ELSEVIER

Theoretical Computer Science 196 (1998) 347–363

Theoretical Computer Science

List scheduling in the presence of branches A theoretical evaluation[1]

Franco Gasperoni[a], Uwe Schwiegelshohn[b,*]

[a] *Télécom Paris-ENST, 46, rue Barrault, 75634 Paris, Cedex 13, France*

[b] *Computer Engineering Institute, University Dortmund, 44221 Dortmund, Germany*

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 conditionals. In this paper we extend previous work on straight line code scheduling by looking at task systems with branches. 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. © 1998—Elevier Science B.V. All rights reserved

Keywords: Instruction scheduling; Performance guarantee; Parallelizing compilers

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 1960s 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 [10].

In the early 1980s Fisher developed an innovative compilation technique called trace scheduling, that went beyond the conditional jump barrier in its quest for parallelism. Fisher subsequently introduced an architectural paradigm, termed VLIW, which by employing a trace scheduling compiler provided high performance at low cost [5].

* Corresponding author. Tel.: +49 231 755 2634; fax: +49 231 755 3251; e-mail: uwe@ds.e-technik.uni-dortmund.de.

[1] A preliminary version of this paper appeared in the Proceedings of Euro-Par 96.

PII S0304-3975(97)00207-7

Today all systems, that boost performance by exploiting fine grain parallelism, combine multiple functional units/single thread of control machines with sophisticated compilers. Further, several compiler algorithms such as percolation scheduling [1] or region scheduling [8] have generalized the ideas behind trace scheduling for nonnumerical programs.

However, for most of these techniques the actual motion of operations beyond conditional branches has received more attention than mechanisms for the selection of the operations to move. Trace scheduling is an exception as operations from the execution path with highest probability are given priority in code motion transformations. 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 [6]. Frequently, a classical heuristic, called list scheduling, is employed to guarantee close to optimum performance. There, operations are first ordered in priority list. Instructions are then constructed in a top down fashion by selecting operations from this priority list and moving them to the instruction under construction. 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 [3].

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

$$\frac{8}{3} - \frac{5}{3m} + \frac{m-1}{2m} \log_2 \frac{m+1}{5}.$$

The remainder of the paper is structured as follows. Section 2 introduces branching task systems which formalize the notion of programs with conditionals. Section 3 explains our machine model and defines schedules containing branches. Next, Section 4 defines optimality while in Section 5 list scheduling is extended to consider branches and the new performance guarantee is derived. 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 [3]. 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 be executed.

