A theoretical and practical approach to instruction scheduling on spatial architectures *

S. Amarasinghe[1], D. R. Karger[1], W. Lee[1], V. S. Mirrokni[1]

Laboratory for Computer Science, Massachusetts Institute of Technology

Abstract. This paper studies the problem of instruction assignment and scheduling on spatial architectures. Spatial architectures are architectures whose resources are organized in clusters, with non-zero communication delays between the clusters. On these architectures, instruction scheduling includes both space scheduling, where instructions are mapped to clusters, and the traditional time scheduling. This paper considers the problem from both the theoretical and practical perspectives. It presents two integer linear program formulations with known performance bounds. We also present an 8-approximation algorithm for constant m and constant communication delays. Then, we introduce three heuristic algorithms based on list scheduling. Then we study a layer partitioning method. Our final algorithm is a combination of layer partitioning and the third heuristic. Two of the better algorithms are evaluated on the Raw machine. Results show that they are competitive with previously published results; for scientific codes, our heuristics can perform an average of 25% better.

1 Introduction

Spatial architectures are becoming increasingly important because they address the problem that wire delays do not scale with technology [1]. Signals already take more than one cycle to cross a chip today, and the delay will only get worse in future technology. Spatial architectures address this problem by organizing their resources into replicated units on the chip. Communication within a unit takes one cycle, but communication between units incurs one more or additional cycles of delays. Examples of spatial architectures include clustered VLIWs, Raw [23], Grid processors [20], and ILDPs [11].

Instruction scheduling is an important optimization problem on spatial architectures. On these architectures, the instruction scheduler has to partition instructions across the computing resources. While instruction schedulers on traditional architectures only need to assign instructions to time, on spatial architectures they need to assign instructions in both space and time. We call this combined scheduling problem *space time scheduling.*

Like most practical instances of instruction assignment and instruction scheduling is known to be NP complete [22]. Thus, in practice space-time schedulers are based on heuristics. To make partitioning decisions, a heuristic scheduler has to understand the proper tradeoff between parallelism and locality. Figure 1 shows an example of this tradeoff. Spatial scheduling by itself is already a more difficult problem than temporal scheduling, because a small spatial mistake is generally more costly than a small temporal mistake. If a critical instruction is scheduled one cycle later than desired, only one cycle is lost. But if a critical instruction is scheduled one unit of distance farther away than desired, cycles can be lost from unnecessary communication delays, additional communication resource contention, and increase in register pressure. In addition, some instructions on spatial architectures may have specific spatial

* Emails: `saman,karger,walt,mirrokni@lcs.mit.edu`

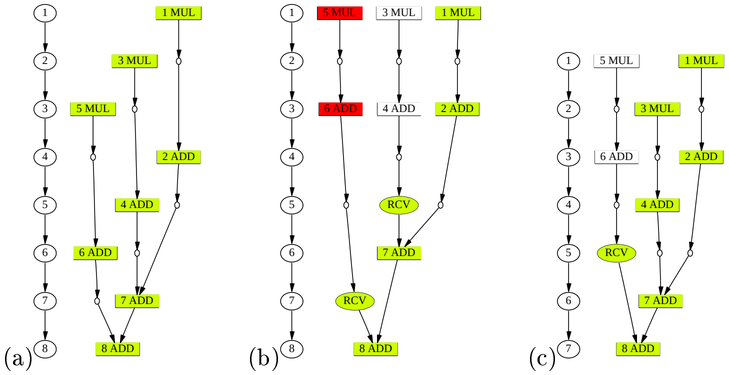

Fig. 1. An example of tradeoff between parallelism and locality on spatial architectures. Rectangles are instructions and edges between rectangles represent data dependences. The circles on the left represent the time axis. Consider an architecture with three clusters, each with one functional unit, where communication takes one cycle of latency plus one receive instruction. In (a), conservative partitioning that maximizes locality and minimizes communication leads to an eight-cycle schedule. In (b), aggressive partitioning leads to too much communication an eight-cycle schedule. The optimal schedule, Figure (c), takes only seven cycles, and it requires a careful tradeoff between locality and parallelism.

requirements. These requirements arise from the need to access specific spatial resources, such as a specific memory bank [3]. A good scheduler must be sensitive to these constraints in order to generate a good schedule.

This paper considers the space-time scheduling problem from both a theoretical and practical perspectives. We present two formulations of the scheduling problem as integer linear programing (ILP). The first formulation is for our general problem, but it uses a large number of integer variables. The second formulation is a $\frac{7}{3}$ approximation algorithm that applies when communication delays are small. We describe the LP-relaxation of the second formulation and derive a $\frac{7}{3}$ approximation algorithm for the case of small communication delays. These formulations can be used to find the optimal solutions for small graphs. The solutions can then be used as baselines for evaluating more practical heuristics approaches. First, we present an 8-approximation algorithm whose running time is exponential in the number of tiles and communication delays, but is polynomial if these parameters are not constant. From the practical side, we present heuristics for estimating the minimum completion time of the subgraphs rooted at each instruction. These estimatations can be used as priority functions for existing list scheduling based algorithms, and they are more accurate than existing heuristics. Then we present several algorithms based on the these heuristics. Finally we introduce a partitioining idea in which we partition the precedence graph into some small layers. We implement these algorithms in Rawcc [13], the Raw instruction level parallelizing compiler, and we show that this algorithm compares favorably with Rawcc's original algorithm.

The rest of the paper is organized as follows. Section 2 defines the problem statement. Section 3 presents related work. Section 4 describes the two integer programming formulations of the problem. Section 5 introduces an 8-approximation algorithm that depends on the number of tiles and communcation delays exponentially, but is polynomial if these parameters are constant. Section 6 introduces several practical heuristic bounds and algorithms. Section 7.1 describes a layer partitioning method and our final algorithm, which is a combination of previous heuristics and this layer partitioning idea. Section 8 presents results of those algorithms on the Raw machine. Section 9 discusses future work and concludes.

2 Problem Statement and Prelemaniries

This section defines the problem statement. We adapt notations described in [7]. A scheduling problem is denoted by $\alpha|\beta|\gamma$. α denotes the machine environment, e.g., P for parallel identical machines; β denotes various side constraints and characteristics, e.g., a list of precedence constraints *prec*. γ denotes an optimality criterion. In our problem, γ is $C_{\max}$, the maximum completion time, or makespan, of a set of instructions.

We are given a set V of instructions and m clusters or tiles[1] on which these instructions should be executed. Each instruction i has processing time p_i. A set of edges E defines the precedence-constraints in V. The graph $G = (V, E)$ is a directed acyclic graph. For each precedence-constrained instruction pair $(i, j) \in E$ and pair of clusters (p, q), we define an associated non-negative delay $l_{i,j,p,q}$ to be the minimum difference between the scheduled time of i and j, if i is scheduled on tile p and j is scheduled on tile q. The output is a schedule mapping from each instruction j to a tile q and a time t, such that for each edge $(i, j) \in E$, if i is scheduled on tile p, it is scheduled at no later than time $t - l_{i,j,p,q}$. Our objective is to find a proper schedule with minimum makespan $(P|prec; l_{i,j,p,q}|C_{\max})$.

Our model allows instructions to be preplaced on a specific cluster. We define F to be a mapping from instructions to clusters. The domain of F is the set of preplaced instructions. Each mapping $i \to p$ in F specifies the constraint that instruction i must be mapped onto cluster p.

We model $l_{i,j,p,q}$ as the sum of two orthagonal components: an instruction dependent component and a cluster dependent component: $l_{i,j,p,q} = delay(i, j) + com(p, q)$. The delay delay$(i, j)$ can be used to model pipelined functional units; for this use its value only depends on i. com(p, q) is used to model the cost of communicating between clusters. In different parts of this paper, we will consider different variants of the problem with restrictions and generalizations. One special case that we are interested in is when communcation delays are small compared to the length of the instructions; more precisely, communication delays are small if they are smaller than all instruction lengths. In many cases, the algorithm considers the preplaced instruction, but in some case it does not.

Throughout the paper, m corresponds to the number of tiles, t_i's correspond to the starting time of the instruction i. We will also use p_i's for instruction lengths, $com(p, q)$ for the communication delay between tile p and tile q and *prec* as the precedence graph.

3 Related Works

The most general problem for which a polynomial time algorithm is known was studied in [6]. If m and $l_{i,j,a,b}$ are constants, instructions are unit lengths, the precedence graph is a tree, and there are no preplaced instructions, Pm|tree; $p_j = 1$; $l_{i,j,a,b} \in \{0, 1, \ldots, D\}$ $|C_{\max}$, the problem is solvable in polynomial time. When the precedence graph is a DAG instead of a tree, it is unknown whether the problem is NP-complete – in fact a special case of this problem is the well-known 3-tile scheduling problem $(P3|prec; p_j = 1|C_{\max})$ whose complexity is open. Generalizing any other part of the above problem yields an NP-complete problem.

