Characterizing Vulnerability of Parallelism to Resource Constraints

V. Vivekanand, K. Gopinath, Computer Science & Automation, IISc, Bangalore
Pradeep Dubey, IBM T.J. Watson Research Center, Yorktown Heights

Abstract

The theoretical available instruction level parallelism in most benchmarks is very high. Vulnerability is related to the difficulty with which we can extract this parallelism with finite resources. This study characterizes the vulnerability of parallelism to resource constraints by scheduling dynamic dependence graphs (DDGs) from traces of several benchmarks using different scheduling algorithms and different number of functional units. It is observed that the execution time of the DDGs does not vary significantly with low-level scheduling algorithms like lazy, slack, etc. Measures of vulnerability based on slack and load were also considered. Although Accslk-Load, which uses a combination of accurate slack and load to make a prediction, has a prediction accuracy of about 85%, the prediction rate is only 42%. On the other hand, even though the prediction accuracy of $\sigma(L_x)$, the standard deviation in the load, is not as high, there is a prediction in all the cases. The DDG execution time is also found to be most vulnerable to the functional unit with the greatest $\sigma(L_x)$.

1 Introduction

Recent advances in VLSI technology are making it increasingly feasible to put multiple (say, 4-8) execution pipelines on the same chip. To make effective use of such systems, high levels of ideal parallelism that is present in many applications [8], [3], [2], [5], [7], [1] should be extractible with finite resources. Informally, we use the term vulnerability to signify the difficulty with which we can extract the high levels of ideal parallelism with finite resources; a formal definition is given later in Section 2.3. Due to variations in the available parallelism, the efficiency with which the resources are used is also of interest.

Future compilers or OS kernels/runtime systems may be able to schedule threads or programs with different vulnerabilities to resource constraints so as to maximize performance by operating them at their higher efficiency regimes. If there are 2 threads or programs, A and B, and both operate at, say, 90% efficiency if *k* functional units are available and, say, 70% efficiency with *2k* functional units, then it may be advantageous[1] to use *k* units each to run A and B together rather than use *2k* units to run each in turn[4]. If the functional units in question are heavily contended such as memory ports, efficiency may increase superlinearly with number of functional units; in such cases, the vulnerability of a program to resource constraints may be high. An intelligent high-level scheduling (at the program (or loop) level rather than at instruction level) requires information about the vulnerability of parallelism to resource constraints.

[1] When factoring effects due to caches, TLBs, etc, this decision may not be entirely correct due to flushing of state or due to contention.

This paper addresses the question of whether high ideal parallelism can be characterized to assess its extractibility with finite resources.

Previous Work: Theobald *et al.* [7] analyzed the *smoothability* of parallelism in instruction traces. The smoothability measure attempts to quantify whether the high ideal parallelism is smoothable enough to be extracted with finite resources. Rauchwerger *et al.* [1] used a measure called *slack* to assess the sustainability of the high ideal parallelism to resource constraints. They found that high ideal parallelism was mostly associated with a not very high average slack, suggesting the vulnerability of this parallelism to resource constraints.

2 Background

Program dependences can be represented by a partially ordered, directed, acyclic graph referred to as the *dynamic dependence graph* (DDG) [2]. The nodes of the graph represent the specific instruction executions (assuming unit execution times) and the edges represent the inter-instruction dependences during the program execution. The height of the topologically sorted DDG is the *critical path length* of the application, which is also equal to the minimum number of steps needed to complete the program execution. A plot of the number of instructions in a level in the topologically sorted DDG gives the *parallelism profile* of the program. The average parallelism in the program is the average number of instructions in a level.

2.1 Slack

An instruction I cannot be scheduled any sooner than the scheduling cycle in which all its source operands become available to be read as inputs. Also, I must be scheduled no later than the cycle preceding the one in which its result gets used as a source operand. Assuming unlimited number of resources, this suggests two scheduling

1094-7256/97 $10.00 © 1997 IEEE

techniques, the *greedy* schedule which attempts to schedule instructions in the earliest possible time slot and the *lazy* schedule which attempts to schedule instructions in the latest possible time slot. The difference between these two scheduling algorithms is defined as the *slack* [1] associated with instruction I. We now define two variations in the slack measure, namely, *accurate slack* and *approximate slack*. Every instruction has an earliest start[2] and a latest start[3]. Slack can be defined as the difference between the latest start time(Lstart) and the earliest start time(Estart). The difference between accurate slack and approximate slack is in the calculation of the latest start times:

$$\begin{aligned} Lstart_{acc}(I) &= MIN(Lstart_{acc}(J)) - 1 \\ Lstart_{app}(I) &= MIN(Estart(J)) - 1 \end{aligned}$$

where J depends on I.