Definition 1 (*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

```
procedure Poly_Roots (in: a,b,c; out: x1,x2,roots)
              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.

task system if the following conditions are valid:

(i) 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 $\mathscr{P}(T)$. For any $op \in O$, $\mathscr{P}(op, T)$ denotes the set of execution paths traversing op.

(ii) For each execution path P, $\prec$ is a partial order on P compatible with its linear ordering, that is if $op \prec op'$, then op must precede op' in P.

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

The precedence relation of the branching task is given in Fig. 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

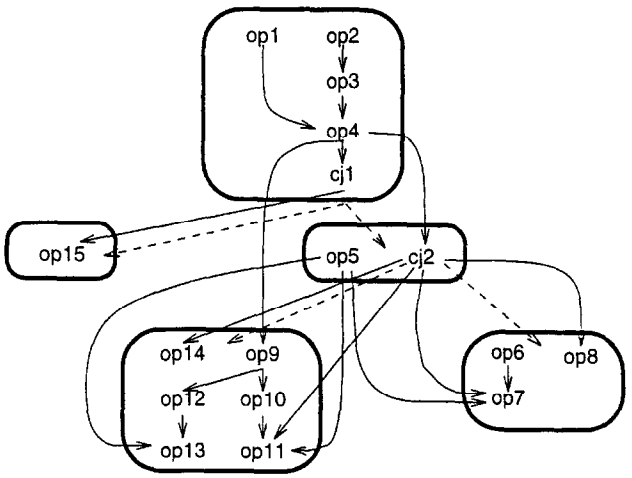

Fig. 2. A branching task system.

static dependencies from appropriate conditional branches to these operations. There are also various other possibilities to address this problem as, e.g. renaming. However, a discussion of these issues is beyond the scope of this paper (see for instance [2]).

The control flow is also given in Fig. 2. Within each block the control flow is defined by the numerical order of the operations 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 multicycle operations is more thoroughly discussed in [7]. 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 (see Section 5).

Note that any acyclic control flow graph $G - \{\zeta\}$ can be transformed into a tree by simply replicating parts of the graph. Therefore, it is sufficient to address only the case where $G - \{\zeta\}$ is a tree from now on. However, while the presented results also hold for general branching task systems, it may then be necessary to sacrifice space performance in order to obtain even a modest speedup [7]. More specifically, a speedup as little as 2 may require exponential code size. Thus for branching tasks, whose control flow graph without the exit vertex is not a tree, time and space performance can be antipodal.

This phenomenon can be intuitively explained by considering the number of execution paths for a control flow graph. If the control flow graph is a tree, the overall 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 $1 \leqslant k \leqslant m$ conditionals, they are arranged to form a decision tree with $k+1$ outgoing branches that specify which instruction must be executed next. This machine model is inspired by the branching paradigm of Karplus and Nicolau [9] and Ebcioğlu [4]. A formal definition is given below:

Definition 2 (*Branching schedule*). A branching schedule σ of a branching task system T comprises a set of instructions $\mathcal{I}(\sigma)$ and a control flow graph $G(\sigma)$ which is an acyclic single entry, single exit di-graph with vertex set $\mathcal{I}(\sigma) \cup \{\xi, \zeta\}$.

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

(ii) 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:

(i) *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 exactly once each operation 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.

