List Scheduling in the Presence of Branches: A Theoretical Evaluation

Franco Gasperoni[1] and Uwe Schwiegelshohn[2]

[1] Télécom Paris-ENST, 46, rue Barrault, 75634 Paris, Cedex 13, France,
gasperoni@inf.enst.fr
[2] Computer Engineering Institute, University Dortmund, 44221 Dortmund, Germany,
uwe@carla.e-technik.uni-dortmund.de

Abstract. The extraction of operation level parallelism from sequential code has become an important problem in compiler research due to the proliferation of superscalar and VLIW architectures. This problem becomes especially hard for code containing a large number of conditional branches. In this paper we extend previous work on straight line code scheduling by looking at branching task systems whose control flow graph is acyclic. First, we define an optimality measure based on the probability of the various execution paths. Then, we apply a list scheduling algorithm to these systems and derive a worst case performance guarantee for this method. Finally, we show that there are branching task systems for which this bound is almost tight.

1 Introduction

With the wide spread use of microprocessors capable of executing multiple operations per cycle, extraction of fine grain parallelism from sequential programs is regaining momentum. This concept dates back to the 60s where machines like the IBM 360/91 or the CDC 6600 provided hardware mechanisms to exploit operation level parallelism automatically. Due to the frequency of conditional jumps in system code, this purely hardware based approach rarely exceeded speedup factors of two or three [9].

In the early 80s Fisher developed an innovative compilation technique called trace scheduling, that went beyond the conditional jump barrier in its quest to extract parallelism. Fisher subsequently introduced an architectural paradigm, termed VLIW, which by employing a trace scheduling compiler was claimed to provide high performance at low cost [4].

Today all systems that boost performance by exploiting fine grain parallelism combine multiple functional units/single thread of control machines with sophisticated compilers. Several new compilation algorithms such as percolation scheduling [1] or region scheduling [7] have generalized the ideas behind trace scheduling for non numerical programs.

However, for most of these techniques the actual motion of operations beyond conditional branches has been given priority over mechanisms for the selection of the operations to move. Trace scheduling is an exception as operations from the execution path with highest probability are always chosen to be the subject of a transformation. But to date no theoretical performance evaluation has been presented for this or any other scheduling heuristic dealing with conditional branches.

This is in contrast with the large body of theoretical results known for scheduling problems in the absence of conditional operations. In general these problems are NP-hard [5]. Frequently, a classical heuristic called list scheduling is employed to guarantee close to optimum performance. There, operations are first ordered in a priority list. Instructions are then constructed in a top down fashion by selecting operations from this priority list and moving them to the instruction. This procedure guarantees in general a final running time of at most $(2 - 1/m)$ times the optimum where m is the number of operations that can be executed concurrently [2].

In this paper we show that a generalization of the list scheduling heuristic in the presence of branches limits the deviation from the optimum to the factor $2 - 1/m + (1 - 1/m) \cdot 1/2 \cdot \lceil \log_2 m \rceil$.

The remainder of the paper is structured as follows. Section 2 introduces branching task systems which formalize the notion of programs containing conditionals. Section 3 explains our machine model and defines schedules containing branches. Next, Section 4 defines optimality while Section 5 explains how list scheduling has been extended in the presence of branches and gives its new performance guarantee. Finally, Section 6 gives an example which shows that the performance bound established in Section 5 is almost tight.

2 Branching Task System

A conventional task system comprises a set of operations O and a precedence relation $\prec$ on O. The operations must be executed so that the dependence constraints dictated by $\prec$ are respected in the final schedule [2]. To formalize the notion of an acyclic program containing branches we extend this definition by adding conditionals, that is operations whose outcome determines the next set of operations to execute.

