily achieve the goal of minimizing the execution time.

General Terms

Matching algorithms, Instruction Set Architecture.

1. INTRODUCTION

Recently in the market, there is explosive demand for the Mobile phones, PDAs, iPads, digital video cameras, audio devoices and other high performance special purpose electronic devices. These devices need to be incorporated with application-specific hardware design is order to meet the challenging cost, power and performance demands. One of the known strategies to provide such special hardware designs is to build a system comprising a low cost core processor, such as an ARM [1] and a number of highly specialized application specific integrated circuits (ASICs) incorporated within it. The ASICs are specially designed hardware units that can accelerate the execution of the computationally demanding portions of the application which would otherwise run too slowly if using just the core processor. Clark *et. al* [2] and Pothineni *et. al.* [3] mentions that while this approach is effective, ASICs are costly to design and offer only a hardwired solution that permits almost no post programmability.

An alternative approach is to have a processor centric system with customized accelerators. Here the core processor is augmented with the instruction set that is capable of significantly improving the performance and power of application in a cost-effective manner, compared to general purpose system. Also the system is post programmable and here the customized instructions with minor modification can easily be generalized to have their use across a set of applications Seeing these potential benefits, couple of commercial efforts [4,5] had also been made to bolster the high level design of custom processors.

An ASIP need to efficiently utilize the instruction level parallelism (ILP) available in the given application, so it can deliver the high performance. The VLIW architecture provides a better opportunity of customization, so we consider a VLIW architecture which consists of some application specific coarse grain functional units that augmented with a core set of functional units (FUs). Particularizing or constructing custom-make FUs for esp. frequent occurring complex operations in a given application can likely lead to very significant performance gains.

2. RELATED WORK

Matching and covering algorithms are well-known in the fields of code generation and logic synthesis. Keutzer [6] was the first to recognize the similarity between the software compiler's task of generating code and the technology mapping problem in automated VLSI design. Both problems can be handled with a matching algorithm, to find all possible instantiations of patterns (instructions or standard cells), followed by a covering algorithm to make a selection of matches that optimizes some criterion (execution time, code size, VLSI area or latency, etc.).

Many researchers had studied the potential utility of customizing the instruction set, but most of them do not describe the methods to automate the process. Some algorithms [7, 8] evaluates each node of the DFG via exploring the corresponding complete binary tree to decide if it can be a possible candidate. Their time complexity being exponential limits the size of DFG that can be considered in order for the algorithm to provide results in timely manner.

For code generation, we need to make the selection from the given set of instructions (including the complex ones) in a way that effectively utilizes the VLIW processor and Application Specific Functional Units (AFU). The goal is minimizing the schedule length. The complexity arises due to the fact that we want covering to be architecture driven and the I/O time-shape of the AFU could be distributed, it gives us to efficiently use the resources (Functional units and registers), thereby reducing the execution time. Most methods [7] generally aim to find the optimal cover within each basic block that minimizes the number of selected matches. Fewer matches translate to fewer operations for the schedule, and it is expected that the increased scheduling freedom leads to better (shorter) schedule.

2.1 Instruction Matching

As described in [9], there are two main approaches to handling the matching problem when performing technology mapping: the Boolean and the structural approach. The Boolean approach can only be applied to networks of Boolean functions and Structural matching will work on networks containing nodes of any type of function.

In a more recent work by Kukimoto et al. [10], a structural matching method was introduced that can handle DAG-shaped subject graphs, allowing reconvergence within the graph. None of the previously described matching algorithms allow pattern graphs to have more than one output. Peymandoust et. al. [11] employs symbolic algebra using commercial symbolic computer algebra systems like Maple and Mathematica, which can perform matching for even multiple-output pattern graphs. But it works only for arithmetic data flow segments, making this approach infeasible for most of the embedded applications. Paolo Ienne et. al. [12] make use of symbolic algebra tools for instruction selection. As opposed to tree covering based algorithms, mapping is performed simultaneously with algebraic manipulations in their algorithms. But as mentioned earlier this is restricted to only arithmetic data flow segments. Matching algorithm proposed by Arnold and Corporaal [13, 14] does not have this restriction, making it possible to exploit a larger family of pattern graphs. They perform a detailed search space exploration. We will experimentally show that the instruction matching algorithm proposed by us is highly efficient in comparison to them.