We also define *block slack* for a block in a DDG that is a connected set of instructions such that each instruction in the block (except the first) has an indegree of one and an outdegree of one. A *Block Graph* is a partially ordered, directed, acyclic graph. The nodes in a block graph represent blocks and the edges represent inter-block dependences.

Let N_b be the number of instructions in block B. Let C_b be the difference between the latest start time of the last instruction in B and the earliest start time of the first instruction in B. The *block slack* of B is defined as the difference between C_b and N_b. Block slack is a measure of the number of the cycles by which the instructions in the block can be delayed without an increase in the critical path length.

2.2 Resource Load

For each instruction I_j in the trace, we know the slots $S_{j,e}$ and $S_{j,l}$ corresponding to the greedy and the lazy scheduling policies [1] of the instruction in the DDG. Thus, at any level x in the DDG, we can define a set Φ of live instructions as,

$$\Phi_x = \{I_j | S_{j,e} \leq x \leq S_{j,l}\}$$

The load L is the mean of the cycle load (L_x). Mean parallelism, MP, of a DDG is the number of instructions in the DDG divided by the critical path length of the DDG.

2.3 Vulnerability

A program A, which is scheduled with P_A resources, is said to be more vulnerable than a program B, which is scheduled with P_B resources, if A loses more of its ideal parallelism when compared to B when $\frac{P_A}{N_A/T_A} = \frac{P_B}{N_B/T_B}$, where N_A and N_B represent the number of instructions in programs A and B respectively with T_A and T_B the time taken. We define the ratio $\frac{P}{N/T}$ as the *normalized number of resources*.

The ability of a program to hold onto its ideal parallelism is measured as the ratio of the execution time with the execution time with infinite resources. This is termed as *Normalized Time*.

[2]corresponding to the greedy schedule
[3]corresponding to the lazy schedule

3 Scope of Study

3.1 Measures of Vulnerability

Earlier work (Rauchwerger *et al.* [1]) suggested that slack could be used as a measure of the vulnerability of parallelism to resource constraints. We have considered both accurate and approximate slacks. In addition, we have considered *filtered slack* to eliminate effects due to *tapering* (see below). We have also studied the standard deviation of load as a candidate measure. Finally, we have also looked at various combinations of load and slack to investigate which of them is suitable and their accuracy and predictive ability.

3.2 Low-level Scheduling Algorithms

We have scheduled the DDGs in this study by four list scheduling algorithms under resource constraints: naive, lazy, slack and block. List scheduling algorithms order all the instructions available for execution in the next time (the ready instructions) step using a local priority function without any backtracking. The various algorithms above differ in the choice of the local priority function. The *naive scheduling* algorithm orders the set of ready instructions in the increasing order of their earliest start times. The *lazy scheduling* algorithm orders the set of ready instructions in the increasing order of their latest start times. In case of a tie, the algorithm gives priority to an instruction with the least earliest start time. The *slack scheduling* algorithm orders the set of ready instructions in the increasing order of their slacks. In case of a tie, the algorithm gives priority to an instruction with the least earliest start time. The *block scheduling* algorithm orders the ready instructions in the increasing order of their block slacks. In case of a tie, the block with a greater number of instructions is given more importance. The block scheduling algorithm is based on the fact that there is only one instruction in the block that can be ready at any given instant of time. The algorithm schedules one instruction from each of the first few blocks in the ready list. The algorithm then modifies the block slacks in the ready list. This is done by decrementing the number of cycles of all the blocks in the ready list. The number of instructions for the blocks from which an instruction has been scheduled is also decremented.

3.3 Functional Units

We have assumed that functional units are capable of executing any instruction in the first part of the study. We later relax this so that different types of functional units are possible (Section 6.6).

4 Simulation Methodology

An instruction level simulation of the RS/6000 instruction set is done using dynamic instruction traces [6] containing each instruction that is executed and the data addresses for all the memory references. We have designed and implemented an analysis tool which accepts as input these dynamic traces and outputs the DDG that could be executed on an abstract machine with unlimited functional units and various constraints on the rest of the resources.