(ii) *Dependencies*: For any pair of operations op, $op' \in O$ with $op \prec op'$, $op \in I$, and $op' \in I'$, instruction I is a proper ancestor of I' in all paths $\phi(P)$, where P is a path traversing both op and op' in G.

In the sequel, an execution path P in T is identified with its corresponding execution path in σ, which will also be denoted P.

Two branching 3-schedules admissible for the branching task system of Fig. 2 are given in Fig. 3. Note that an operation op need not be scheduled in a single

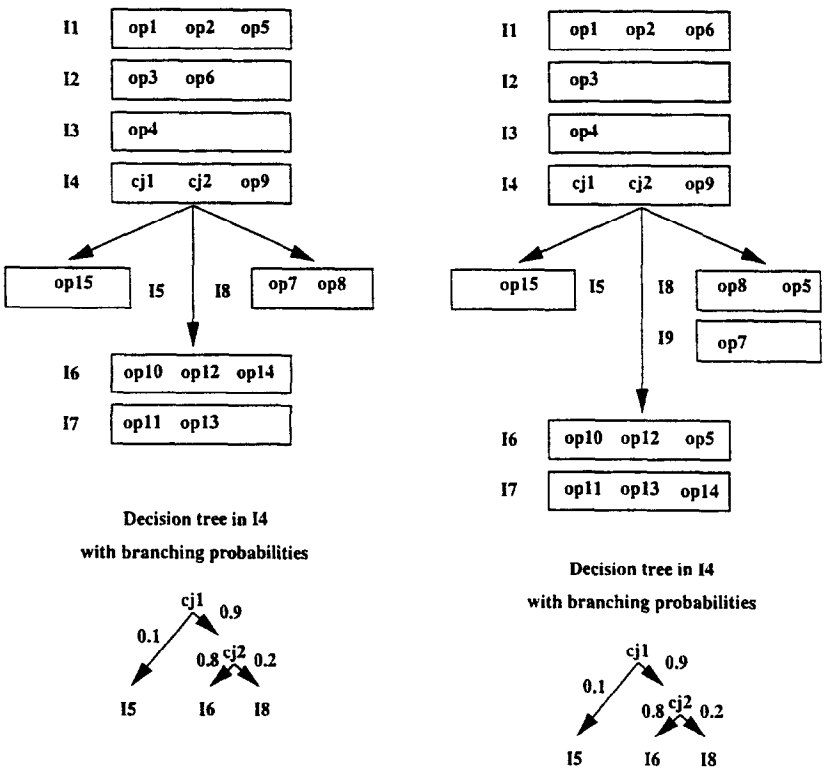

Fig. 3. Two branching schedules.

instruction. For instance in the schedule to the right, 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, the average execution time based on an appropriate weight function is used as a measure for the quality of a branching schedule.

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

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

When $G-\{\zeta\}$ is a tree, we can easily construct a function $\hat{w}$ mapping the edges e_i of G into the non-negative reals such that

$$w(P)=\prod_{i=1}^{k}\hat{w}(e_i) \quad \forall P=(e_1,e_2,\ldots,e_k)\in\mathscr{P}(T)$$

by evaluating the tree in bottom up fashion.

Definition 5 (*Optimality*). The weighted average running time $t(\sigma)$ of a schedule σ admissible for a branching task T is defined as

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

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

Usually, these above-defined weights are execution path probabilities. For instance, the probability in Fig. 3 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`. Then the average running time of the schedule on the left of Fig. 3 is `5.72`, whereas the average running time of the schedule on the right of Fig. 3 is `5.9`.

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 [3] to branching task systems.

When generating instructions during scheduling, 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 combined weight. We call such a heuristic 'greatest weight first' (GWF).

Note that the schedule given on the left of Fig. 3 will always be a GWF schedule independent of branching probabilities. On the other hand the one on the right will never be one as op5 should have been given priority over op6, if the execution path containing op14 has a positive weight.

While the GWF heuristic yields provably good results for unit cycle operations (see Theorem 6), GWF can yield very poor performance for multicycle operations in the absence of preemption. Consider for instance, the example given in Fig. 4 for $m=3$ processors. On the top of the figure the input branching task, its control flow graph, its dependencies, and the weights of its edges are shown. The length of the optimum schedule (on the right-hand side of the figure) is

$$D+1+\varepsilon(D+1)+\varepsilon(1-\varepsilon)D < D+3 \quad \text{for } \varepsilon = 1/(D+1)$$

while the length of the GWF schedule (on the left-hand side of the figure) is

$$D+1+(1-\varepsilon)D+(1-\varepsilon)^2(D-2) > 3D-4 \quad \text{for } \varepsilon = 1/(D+1).$$

Thus, the ratio between the GWF schedule and the optimum schedule approaches 3 as D goes to infinity. It is easy to generalize this example to obtain a worst case performance factor of m for m processors.