3. PROBLEM DEFINITION

Let *CFU-lib* be a library of Complex Functional Units (may be multiple-output), $G_{pat\text{-}i}$ be a data flow graph corresponding to the ith customized instruction (or CFU) in the *CFU-lib*(Figure 1) and G_{sub} be a data flow graph (DFG)(Figure 2a) within the control flow graph of a C application,

Instruction Matching:

Given G_{sub} and CFU-lib, find for each $G_{pat\text{-}i}$ in CFU-lib, all those dataflow segments of G_{sub} that match with the $G_{pat\text{-}i}$.

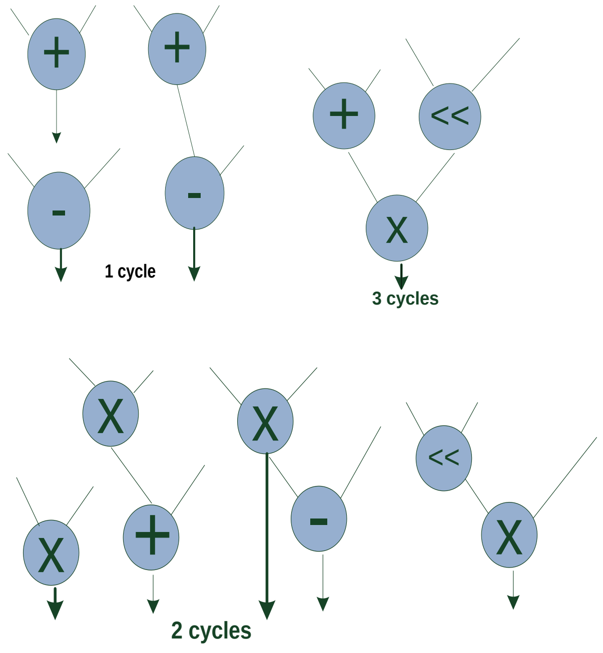


Figure 1: ***CFU-lib*****: A library of functional units**

Instruction Selection:

Given subject graph, G_{sub} with full matches found for all CFUs, find the subset of full matches that when implemented, minimizes the data ready time of the slowest subject graph output in the VLIW processor.