In order to obtain the oracle limit of parallelism present in a program, we assume perfect branch prediction. Using the option of unlimited renaming of registers and memory, the tool is capable of providing a DDG containing only essential dependences. For simplicity, we assume that all the operations are single cycle operations. The scope of concurrency detection (instruction window) is unlimited. Thus, two instructions which are arbitrarily far apart in the trace can be present in the same level in the DDG.

4.1 Construction of the DDG

For every instruction that is executed, the trace contains the opcode of the instruction. If the instruction makes a memory reference, the memory address accessed is also a part of the trace. The parallelizer extracts the instruction from the trace and computes the earliest time an instruction can be executed, taking into account all the data dependences that have to be satisfied, and places the instruction in the appropriate level in the DDG. Essential dependence is enforced by simply waiting for all the input resources. Anti and output dependences are enforced by waiting for the destination resources also. The *ready time* of a resource is the time at which the data contained in the resource is valid. The *definition time* of an instruction is the time when all the source operands and all the destination operands are ready. This gives the earliest start time of an instruction. The *first use time* of an instruction is the time of first use of any of the resources defined by the instruction. This gives the latest start time[4] of the instruction. The IBM RS/6000 instructions can potentially define several resources at the same time. For example, the load update instruction (lu) defines a memory address and also a register variable. In order to keep an accurate first use stamp of an instruction, we have to keep links between these simultaneously defined resources. When a resource is being redefined these links are broken and new links are established. Every resource has an instruction id, which identifies the defining instruction. When a resource that has a valid instruction id is being redefined and it has no links to other registers or memory locations, then that instruction is eligible to be *emitted*. An instruction that is emitted is inserted into its corresponding level in the DDG. If a resource is never redefined during the execution of the rest of the program, then it will wait for emission at the end of the program.

[4] $Lstart_{app}$ mentioned in Section 2.1

4.2 Data Structures

In order to enforce the dependence relations, we declare shadow structures for all the visible system resources, the registers and memory. There is a shadow structure for every register and every memory object in use. Each such storage resource has a shadow structure whose fields record: ready time, instruction id, last use time stamp and link to co-defined resources.

The DDG is stored as a hash table indexed by the earliest start time of an instruction. The DDG also needs to keep track of the dependences between various nodes in the graph, since this information is required while scheduling the DDG for a given number of resources.

The size of the structure for an instruction in the DDG with two operands is 28 bytes. Hence, the amount of space needed to maintain the shadow structures and the DDG is substantial. Due to the large amount of space needed to maintain the DDG and the shadow structures and the unlimited scope of concurrency detection, it is not possible to construct the DDG for the entire trace. The analysis tool can generate the DDG for a program, after tracing a fixed number of instructions. To carry the study further, we have developed a method of identifying loops from a trace stream. Since loops are likely to be the most executed portions of a program, latching into loops will help us to trace regions of high dynamic frequency execution. This handles another problem: program startup is often anomalous and so a million-instruction sample should be taken from the middle of the benchmark, perhaps after a large random number of instructions have already been executed. Detecting loops helps in achieving this objective.

4.3 Loop Detection

Due to the difficulty of generating traces for a large number of instructions due to memory constraints, we have generated traces for those portions of the program that are more frequently executed by latching onto loop bodies and scheduling the DDGs generated using these traces.

This section describes our algorithm for detecting loops given a dynamic execution trace of a program. The *source* of a branch instruction is the address of the branch instruction and the *target* of a branch instruction is the target address of the branch. We define a *backward branch* as a branch whose *target* is lesser than the *source*. The algorithm assumes that a branch forms the tail of a loop if it is *backward branch* and the branch is taken atleast twice. The *target* of this branch is considered the head of the loop.

The algorithm groups all tails of a branch at the tail with the greatest value of the *source*. Let the *target* for branch B be represented by T_B. During the execution of a backward branch B, the program may or may not be in a loop with the loop head given by T_B. If the program is in a loop with the loop head given by T_B, the branch B is nothing but another of the tails of the loop with the head at T_B. If the program is not in a loop with the loop head given by T_B, the branch B could be the tail of another loop.

A branch, B_1 belongs to the same group as another branch B_2 if they both B_1 and B_2 have the same *target* and the program is already in a loop due to the branch B_2. For each backward branch, the group to which the branch belongs is identified. If a branch to the *target* of this group has been taken by members of this group more than once, the program is in a loop.