Definition1 Branching Task System. A triple $T = (O, G, \prec)$ consisting of a set of operations O, a control flow graph G, and a dependence relation $\prec$ is called a branching task system if the following conditions are valid:

1. G is an acyclic single entry, single exit di-graph with vertex set $O \cup \{\xi, \zeta\}$ such that no operation in G has out-degree greater than 2. Operations with out-degree 2 are called conditionals. ξ is G's entry and has out-degree one, while ζ is G's exit. A path from the entry ξ to the exit ζ is called an execution path of T. The set of all such paths is denoted $\mathcal{P}(T)$. For any $op \in O$, $\mathcal{P}(op, T)$ denotes the set of execution paths traversing op.
2. For each execution path P, $\prec$ is a partial order on P compatible with its linear ordering, that is $op \prec op'$ only if op precedes op' in P.

An example of a branching task system T is given in Figures 1 and 2. Figure 1 gives the low-level code generated for a procedure computing the square roots of the polynomial $\mathrm{a} \cdot x^2 + \mathrm{b} \cdot x + \mathrm{c}$ with $\mathrm{a} \neq 0$.

The precedence relation of the branching task is given in Figure 2. The relation is portrayed in the form of a dependence graph where a solid edge from an operation op to an operation op' denotes $op \prec op'$. Note that output dependencies between operations on different execution paths, as e.g. between $op15$ and $op14$ are realized by introducing

```
procedure Poly_Roots (a,b,c: incoming; x1,x2,roots: outgoing) is
   r1 := b * b;              -- op1
   r2 := 4 * a;              -- op2
   r3 := c * r2;             -- op3
   r4 := r1 - r3;            -- op4
   if r4 >= 0.0 then         -- cj1
      r5 := 2 * a;           -- op5
      if r4 = 0.0 then       -- cj2
         r6 := -b;           -- op6
         x1 := r6 / r5;      -- op7
         roots := 1;         -- op8
      else
         r7 := sqrt (r4);    -- op9
         r8 := r7 - b;       -- op10
         x1 := r8 / r5;      -- op11
         r9 := -r7 - b;      -- op12
         x2 := r9 / r5;      -- op13
         roots := 2;         -- op14
      end if;
   else
      roots := 0;            -- op15
   end if;
end Poly_Roots;
```

Fig. 1. Code to compute the roots of a degree 2 polynomial.

static dependencies from appropriate conditional branches to these operations. There are various other possibilities to address this problem as e.g. renaming. However, a discussion of these issues is beyond the scope of this paper.

The control flow can also be extracted from Figure 2. Within each block the control flow is defined by the numerical order of the operation with the conditional operation being the last. Between blocks the control flow is represented by dashed edges. Vertex ξ is the single predecessor of op1 while vertex ζ is the successor of operations op8, op14, and op15.

Note that time is considered to be a discrete, rather than a continuous entity. Further it is assumed that every operation requires a single unit of time to execute. The use of multi cycle operations is more thoroughly discussed in [6]. In general it can be stated that the bound derived in this paper is no longer valid when operations have arbitrary durations. In this case list schedules may yield arbitrarily poor performance.

For the sake of simplicity we will also require that the control flow graph $G - \{\zeta\}$ is a tree although the presented results apply to arbitrary control flow graphs as well. For more general branching task systems it may be necessary to sacrifice space performance in order to obtain even a modest speedup [6]. More specifically, a speedup as little as 2 may require exponential code size. Thus for branching tasks, whose control flow graph is not a tree, time and space performance can be antipodal.

This phenomenon can be intuitively explained by considering the number of execution paths of a control flow graph. If the control flow graph is a tree, the overall