We will first state the main result, give an informal explanation for the worst-case bound, and then give its full proof.

Theorem 6. *Let $T=(O,G,\prec)$ be a branching task system with $G-\{\zeta\}$ being a tree, 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*

$$\frac{t(\sigma)}{t(\sigma_{\text{opt}})} \leqslant \frac{8}{3} - \frac{5}{3m} + \frac{m-1}{2m} \log_2 \frac{m+1}{5}.$$

In the absence of branches a GWF scheduler is nothing more than a list scheduler. Hence, it suffers from the standard list scheduling problem that an operation of little importance, for instance an operation not on the critical path, is given precedence over a more important operation.

In the presence of branches the generalization of this problem is illustrated by the worst-case situation explained in Section 6 and portrayed in Fig. 5.

More specifically, assume that during assembly by the GWF scheduler the current instruction contains $m-1$ free processor slots. Suppose that some basic block contains $m-1$ completely independent operations ready for scheduling and that each of these operations has a probability of p_1 of being executed. Further, suppose that in another basic block a single operation op_0, whose probability of execution is $p_0 < p_1$, is also available for scheduling. The GWF scheduler will select the $m-1$ operations for scheduling in the current instruction, even though these $m-1$ operations may not be on any critical path while op_0 may be on a critical path.

In the worst case, if there are $m-1$ operations in op_0's situation, this mistake would entail a loss of roughly $(m-1)(p_0 - p_1/m)$ execution cycles for the overall average

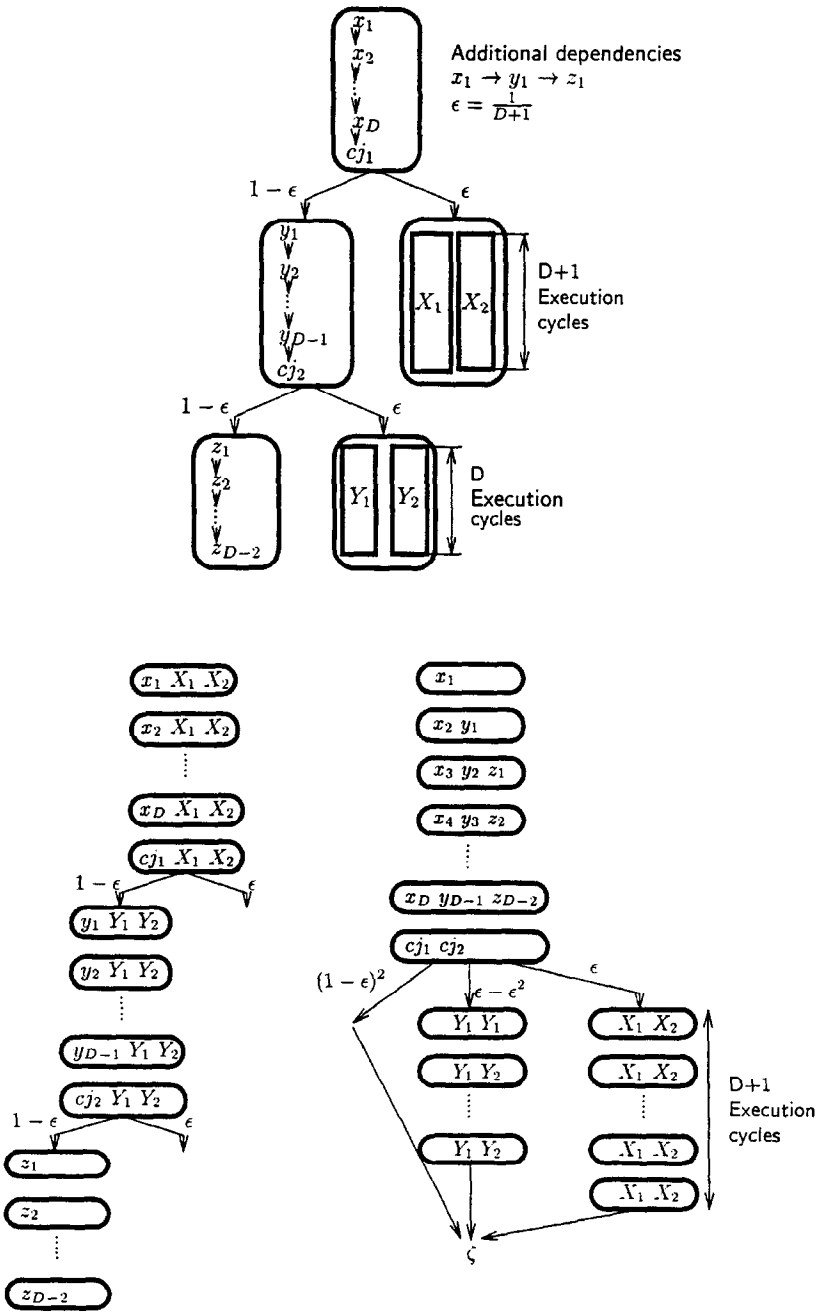

Fig. 4. GWF can perform poorly for multicycle operations. Top: branching task; Left: GWF schedule; Right: optimum schedule.

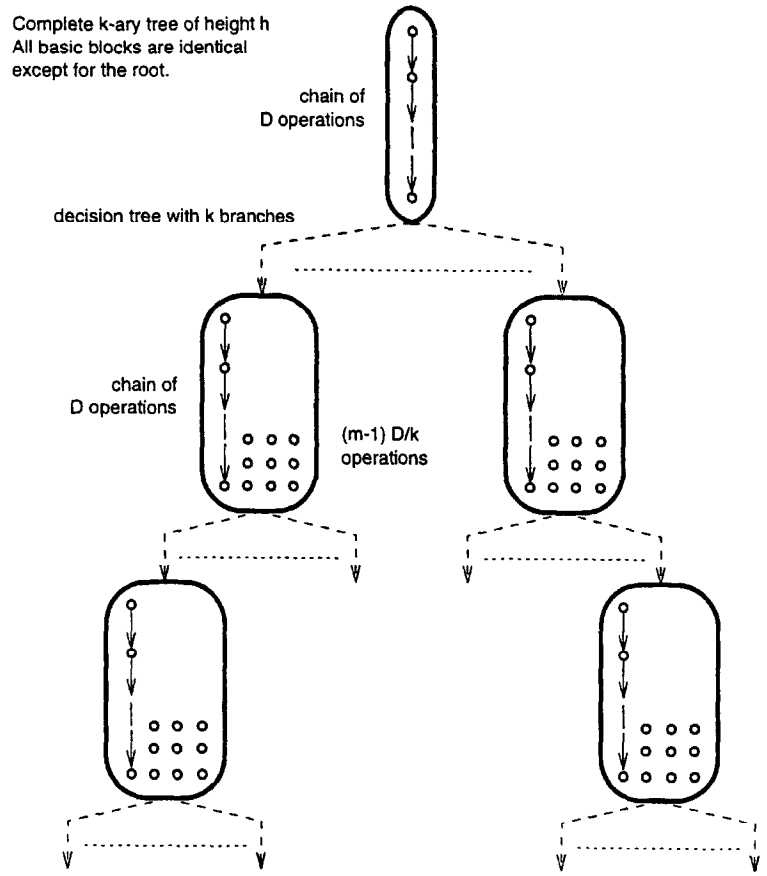

Fig. 5. Branching task system on which GWF performs poorly.

execution time performance. Furthermore, this mistake could be repeated at every level in the input control flow graph.

Thus, if the control flow graph has h levels (i.e. it is a tree of height h), the overall error would be $(m-1)h(p_0 - p_1/m)$.

If the control flow graph is a complete q-ary tree (each node has q descendents) and if p_0 is arbitrarily close to p_1, we can rewrite the previous formula as $(m-1)h1/q(1-1/m)$. Henceforth, determining the worst-case bound boils down to an optimization problem where we try to maximize the expression h/q under the constraint that $q<m$ with m being the number of overall processors and h being the height of a q-ary tree. Lemma 9 shows that the upper bound of a somewhat more sophisticated scenario is in the order of $\log_2 m$.

The actual details of the full proof are a bit more intricate because the actual accounting of execution cycles, that are potentially lost, is more accurate than the above rough reasoning.

Before giving the full proof of our main result, we introduce two lemmas which are used later.

Lemma 7. *Let* $(a_i)_{1\leqslant i\leqslant n}$ *and* $(b_i)_{1\leqslant i\leqslant n}$ *be two sequences of* $n\geqslant 1$ *positive numbers. Then*