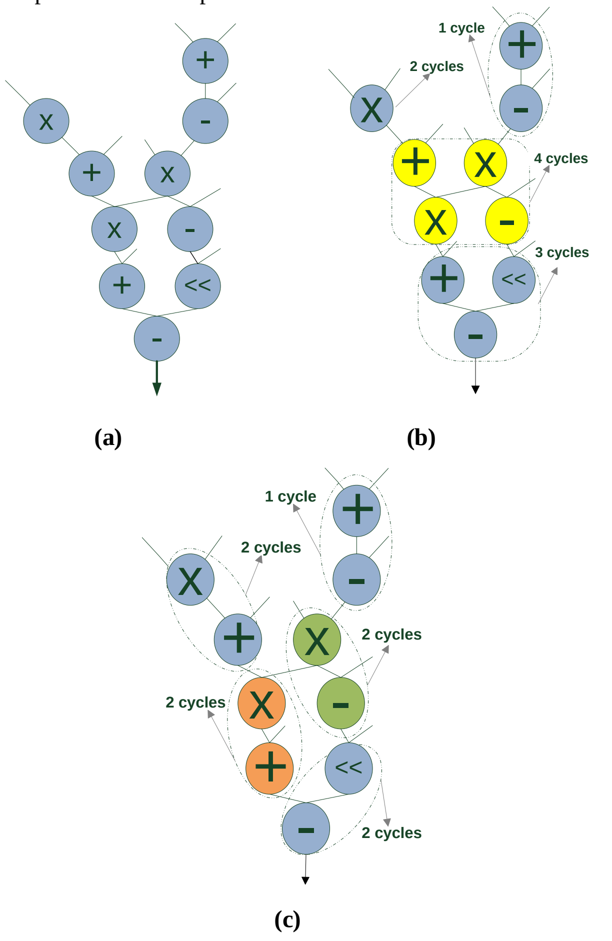


Figure 2: (a) G_{sub} Subject Graph (b) Cover selected when purpose is to minimize number of patterns used to cover - no. of cycles taken=9 (c) Cover selected when purpose is to minimize the data ready time - no. of cycles taken=7

4. ALGORITHM FOR INSTRUCTION MATCHING

Given G_{sub} and CFU-lib, we must try to find all matches between pattern graphs G_{pat} (from pattern library CFU-lib) and sub graphs of G_{sub} -The strategy for this is described in Section 4.1.

After all matches are found, we try to find the best cover, or selection of matches, that is, the set of matches that, when implemented, minimizes the data-ready time of the longest path through the subject graph. The covering approach is described in Section 4.2.

4.1 Instruction matching

Primary nodes: These are those nodes of G_{pat}, who's all input operands belong to the set of source operands of the customized instruction, corresponding to that particular G_{pat}.

Reachability matrix: From the adjacency matrix of a directed graph, we compute the Reachability matrix where an element Rij in Reachability matrix is 1, if there exists a path from node i to node j. Reachability[i][i] is considered to be 1.

compute reachability ( )

```
{
for (i = 0; i < num_ nodes; i ++)
  Reachability[i][i] = 1
for (j = 1; j < num _nodes; j ++)
  for (i = j-1; i >= 0; i--)
  {
    Reachability[i][j] = Adj_matrix[i][j]
     if (Reachability[i][j] == 0)
       for (k = j; k > i; k--)
         Reachability[i][j]+ = (Adj_matrix[i][k]
*Adj_matrix[k][j])
  }
}
```

Algorithm 3.1: Reachability

CommonSink matrix: CommonSink[i][j] = 1 if there exist a k such that Reachability[i][k] = 1 and Reachability[j][k] = 1. We will attach an attribute degree (> 0) to this Commonsink matrix.

CommonSink degree 1[ ][ ] = CommonSink[ ][ ].
CommonSink degree d[ ][ ] = (CommonSink[ ][ ])^d.

```
Find match (G_sub, G_pat)
 {
for i = 1 to p,
  find nodes in G_sub matching with primary input nodes $x_i$
//If primary node $x_i$ matches with $k_i$ nodes in G_sub
//then num_partial_matches = $k_1 * k_2 * k_3 * ::: * k_p$
//(partial matches)
for i = 1 to num_partial_matches
{
  //pruning the search space by applying heuristics
  //check if partial_match_i is an eligible candidate
  if(partial_match:Iseligible) num_eligible_candidates + +
}
for each_eligible_candidate {
  //validate this eligible_candidate for full match by traversing
along the data-flow edges.
  if (eligible_candidate_match:Isvalid) {
    n_valid_matches + +
    // update the match_vector list
            }
      }
}
```

Algorithm 3.2: find_match (G_{sub}, G_{pat})

4.2 Two algorithms for partial match identification

Let us consider for analysis that G_{sub} and G_{pat} has n and m number of total nodes. And no of primary nodes in each G_{pat} is p. Let us compare our algorithm (Algorithm 3.4) for identifying partial matches with the partial match identification algorithm (Algorithm 3.5) proposed by Arnold [13].

```
Insn_matching ( )
 {
For each basic block G_sub
 {
  compute Adjacency, Reachability and Commonsink
matrices for G_sub
    for each CFU graph, G_pat
      find match (G_sub, G_pat)
}}
```

