Buffer-Sizing for Precedence Graphs on Restricted Multiprocessor Architectures

Thomas Huining Feng and Yang Yang
Mentors: Qi Zhu and Abhijit Davare

December 7, 2005

1 Overview

This work solves a sub-problem required to enable software synthesis for Parallel Heterogeneous Platforms (PHPs). This sub-problem is concerned with determining the buffer sizes between processing elements.

In the design flow, task allocation and scheduling are carried out before buffer sizing. Given a PHP with a number of processors and a set of tasks, a heuristic is used to allocate tasks onto processors and to schedule them. The input of this algorithm is a *precedence DAG*, whose vertices represent the tasks and whose edges represent data dependencies.

The scheduling algorithm [1] assumes unlimited FIFO sizes between processors, but this is not true in reality. Architectural platforms have finite-depth FIFOs between processors. If the FIFO depth is too small, execution may deadlock. We call this kind of deadlock *artificial deadlock* [2], as compared to *real deadlock* which can occur even if the FIFO depth were unlimited.

Prior work in this field mainly focuses on the buffer sizing problem in uni-processor platforms [3]. The previous work that deals with multiprocessor buffer minimization [4] does not consider interleaving communication, where two active tasks on different processors can communicate large amounts of data using one-place buffers. In this work, we develop algorithms to address this problem. Theoretical as well as practical results are provided.

2 Problem Statement

Before our algorithms are presented, we first formalize the problem. An instance of this problem is a 5-tuple $\langle V, P, M, E, W\rangle$. $V = \{v_1, v_2, \ldots, v_m\}$ is the set of vertices in the precedence DAG. $P = \{p_1, p_2, \ldots, p_l\}$ is the set of processors. $M : V \to P$ is the mapping from vertices to the processors that they are scheduled on. $E = \{e_1, e_2, \ldots, e_n\}$ is the set of edges. We distinguish two disjoint subsets of E: $S = \{e | e \in E \wedge M(src(e)) = M(des(e))\}$ is the set of schedule edges, and $D = \{e | e \in E \wedge M(src(e)) \neq M(des(e))\}$ is the set of data edges. $W : D \to \Re^+$ is the weight function. M and E are acquired from the scheduling algorithm.

We try to compute valid FIFO sizes according to some minimum criteria without giving rise to artificial deadlock. If we use function $F : P \times P \to \Re^+$ to denote the FIFO sizes, the two problems that we are going to solve are:

- *Min max*: with $\langle V, P, M, E, W\rangle$ given, find a valid F such that $\max\{F(p_i, p_j) | \forall i, j\}$ is minimized.
- *Min total*: with $\langle V, P, M, E, W\rangle$ given, find a valid F such that $\sum\{F(p_i, p_j) | \forall i, j\}$ is minimized.

3 Solving the Min Max Problem

Because any valid FIFO assignment should not produce artificial deadlock, we need to study how artificial deadlocks occur. A deadlock occurs when there is cyclic dependency. An artificial deadlock is a special kind of deadlock where the cyclic dependency exists only because of FIFO depth. With the observation that a data edge implies bidirectional dependency if there is not enough FIFO space for it, we transform the precedence DAG into a *dependency graph* by making all the data edges bidirectional. We then prove the following theorem about artificial deadlock.

Theorem 1: *Artificial deadlock exists if and only if there is a cycle in the dependency graph (*dependency cycle*).*

We further observe that a dependency cycle must contain at least one data edge, because the precedence graph is acyclic. In addition, schedule edges do not affect FIFOs. Combining all these results, our algorithm to solve the min max problem only needs to deal with data edges in dependency cycles.

Our algorithm iterates over all the edges in the dependency graph. One edge is resolved and removed each time (hence, E changes over time). In iteration i, if we let $V_i^{free} = \{v | v \in V \wedge \overline{\exists} e \in E_i . des(e) = v\}$ and $E_i^{free} = \{e | e \in E_i \wedge src(e) \in V_i^{free}\}$, then our algorithm only needs to consider edges in E_i^{free}. Among all the edges in E_i^{free} that are also in dependency cycles, the algorithm always chooses the one e_i such that the FIFO size required to complete the communication on e_i, $F_i(M(src(e_i)), M(des(e_i)))$, is minimal. It builds F_i by making it the same as F_{i-1} (initially, F_0 always returns 0), except that $F_i(M(src(e_i)), M(des(e_i)))$ becomes this new FIFO size.

We call the above algorithm A_m, and give the following theorem with detailed proof:

Theorem 2: *Assume that A_m terminates after iteration k. Let F be F_k computed by A_m. F is a valid FIFO assignment, and $\max\{F(p_i, p_j) | \forall i, j\}$ is minimized.*

To detect whether a data edge is in any dependency cycle, we develop an $O(|E|)$-time algorithm A_c that returns a boolean. With this, we show the complexity of A_m to be $O(|E|^2)$.

4 Solving the Min Total Problem

For the min total problem, it can be proved that in any intermediate step i, it cannot be determined which edge in E_i^{free} should be resolved so as to guarantee the total FIFO size to be minimized at the end. This problem turns out to be an *NP*-hard problem due to the following theorem:

Theorem 3: *Any instance of the Feedback Arc Set (FAS) Problem [5], which is* NP*-hard, can be reduced to a min total problem in polynomial time.*

Because of this, we are content with another algorithm, A_t, that solves the min total problem in exponential time.

5 Implementation and Experimental Result

The algorithms are implemented in C++ with the Boost Graph Library (BGL). We have manually built a set of simple precedence DAGs, which covers most of the corner cases. We have also generated bigger random precedence DAGs with Task Graph For Free (TGFF) [6]. For those graphs, the above-described algorithms return the correct results to the min max problem and the min total problem, respectively. The big difference in time complexity is reflected with both small and large test cases. Furthermore, from the experiments, we obvserve that the results given by A_m are good heuristics for those given by A_t.

References

[1] Gilbert C. Sih and Edward A. Lee. Compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *j-IEEE-TRANS-PAR-DIST-SYS*, 4(2):175–187, February 1993.

[2] M. Geilen and T. Basten. Requirements on the Execution of Kahn Process Networks. In P. Degano, editor, *Proc. of the 12th European Symposium on Programming*, 2003.

[3] S. S. Bhattacharyya, R. Leupers, and P. Marwedel. Software synthesis and code generation for signal processing systems. Technical Report CS-TR-4063, 1999.

[4] Marleen Adé, Rudy Lauwereins, and J. A. Peperstraete. Data memory minimisation for synchronous data flow graphs emulated on DSP-FPGA targets. In *DAC*, pages 64–69, 1997.

[5] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[6] Robert P. Dick, David L. Rhodes, and Wayne Wolf. TGFF: task graphs for free. In *CODES*, pages 97–101, 1998.