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

Proof. The lemma is true for $n=1$. For $n=2$ assume $a_1/b_1 \geqslant a_2/b_2$. Then we have

$$\frac{a_1+a_2}{b_1+b_2} \leqslant \frac{a_1(1+b_2/b_1)}{b_1+b_2} = \frac{a_1}{b_1}.$$

The correctness for arbitrary n follows by induction. □

For the next lemma we need to define a specially weighted tree. Intuitively, the weights of the tree represent branching probabilities.

Definition 8 (*Weighted Tree*). Let $\mathcal{T}$ be a directed tree 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. Then, 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 it is the product of the weights of the edges from $\mathcal{T}$'s root to x.

Note that for any q vertices $x_1,\ldots,x_q$, such that no x_i is an ancestor of any other x_j in $\mathcal{T}$, we have $\sum_{i=1}^{q} w(x_i)\leqslant 1$.

Lemma 9 (Heaviest subgraph in a tree). *Assume that* k *vertices* $x_1,\ldots,x_k$ *of a weighted tree* $\mathcal{T}$ *are marked. For every vertex* x *in* $\mathcal{T}$ *let* $\mathcal{T}(x)$ *denote the subtree of* $\mathcal{T}$ *rooted at* x *and* $v(x)\neq x$ *the vertex in* $\mathcal{T}(x)$ *which is marked and has maximum weight. If* $\mathcal{T}(x)$ *contains no marked vertex apart from possibly* x*, we say* $w(v(x))=0$*. Then the following statement holds:*

$$S(x_1,\ldots,x_k)=\sum_{i=1}^{k} w(x_i)-w(v(x_i)) \leqslant \frac{5}{3}+\frac{1}{2}\log_2\frac{k+1}{5}.$$

Proof. First, we show that it is sufficient to consider only those trees where all vertices are marked. Assume that root r is not marked while n vertices x_i are marked with $w(x_i)\geqslant w(x_{i+1})$ for $1\leqslant i<n$. Then we mark r and unmark x_1. This will increase S by $1-w(x_1)\geqslant 0$ if there is a path from x_1 to x_2 or if $n=1$. Otherwise, we have $w(x_1)+w(x_2)\leqslant 1$ and S will be increased by at least $1-w(x_1)-w(x_2)\geqslant 0$.

Now that the root is marked we remove all unmarked vertices while modifying the structure of the tree and the edge weights appropriately so that the weight of any marked vertex will not decrease. Hence, S cannot decrease as well.

Next by examining all trees with at most 4 vertices, it can be verified that the claimed statement holds for these cases and that the bound is tight for a tree consisting of a root and 3 children where all edges have the uniform weight of $\frac{1}{3}$.

The remainder of the proof is done by induction on the number of vertices in the tree. The claim holds for trees consisting of at most 4 vertices. For the inductive step, we combine all subtrees of the q children $x_1,\ldots,x_q$ of the root r of $\mathcal{T}$ with r. As induction hypothesis we assume that the claim holds for the subtrees rooted in those x_i. If k_i denotes the number of vertices in $\mathcal{T}(x_i)$ we must therefore prove

$$1 - w(v(r)) + \sum_{i=1}^{q} w(x_i)\left(\frac{5}{3} + \frac{1}{2}\log_2\frac{k_i+1}{5}\right) \leqslant \frac{5}{3} + \frac{1}{2}\log_2\frac{\sum_{i=1}^{q} k_i + 2}{5}.$$

After some transformations the above translates into

$$2 - 2w(v(r)) \leqslant \log_2 \frac{\sum_{i=1}^{q} k_i + 2}{\prod_{i=1}^{q}(k_i+1)^{w(x_i)}}$$