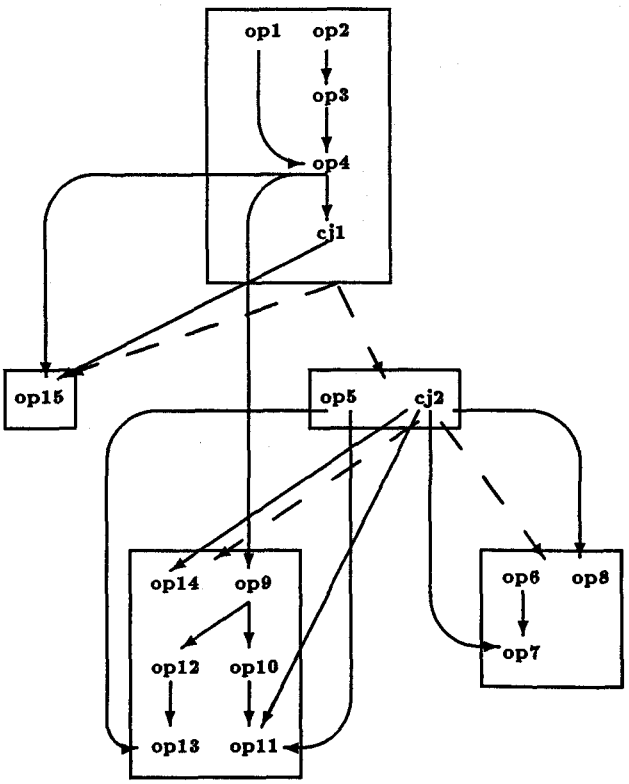

Fig. 2. A branching task system

number of execution paths is equal to the number of leaf operations in the graph. However, in an arbitrary control flow graph with n operations there can be close to 2^n execution paths.

3 Machine Model and Branching Schedules

Our machine model is capable of executing m arbitrary operations per time unit. The set of operations executed in a given time instant is called an instruction. When an instruction I contains $1 \leq k \leq m$ conditionals, these are arranged to form a decision tree with $k+1$ outgoing branches that specifies which instruction must be executed next. This machine model is inspired by the branching paradigm of Karplus & Nicolau [8] and Ebcioğlu [3]. A formal definition is given below:

Definition2 Branching Schedule. A branching schedule σ of a comprises a set of instructions $\mathcal{I}(\sigma)$ and a control flow graph $G(\sigma)$.

1. An instruction is a set of operations. If every instruction contains at most m operations, σ is said to be an m-schedule.

2. $G(\sigma)$ is an acyclic single entry, single exit di-graph with vertex set $\mathcal{I}(\sigma) \cup \{\xi, \zeta\}$. An instruction I has out-degree k iff it contains $k-1$ conditionals. A path from the entry ξ to the exit ζ is called an execution path of σ. The set of all such execution paths is denoted $\mathcal{P}(\sigma)$. The length $d(P, \sigma)$ of an execution path $P \in \mathcal{P}(\sigma)$ is the number of instructions traversed by P.

As we have assumed that the control flow graph of a branching task is a tree, the control flow graph of a branching schedule will also be a tree.

Definition 3 Admissibility. Let $T = (O, G, \prec)$ be a branching task system and σ be a branching schedule. σ is said to be admissible for T iff the following constraints are met:

1. **Branching:** There is a bijective function ϕ mapping $\mathcal{P}(T)$ into $\mathcal{P}(\sigma)$ such that for all $P \in \mathcal{P}(T)$ the instructions traversed by $\phi(P)$ in σ contain all the operations traversed by P in T. Furthermore, if conditional cj is an ancestor of conditional cj' in T then either cj and cj' are scheduled in the same instruction in σ or cj is scheduled in an instruction which is an ancestor of the instruction where cj' is scheduled.
2. **Dependencies:** For any pair of operations $op, op' \in O$ with $op \prec op'$, $op \in I$ and $op' \in I'$ follows that I is a proper ancestor of I'.

Two branching 3-schedules admissible for the branching task system of Figure 2 are given in Figure 3. Note that an operation op need not be scheduled in a single instruction. For instance in the right schedule, operation op5 is scheduled both in I6 and I8.

If an operation op is scheduled in an instruction I but there exists a path P traversing I such that $P \notin \mathcal{P}(op, T)$, we say that op is speculatively scheduled in I. This means that the execution of op will not be useful if execution path P is taken. For instance in the left schedule operation op5 is scheduled speculatively in I1.