Since most scheduling problems are NP-complete, approximation algorithms have been derived for them. In the presence of precedence delays, the best approximation factor for

[1] Throughout this paper, we will use clusters and tiles interchangeably.

the problem $P|prec;\ delays|C_{\max}$ is $2 - \frac{1}{m}$ [19]. In the presence of communication delays, most studies have assumed that the delay is *uniform*: zero if two instructions are on the same cluster, and a constant c if two instructions are on different clusters. For c smaller than instruction execution times and no precedence (pipeline) delays, there is a $\frac{7}{3}$-approximation algorithm [17]. We will extend this algorithm to handle precedence delays as well. For larger communication delays, there is no known constant factor approximation algorithm.

Many heuristics approaches have been developed for space-time scheduling. UAS (Unified Assign-and-Schedule) performs space-time scheduling on clustered VLIWs in a single step, using a greedy, list-scheduling-like algorithm [21]. Desoli describes an algorithm targeted for graphs with a large degree of symmetry [4]. Leupers describe an iterative combined approach to perform scheduling and partitioning on a VLIW DSP [15]. The approach is based on simulated annealing.

However, there have been far fewer space-time scheduling algorithms that take into account preplaced instructions. One such algorithm is BUG (Bottom-Up-Greedy). BUG is implemented on for ELI, one of the earliest spatial architectures that relies on the compiler for space-time scheduling [5]. BUG only performs space-scheduling; time-scheduling is done via traditional list scheduling. BUG is a two-phase algorithm: the algorithm first traverses a dependence graph bottom-up to propagate information about preplaced memory instructions. Then, it traverses the graph top-down and greedily map each instruction to the clusters that can execute it earliest. The Multiflow compiler uses a variant of BUG [16], but it does not account for preplaced instructions. Lee also handles preplaced instructions [13]. He borrows his general approach from multiprocessor task graph scheduling [13]. Like Ellis, Lee uses a separate list scheduler to perform time-scheduling. Space-scheduling is performed in three steps. Clustering groups together instructions that have little parallelism; merging reduces the number of clusters through merging; placement maps clusters to tiles. Constraints from preplaced instructions are mainly handled during placement.

4 ILP Based Methods

In this section, we study two kinds of integer linear programming formulation for our problem. One is a naive approach for formalizing our general problem; the other is a formulation for a special case that is useful to derive an approximation algorithm. The advantage of the first formulation is that it works for the general case, but the size of the ILP is large. The second formulation is for a special case, but its size is much smaller, and we can derive an approximation algorithm from its LP-relaxation.

4.1 Time-Indexed and Interval-Indexed Integer Programming

This section presents integer programming formulations for our general problem. The formulation are generalizations of the Time-Indexed and Interval-Indexed integer programming formulation for the $P|prec;|\sum w_i C_i$ problem studied in [9] and [10], generalized to account for communication and precedence delays.

The idea is to associate a zero-one variable to each triple of instruction, tile, and time. Each such variable indicates whether an instruction is completed on the specific cluster at the specific time. This approach is called Time-Indexed Linear Programming. Here, variable y_{ijt} is equal to 1 if instruction i is completed on cluster j at time t. Variable x_{ia} is equal to 0 if instruction i is scheduled on cluster a and 1 otherwise. We use T to represent the makespan($C_{\max}$), and M is an arbitrary, conservative upper bound of T.

If y_{ijt}'s are zero-one variables, $\sum_{j=1}^{m} ty_{ijt}$ is exactly equal to completion time of instruction i. Thus, for each instruction i, the completion time of instruction i must be less than T, *i.e.*,

$$T \geq \sum_{t=1}^{M} \sum_{j=1}^{m} t.y_{ijt}$$

Each instruction must be completed at a unique time and before makespan T, thus

$$\forall\, 1 \leq i \leq n: \ \sum_{j=1}^{m} \sum_{t=1}^{M} y_{ijt} = 1$$

From their definitions, x_{ia} and y_{iat} are related by the following:

$$\sum_{t=1}^{M} y_{iat} = 1 - x_{ia}$$

If $(i, j) \in E$ and instruction i is scheduled on cluster a at time t, then the earliest time instruction j can be completed on cluster b is $t + l_{ijab}$:

$$\sum_{s=1}^{t} y_{ias} + x_{ia} \geq \sum_{s=1}^{t+p_j+l_{ijab}} y_{jbs}$$

At each time $1 \leq t \leq T$, there can be at most one instruction completed at that time per cluster, thus

$$\sum_{i=1}^{n} \sum_{s=t}^{t+p_i-1} y_{ijs} \leq 1$$

Therefor we have the following integer program:

$$\begin{array}{ll}
\text{minimize} & T \\
\text{subject to} & \forall\, 1 \leq i \leq n: \ T \geq \sum_{t=1}^{M} \sum_{j=1}^{m} t.y_{ijt} \\
& \forall\, 1 \leq i \leq n: \ \sum_{j=1}^{m} \sum_{t=1}^{M} y_{ijt} = 1 \\
& \forall\, 1 \leq i \leq n, 1 \leq a \leq m: \ \sum_{t=1}^{M} y_{iat} = 1 - x_{ia} \\
& \forall\, \text{instructions } i \rightarrow j, p_i \leq t \leq T - p_j - l_{ijab}, 1 \leq a, b \leq m: \ \sum_{s=1}^{t} y_{ias} + x_{ia} \geq \sum_{s=1}^{t+p_j+l_{ijab}} y_{jbs} \\
& \forall\, 1 \leq j \leq m, 1 \leq t \leq T: \ \sum_{i=1}^{n} \sum_{s=t}^{t+p_i-1} y_{ijs} \leq 1 \\
& \forall\, j, t: \ y_{ijt} \in \{0, 1\}, x_{ia} \in \{0, 1\}
\end{array}$$

This formulation can easily be extended to handle the preplaced instruction constraints in F. In order to do so, for a preplaced instruction i that is preplaced on tile k, the following equality holds: $\sum_{t=1}^{M} y_{ikt} = 1$.

The problem with the above ILP-formulation is that the number of variables and inequalities is large. In order to decrease the number of variables and constraints, we use interval-indexed formulation [10]. Define Interval l to be the interval $[(1+\epsilon)^{l-1}, (1+\epsilon)^{l}]$ and $\tau_l = (1+\epsilon)^{l}$. In the interval ILP-formulation, variable x_{ijl} is 1 if instruction i is completed on cluster j in interval l. With this formulation, we can trade off the size of the LP with the optimality of the output schedule. For example, by setting $\epsilon = 1$, the number of variables is $mn \lg(T)$ instead of mnT, and the algorithm guarantees that the schedule it finds will be within a factor of two of optimal.

4.2 Approximation Algorithm for Small Communication Delays

This section presents an approximation algorithm for clusters with uniform communication delays, where the maximum communication delay is less than the minimum instruction length. We denote this problem as $P|prec; p_i; l_{ijab};$ com[i, j]=τ; small $\tau|C_{\max}$.

Our algorithm is based on the $\frac{4}{3}$ approximation algorithm presented in [18] for the scheduling problem without precedence delays (delay(i, j)) and with small communication delays (com(a, b)). Their method is based on an LP-relaxation of the problem followed by an LP-rounding method. We first extend their result for unbounded number of clusters. Then, using the order of instructions in this schedule for unbounded number of clusters, we extend the previous $\frac{7}{3}$-approximation algorithm for bounded number of clusters in the presence of communication delay. We present the algorithm and sketch the proof of correctness for this more general case.

First we model the $P\infty|prec; p_i; l_{ijab};$ com[i, j] $= \tau$; small $\tau|C_{\max}$ problem with an integer program. The term $P\infty$ means there is no constraint on the number of tiles. In the following integer linear program, T corresponds to the makespan($C_{\max}$), t_i's correspond to starting time of instruction i, and x_{ij}'s indicate whether instruction j is scheduled in the same cluster as instructions i, with 0 representing yes and 1 representing no. As before p_i is the length of instruction i. $\Gamma^+(i)$ is the set of successors of node i, and $\Gamma^-(i)$ is the set of predecessors of node i.

The makespan is greater than the completion time of all instructions, thus for each instruction i, $t_i + p_i \le T$.