For every branch, if the *target* of the branch exceeds the *target* of the innermost nested loop, we assume that we have exitted from the innermost nested loop. Thus, the algorithm speculatively exits from a loop.

5 Experiments Conducted

`compress`	Performs data compression on a file
`eqntott`	Translates boolean eqn into table
`hydro2d`	Solves Navies-Stokes equations
`li`	A lisp interpreter C solving 9 queens
`lud`	Solves $Ax = b$ by LU factorization
`ocean`	Simulates eddy currents in a basin
`spice`	Simulates an electronic circuit
`trfd`	A kernel simulating a 2-electron integral transformation
`uncompress`	Uncompresses a compressed file

Table 1: The benchmarks used

The DDG generated by the analysis tool is scheduled by the scheduler under various constraints on the number of functional units. The functional units are assumed to be capable of executing any instruction. (We later assume different types of functional units in Section 6.6.) The scheduler also assumes that all instructions execute in a single cycle. We have scheduled the DDGs by the following four list scheduling algorithms: naive, lazy, slack and block. Results of using traces from loop is discussed in Section 6.7.

We have compared various parameters including slack, $\sigma(L_x)$ as possible measures to predict vulnerability. For the purposes of this study the DDGs have been obtained from the first one million instructions of the benchmarks in Table 1. The results are presented in Section 6.

We have also scheduled the DDGs assuming different types of functional units. For this purpose, the IBM RS/6000 instruction set has been divided into four categories as follows: *logical* (all branch & those that operate on condition register; executed by instruction and cache unit), floating-point (executed by floating point unit), memory (ld/st; executed by fixed point unit) and integer (arithmetic or logical operations on integer operands which are stored in the GPRs; also executed by fixed point unit).

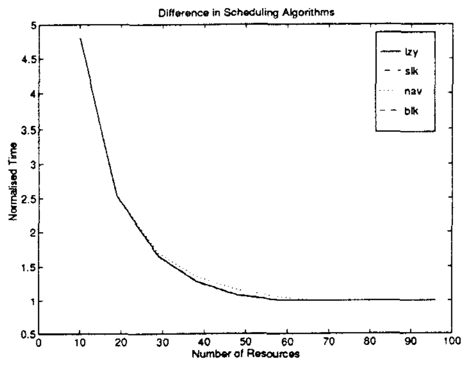

Figure 1: Difference between Scheduling Algorithms

6 Results

6.1 Effect of Scheduling Algorithms

In order to study the effect of different scheduling algorithms on the vulnerability, we scheduled all the DDGs using the four scheduling algorithms, naive scheduling, lazy scheduling, slack scheduling and block scheduling. Our results indicate that there is very little difference between the execution times for the schedules generated using different scheduling algorithms. Figure 1 shows the difference between the various scheduling algorithms for the DDG obtained from `spice`. From this, we have concluded that the differences in the scheduling algorithms can be neglected for the purposes of this study.

Though the various scheduling algorithms did not have a perceptible impact on the execution times, it should be emphasized that a higher-level scheduling of a program based on its vulnerability is the issue that this study addresses.

6.2 Tapering

It has been observed in a few cases that some levels in a DDG have fewer instructions than the remaining levels. On closer inspection, the profile of instructions per level (*parallelism profile*) of the DDG reveals that the levels in which there are fewer number of instructions are significant and clustered. This phenomenon was first observed by sampling the parallelism at various levels (corresponding to 5%, 10% ... 100% of the total number of instructions) of the DDG. The plot of the parallelism as a function of the cumulative number of instructions scheduled is called *parallelism plot*. This phenomenon has been named tapering effect due to the effect it has on the parallelism plot of the DDG. Figure 2 and Figure 3 show the parallelism

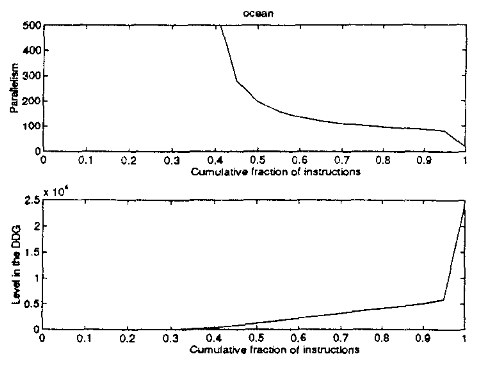

Figure 2: Parallelism plot of the DDG for ocean

% of instructions	parallelism	DDG Level
70	109.5	3139
75	101.76	3619
80	95.86	4099
85	91.17	4579
90	87.43	5056
95	80.69	5782
100	19.94	24626