4 Optimality Definition

Depending on the outcome of the conditionals contained in a branching task system T, the actual path followed during execution varies. Consequently an admissible schedule for T may require different completion times for different executions. Therefore for two execution paths P_1, P_2 of T and two admissible m-schedules σ, σ' for T, $d(\phi_\sigma(P_1), \sigma) < d(\phi_{\sigma'}(P_1), \sigma')$ and $d(\phi_\sigma(P_2), \sigma) > d(\phi_{\sigma'}(P_2), \sigma')$ is possible. Consequently, a weight function is typically employed to define the average execution time of a branching schedule.

Definition 4 Weight Function. Let T be a branching task and G its control flow graph. A function w mapping $\mathcal{P}(T)$ into the non-negative reals is called a weight function for T if and only if

$$\sum_{P \in \mathcal{P}(T)} w(P) = 1.$$

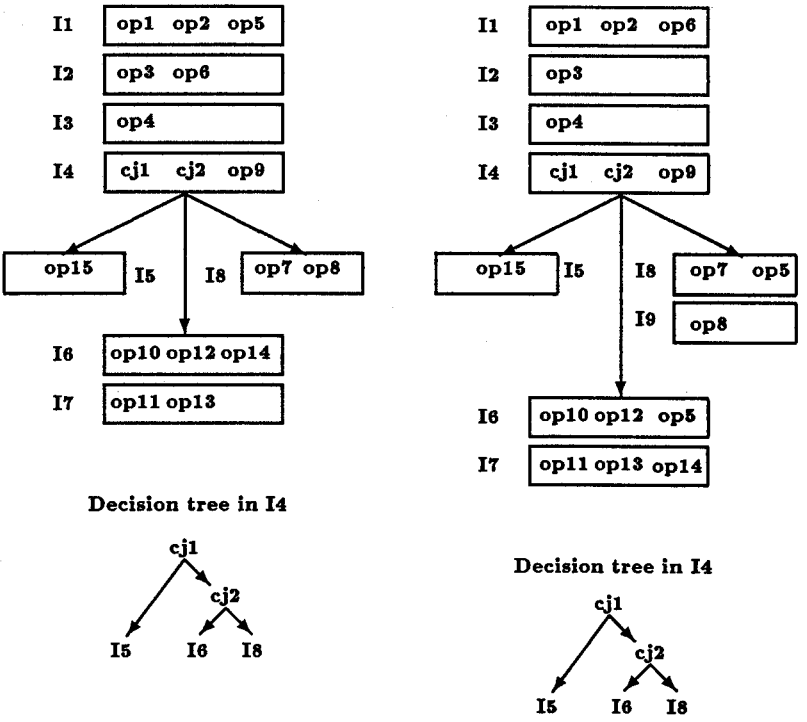

Fig. 3. Two branching schedules.

When $G - \{\zeta\}$ is a tree there exists a function $\hat{w}$ mapping the edges of G into the non-negative reals such that:

$$\forall\, P = (e_1, e_2, \cdots, e_k) \in \mathcal{P}(T) \qquad w(P) = \prod_{i=1}^{k} \hat{w}(e_i)$$

Definition 5 Optimality. The weighted average running time $t(\sigma)$, is defined as

$$t(\sigma) = \sum_{P \in \mathcal{P}(T)} w(P) \cdot \sigma(P).$$

σ is said to be m-optimum for w iff there exists no admissible m-schedule σ' for T such that $t(\sigma') < t(\sigma)$.

Usually weights are taken to be execution path probabilities. If the probability to take the `'if r4 >= 0.0 then'` branch is `0.9` and the probability to take the `'if r4 = 0.0 then'` branch is `0.2` in Figure 2 then the average running time of the schedule on the left of Figure 3 is `6.72`, whereas the average running time of the schedule on the right of Figure 3 is `6.9`.