The fact that all communication delays are small along with the fact that there is at most one successor of i which is scheduled in the same tile as i immediately after i implies for at most one successor j of i, we have $t_i + p_i + \text{delay}_{ij} + \tau > t_j$ and for the others $t_i + p_i + \text{delay}_{ij} + \tau \le t_j$. Note that this works when communication delays are small, because in this case after completion of the instruction that is scheduled immediately after i all the other successors of i can be scheduled on the same tile as i, because their communication delay is smaller than the length of instruction j. Using x_{ij}, we can capture both cases by the following inequality: $t_i + p_i + \text{delay}_{ij} + x_{ij}\tau \le t_j$. Similarly, there is at most one predecessor of i that is scheduled in the same tile as i immediately before i. These two facts are captured by these inequalities: for all $(i, j) \in E(G)$, $\sum_{j \in \Gamma^+(i)} x_{ij} \ge \Gamma^+(i) - 1$ and $\sum_{j \in \Gamma^-(i)} x_{ji} \ge \Gamma^-(i) - 1$ (for more details please refer to [18]). We relax the integer constraints $x_{ij} \in \{0, 1\}$ to come up with the following linear program:

$$\begin{array}{ll}
\text{minimize} & T \\
\text{subject to} & \forall\, i \in V:\ t_i + p_i \le T \\
& \forall\, i \in V:\ t_i \ge 0 \\
& \forall\, (i, j) \in E(G):\ t_i + p_i + \text{delay}_{ij} + x_{ij}\tau \le t_j \\
& \forall\, (i, j) \in E(G):\ \sum_{j \in \Gamma^+(i)} x_{ij} \ge \Gamma^+(i) - 1 \\
& \forall\, (i, j) \in E(G):\ \sum_{j \in \Gamma^-(i)} x_{ji} \ge \Gamma^-(i) - 1 \\
& \forall\, 1 \le i, j \le |V|:\ 0 \le x_{ij} \le 1.
\end{array}$$

Now suppose the optimal solution of above linear program is $t_i^{opt}, T^{opt}, x_{ij}^{opt}$. From this solution, we compute integer values α_{ij}'s in this way: if $x_{ij}^{opt} < \frac{1}{2}$, then $\alpha_{ij} = 0$, otherwise $\alpha_{ij} = 1$. It is easy to see that for each i the number of zero α_{ij}'s is at most 1. We call j the favored successor of i iff $\alpha_{ij} = 0$.

The following list scheduling heuristic algorithm uses this optimal solution to schedule instructions. Let t_i^h denote the starting time of instruction i determined by this list scheduling.

Suppose R is the set of all available instructions at the current time. The algorithm increases time and at each time it decides which instructions to be scheduled on which clusters.

- For current time $\delta = 1$ to T do.
  1. Let S be the subset of R that can be processed at time δ.
  2. Let S_1 and S_2 be subsets of R such that $S = S_1 \cup S_2)$
     - $S_1 : (i \in S_1 \Leftrightarrow \forall j \in \Gamma^-(i) : t_j^h < \delta - 1) \Rightarrow$ all instructions $i \in S_1$ can be executed at time δ on new clusters.
     - $S_2 : (i \in S_2 \Leftrightarrow \exists! k \in \Gamma^-(i) : t_k^h = \delta - 1$ and $\forall j \in \Gamma^-(i)(j \neq k) : t_j^h < \delta - 1)$ (prec$(i) := k$) if prec(i_1)=prec(i_2)=…=prec$(i_t) = k$,

     Schedule all instructions in S_1 at time δ in a different new cluster and for instruction k in S_2 choose i_α such that x_{k,i_α}^{opt} is minimum and schedule i_α in the same cluster as k.

We claim that the output of the above scheduling algorithm is at most a factor of $\frac{4}{3}$ of the optimum solution.

The proof of above claim is based on the following lemma, which we state without proof:

Lemma 1. *For all* $(i, j) \in E(G)$*:* $p_i + d_{ij} + \alpha_{ij}\tau \leq \frac{4}{3}(p_i + d_{ij} + x_{ij}^{opt} c_{ij})$.

Based on above lemma and with an induction argument and using the fact that the solution for linear program is a lower bound on the solution for integer program, we can prove the following theorem.[2]

Theorem 1. *There exists a* $\frac{4}{3}$*-approximation algorithm for the following problem,* $P\infty|prec; p_i; l_{ijab};$ *com[i, j]* $= \tau;$ *small* $\tau|C_{\max}$.

Now, in order to solve the problem $P|prec; p_i; l_{ijab};$ com[i,j] = c small$|C_{\max}$, we first solve the problem with unbounded number of clusters: $P\infty|prec; p_i; l_{ijab};$ com[i,j] = c small$|C_{\max}$. We call a successor j of i the favored successor if in the scheduling with unbounded number of clusters j is scheduled in the same cluster as i immediately after i. As before, we set $\alpha_{ij} = 0$ if j is the favored successor of i and $\alpha_{ij} = 1$ otherwise. We use a variant of Graham's list-scheduling rule that takes communication delays into account. Suppose the completion time of instruction i in the schedule produced by the list scheduling algorithm is denoted by C_i^h. We define instruction j to be available at time t if $t \geq C_i^h + d_{ij} + (1 - \alpha_{ij})\tau$. Then the algorithm proceeds as follows: at each time we choose m arbitrary available instructions to be scheduled on these m clusters.

Using above theorem and the fact that the solution for unbounded number of clusters is a lower bound on the solution for m clusters (similar to the method in [18]) we can prove the approximation factor of the above algorithm is at most $\frac{7}{3}$. Thus we have the following theorem:

Theorem 2. *There exists a* $\frac{7}{3}$*-approximation algorithm for the following problem,* $P|prec; p_i; l_{ijab};$ *com[i, j]* $= \tau;$ *small*

[2] Due to space constraints, we omit the proofs. Interested readers can refer to the similar proofs in [18].

5 Constant Factor Approximation

This section presents a polynomial time constant factor approximation algorithm for the case of constant communciation delays $com(i,j) = \rho$ and constant number of tiles. However the running time is exponential in terms of ρ, the advantage of this algorithm is that the approximation factor does not depend on ρ. We present this algorithm in the case of unit execution time.

Definition 1. *For all $v \in V(G)$, height(v), the longest path from a source, is defined inductively as follows: if indegree(v) $= 0$, height(v) $= 1$. Otherwise, if $v_1, v_2, \ldots, v_k$ are predecessors of v: height(v) $= \max_{1\leq i\leq k}(height(v_i)+1)$.*

An edge $(u,v) \in E(G)$ in the precedence graph is called saturated iff height(v) $= k\rho$ for any integer k. A scheduling of instructions on m tiles is called saturated scheduling *iff for any saturated edge (u,v), starting time of instruction v is at least the completion time of u plus ρ.*

Theorem 3. *If the makespan of the optimum scheduling is T and the makespan of the optimum saturated scheduling is T', then $T' \leq 2T$.*

Proof. Consider the optimum scheduling OPT in which the makespan equals T and the starting time of instruction i is t_i. We construct a saturated scheduling for which the starting time of instruction i, namely t'_i, is at most $2t_i$. t'_i is computed as follows: $t'_i = t_i + \lfloor \frac{\text{height(i)}}{\rho} \rfloor \rho$. This new solution is saturated, because for a saturate edge (i,j), we know that $t_j \geq t_i + 1$. Now, $\text{height}(j) = k\rho$ implies $t'_j = t_j + k\rho \geq 1 + t_i + (k-1)\rho + \rho = 1 + t'_i + \rho$. It is clear that t'_i's is a feasable solution. Using this fact that $t_i \geq \text{height}(i)$, it turns out that $t'_i \leq t_i + \text{height(i)} \leq 2t_i$, thus the makespan of this new solution is at most $2T$.

Above theorem implies that it is sufficient to design a constant factor approximation algorithm for the optimum saturated scheduling problem. After removing all saturated edges, the graph is partitioned into some layers in each of them the longest path from any source node to any leaf is at most ρ. Scheduling each layer separately and putting ρ empty slots between them yeilds saturated scheduling. In order to schedule each layer, we observe that the longest path between any source and leaf node is at most ρ. Thus after putting a delay of length ρ on each edge, the length of the scheduling increases at most $\rho * \rho = \rho^2$. If the number of instructions is less than $m\rho^2$, then the optimum scheduling can be found using the brute-force method (for example, the integer programming formulation). Otherwise, the optimum makespan is at least $\frac{m\rho^2}{m} \geq \rho^2$, thus by adding ρ^2 to the makespan, we only lose a factor 1 in total makespan and still it is a 2-approximation. Scheduling this new precedence graph with ρ delay on each edge is easy because there is no communication delay and we can use the Graham list scheduling to get a 2-approximation algorithm**??**. From this discussion, we have the following theorem:

Theorem 4. *If ρ and m are bounded by constants, there exist an 8-approximation algorithm for the instruction scheduling problem in the case of uniform communication delays $Pm|prec; delay_{ij}; com_{ij} = \rho|C_{\max}$.*

Proof. From above discussion, we can easily prove the approximation factor 8. In fact, we lose a factor of 2 in Theorem 3, a factor of 2 by adding ρ^2 to the makespan between different layers, and a factor of 2 from using list scheduling.

6 Heuristic Algorithms

In this section, we give several heuristic bounds and algorithms to solve our general problem. In subsection 6.1, we describe several bounds that are useful as a priority function in scheduling. In subsection 6.2, we describe some algorithms using these bounds. In section 7.1, we will introduce a partitioning idea of the precedence graph and will see using the combination of all these ideas, we can design an algorithm which has all advantages of these heuristics. In other words, using several ideas from theoretical point of view, we design an algorithm whose performance and running time is reasonable. While describing algorithms, we study their theoretical justification in terms of worst-case performance gaurantees.

6.1 Heuristic Bounds

In this section, we design recursive algorithms to find the following: lower bounds or an estimation for minimum remaining time of scheduling after starting one instruction; and the earliest time that one instruction can be scheduled in any proper scheduling. We will use these bounds to design heuristic algorithms.

We first define some notations.