with $0 \leqslant w(x_i) \leqslant w(v(r))$ for $1 \leqslant i \leqslant q$. Assume that the subtrees are ordered such that $k_i \geqslant k_{i+1}$ for $1 \leqslant i < q$. Then $\prod_{i=1}^{q}(k_i+1)^{w(x_i)}$ is maximum if

$$w(x_i) = \begin{cases} w(v(r)) & \text{for } 1 \leqslant i \leqslant p = \lfloor \frac{1}{w(v(r))} \rfloor, \\ 1 - pw(v(r)) & \text{for } i = p+1, \\ 0 & \text{for } i > p+1. \end{cases}$$

Thus, if we let $y = w(v(r))$ with $1/(p+1) \leqslant y < 1/p$ it is sufficient to consider the inequality

$$2 - 2y \leqslant \log_2 \frac{\sum_{i=1}^{p+1} k_i + 2}{(\prod_{i=1}^{p}(k_i+1))^y (k_{p+1}+1)^{1-p\cdot y}}$$

which can be transformed into

$$4(k_{p+1}+1)\left(\frac{1}{4}\prod_{i=1}^{p}\frac{k_i+1}{k_{p+1}+1}\right)^y \leqslant \sum_{i=1}^{p+1} k_i + 2.$$

While the right-hand side of the inequality is independent of y the left-hand side becomes maximal either for $y \to 1/p$ or for $y = 1/(p+1)$. Note that in both cases $w(x_i)$ is either $w(v(r))$ or 0. Hence, we further assume that $py = 1$ which results in

$$\frac{4}{4^y}\left(\prod_{i=1}^{p}(k_i+1)\right)^y \leqslant \sum_{i=1}^{p} k_i + 2.$$

As $\sum_{i=1}^{p} k_i$ is equal to some fixed constant k, the product on the left-hand side is maximum when all k_i are equal to some constant k_0. This gives

$$\frac{4}{4^{1/p}}(k_0+1) \leqslant pk_0 + 2$$

which is true for all positive integers p if $pk_0 \geqslant 5$ and $k_0 \geqslant 1$. □

Now, we are ready to address the proof of Theorem 6.

Proof of Theorem 6. For every operation op let $w(op)$ be the overall weight of the execution paths traversing op in T, that is

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

Likewise for each instruction I, $w(I)$ denotes the sum of the weights of the execution paths traversing I. Therefore, we have

$$t(\sigma) = \sum_{P \in \mathscr{P}(T)} w(P)d(P,\sigma) = \sum_{I \in \sigma} w(I).$$

Further, for each $op \in I$ we define $w(op,I) = \min(w(op), w(I))$. Thus, if an operation op is scheduled in instructions $I_1, \ldots, I_n$, then

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

since op is not executed twice on any path of the schedule.

Note that $w(op,I)$ will be strictly less than $w(I)$, if op is executed speculatively in I. Likewise, $w(op,I)$ will be strictly less than $w(op)$, if in the original sequential task op was to be executed above a conditional branch cj, while in the GWF schedule σ, op is scheduled below cj on both its true and false branches. Finally, define $w_{\min}(I)$:

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

Let I be some instruction in σ containing $1 \leqslant k \leqslant m$ vertices. Then, we use the following identity:

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

As σ is a GWF schedule, at least one nonspeculative operation must be executing in I. Thus, there exists at least one operation $op_0 \in I$ such that $w(op_0,I) = w(I)$. This implies

$$mw(I) \leqslant \sum_{op \in I} w(op,I) + (m-1)(w(I) - w_{\min}(I)).$$

Therefore, we have

$$mt(\sigma) = m \sum_{I \in \sigma} w(I)$$
$$\leqslant \sum_{I \in \sigma} \sum_{op \in I} w(op, I) + (m-1) \sum_{I \in \sigma} (w(I) - w_{\min}(I)).$$