5 Optimum Performance Approximation

As pointed out in the introduction, the problem of generating optimum m-schedules for tasks without conditionals is NP-complete. In these cases the approach is frequently taken to devise simple heuristics that always produce a result within a constant factor from the optimum. By introducing a new list scheduling heuristic we extend Graham's result on the performance of list scheduling algorithms [2] to branching task systems.

When generating instructions for branching or straight line codes, frequently several operations are available for execution in the same instruction. In the case where such operations cannot all be executed together a selection criterion must be employed. For a straight line task system a random choice guarantees a bound of $2-1/m$ from the optimum. In the presence of conditionals such a selection process may produce disastrous results as available operations may belong to different computational paths with disparate execution weights. The obvious generalization of the random heuristic is to give priority to operations belonging to the execution paths with greatest weight. We call such a heuristic greatest weight first (*GWF*).

Note that the schedule given on the left of Figure 3 will always be a GWF schedule independent of branching probabilities, whereas the one on the right will never be one as op5 should have been scheduled in I1.

Before stating our main result, we introduce two lemmas which are used later.

Lemma 6. *Let $(a_i)_{1\leq i\leq n}$ and $(b_i)_{1\leq i\leq n}$ be two sequences of $n \geq 1$ positive numbers. Then*

$$\frac{\sum_{i=1}^{n} a_i}{\sum_{i=1}^{n} b_i} \leq \max_{i\in[1,n]} \frac{a_i}{b_i}$$

Proof. It is easy to see that the lemma is true for $n \leq 2$. The correctness for arbitrary n follows by induction.

Lemma 7 Heaviest Subgraph in a Tree. *Let k be a positive integer, $\mathcal{T}$ a directed binary tree with at least k vertices, and $\hat{w}$ a weight function which maps every edge of $\mathcal{T}$ into the non-negative reals such that the sum of the weights of the edges sharing the same tail vertex is 1. The weight $w(x)$ of a vertex x of $\mathcal{T}$ is defined to be 1 if x is the root of $\mathcal{T}$ and otherwise the product of the weights of the edges from $\mathcal{T}$'s root to x.*

Assume that k vertices $x_1, \cdots, x_k$ of $\mathcal{T}$ are marked. For every marked vertex x_i, $i \in \{1, \cdots, k\}$ define $w_{\max}(x_i)$ to be the maximum of the weights of all marked vertices reachable from x_i. Then the following condition holds:

$$s(x_1, \cdots, x_k) = \sum_{i=1}^{k} w(x_i) - w_{\max}(x_i) \leq 1 + \frac{1}{2} \cdot \lceil \log_2 k \rceil$$

Proof. We just provide a sketch of the proof. First, we consider only balanced trees where each edge has the weight 1/2. For these trees we assume that the nodes are marked by levels in top down fashion such that at most one node has exactly one marked child. It is easy to see that the claim holds for this kind of marking. Finally, we map an arbitrary directed tree into a tree of the above described type such that the number of marked nodes will not increase while s will not decrease.

Theorem 8. *Let $T = (O, G, \prec)$ be a branching task, w a weight function for T, σ a GWF admissible m-schedule for T and σ_{opt} an m-optimum schedule for T and w. Then we have*

$$\frac{t(\sigma)}{t(\sigma_{opt})} \leq 2 - \frac{1}{m} + \frac{m-1}{2 \cdot m} \cdot \lceil \log_2 m \rceil.$$

Proof. For every operation op let $w(op)$ be the overall weight of the execution paths traversing op in T, that is

$$w(op) = \sum_{P \in \mathcal{P}(op,T)} w(P).$$

] ewise for each instruction I, $w(I)$ denotes the sum of the weights of the execution] hs traversing I. Further for each $op \in I$ we define $w(op, I) = \min(w(op), w(I))$.] te that if op is scheduled in instructions $I_1, \cdots, I_n$, then

$$w(op) = \sum_{j=1}^{n} w(op, I_j).$$

] ally, define $w_{\max}(I)$:

$$w_{\max}(I) = \begin{cases} 0 & \text{if } I \text{ contains less than } m \text{ operations} \\ \min_{op \in I} w(op, I) & \text{otherwise} \end{cases}$$