Definition 2. *Given a precedence graph $G(V, E)$, for a node $v \in V(G)$, children of v, namely $CH(v)$, the subgraph above v, namelay $GA(v)$, and the subgraph below v, namely $GB(v)$ are defined as follows:*

- $CH(v) = \{u \in V(G) | (v, u) \in E(G)\}$.
- $GA(v) = \{u \in V(G)|$ *there is a directed path from* u *to* v *in* $G\}$.
- $GB(v) = \{u \in V(G)|$ *there is a diretted path from* v *to* u *in* $G\}$.

In order to clarify algorithms, we will show their results on some sample precedence graphs. Three sample graphs are depicted in figure 2. We also consider a network of two tiles with communication delay 5 between them. In T_1, T_2 and T_3 all instructions are unit length, all pipeline delays ($delay(u, v)$) are 2, and the only preplaced node is the bottom node in T_3. T_1 is called the *fork graph* and T_2 and T_3 are called *join graph.*

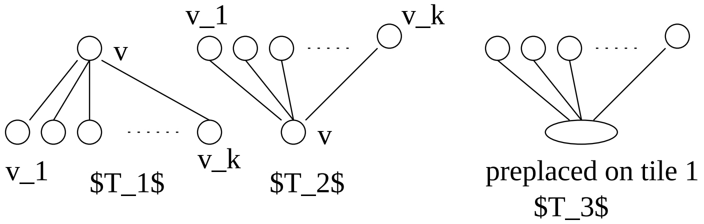

Fig. 2. Sample precedence graphs.

First, we describe a simple bound that works only for a symmetric network of tiles. Then we extend this bound so that it works for general networks of tiles, where the bound for each tile is computed separately and it is suitable to handle preplaced instructions as well.

A Bound for Symmetric Networks First we define the following property:

Definition 3. *Let C_i be the set of communication delays of a tile i to the other tiles. A* symmetric network *is a network of tiles satisfying the following property: for each pair of tiles a, b, $C_a = C_b$. Examples of symmetric networks include cycles and complete graphs.*

Given an instance of the problem for a symmetric network, our purpose is to find a lower bound on the remaining time of the scheduling after scheduling one instruction. In other words, we want to find a function mintime : $V \rightarrow N$ such that in any schedule, if v is started at time t_v the scheduling can not be completed before time $\text{mintime}(v) + t_v + p(v)$.

This bound mintime(v) is simply equal to $p(v)$, if v is a leaf. Otherwise, assume we have computed the mintime value for all children of node v, say $v_1, v_2, \ldots, v_k$. Now, assume we schedule v at time t on machine 1. The constraints of the rest of the scheduling are as follows: 1) All v_i's should be scheduled after time $t + p_v$, 2) instruction v_j can be scheduled on machine i after time $t + delay(v, v_j) + com(1, i) + p_v$ and 3) if instruction v_i is completed at time C_i, the scheduling may not be completed before time $C_i + \text{mintime}(v_i)$. Now consider $D_i = \text{mintime}(v_i)$ as the delivery time and $r_{ji} = delay(v, v_j) + com(1, i) + p_v$ as the release time of instruction j on machine i. From above discussion, mintime(v) is at least the minimum solution to the following problem: $Q|p_j; r_{ji}| \max(C_i + D_i)$ (Q for non-identical processors, r_{ji} for release date of j on machine i and the problem is to minimize completion time plus delivery time). In general, this problem is NP-complete (by a simple reduction from PARTITIONING [14]). Our next purpose is to solve the problem for special cases. In the following, we present a greedy algorithm for the case in which p_i's (instruction lengths) are equal and $delay(v, v_i) = delay(v, v_j)$ for all $1 \leq i, j \leq k$, as in pipeline delays.

Let $C = \{c_1, c_2, \ldots, c_m\}$ be the set of m communication delays from a cluster to all other clusters. For convenience, we use $p(v)$ instead of p_v for length of instruction v. Let small(C) denote the smallest member of the set C.

Algorithm GSM: Greedy algorithm for finding Simple Mintime

```
Input:   A precedence graph $G$.
         A network of tiles with communication delays between them.
         Pipeline delays between instructions.
         A node $v$.
Output:  mintime($v$).
begin
    if $v$ is a leaf, mintime($v$) = $p(v)$ and return
    let $v_1, \ldots, v_k$ be children of node $v$ ($CH(v)$), ordered in decreasing value of $\text{mintime}(v_i) + delay(v, v_i)$
    // We compute $O_1, \ldots, O_k$ and then mintime($v$) as follows:
        let $C = \{c_1, c_2, \ldots, c_m\}$ = the set of $m$ communication delays from a cluster to all other clusters.
        for $i = 1$ to $k$ do
            let $c_t = \text{small}(C)$
            $O_i = \text{mintime}(v_i) + delay(v, v_i) + p(v) + c_t$
            $c_t = c_t + p(v_i)$ (update $C$)
    mintime($v$) = $\max(O_1, O_2, \ldots, O_k)$. end
```

The result of this algorithm for graph T_1 is: $\text{mintime}(v_i) = p(v_i) = 1$ and

$$\text{mintime}(v) = \begin{array}{cl} k+3 & if k < 5 \\ \lfloor \frac{k-5}{2} \rfloor + 8 & if k \geq 5 \end{array}$$

and the result for T_1 is clearly $\text{mintime}(v) = 1$ and $\text{mintime}(v_i) = 3$.

Lemma 2. *If p_i's (instruction lengths) are the same and delay(v, v_i) = delay(v, v_j) for all $1 \leq i, j \leq k$, then the* **GSM** *gives a lower bound on the remaining time after scheduling instruction v.*

Proof. The output of the above algorithm is a scheduling of all instructions, $v_1, \ldots, v_k$, say OUTPUT. From above discussion, it is sufficient to prove that the output is the solution of the following minimization problem: $Q|r_{ji}; p_i| \max(C_i + \text{mintime}(i))$. Suppose for contradiction that in the optimum solution, OPT, instructions are scheduled in a different order. Then, consider the first place that OPT is different from OUTPUT in which instruction j is scheduled on machine i in OUTPUT and instruction j' is scheduled on machine i in OPT. It is not hard to see that switching instructions i and i' in OPT yield another feasible scheduling because pipeline delays are the same and release times of different instructions are the same and the cost of the output is not increased, because this place is the first place in which OPT and OUTPUT are not the same and the instructions are sorted in decreasing order of their mintime value in the OUTPUT. This proves that we can modify the solution OPT and don't increase the cost of the solution until we get OUTPUT.

This bound is more accurate than the longest path bound as a priority function to schedule instructions because it considers the width of the subgraph below a node i in the precedence graph in addition to its height. We will see that the list scheduling algorithm with this priority function is in fact a good approximation algorithm in several special cases.

More General Bounds There are two problems with the simple bound above. Firstly, it can be used only on symmetric networks. Secondly, it does not take into account *preplaced instructions*. Preplaced instructions are instructions that must be placed on a specific tile. They arise when an instruction needs to access a specific resource that is only available on a tile, *e.g.,* the memory back of a specific tile. These instructions force other close instructions to be scheduled on the same tile as well. In order to capture these more general problems and take into account different remaining time for different tiles, we need to separately compute the mintime for each potential tile. Now we want to find a tile-sensitive lower bound/estimation mintime(v, j) for instruction v and cluster j, as follows:

Definition 4. *For an instruction i and tile j, mintime(i, j) is a lower bound on the minimum remaining time of scheduling after starting instruciton i on cluster j.*

Similar to the previous discussion for the simple bound, it turns out that the mintime value can be captured as the optimum solution for the following scheduling problem: $Q|r_{vj}, p_v|max(C_v + D_{v,j})$ where delivery times $D_{v,j} = \text{mintime}(v, j)$ and release dates r_{vj} come from communication and pipeline delays (precise discussion is in the previous section and we will see more details in the following algorithms). Assuming the number of tiles and the out-degree of all vertices in the precedence graph are bounded by a constant, we describe how to find this lower bound in general.

Algorithm RTM: Recursive algorithm to find Tile-sensitive Mintime

- If v is a leaf, for all j mintime$(v, j) = p(v)$. If v is preplaced on cluster k, for all clusters j other than k, mintime$(v, j) = \infty$.
- Otherwise, let $v_1, \ldots, v_k$ be the successors of node v. Again, if v is preplaced on tile k, for all clusters j other than k, mintime$(v, j) = \infty$, otherwise we compute mintime(v, j) as follows:

- we consider all different tile assignments of $v_1, v_2, \ldots, v_k$ and for each assignment $\mathcal{A} = (j_1, j_2, \ldots, j_k)$, we compute value$(\mathcal{A})$ i.e., the minimum remaining time of v with this assignment, as follows:
  1. For each tile p, let $S_p = \{u_1, \ldots, u_t\}$ be the set of instructions assigned to p. Now, we compute the minimum value of $C_i +$ mintime(u_i, p) where C_i is the completion time of instruction u_i in a scheduling of instructions $\{u_1, \ldots, u_t\}$ on cluster p such that instruction u_i has a release date of com$(j, p)+$ delay(v, u_i) (In fact, this problem can be formalized as $1|r_{ij}; p_i|Max(C_i + w_i)$ and we can solve it by assuming t, k and m are constants) Let O_p be the output of this minimization problem.
- value$(\mathcal{A}) = \max_{i \in \{1, \ldots, m\}} O_i$

– mintime$(v, j) = \min_{\text{all assignments}\mathcal{A}}$ value$(\mathcal{A})$

Notice that the above algorithm is polynomial time only if the out-degree of every node in the precedence graph and the number of processors are constants. For sample graphs T_1 and T_2, the result of **RTM** is the same as **GSM**. For T_3, mintime$(v, 1) = 1$, mintime$(v, 2) = \infty$, mintime$(v_i, 1) = 4$ and mintime$(v_i, 2) = 9$.