Table 2: Parallelism in the DDG for ocean

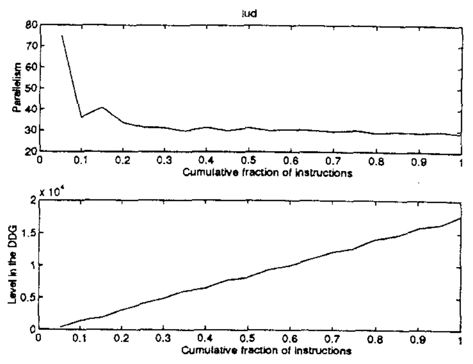

Figure 3: Parallelism plot of the DDG for lud

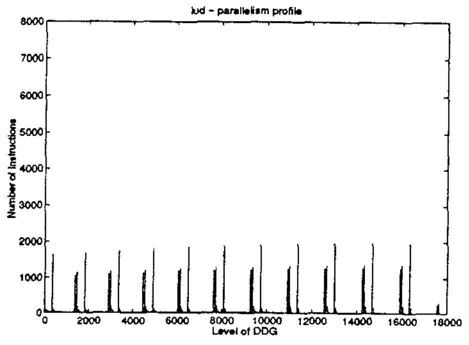

Figure 4: Parallelism profile of lud DDG for 512K instrs

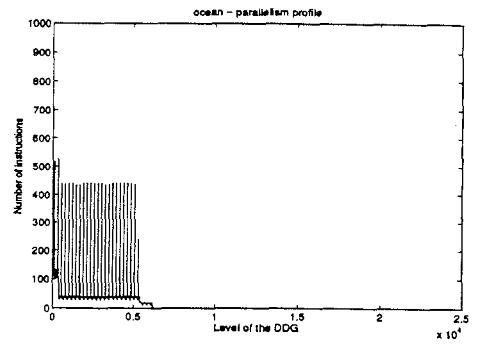

Figure 5: Parallelism profile of ocean DDG for 512K instrs

plots for the DDGs of ocean and lud respectively. It can be observed from Figure 2 and Table 2 that the instructions in the last 75% of the levels in the DDG of ocean are only 5% of the total instructions. It can be seen from Figure 3 that tapering is not observed in the parallelism plot of the DDG obtained from lud. The effect of tapering on the parallelism profile is a sudden sharp decline in the number of instructions per level of the DDG. Figure 5 and Figure 4 show the parallelism profiles of the DDGs of ocean and lud respectively. The effect of tapering on the parallelism profile can be observed in Figure 5, where the number of instructions in most levels of the DDG in the range 5000 to 25000 is close to zero.

Since the number of resources for which the DDG is scheduled depends on the amount of parallelism in the DDG, tapering effect plays an important role. Table 2 shows that there is considerable difference between the parallelism for 95% of the instructions and for 100% of the instructions. Hence, we schedule only that portion of the DDG which is not affected by the tapering effect. For the example in Table 2, the parallelism is chosen to be equal to 81 and only the first 5782 levels of the DDG are scheduled. Figure 6 shows the parallelism profile for the DDG of ocean when only the initial 5782 levels of the DDG are considered. It can observed that the tapering effect is absent in Figure 6.

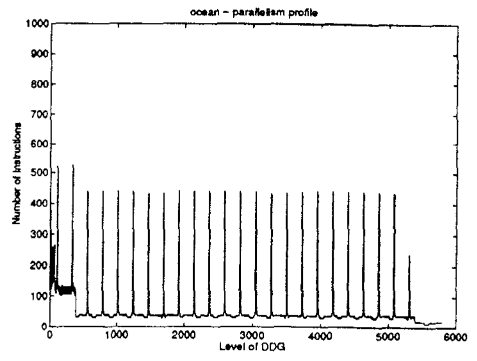

Figure 6: Parallelism profile of ocean DDG for 1st 5782 levels

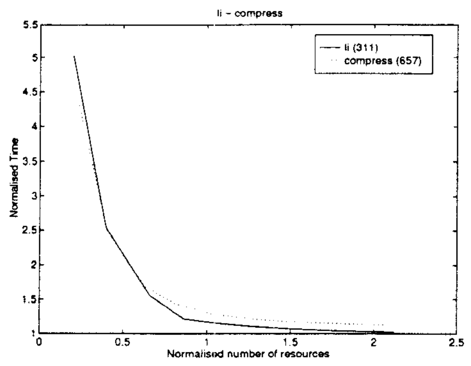

Figure 7: `li`'s DDG is less vulnerable than `compress`'s DDG (number in bracket refers to accurate slack of DD G)