] m the above it is easy to prove that

$$t(\sigma) = \sum_{P \in \mathcal{P}(T)} w(P) \cdot \sigma(P) = \sum_{I \in \sigma} w(I)$$

] I be some instruction in σ containing $k \leq m$ vertices. Then after some simple ɛ ɛbraic manipulations we have

$$m \cdot w(I) = \sum_{op \in I} w(op, I) + \sum_{op \in I} (w(I) - w(op, I)) + (m - k) \cdot w(I).$$

, σ is a GWF schedule, at least one non speculative operation must be executing in] Thus, there exists at least one operation $op_0 \in I$ such that $w(op_0, I) = w(I)$. This i ɔlies

$$m \cdot w(I) \leq \sum_{op \in I} w(op, I) + (m - 1) \cdot (w(I) - w_{\max}(I)).$$

' ɛrefore, we have

$$m \cdot t(\sigma) = m \cdot \sum_{I \in \sigma} w(I) \leq \sum_{I \in \sigma} \sum_{op \in I} w(op, I) + (m - 1) \cdot \sum_{I \in \sigma} (w(I) - w_{\max}(I)).$$

] ɛt, it is easy to see that

$$\sum_{I \in \sigma} \sum_{op \in I} w(op, I) = \sum_{op \in O} w(op) \leq m \cdot w(\sigma_{opt}).$$

Thus to bound $t(\sigma)$ in terms of $t(\sigma_{opt})$ it suffices to bound

$$H = (m-1) \cdot \sum_{I \in \sigma} (w(I) - w_{\max}(I)).$$

Let S denote the set of all the instructions $I \in \sigma$ which either contain less than m operations or an operation which is scheduled speculatively in I. For these instructions we have $w_{\max}(I) < w(I)$. Now, consider the following branching task system $T' = (O', G', \prec)$ with

1. $O' = O_s \cup O_c$ with $O_s = \{op \in I | I \in S\}$ and O_c is the set of conditional operations not in O_s.
2. The control flow graph G' is obtained from the control flow graph of T by deleting every operation not in O'.
3. The dependence relation of T' is the restriction of the dependence relation of T to the operations in O'.

As T' has the same set of execution paths as T w can also be used as weight function for T'. Consider the m-optimum schedule σ_o admissible for T' such that each conditionals in O_c is scheduled alone in an instruction of σ_o. If R denotes the set of instructions in σ_o which only contain operations from O_s, then

$$\sum_{I_o \in R} w(I_o) \leq t(\sigma_{opt}).$$

For each $I \in S$ define its representative operation $op(I)$ to be some operation scheduled in I. These representative operations can be selected so that for each instruction $I_o \in R$ and for all representative operations $op(I)$ and $op(I')$ scheduled in I_o there is:

1. $w(I_o) \geq w(I)$ and $w(I_o) \geq w(I')$.
2. If I is an ancestor of I' in σ then $w_{\max}(I) \geq w(I')$.

For an instruction $I \in S$ let cj be the first conditional which is scheduled non speculatively in an instruction following I in σ. Note that cj may not necessarily exist. Initially, $op(I)$ is defined as follows:

$$op(I) = \begin{cases} cj & \text{if } cj \in I \\ op_d & \text{with } op_d \in I \text{ and } op_d \prec cj \text{ if } cj \text{ exists.} \\ op & \text{any operation } op \in I \text{ such that } w(op, I) = w(I) \text{ otherwise} \end{cases}$$

If cj exists and $cj \notin I$ then op_d must exist as σ is a GWF schedule. Also, op_d cannot be scheduled speculatively in I. Further, there must be at least one operation $op \in I$ which is scheduled non speculatively. Hence we have $w(op(I), I) = w(I)$ in all of the above cases. Also due to the initial choice of $op(I)$ for each $I \in S$ no conditional branch following $op(I)$ in σ can be executed before $op(I)$ in any schedule admissible for T. Therefore, for each $I_o \in \sigma_o$ such that $op(I) \in I_o$ initially, we have $w(I_o) \geq w(op(I), I) = w(I)$.