Earliest Scheduling Time on a Tile In addition to the preplaced instructions, a good bound for the remaining execution time should take into account the tile placement of the instructions that have been already scheduled. For this purpose, we define another bound, earlytime(v, j). Earlytime(v, j) is the earliest time that instruction v can be scheduled on tile j given the instructions that have already been scheduled. This bound is computed very similar to mintime(v, j), except that the recursive computation is done top-down instead of bottom-up. For breviry, we omit repeating the details. Similar to **GSM** there is an algorithm called **GSE** and similar to **RTM** there is **RTE** algorithm for computing earlytime instead of mintime values. Notice that in order to find earlytime using **RTE**, it is necessary that the in-degree and the number of processors are bounded by a constant.

The output of **RTE** and **GSE** for T_1 is the following: $earlytime(v, j) = 1$ and earlytime$(v_i, j) = 4$ for $1 \leq j \leq 2$.

Now, we describe another property of this bound which is useful for justifying the final algorithm.

Definition 5. *Let G_ρ be the induced subgraph of G on the vertices with eartlytime value not greater than ρ, i.e., $V(G_\rho) = \{v \in V(G) | earlytime(v) \leq \rho\}$. Let eartlytime$(G) = \max\{eartlytime(v) | v \in V(G)\}$ and $G \backslash G_\rho$ is the the induced subgraph of G on vertices $V(G) - V(G_\rho)$.*

Theorem 5. *For any precedence graph G, eartlytime$(G) \geq$ eartlytime$(G \backslash G_\rho) + \rho$.*

Proof. We prove a stronger fact that is for all vertices $v \in V(G) - V(G_\rho)$, eartlytime$_G(v) \geq$ eartlytime$_{G \backslash G_\rho}(v) + \rho$. The proof is by induction. The argument is clear for all vertices without predecessor in $G \backslash G_\rho$, because eartlytime$_G(v) > \rho$ and eartlytime$_{G \backslash G_\rho}(v) = 0$. The eartlytime value of v in G and $G \backslash G_\rho$ are the minimum value of two minimization problems of the form $Q|r_i; p_i| \max(C_i + D_{ij})$ in which all parameters are the same except the release times, r_i's. In one problem $r_i =$ eartlytime$_G(v_i)$ and in the other one $r'_i =$ eartlytime$_{G \backslash G_\rho}(v_i)$. Using induction hypothesis, one can see that $r_i > r'_i + \rho$. Consider the optimum solution for $G \backslash G_\rho$. By staring all instrutions ρ time slots sooner, we get a feasable solution for G, because all realease times are at least ρ units smaller than that of $G \backslash G_\rho$. Thus, if the cost of the optimum solution of the problem for r'_i's is at least the cost of the optimum solution for r_i's plus ρ. It shows that eartlytime$_G(v) \geq$ eartlytime$_{G \backslash G_\rho}(v) + \rho$.

6.2 Heuristic Algorithms Based on Above Bounds

In this section, we use the above bounds to design heuristic algorithms.

A Simple Algorithm Using Simple Bound First, we define the following:

Definition 6. *An instruction v is* available *if all its predecessors have been scheduled. The availability time of instruction v on tile j, denoted by avtime(v, j), is the earliest time t that instruction v can be scheduled on tile j, given the existing state of the schedule and all precedence and communication delays.*

The first observation is that simple mintime(v) is a good priority function for instructions that can be scheduled at the any time. When there are two or more candidates to be scheduled in a time slot, we should select the one with greater mintime(v). This idea leads us to the following simple list scheduling algorithm. We visit each scheduling slot $(tile, time)$ in time order. If there are more than one instruction that can be scheduled in that slot, we select the one with the greatest mintime value. Thus, we have the following list scheduling algorithm:

Algorithm LSSM: List Scheduling algorithm using Simple Mintime
Input:

- A precedence graph G.
- A network of tiles with communication delays between them.
- Pipeline delays between instructions.

Output: A schedule of grpah G.

```
– $t = 0$
– Until there exists an unscheduled instruction do
    • $t = t + 1$
    • for all tiles $i$ do
        1. Let $A$ be the set of unscheduled instructions that can be scheduled at time $t$ on
           tile number $i$. Let instruction $k$ be the instruction with maximum mintime value
           in the set $A$. Schedule $k$ on cluster $i$ at time $t$ and update scheduling.
```

This algorithm works well when communication delays are small relative to execution time of the instructions. The well-known Lawler's algorithm [8] is a special case of **LSSM**. In fact for the case of out-tree precedence graphs, unit execution time and unit communication delays, it is proved that the output of Lawler's algorithm is at most $\frac{m-2}{2}$ plus the optimum solution [8]. It is not hard to extend that proof in the case of small communication delays. However, this algorithm doesn't work well when communication delays are larger and more crucial as in Raw machine. It is better not to schedule an instruction in an empty slot on an inconvenient tile and it might be better to wait and schedule it sometimes later on a more appropriate tile. Particularly, in the case of preplaced instructions in the bottom part of the precedence graph, these constraints don't propagate to the top part.

As for our sample graphs, **LSSM** works well for T_1. For T_2, one can see the problem of this algorithm when $k = 2$. **LSSM** schedules both v_1 and v_2 and the bottom node cannot be scheduled before time 8 whereas by scheduling v_1 and v_2 on tile 1 at time 1 and 2, the bottom node can be scheduled at time 5 on tile 1. **LSSM** has the same problem for T_3 as well.

Bottom-up Constraint Propagation The above algorithm works well when communication delay is small relative to the execution time of instructions. When communication delay is larger than execution time, as in a spatial architecture, this algorithm works poorly. Intuitively, the reason is that for such communication delay, often even if an instruction can be scheduled in an early slot, it may be better to schedule it in a later slot on a more convenient tile.

In this section, we describe a scheduling algorithm based on mintime(v, j) ,i.e., the tile-sensitive mintime. (mintime and avtime are defined in Definitions 4 and 6).

Definition 7. *Instruction v is* convenient *for tile j, if according to the current situation of scheduling $avtime(v, j) + mintime(v, j) \leq avtime(v, i) + mintime(v, i)$ for all tiles $1 \leq i \leq m$.*

The algorithm is as follows:

Algorithm LCTM: List scheduling of Convenient insturctions using Tile-sensitive Mintime
Input:

- A precedence graph G.
- A network of tiles with communication delays between them.
- Pipeline delays between instructions.

Output: A schedule of grpah G.

```
– Until there exists an unscheduled instruction do
    • Update availability times of instructions on tiles.
    • for all tiles $i$ do
        1. Let $A$ be the set of unscheduled instructions that are convenient for the tile
           number $i$. Let instruction $k$ be the instruction with maximum mintime value in
           the set $A$. Schedule $k$ on cluster $i$ at time avtime$(v, i)$ and update scheduling.
```

Note that as the algorithm proceeds there might be a situation in which there is an instruction v that is available but not convenient for tile j, but after scheduling other instructions on the other tiles, v becomes convenient for tile j, because availability time of instruction v will be changed on the other tiles. Our algorithm proceeds as follows: in each step and for each tile, we check all instructions that are convenient for this tile and select the instruction with the highest priority *i.e.,* the largest mintime value on this specific tile.

The algorithm **LCTM** has the same problem as **LSSM** for the sample graph, T_2, but it works better for T_3 because it considers the bottom preplaced node. In fact, if $k \leq 5$ it does not schedule any instruction on tile 2 as desired. As for large k's, it uses both tiles which yeilds an optimum solution.

With the above definition of convenient instructions and the strategy of selecting instructions with larger mintime value, the above algorithm takes into account the following considerations:

- It is better to schedule an instruction on a cluster with smaller mintime.
- If there are more than one candidate instructions for an empty slot, the instruction with higher mintime value has the higher priority.
- If it is impossible to schedule an instruction on a tile with smaller mintime value soon, it might be better to schedule this instruction on a less convenient tile much sooner.

Top-down Constraint Propagation The main flaw of the algorithm **LCTM** is that it doesn't take into account existing tile assignment of instructions. More precisely, we can describe the problem with the following example: if instruction 3 is the successor of instructions 1 and 2 and instruction 1 has been scheduled on tile a, then instruction 2 should not be scheduled on a tile far from a, because in that case the earliest time of scheduling of instruction 3 on any tile will be larger. In order to resolve this flaw in the algorithm, we define estimated time of scheduling when we schedule instruction v on the tile j.