Algorithm 3.3: Instruction Matching

```
FindPartialMatches_algorithm1 ( ) {
for each pattern G_pat in pattern library
 for each pattern node N_prime_pat of G_pat{
   For each node N_sub of G_sub
     //check for opcode and outdegree constraint
        if Nodematch (N_sub, N_prime_pat)
      //create a new match
         new Match(N_sub, N_prime_pat)
         //let us say, i^th prime node matched with $k_i$ nodes in
G_sub
}
Merge_matches( )
  //This identify all partial matches.
  // Num_partial_matches = $\prod_i k_i$, i. e of $O(n^p)$
}
```

Algorithm 3.4: Partial Matching Algorithm 1

```
FindPartialMatches_algorithm2( ) {
  for each pattern G_pat in pattern library
  for each N_sub of G_sub
    for each N_pat of G_pat
      if Nodematch(N_sub, N_pat)
      //create a new match
        new Match(N_sub, N_pat)
          //let us say, i^th node (N_pat, i) matched with $k_i$ nodes in
G_sub
Merge_matches( )
  //This identify all partial matches.
  // Num_partial_matches = $\prod_i k_i$, i. e of $O(n^m)$
}
```

Algorithm 3.5: Partial matching algorithm 2

4.3 Motivating example:

Example 4.1: The example as shown in Figure 2 illustrates that the common assumption, (heuristic used by many current methods) that lesser the number of patterns used, better will be the scheduling time, may not always hold true.

In Fig 2(b), number of FUs used = 4 but it takes 9 cycles.
In Fig 2(c), number of FUs used = 5 but it takes 7 cycles only.

Fewer matches translate to fewer operations for the schedule – this is generally considered the prime assumption for developing efficient covering algorithms. So current covering algorithms, generally aim to find the optimal cover within each basic block. The above example clarifies that that this assumption need not necessarily achieve the goal of minimizing the execution time.

5. EVALUATION FRAMEWORK

For evaluating the efficiency of the Instruction Matching Algorithm, we need to compile the given application code into an intermediate representation where the C-code is reduced into a DFG/CFG representation closer to assembler, although still largely architecture independent. Dataflow nodes should resemble generic assembler operations. This process would ease the process of instruction matching, since the algorithm is completely dependent on the topological characteristics of the DAG constructed. Any of the two compiler infrastructures, MachSuif[15] and Trimaran[16], could be selected for this purpose. But because of inherent complexities involved in Trimaran and being already familiar with Machsuif framework, Machsuif is selected for the purpose of establishing the efficiency of proposed instruction matching algorithm. Afterwards, it is adopted in Trimaran frame-work with some changes in data-structures, used for traversing along the data-flow edges.

6. ANALYSIS AND RESULTS

For evaluating the performance of proposed instruction matching algorithm, we used the bitwise benchmarks new life, histogram and bubble sort. Our CFU-lib consists of 6 patterns (or customized instructions) as shown in Fig 3.

Table 1 shows the number of complete (valid) matches found in a particular Basic block of that application.

Number of partial matches is of O(n^p) but experimentally we observed that it is much less than O(n^p). Also we analysed the effect of heuristics in pruning the search space.

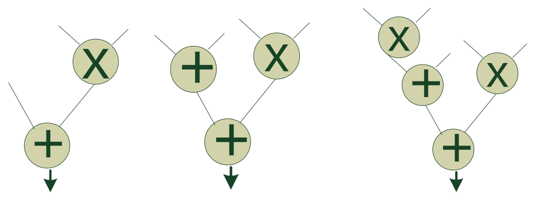
The Table 2 shows the fraction of eligible matches for different values of p and for different benchmarks. The eligible matches are those partial matches that qualify the outdegree constraint, Common sink constraint, etc. Fraction of eligible matches is computed as Number of eligible matches / number of partial matches. Best case = 0% for a certain benchmark and p, means that for a certain basic block (DFG) in that benchmark, we observed that after applying the heuristics, none of the partial match become eligible candidate. Without going through the complicated task of traversing through the data flow edges, we have filtered out lot of unsuitable partial matches. It is important to note that for p = 3, number of matches that would be considered for full match are only 0.1 to 2% of the partial matches identified in the stage1 of our algorithm. It means that for higher values of p, the eligibility criteria imposed is very effective in pruning the search space.