Next, this initial value of $op(I)$ is updated such that both conditions above are respected. To perform this update the control flow graph of σ_o is traversed in bottom up fashion.

If there exist two representative operations $op(I)$ and $op(I')$ scheduled in some $I_o \in \sigma_o$ such that I is an ancestor of I' with $w_{\max}(I) < w(I')$ then, there must exist an operation $op'' \in I$ such that $op'' \prec op(I')$ as σ is a GWF schedule. Change $op(I)$ so that $op(I) = op''$. Note that op'' must be scheduled in σ_o in an instruction I'_o preceding I_o with $w(I) \leq w(I_o) \leq w(I'_o)$. Furthermore, I'_o has not yet been explored by our bottom tree traversal. Thus as we proceed up the control flow graph of σ_o both, conditions for representative operations are preserved by this transformation. Therefore, we can write:

$$H = (m-1) \cdot \sum_{I \in S} w(I) - w_{\max}(I) = (m-1) \cdot \sum_{I_o \in R} \sum_{\{I : op(I) \in I_o\}} w(I) - w_{\max}(I)$$

which implies that

$$\frac{H}{m \cdot t(\sigma_{opt})} \leq \frac{(m-1) \cdot \sum_{I_o \in R} \sum_{\{I : op(I) \in I_o\}} w(I) - w_{\max}(I)}{m \cdot \sum_{I_o \in R} w(I_o)}$$

Using Lemma 6 we obtain

$$\frac{H}{m \cdot t(\sigma_{opt})} \leq \frac{m-1}{m} \cdot \max_{I_o \in R} \underbrace{\sum_{\{I : op(I) \in I_o\}} \frac{w(I) - w_{\max}(I)}{w(I_o)}}_{\text{call this } X(I_o)}$$

Because of the first condition for representative operations $w(I) \leq w(I_o)$ whenever $op(I) \in I_o$. Thus to bound $X(I_o)$ it suffices to find the upper bound U to the solution of the graph theoretical problem given in Lemma 7. This lemma establishes that $X(I_o) \leq U$ resulting in

$$H \leq (m-1) \cdot U \cdot t(\sigma_{opt}).$$

Therefore, we finally get

$$m \cdot t(\sigma) \leq m \cdot t(\sigma_{opt}) + (m-1) \cdot U \cdot t(\sigma_{opt})$$

and

$$\frac{t(\sigma)}{t(\sigma_{opt})} \leq 1 + \frac{m-1}{m} \cdot U.$$

6 Tightness of the Bound

The bound of Theorem 8 is almost tight. A deviation from the optimal case may occur if two operations op_1 and op_2 which do not belong to the same execution path are both ready for scheduling. In this case GWF systematically selects the operation with the highest weight, say op_1, whereas their weights might be close and op_2 could be a critical operation for the execution paths containing it. This kind of behavior is illustrated in the example of Figure 4.