We can find the estimated makespan for assigning instruction v to the tile j, denoted by *estimate*(v, j), as follows:

Algorithm FE: Find Estimated time

```
Input:  Precedence graph $G$.
        A network of tiles with communication delays between them.
        Pipeline delays between instructions.
        node $v$ and tile $j$.
        The current scheduling $S$, current mintime and earlytime values.
Output: estimate$(v, j)$.
  Assign instruction $v$ to the tile $j$ at its available time.
  **for** all unscheduled instructions $u \in GA(v)$ **do**
      compute earlytime$(u)$ according to the new scheduling using **Algorithm RTE**.
  **for** all unscheduled instructions $u \in GA(v)$, value$(u) = \min_{i\in\{1,\ldots,m\}}$ mintime$(u, i)$ + earlytime$(u, i)$
  estimate$(v, j) = \max_{\text{unscheduled instructions } u}$ value$(u)$
end
```

Remark 1. According to the current situation of scheduling, estimate(v, j) is the minimum makespan of any scheduling in which instruction v is scheduled on tile j.

With the above definition of estimated time of scheduling, it is easy to see that the tile j with the smallest value of estimate(v, j) is heuristicly the most convenient tile for scheduling v. Similar to the algorithm **LCTM**, we want to take into account this fact if we cannot schedule an instruction on a very convenient tile very soon, we would schedule it on a less convenient tile that is available much sooner. Thus, we define the new convenience property similar to the previous one as follows:

Definition 8. *Instruction v is* convenient *for tile j, if according to the current partial schedule avtime*(v, j) + *estimate*$(v, j) \leq$ *avtime*(v, i) + *estimate*(v, i) *for all tiles i.*

The new **algorithm LCTE** is the same as the **algorithm LCTM** except that we replace the old convenience definition with this new one. Note that estimate(v, j) should be recomputed after scheduling each instruction. Thus the running time of **LCTE** is more than that of **LCTM**. However, the algorithm only needs to dynamically update the estimate(v, j) attribute of available instructions. Suppose the running time of computing mintime(v, j) and earlytime$(v, j))$ is $O(M_v)$ and $O(E_v)$ respectively which are polynomials on the $|GB(v)$ and $|GA(v)|$. We find mintime(v, j) for each processor one time at the begining. In order to compute estimate(v, j), we need to update earlytime(u, i) for all vertices in the subgraph above the vertex v, namely $GA(v)$. Thus, computing estimate(v, j) takes $\sum_{u\in GA(v), 1\leq i\leq m} O(E_u)$ time. This is clearly a polynomial of n, if E_v is a polynomial. As mentioned before, in our method M_u and E_v are polynomials if in-degrees, out-degrees and the number of processors are constants. With the above discussion, we conclude that the last list scheduling algorithm **LCTE** with the definition of convenient processors and estimate function is not practically implementable when the number of tiles, in-degrees, or out-degrees are large. In the following, we will introduce a clustering idea by which we can solve all above problems and still get a good performance.

7 Final Algorithm

Here, we first present a layer partitioning idea by which we can use our heuristics on small graphs and then construct the whole solution by combining the solutions for all layers. We will prove that if the algorithm for small graphs has a good performance, this idea leads to an algorithm with a good performance for general graphs. Then we present our final algorithm which is an improvement of layer partitioning algorithm.

7.1 Layer Partitioning

In this subsection, we introduce a partitioning idea based on the earlytime bound. This algorithm is a warmup for our final algorithm. The idea is to schedule the graph G by scheduling iteratively the subgraphs G_ρ. It means that our algorithm first partition the graph into layers of earlytime smaller than ρ, schedule each layer separately and put all these scheduling together. The main point is to guarantee that the scheduling of layers are independent of each other. In the first algorithm, it is done by inserting a communication phase of $\mathcal{C}+1$ where $\mathcal{C}$ is the maximum communication delay between two instructions and tiles ($\mathcal{C}+1$ is sufficiently large and bounded by a constant).

We assume in this section that we are given a scheduling algorithm $\mathcal{A}$ for the *small graphs*, i.e., graphs of earlytime smaller than ρ. Then we discuss that if ρ is bounded by a constant, these small graphs have the desired properties to use the heuristic **LCTE**.

Algorithm PSSG: layer Partitioning and Scheduling Small Graphs

```
Input:   A precedence graph $G$, and a scheduling algorithm $\mathcal{A}$ for small graphs than $\rho + 1$.
         A network of tiles with communication delays between them.
         Pipeline delays between instructions.
Output:  A schedule of graph $G$.
begin
    let $H = G$
    let $i = 0$
    while $H \neq \emptyset$
        let $i = i + 1$
        let $G_i = H_\rho$
        let $H = H \backslash H_\rho$
    let $k = i$, $i = 1$
    while $i \leq k$
        Use algorithm $\mathcal{A}$ to schedule $G_i$
        Insert a communication phase of $\mathcal{C}$ time units.
end
```

Remark 2. According to the Theorem 5 and **algorithm PSSG**, it is straightforward that $\sum_{i=1}^{k} \text{earlytime}(G_i) \leq \text{earlytime}(G)$.

First, we observe that if we select $\rho \geq \mathcal{C}$ then the length of the optimum schedule for each G_i is at least $\rho \geq \mathcal{C}$. Now, we want to show that if we have an algorithm $\mathcal{A}$ with good performance for small graphs, **algorithm PSSG** has a a good performance as well. We formulate and prove it in the following theorem:

Theorem 6. *Let $L(G) = \sum_{v \in V(G)} p(v)$. If algorithm $\mathcal{A}$ can schedule each small graph G_i in at most $\alpha|\frac{L(G_i)}{m}| + \beta\rho + \gamma$earlytime$(G_i)$, then the* **PSSG** *can schedule any graph G in $\alpha|\frac{L(G)}{m}| + \beta\rho + (\beta+\gamma+1)$earlytime$(G)$, i.e., an $(\alpha + 2\beta + \gamma + 1)$-approximation.*

Proof. Suppose the graph G is partitioned into layers, $G_1, G_2, G_3, \ldots, G_k$. The makespan of the output of **PSSG** is less than or equal to $\mathcal{M} = \alpha \sum_{i=1}^{k} |\frac{L(G_i)}{m}| + (k-1)\mathcal{C} + k\beta\rho + \gamma \sum_{i=1}^{k} \text{earlytime}(G_i)$. First, $\sum_{i=1}^{k} |\frac{L(G_i)}{m}| = L(G)$ and according to the Remark 2, $\sum_{i=1}^{k} \text{earlytime}(G_i) \leq \text{earlytime}(G)$. Finally, $\frac{\text{earlytime(G)}}{\rho} + 1 \geq k$, thus $k\rho \leq \text{earlytime}(G) + \rho$ and $\rho \geq \mathcal{C}$. From all these inequalities, it turns out that the output of **PSSG** is less than

$$\alpha|\frac{L(G)}{m}| + \beta\rho + (\beta+\gamma+1)\text{earlytime}(G)$$

as desired. This is an $(\alpha + 2\beta + \gamma + 1)$-approximation, because $L(G)$, earlytime(G), and ρ are all lower bounds on the optimum solution.

We conclude this section by defining *small graphs* formally and proving their properties.

Definition 9. *The precedence graph H is* small *if and only if for all vertices $v \in V(H)$, earlytime$(v) \leq \rho$ and mintime$(v) \leq \rho$.*

Note that the above definition of small graphs is stronger than what we had in **PSSG** algorithm. Here, not only earlytime of every vertex is less than ρ, but also their mintime value is less than ρ. As we will see in the next chapter, it is not hard to partition the graph in small layers.

Remark 3. For any small graph H and any vertex $v \in V(H)$, in-degree and out-degree of v is at most ρ. The reason is that in-degree$(v) \leq$ earlytime(v) and out-degree$(v) \leq$ mintime(v).

The Remark 3 shows that by choosing an appropriate ρ (bounded by a constant), the running time of our heuristics in the previous section is polynomial, thus we can use **Algorithm LCTE** to schedule a small graph.

7.2 Partitioning, Clustering and List Scheduling of Clustered Graph

In this subsection, we present our final algorithm by combining all the ideas. From Section **??**, we had several heuristics: **LSSM** that is suitable for small communication delays and **LCTE** that is applicable for graphs with small in and out-degrees, e.g., small graphs. From Section 7.1, we know how to partition the graph into some layers of small graphs. In this section, we use this partitioning idea, However, instead of inserting $\mathcal{C}$ empty slots between layers, we use the output of small graph scheduling as a clustering, construct a new precedence graph from this clustering and then in the new precedence graph we use an algorithm like **LSSM** that is suitable for small communication delays. Although the provable worst-case performance ratio is the same as **PSSG**, the difference is clear from the practical results.

In order to formalize our algorithm, we need to define *clustering* and *clustered graph* formally.