	Approx Slack	Filtslk			
		85%	90%	95%	99%
`compress`	381.90	91.67	90.56	89.14	88.29
`eqntott`	332.55	89.48	103.51	125.56	148.75
`hydro2d`	52.26	17.74	17.84	25.77	58.02
`li`	180.98	43.56	40.09	36.37	33.09
`lud`	53.46	30.66	50.40	51.01	46.12
`ocean`	784.61	72.58	77.55	80.63	71.75
`spice`	172.13	27.14	27.32	28.66	37.64
`trfd`	927.06	72.95	67.22	59.17	53.11
`uncompr`	434.23	158.00	177.30	278.76	368.80

Table 3: Filtered slack for the various DDGs

6.3 Slack as a Measure of Vulnerability

Rauchwerger *et al.* [1] suggest that slack could be used as a measure of the vulnerability of parallelism to resource constraints. A higher slack in the DDG implies greater freedom in scheduling instructions, which could lead to better utilization of the resources. However, although there are instances where a DDG with greater slack is less vulnerable than a DDG with lesser slack, there are also cases where the opposite is true. It can be seen in Figure 7 that a DDG with a slack of 311 is less vulnerable than a DDG with a slack of 657.

One of the reasons for slack not being an accurate measure of the vulnerability is that a few instructions with very high slacks can increase the slack of the entire DDG. To evaluate the effect of instructions with very high slack, we have computed the slack of the DDG by neglecting the slacks of instructions which have either very high or very low slacks. This measure of the slack is called *filtered slack*. Filtered slack can be calculated by not considering the slacks of various percentages of instructions. A filtered slack of $x\%$, implies that $x\%$ of the total number of instructions have been considered in calculating the filtered slack. The slacks of the outliers ($(1-x)\%$ of the instructions) are ignored in calculating a filtered slack of $x\%$ as they either have a very high or a very low slack. The filtered slacks of various programs are given in Table 3, which illustrates the effect on slack due to instructions with a very high slack.

A large slack in a level of the DDG accompanied by a small number of instructions, implies that even though these instructions can be scheduled in later cycles, they are scheduled in this cycle because the number of resources is sufficient. Hence, for slack to be an accurate measure of the vulnerability, it is essential for the the large slack to be accompanied by a large number of instructions. We have observed that there is no strong correlation between the *slack profile* and the parallelism profile, implying that a large slack is not always accompanied by a large number of instructions. *Slack profile* is the profile of the average slack of the instructions in a level of the DDG. The correlation between the slack profile and the parallelism profile is shown in Table 4.

benchmark	correlation
`compress`	0.036
`eqntott`	0.329
`hydro2d`	0.289
`li`	0.066
`lud`	0.175
`ocean`	0.377
`spice`	0.08
`trfd`	-0.001
`uncompress`	0.571

Table 4: Correlation between slack and parallelism profile

6.4 Load as a Measure of Vulnerability

For every scheduling algorithm, S, we define $T_{S,N}(D)$ to be the execution time of the DDG D, when D is scheduled with N resources.

Theorem 1 *If in a dependence graph, D, $\sigma(L_x)$[5] $= 0$, then $\exists$ a scheduling algorithm, S, such that $T_{S,MP}(D) = T_{S,\infty}(D)$, where MP is the mean parallelism of D.*

Due to the above theorem (stated without proof due to lack of space), one could hypothesize that a DDG with a lesser value of $\sigma(L_x)$ indicates that the DDG is less vulnerable to resource constraints. However, even $\sigma(L_x)$ is not an accurate measure of the vulnerability.