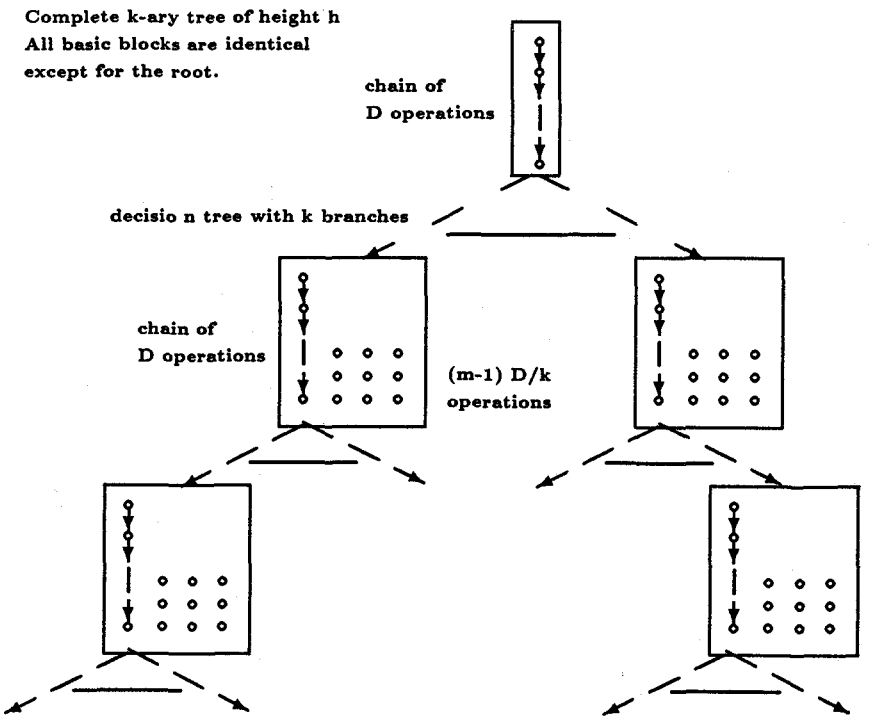

Fig. 4. Branching task system on which GWF performs poorly.

The branching schedule of the figure consists of a control flow graph whose basic blocks form a complete k-ary tree of height h. Each block contains a chain of D dependent operations, whose last operation is a conditional. Apart from the root block, every other block also contains $(m-1) \cdot D/k$ independent operations. If we assume that D is very big, we can regard the last conditional in a block as having out-degree k. However, this means that a decision tree of $k-1$ conditionals jumps is at the end of a block.

Next, let us assume that all execution paths in the branching task system are equally likely. Further, let $k = h = \log m / \log \log m$. Consider the schedule σ_1 where the independent operations are scheduled in the block immediately above. Clearly this schedule is GWF and its average execution time $t(\sigma_1) = h \cdot D$.

Consider now the schedule σ_2 where all the chains of operations are scheduled in the root block, and the independent operations are scheduled in the k^h leaf basic blocks. Then, the average execution time of σ_2 is approximately $2 \cdot D$ resulting in $t(\sigma_1)/t(\sigma_2) \geq 1/2 \cdot \log m / \log \log m$.

References

1. A. AIKEN AND A. NICOLAU, *A development environment for horizontal microcode*, IEEE Transactions on Software Engineering, 14 (1988), pp. 584–594.
2. E. G. COFFMAN, *Computer and Job-shop Scheduling Theory*, John Wiley and Sons, New York, New York, 1976.
3. K. EBCIOĞLU *Some design ideas for a VLIW architecture for sequential-natured software*, in Proc. IFIP WG 10.3 Conf. on Parallel Processing, 1988, North-Holland, pp. 1–21.
4. J. A. FISHER, J. R. ELLIS, J. C. RUTTENBERG, AND A. NICOLAU, *Parallel processing: A smart compiler and a dumb machine*, in Proc. SIGPLAN 1984, June 1984, ACM, pp. 37–47.
5. M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability - A Guide to the Theory of NP-Completeness*, Freeman, New York, New York, 1979.
6. F. GASPERONI, *Scheduling for Horizontal Systems: The VLIW Paradigm in Perspective*, PhD thesis, New York University, New York, New York, July 1991.
7. R. GUPTA AND M. L. SOFFA, *Region scheduling: An approach for detecting and redistributing parallelism*, IEEE Transactions on Software Engineering, 16 (1990), pp. 421–431.
8. K. KARPLUS AND A. NICOLAU, *A compiler-driven supercomputer*, Applied Mathematics and Computations, 20 (1986), pp. 95–110.
9. A. NICOLAU AND J. A. FISHER, *Measuring the parallelism available for very long instruction word architectures*, IEEE Transactions on Computers, C-33 (1984), pp. 968–976.