Definition 10. *Given a precedence graph G of instructions, a* clustering *is defined as a family of partial scheduling of instructions, namely $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ where $\mathcal{S}$ is a partitioning of all vertices of G and all instructions in S_i should be scheduled in a prespecified order on the same tile, i.e., there is a specific order for the instructions of S_i and these instructions should be scheduled in this order.*

Given a clustering of G, we want to construct a new precedence graph by putting a vertex for each cluster and an edge between two clusters if there was an edge between a vertex in the source cluster to a vertex in the destination. Then the delay between two clusters can be found using the delays between their vertices as follows:

Definition 11. *Given two clusters $S = (s_1, s_2, \ldots, s_p)$ and $T = (t_1, t_2, \ldots, t_q)$ of G, if the completion time of instruction s_i is $C(s_i)$ and completion time of t_i in T is $C(t_i)$, then the delay between S and T, namely $delay(S,T)$, is defined as* $\max_{(s_i,t_j)\in E(G)}\{delay(s_i,t_j) - C(t_{j-1}) - C(s_p) + C(s_i)\}$.

Intutively, the above delay is the minimum needed delay between clusters S and T.

Definition 12. *Given a precedence graph G, communication and pipeline delays $delay(i,j)$ and $com(i,j)$, and a clustering, $S_1, S_2, \ldots, S_t$, the* clustered graph*, namely CG is defined as follows: $V(CG) = \{S_1, S_2, \ldots, S_t\}$ and $(S,T) \in E(CG)$ iff there exist vertices $i \in S$ and $j \in T$ such that $(i,j) \in E(G)$ and $delay(S,T)$ is defined in Definition 11.*

Similar to the definition of G_ρ, we define G^ρ as follows: $V(G^\rho) = \{v \in V(G)|\text{mintime}(v) \leq \rho\}$.

This is our final algorithm:

Algorithm PCLC: layer Partitioning, Clustering and List scheduling of Clustered graph

```
Input:   A precedence graph $G$, and a scheduling algorithm $\mathcal{A}$(LCTE) for graphs of mintime smaller than $\rho + 1$.
         A network of tiles with communication delays between them.
         Pipeline delays between instructions.
Output:  A schedule of graph $G$.
begin
  let $H = G$
  let $i = 0$
  while $H \neq \emptyset$
      let $L = H_\rho$
      while$L \neq \emptyset$
          let $i = i + 1$
          let $G_i = L^\rho$
      let $H = H \backslash H_\rho$
  let $k = i$, $i = 1$
  while $i \leq k$
      Use algorithm $\mathcal{A}$(LCTE) to schedule $G_i$
      let $S_{ij}$ be the set of vertices in $G_i$ that are scheduled on tile $j$ in the output of $\mathcal{A}$(LCTE).
  let CG be the clustered graph corresponding to the clustering of $S_i$'s.
  Scheduling $CG$ using LSSM
  //any scheduling suitable for small communication delays can be used here, like ILP-based ones.
end
```

Note that since $\rho \geq \mathcal{C}$, there exist a cluster in each layer with execution time more than $\mathcal{C}$. Thus, using **LSSM** is reasonable. Although we can not prove better performance guarantee for **PCLC** compared to **PSSG**, it is clear the output of **PCLC** is better than that of **PSSG** and using it in practice is more reasonable.

We conclude this section by emphasizing that **PCLC** is based on the following facts: the heuristic algorithm **LCTE** algorithm for small graphs, a suitable algorithm (like the ILP-based algorithm or **LSSM**) for small communication delays, a layer partitioning method and schedule each layer separately and finally scheduling the clustered graph.

8 Experimental Results

This section presents results of the algorithms described in Section 6. The algorithms are implemented in Rawcc, the instruction level parallelizing compiler for the Raw machine, which is a spatial architecture.

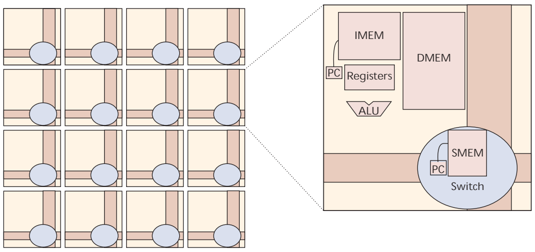

Fig. 3. The Raw machine.

Experimental setup Experiments are performed on Beetle, a validated, cycle-accurate simulator of the Raw machine. Figure 3 shows a picture of the Raw machine [?]. The Raw machine comprises tiles organized in a two dimensional mesh. The actual Raw prototype has 16 tiles in a 4x4 mesh. Each tile has its own instruction memory, processor pipeline, ALUs, data memory, and 28 registers. Its instruction set is based on Mips R4000.

The tiles communicate with each other via point-to-point, mesh networks. In additional to a traditional, wormhole hole dynamic network, Raw has a programmable, compiler-controlled *static network* that is used to route scalar values between the register file/ALUs on different tiles. Network ports are register mapped (the count of 28 registers do not include these network ports). Latency on the static network is three cycles for two neighboring tiles; each additional hop takes an extra cycle of latency.

Our scheduling algorithms are implemented in Rawcc, the instruction level parallelizing compiler for the Raw machine. Rawcc takes a sequential C or Fortran program and parallelizes it across the Raw tiles. Each program is divided into one or more scheduling traces. For each trace, Rawcc constructs the data precedence graph for the instructions and performs space-time scheduling on the graph. After space-time scheduling is performed on all the scheduling regions, the code on each tile is run separately through a traditional register allocation [?].

Rawcc employs congruence transformation and analysis to increase and analyze the predictability of memory references [3, 12]. This analysis creates memory reference instructions that must be placed on specific tiles. For dense matrix loops, the congruence pass usually unrolls the loops by the number of clusters or tiles. This unrolling also increases the size of the scheduling regions, so that no additional unrolling is necessary to expose parallelism.

Sources of benchmarks include the Raw benchmark suite (Jacobi, Life) [2], Spec95 (Swim), and Nasa7 of Spec92 (Cholesky, Vpenta, and Mxm). Sha is an implementation of Secure Hash Algorithm. Fpppp-kernel is the inner loop of fpppp in Spec95 that accounts for 50% of the execution time. Some problem sizes have been changed to cut down on simulation time, but they do not effect the results qualitatively.

Comparison We compare two algorithms in Section 6 with two existing space-time scheduling algorithms. The first existing algorithm is BUG, one of the earliest space-time scheduling for instruction level parallelism. The second existing algorithm is the algorithm implemented in Rawcc described in [13]. This algorithm uses a three-phase algorithm to map instructions to tiles, then a separate list scheduling algorithm to order assign instructions to time slot. Our new algorithms are implemented in place of the space scheduling algorithms of Rawcc – Rawcc does its own time scheduling via list scheduling.

		Original Rawcc				BUG			
Benchmark	Seq. Time	N=2	N=4	N=8	N=16	N=2	N=4	N=8	N=16
Jacobi	238663	1.140	2.152	3.815	6.587	1.237	2.159	3.771	5.655
Life	2043360	1.201	2.309	4.085	6.939	1.312	2.070	3.308	5.353
Cholesky	1835081	0.975	2.164	3.455	4.889	1.110	1.551	2.733	3.523
Vpenta	6490961	1.210	2.109	4.150	6.343	1.632	2.464	3.938	6.161
Mxm	1570607	1.145	2.086	4.056	7.186	1.612	2.491	3.683	5.066
Swim	83132395	1.224	1.831	3.251	5.658	1.360	2.245	2.837	4.868
Sha	955237	1.180	1.580	1.704	2.132	1.470	1.540	1.726	1.967
Fpppp-kernel	150395	1.479	2.675	5.072	6.073	1.751	2.563	3.613	4.113

Table 1. Speedup of Rawcc's original algorithm and BUG. "Seq. Time" is the sequential run-time of each benchmark on a single tile. All speedups are measured relative to execution time on one tile.

	Mintime-based				Estimate-based			
Benchmark	N=2	N=4	N=8	N=16	N=2	N=4	N=8	N=16
Jacobi	1.323	2.318	4.725	7.794	1.128	2.089	3.906	7.131
Life	1.314	2.276	4.863	6.749	1.260	2.331	4.406	8.061
Cholesky	1.408	2.012	3.624	4.953	1.120	1.869	3.123	4.071
Vpenta	1.520	2.306	4.152	6.860	1.480	2.471	4.270	6.573
Mxm	1.601	2.197	4.221	8.500	1.527	2.193	4.100	7.478
Swim	1.536	2.160	4.446	7.930	1.425	2.108	4.101	7.260
Sha	1.051	0.945	1.032	1.034	1.333	1.346	1.552	1.575
Fpppp-kernel	2.083	2.005	2.001	2.023	1.577	2.698	3.357	4.779

Table 2. Speedup of *mintime*-based and *estimate*-based algorithms.

Table 1 shows the performance of Rawcc with the existing algorithms, and Table 2 presents the performance of our *mintime*-based and *estimate*-based algorithms. Figure 4 displays the relative performance of each algorithm on 16 tiles. We note in passing that of the two existing algorithms, Rawcc's original algorithm performs better than BUG. We find that BUG performs relatively poorly for two reasons: first, it does not take into account for constraints from preplaced instructions very well; second, it tends to traverse the graph depth first while making greedy decisions. The consequence of the depth first traversal is that the traversal tends to expose fine-grained parallelism before coarser-grained parallelism. As a result, unnecessary communication is often introduced. BUG was designed and evaluated on a spatial architecture whose functional units are connected via a single-cycle crossbar. It is less suitable for architectures with more costly communication, such as Raw.