6.5 Some More Measures of Vulnerability

We have seen that neither slack nor $\sigma(L_x)$ are accurate measures of the vulnerability of parallelism to resource constraints. It is clear that a high slack and a low $\sigma(L_x)$ are desirable if a DDG is to be less vulnerable to resource constraints. Based on this, we have considered a few other

[5] *sigma*(x) is the standard deviation of the random variable x

Measure	Corr	Wrong	Inconsis	Unpred
AccSlk	47.2	36.1	16.6	0
AppSlk	41.6	41.6	16.6	0
FiltSlk	50	33.3	16.6	0
StdLoad	69.4	13.8	16.6	0
FiltLoad	55.5	27.7	16.6	0
AccSlk-Load	86.6	6.6	6.6	58.3
AppSlk-Load	80	13.3	6.6	58.3
FiltSlk-Load	78.9	10.5	10.5	47.2
FiltSlk-FiltLoad	59.1	22.7	18.2	38.8

Table 5: Prediction accuracies (percentages) of various measures

measures, which are combinations of slack and load, to predict the vulnerability. Let A and B be two different DDGs. Consider the following cases:

- $slack(A) > slack(B)$ & $\sigma(L_x(A)) < \sigma(L_x(B))$: In this case we predict that A is less vulnerable to resource constraints than B.
- $slack(A) > slack(B)$ & $\sigma(L_x(A)) > \sigma(L_x(B))$: In this case nothing is predicted regarding the vulnerabilities of A and B.
- $slack(A) < slack(B)$ & $\sigma(L_x(A)) < \sigma(L_x(B))$: In this case also nothing is predicted regarding the vulnerabilities of A and B.
- $slack(A) < slack(B)$ & $\sigma(L_x(A)) > \sigma(L_x(B))$: In this case A is predicted to be more vulnerable than B.

To summarize, while using both slack and load to predict vulnerability, a prediction is made if and only if the predictions made using slack and $\sigma(L_x)$ agree.

We have compared the following as measures of vulnerability with a view of comparing their prediction accuracies: accurate, approx slack (AccSlk, AppSlk); 95% filtered slack (FiltSlk); $\sigma(L_x)$ (StdLoad); filtered load (FiltLd) [6]; accurate, approx slack, FiltSlk each with $\sigma(L_x)$ (AccSlkLoad, AppSlkLoad, FiltSlkLoad); both FiltSlk & FiltLd (FiltSlk-FiltLoad). The outcome of the prediction by each of these measures can be one of the following: *corect*, *wrong*, *inconsistent* (there is no discernible difference between the DDGs compared, but a prediction is made which is inconsistent with the observation), *unpredicted* (the measure used is unable to make any prediction regarding the vulnerabilities; applies measures which are combinations of slack and load).

We have made pairwise comparisons of the DDGs of the nine benchmarks to obtain the prediction accuracies

[6]We have neglected cycles with either a very high or a very low load in the calculation of $\sigma(L_x)$. This is analogous to filtered slack and is included here only for the purpose of completion. Only 95% of the levels of the DDG are considered in calculating $\sigma(L_x)$.

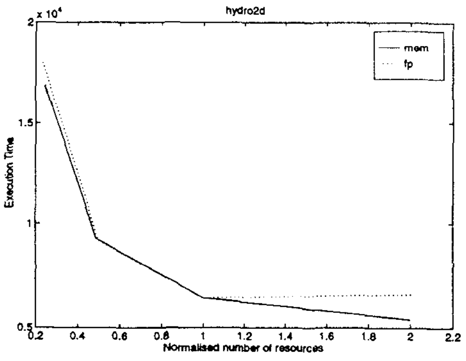

Figure 8: Scheduling a hydro2d DDG for different memory and FP units

of the various measures considered. Table 5 gives the prediction accuracies of each of the measures considered. In Table 5, %Correct, %Wrong and %Inconsistent are calculated as percentages of the number of predictions made. From Table 5, the highest prediction accuracy is obtained when both accurate slack and Load are used as a measure of vulnerability. Also, although the prediction accuracy is high using this measure, in more than 50% of the instances, there is no prediction.

Although the prediction accuracy of AccSlk-Load is very high, the rate of prediction is low. On the other hand, while using Load as the measure to predict we get a good prediction accuracy with a prediction in all cases.

The above results have been obtained by studying the vulnerability of the DDGs obtained from the initial one million instructions of the trace. Since, these instructions are all in the initial stages of execution of the program, we constructed DDGs for a set of instructions which are executed after 10 million instructions. We have compared the vulnerability of these DDGs with the predictions made by the various measures considered and now find that $\sigma(L_x)$ has a prediction accuracy of 61%.