Next, it is easy to see that

$$\sum_{I \in \sigma} \sum_{op \in I} w(op, I) = \sum_{op \in O} w(op) \leqslant mt(\sigma_{\text{opt}}).$$

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

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

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 $w_{\min}(I) < w(I)$. Now, consider the following branching task system $T' = (O', G', \prec)$ with

(i) $O' = O_s \cup O_c$ with $O_s = \{op \in I \mid I \in S\}$ and O_c being the set of conditional operations not in O_s.
(ii) The control flow graph G' is obtained from the control flow graph of T by deleting and bypassing every operation not in O'.
(iii) The dependence relation of T' is the restriction of the dependence relation of T to the operations in O'.

As T' and T have the same set of execution paths, w can also be used as weight function for T'. Consider the m-optimum schedule σ_o admissible for T' such that each conditional 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

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

For each $I \in S$ define its representative operation $op(I)$ to be some operation scheduled in I. We show that 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 we have

(i) $w(I_o) \geqslant w(I)$ and $w(I_o) \geqslant w(I')$.
(ii) If I is an ancestor of I' in σ then $w_{\min}(I) \geqslant w(I')$.

Initially, we choose each representative operation $op(I)$ for each $I \in S$ as follows. Let cj be the first conditional which is scheduled nonspeculatively in an instruction following I in σ. Note that cj may not necessarily exist. Initially, $op(I)$ is defined as follows:

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

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. In any case there must always be at least one operation $op \in I$ which is scheduled nonspeculatively. 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 $I_0 \in \sigma_0$ such that $op(I) \in I_0$ initially, we have $w(I_0) \geqslant w(op(I), I) = w(I)$ as the control flow graph without ζ is a tree.

This initial choice respects condition 1 above but does not necessarily guarantee the correctness of condition 2 as well. We therefore update this initial value of $op(I)$ so that both conditions 1 and 2 above are respected. To perform this update the control flow graph of σ_0 is traversed in a bottom-up fashion.

If there exist two representative operations $op(I)$ and $op(I')$ scheduled in some $I_0 \in \sigma_0$ such that I is an ancestor of I' with $w_{\min}(I) < w(I')$, then there must exist an operation $op'' \in I$ such that $op'' \prec op(I')$ since σ is a GWF schedule. Change $op(I)$ so that $op(I) = op''$. Note that op'' must be scheduled in σ_0 in an instruction I_0' preceding I_0 with $w(I) \leqslant w(I_0) \leqslant w(I_0')$. Furthermore, I_0' has not yet been explored by our bottom-up tree traversal. Thus as we proceed up the control flow graph of σ_0, condition 1 for representative operations is preserved by this transformation while we keep updating representative operations until condition 2 is met. This must eventually occur since we traverse the tree bottom up and new representative operations are always scheduled in σ_0 in instructions not already explored by our bottom up search. Thus, we can write