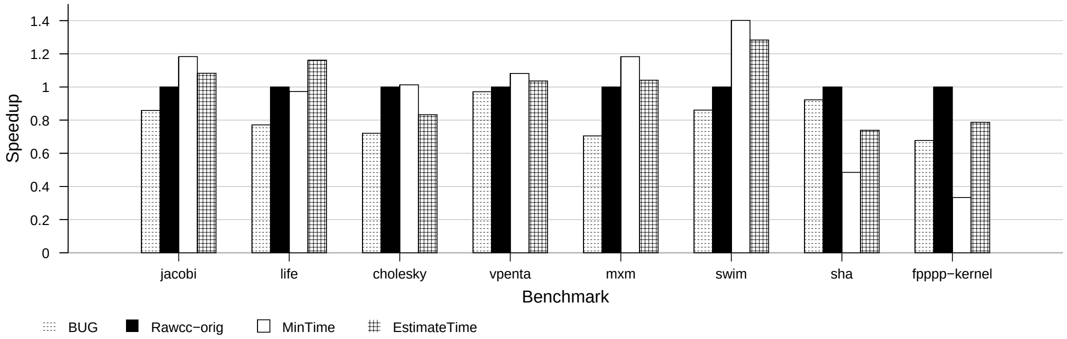

Fig. 4. Performance improvement of the algorithms on 16 tiles, relative to Original Rawcc.

For our new algorithms, we focus our attention on their performance relative to Rawcc's original algorithm. Results show that these new algorithms perform competitively overall, with an average speedup of 5.7 to 5.8 on 16 tiles. Closer examination reveal that the results can be roughly divided into two classes. Dense matrix applications include Jacobi, Life, Cholesky, Vpenta, Mxm, and Swim. These applications have unrolled loops that are highly regular. In addition, congruence analysis is able to identify many preplaced memory instructions. For these applications, preplaced instructions give very good guides about how instructions should be partitioned. The *mintime* and *estimate* based algorithms are designed to take advantage of this information, and they successfully make use of this information and achieve better results than the original algorithm. For these benchmarks, average improvements of our algorithms over the Rawss's original algorithm are 25% and 14%, respectively. Sha and Fpppp-kernel, however, are less regular and have no preplaced instructions. For these benchmarks, our algorithms perform worse. However, note that the *estimate*-based algorithm, which uses a relatively accurate completion time estimates that propagates scheduling information both upwards and downwards, is able to perform more than 75% better than *mintime* on the irregular benchmarks, at a cost of only 10% worse for the regular benchmarks.

9 Conclusion and Future Works

In this paper, we discussed different methods for solving the general instruction scheduling problem in a network of tiles.

We discussed two different integer programming approaches, one of which is suitable to formalize our general problem and the other one with smaller size gives an approximation algorithm for a special case. In order to decrease the number of variables and constraints in the first formulation, one potential research is using ideas in [24]. Using LP-relaxation and LP-rounding method in order to design approximation algorithm based on this LP is an interesting theoretical problem that needs understanding the LP-relaxation better. The second formulation only works for the case of fully connected network of tiles and small communication delays.

We described several heuristic bounds and algorithms based on them. Theoretical analysis of these algorithms would be a nice theoretical problem. No constant-factor approximation algorithm is known for the case of large communication delays. Finding a constant factor approximation algorithm in this case or proving a lower bound of approximation is also an interesting theory problem. In the case of small communication delays the lower bound of $\frac{7}{6}$ approximation factor is known but the best known approximation factor is $\frac{4}{3}$.

We implement these algorithms separately. Using these bounds along with other previous algorithms, such as local optimization, might give us better practical results. Another application of aforementioned bounds is for branch and bound backtracking method and we can use them as a branch hint for searching state space in a more appropriate order and use these bounds as an estimation or a lower bound for the best solution that we can get from the current assignment.

Spatial architectures are becoming increasingly important because they are a natural way to address the lack of scalability of wire delays with technology. Therefore, we believe that it is important to have a better understanding of the problem both theoretically and practically. This paper contributes toward this goal.

References

1. Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *ISCA*, pages 248–259, 2000.
2. Jonathan Babb, Matthew Frank, Victor Lee, Elliot Waingold, Rajeev Barua, Michael Taylor, Jang Kim, Srikrishna Devabhaktuni, and Anant Agarwal. The raw benchmark suite: Computation structures for general purpose computing. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 1997.
3. Rajeev Barua, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Maps: A Compiler-Managed Memory System for Raw Machines. In *Proceedings of the 26th International Symposium on Computer Architecture*, Atlanta, GA, May 1999.
4. Giuseppe Desoli. Instruction assignment for clustered vliw dsp compilers: a new approach. Technical Report HPL-98-13, Hewlett Packard Laboratories, January 1998.
5. John R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* PhD thesis, Yale University, 1985.
6. D.W. Engels, J. Feldman, D. R. Karger, and M. Ruhl. Parallel processor scheduling with delay constraints. In *ACM-SIAM Symposium on Discrete Algorithms (SODA '01)*. Jan. 2001.
7. R.L Graham, E.L. Lawler, J.K. Lenstra, and K Rinnooy. Optimization and approximation in deterministic sequencing and scheduling. *Annals of Discrete Mathematics*, 5(14):287–326, 1979.
8. F. Guinand, C. Rapine, and D. Trystram. Worst case analysis of lawler's algorithm for scheduling trees with communication delays. *IEEE Transaction On Parallel and Distributed Systems*, 8(10):1085–1086, 1997.
9. L. A. Hall, A. S. Schulz, D. B. Shmoys, and J. Wein. Scheduling to minimize average completion time: Off-line and on-line approximation. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*. Jan. 1996.
10. Leslie A. Hall, David B. Shmoys, and Joel Wein. Scheduling to minimize average completion time: Off-line and on-line algorithms. In *SODA*. Jan. 1996.
11. Ho-Seop Kim and James E. Smith. An Instruction Set and Microarchitecture for Instruction Level Distributed Processing. In *Proceedings of the 29th International Symposium on Computer Architecture*, Anchorage, AL, May 2002.
12. Sam Larsen and Saman Amarasinghe. Increasing and detecting memory address congruence. In *Proceedings of 11th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Charlottesville, VA, September 2002.
13. Walter Lee, Rajeev Barua, Matthew Frank, Devabhatuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. In *Proceedings of the Eighth ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, San Jose, CA, October 1998.
14. J. K. Lenstra, A. Rinnooy, and P. Brucker. Complexity of machine scheudling problems. *Annals of Discete mathematics*, 1(2):343–362, 1977.

15. Rainer Leupers. Instruction scheduling for clustered vliw dsps, 2000.
16. P. Lowney, S. Freudenberger, T. Karzes, W. Lichtenstein, R. Nix, J. O'Donnell, and J. Ruttenberg. The Multiflow Trace Scheduling Compiler. In *Journal of Supercomputing*, pages 51–142, January 1993.
17. Rolf R. H. Mhring, Markus W. Schffter, and Andreas S. Schulz. Scheduling jobs with communication delays - using infeasible solutions for approximation. In *Proceedings of the Fourth European Symposium on Algorithms, Lecture Notes in Computer Science, Springer-Verlag*. Sep. 1996.
18. A. Munier and J.C. Konig. A heuristic for a scheduling problem with communication delays. *Oper. Res. 45*, 45(1):145–147, 1997.
19. A. Munier, M. Queyranne, and A.S. Schulz. Approximation bounds for a general class of precedence constrained parallel machine scheduling problems. In *IPCO*, pages 404–413. May 1998.
20. R. Nagarajan, K. Sankaralingam, D. Burger, and S. Keckler. A design space evaluation of grid processor architectures, 2001.
21. Emre Ozer, Sanjeev Banerjia, and Thomas M. Conte. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *International Symposium on Microarchitecture*, pages 308–315, 1998.
22. Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989. In the series, Research Monographs in Parallel and Distributed Computing.
23. Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring It All to Software: Raw Machines. *IEEE Computer*, 30(9):86–93, September 1997. Also available as MIT-LCS-TR-709.
24. K. Wilken, J. Liu, and M. Heffernan. Optimal instruction scheduling using integer programming. In *PLDI*, pages 121–133. 2000.

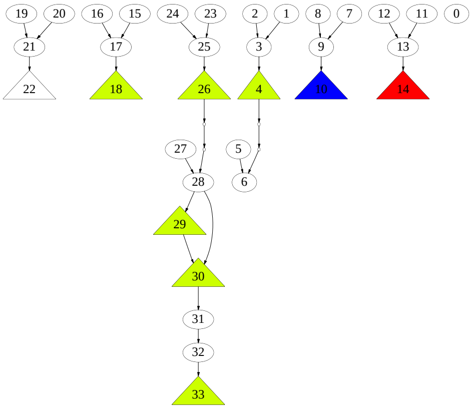

Fig. 5. Please ignore this picture. Without I couldn't generate a valid pdf file!!!