6.6 Different Types of Functional Units

The above results have been obtained by assuming that the functional units are capable of executing any instruction. In this section, we present results for the case when a given functional unit is capable of executing only certain types of instructions.

We have scheduled each DDG with varying number of functional units of each kind. The number of functional units used of each type is proportional to the number of instructions of each type. We have found that the DDG is more vulnerable to the availability of the functional unit with a greater $\sigma(L_x)$. Figure 8 compares the effect of scheduling a DDG of hydro2d with *memory units* and *floating point units*. The $\sigma(L_x)$ values for the *memory instructions* and *floating point instructions* are 154 and 201 respectively. ¿From the Figure 8, it can be seen that the DDG is more vulnerable to the functional unit with a greater $\sigma(L_x)$.

6.7 Scheduling DDGs of loops

We have obtained the DDGs of various benchmarks by tracing only the loops of the benchmark. A loop is only traced once, thus reducing the number of DDGs generated. We have scheduled the DDGs obtained from these loops and have observed that $\sigma(L_x)$ predicts vulnerability with an accuracy of 65%. AccslkLoad has a greater accuracy of prediction, but the prediction rate is lesser than 50%. Since these results are close to other results that are not based on loop detection, we believe that our results reported previously are qualitatively valid.

7 Conclusions

In this paper, we have attempted to characterize the vulnerability of parallelism to resource constraints by constructing DDGs from instruction traces and scheduling these DDGs for various numbers of functional units. The effect of scheduling algorithms on the execution time of the DDGs for the benchmarks we have considered is negligible.

We have considered a number of candidate measures of vulnerability but none of these measures is an accurate measure of the vulnerability. Although Accslk-Load has a high prediction accuracy of about 85%, the rate of prediction is only 42%. On the other hand, even though $\sigma(L_x)$ has a prediction accuracy of 70%, there is a prediction in all cases.

We have also scheduled each DDG assuming the existence of four different types of functional units, *logical units*, *floating point units*, *memory units* and *integer units*. We have observed that the DDG is more vulnerable to the functional unit with a greater $\sigma(L_x)$.

7.1 Future Directions

In the DDGs that have been used for this study, the number of instructions is $512k$, which is small compared the total number of instructions executed by the benchmarks. The limitations of the size of memory for keeping the DDG in memory currently prevent any larger set of instructions to be considered (our machine for the simulations has been a RS6000 server 590 with 256MB of memory). Since the choice of the scheduling algorithm does not play an important role in deciding the vulnerability, one could dispense with the DDG and schedule the instructions while tracing the benchmarks. This approach has the additional advantage of eliminating the effect due to tapering.

The measure with the highest prediction accuracy (Accslk-Load) has a very low prediction rate while both slack and $\sigma(L_x)$ make a prediction all the time. Since Accslk-Load makes a prediction if and only if both slack and $\sigma(L_x)$ make the same prediction, a low prediction rate of Accslk-Load implies that the predictions using slack and $\sigma(L_x)$ disagree most of the time. A closer examination of the circumstances in which slack and $\sigma(L_x)$ make correct predictions will help in obtaining a better measure of vulnerability.

This study assumes that all instructions have unit cycle latencies. One could study the effect of instructions with non-unit cycle latencies on the prediction accuracies of the various measures that have been considered.

References

[1] L. Rauchwerger, P. K. Dubey, R. Nair. Measuring Limits of Parallelism and Characterizing its Vulnerability to Resource Constraints. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 105–117, Dec 1993.

[2] T. Austin and S. Sohi. Dynamic Dependency Analysis of Ordinary Programs. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 342-351, May 1992.

[3] M. Butler, T. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebarow. Single Instruction Stream Parallelism is Greater Than Two. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 276–286, May 1991.

[4] Shankar Ramaswamy *et al.*. A Convex Programming Approach for Exploiting Data and Functional Parallelism on Distributed Memory Multicomputers. In *Proceedings of 1994 ICPP, Chicago*, 1994.

[5] M. Lam and R. Wilson. Limits of Control Flow on Parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 46–57, May 1992.

[6] Ravi Nair. Profiling IBM RS/6000 Applications. *International Journal of Computer Simulation*, 6(1):101-111, 1996.

[7] K. Theobald, G. Gao, and L. Hendren. On the Limits of Program Parallelism and its Smoothability. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 10–19, Dec 1992.

[8] D. Wall. Limits of Instruction Level Parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, April 1991.