For applying the eligibly criteria, it is required to compute the Reachability matrix, the Commonsink matrix beforehand. The overhead involved is Reachability matrix and Commonsink matrix computation time but as it is to be done for each basic block only once and also, it is very helpful in effectively pruning the search space, and we can easily choose to pay for this overhead.

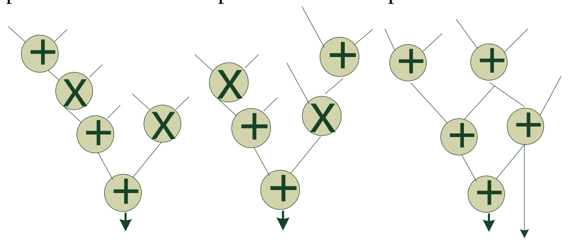
Theoretically, the number of comparisons performed while checking for eligibility criteria is num_partial_matches*(p*(p-1)/2). We compare this with the number of comparisons actually done. On an average, for p = 2, we found the ratio of actual number of comparisons and num_partial_matches*(p*(p-1)/2) to be about 2.5 and for p = 3, this ratio is 1 (Table 3).

Comparing two algorithms of finding out partial matches: We have described in Section 4.2, the two algorithms for finding partial matches. Table 4 shows for different benchmarks, the comparison between the two algorithms in terms of the number of partial matches (that are to be evaluated for eligibility and validity) found.

We observed that the Algorithm 3.4 to be very efficient than Algorithm3.5. The number of partial matches identified in Algorithm3.5 is much more than the number of partial matches identified in algorithm3.4 by us.



(a) CFU 0, m=2, p=1 (b) CFU 1, m=3, p=2 (c) CFU 2, m=4, p=2



(d) CFU 3, m=5, p=2 (e) CFU 4, m=5, p=2 (f) CFU 5, m=5, p=3

Figure 3: An example pattern Library CFU-lib

Table 1: Matches found

Bubble sort

Patid	BB	Valid_matches
0	3	2
0	8	2
0	9	2
0	14	2

Histogram

Patid	BB	Valid_matches
1	5	1
0	9	1
1	14	1
0	5	2
0	18	2
0	14	3
1	23	3
0	23	7

New life

Patid	BB	Valid_matches
1	32	1
1	5	2
1	11	2
0	32	2
3	19	3
4	19	3
3	25	3
4	25	3
0	5	4
5	19	4
5	25	4
0	11	5
1	19	10
1	25	10
0	19	20
0	25	20

Table 2: Results: Effects of Heuristics

	P	Best case (%)	Worst case (%)	Average (%)
Bubblesort	2	0	84	41.67
	3	0	11.11	**2.23**
Histogram	2	0	67	32.97
	3	0	1	**0.12**
Newlife	2	10.74	75	31.05
	3	0	5.56	**0.21**

Table 3: Ratio of experimental to Theoretical number of comparisons in eligibility check

Bubble sort

Patid	BB	Matches1	Matches2
0	3	2	10
0	8	2	6
0	9	2	4
0	14	2	6

Histogram

Patid	BB	Matches1	Matches2
0	9	1	2
0	5	2	6
0	14	3	15
0	18	3	15
1	5	6	18
0	23	7	63
1	14	15	75
1	23	63	567

New Life

Patid	BB	Matches1	Matches2
0	32	2	8
0	5	4	20
0	11	5	30
1	32	8	32
1	5	20	100
0	19	20	680
0	25	20	680
1	11	30	180
1	19	680	23120
3	19	680	$>10e^6$
4	19	680	$>10e^6$
1	25	680	23120
3	25	680	$>10e^6$
4	25	680	$>10e^6$
5	19	39304	$>10e^6$
5	25	39304	$>10e^6$