$$\begin{aligned} H &= (m-1) \sum_{I \in S} (w(I) - w_{\min}(I)) \\ &= (m-1) \sum_{I_0 \in R} \sum_{\{I : op(I) \in I_0\}} (w(I) - w_{\min}(I)) \end{aligned}$$

which implies that

$$\frac{H}{t(\sigma_{\text{opt}})} \leqslant \frac{(m-1) \sum_{I_0 \in R} \sum_{\{I : op(I) \in I_0\}} (w(I) - w_{\min}(I))}{\sum_{I_0 \in R} w(I_0)}.$$

Using Lemma 7 we obtain

$$\frac{H}{t(\sigma_{\text{opt}})} \leqslant (m-1) \max_{I_0 \in R} \underbrace{\sum_{\{I : op(I) \in I_0\}} \frac{w(I) - w_{\min}(I)}{w(I_0)}}_{\text{call this } X(I_0)}.$$

Because of the first condition for representative operations we have $w(I) \leqslant w(I_0)$ whenever $op(I) \in I_0$. As all paths through I must also pass through I_0, it suffices to find the upper bound U to the solution of the graph theoretical problem given in Lemma 9 in order to bound $X(I_0)$. With this upper bound U we then have

$$H \leqslant (m-1) U t(\sigma_{\text{opt}}).$$

Therefore, we finally get

$$mt(\sigma) \leqslant mt(\sigma_{\text{opt}}) + (m-1) U t(\sigma_{\text{opt}})$$

and

$$\frac{t(\sigma)}{t(\sigma_{\text{opt}})} \leqslant 1 + \frac{m-1}{m} U. \qquad \square$$

6. Tightness of the bound

In this section we show that the bound of Theorem 6 is almost tight. In general 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 Fig. 5.

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 $(m-1)D/k$ independent operations. If we assume that D is sufficiently large, 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 $t(\sigma_1) = hD$.

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 $2D$ resulting in $t(\sigma_1)/t(\sigma_2) \geqslant 1/2 \log m / \log \log m$.

References

[1] A. Aiken, A. Nicolau, A development environment for horizontal microcode, IEEE Trans. Software Eng. 14 (1988) 584–594.

[2] F. Allen, M. Burke, P. Charles, R. Cytron, J. Ferrante, An overview of the PTRAN analysis system for multiprocessing, J. Parallel Distributed Comput. 5 (1988) 617–640.

[3] E.G. Coffman, Computer and Job-shop Scheduling Theory, Wiley, New York, 1976.

[4] K. Ebcioğlu, Some design ideas for a VLIW architecture for sequential-natured software, in: Proc. IFIP WG 10.3 Conf. on Parallel Processing, North-Holland, Amsterdam, 1988, pp. 1–21.

[5] J.A. Fisher, J.R. Ellis, J.C. Ruttenberg, A. Nicolau, Parallel processing: A smart compiler and a dumb machine, in: Proc. SIGPLAN 1984, June 1984, ACM, pp. 37–47.

[6] M.R. Garey, D.S. Johnson, Computers and Intractability – A Guide to the Theory of NP-Completeness, Freeman, New York, 1979.

[7] F. Gasperoni, Scheduling for horizontal systems: the VLIW paradigm in perspective, Ph.D. thesis, New York University, New York, July 1991.

[8] R. Gupta, M.L. Soffa, Region scheduling: An approach for detecting and redistributing parallelism, IEEE Trans. Software Eng. 16 (1990) 421–431.
[9] K. Karplus, A. Nicolau, A compiler-driven supercomputer, Appl. Math. Comput. 20 (1986) 95–110.
[10] A. Nicolau, J.A. Fisher, Measuring the parallelism available for very long instruction word architectures, IEEE Trans. Comput. C-33 (1984) 968–976.