Table 4: Algorithm1(3.4) versus Algorithm2(3.5)

	Avg	Max	Avg P=2	Max P=2	Avg P=3	Max P=3
Bubblesort	2.20	3.50	2.58	3.50	**0.98**	**1.29**
Histogram	2.07	3.33	2.52	3.33	**0.96**	**1.38**
Newlife	2.01	3.50	2.57	3.50	**1.19**	**1.64**

7. CONCLUSION AND FUTURE WORK

We have presented a novel and very efficient algorithm for instruction matching. It successfully matches even multi-output complex Function units with a sub graph in the DFG of an application. We observed that the concept of Commonsink (or common descendent) plays very significant role in effectively pruning the search space. Matching only primary input nodes of the Graph Gpat corresponding to customized instruction and the concept of commonsink constitute the crux of the algorithm. We evaluated the performance of the matching algorithm with many benchmarks and compared its efficiency with some already existing algorithms.

At present we did not include arithmetic-logic reduction in our instruction matching algorithm. We provide support for handling commutative cases in instruction matching algorithm but not for complex arithmetic-logic reductions. The algorithm can be extended in future to match the Customized instruction (G_{pat}) with a sub graph (G_{sub}) in DFG, where the same computation is performed as in Gpat but the topology of G_{pat} and G_{sub} may not be the same.

8. ACKNOWLEDGEMENTS

The authors like to acknowledge Prof. Anshul Kumar and Embedded Systems group of Department of Computer Science and Engineering at IIT-Delhi for their kind support and guidance.

9. REFERENCES

[1] D. Seal, *ARM Architecture Reference Manual*. Addison – Wesley, 2000.

[2] Nathan T. Clark, Hongtao Zhong, Scott A. Mahlke, Automated custom instruction generation for domain-specific processor acceleration. IEEE Transactions on Computers, October 2005.

[3] N. Pothineni et. al., Exhaustive Enumeration of Legal Custom Instructions for Extensible Processors, 21st Int'l Conf. on VLSI design 2008.

[4] R. E. Gonzalez. Xtensa: A configurable and extensible processor. IEEE Micro, 2000

[5] T. R. Halfhill, Mips embraces configurable methodology. March 2003

[6] K. K. Dagon. Technology binding and local optimization by dag matching. In Proceedings of the design Automation Conference, pages 617-623, May 1987.

[7] A. Aho, R. Sethi and J. Ullman. Compilers: Principles, Techniques and Tools. Addison-Wesley, 1986

[8] K. Atasu et al., "Automatic Application-Specific Instruction Set Extensions under Microarchitectural Constraints," Proc. 40th Design Automation Conf., June 2003.

[9] Giovanni De Micheli. Synthesis and optimization of digital circuits. McGraw Hill, 1994

[10] Y. Kukimoto, R. K. Brayton, and P. Sawkar. Delay-optimal technology mapping by dag covering. In Proceedings of the Design Automation Conference, 1998.

[11] Armita Peymandoust and Giovanni De Micheli. Symbolic algebra and timing driven data-flow synthesis. In Proceedings of International Conference on ComputerAidedDesign,2001.

[12] Paolo Ienne, Laura Pozzi, and M. Vuletic. On the limits of processor specialization by mapping data flow sections on ad-hoc functional units. Technical Report CS Technical Report 01/376, LAP, EPFL, Lausanne, December 2001.

[13] Marnix Arnold. Matching and covering with multiple output patterns. Technical Report 1-68340-44(1999)-01, Delft University of Technology, January 1999.

[14] Marnix Arnold and Henk Corporaal. Designing domain-specific processors. ACM, 2001.

[15] Machinesuif. http://www.eecs.harvard.edu/hube/software.

[16] The trimaran compiler infrastructure. http://www.trimaran.org