# Extraction of Massive Instruction Level Parallelism

Augustus K. Uht
Dept. of Electrical Engineering
Kelley Hall
Univ. of Rhode Island
Kingston, RI 02881, USA

## Abstract

*Our goal is to dramatically increase the performance of uniprocessors through the exploitation of instruction level parallelism, i.e. that parallelism which exists amongst the machine instructions of a program. Speculative execution may help a lot, but, it is argued, both branch prediction and eager execution are insufficient to achieve performances in speedup factors in the tens (with respect to sequential execution), with reasonable hardware costs.*

*A new form of code execution,* Disjoint Eager Execution (DEE)*, is proposed which uses less hardware than pure eager execution, and has more performance than pure branch prediction; DEE is a continuum between branch prediction and eager execution. DEE is shown to be optimal, when processing resources are constrained.*

*Branches are predicted in DEE, but the predictions should be made in parallel in order to obtain high performance. This is not allowed, however, by the use of the standard insrtruction stream model, the dynamic model (the order is as indicated by the contents of the Program Counter).*

*The use of the static insruction stream is proposed instead. The static instruction stream oreder is the same as the order of the code in memory, and is independent of the execution of branches. It allows reduced branch dependencies, as well.*

*It is argued that a new version, Levo, of an old machine model, CONDEL-2, will be able to attain massive Instruction Level Parallelsim.*

## 1 Introduction

This document is part of a research proposal; background problems are discussed, and potential solutions are given. Details of the proposed research suggested by this paper may be found in[20].

The primary goal of this research is to improve the performance of uniprocessors. The main method employed to achieve the improvement is the extraction of concurrency or parallelism from the single, standard, imperative instruction stream of the computer. The degree of concurrency to be extracted is a speedup factor in the tens. The classic methods of concurrency extraction[14, 16] have sought to obtain their concurrency through the reduction of data dependencies alone. This produced modest gains.

The most potent and novel concurrency extraction techniques work via the lessening of the ill effects of branches, or conditional instructions, in the input code. Classically, all instructions dynamically after a branch are dependent on the branch. In recent work [18] a new model of *reduced branch dependencies* was constructed and exploited to significantly reduce the execution time of programs. In this model, instructions not in a branch's *domain*[1] may be independent of the branch, and able to execute concurrently with it; branches may execute concurrently with branches.

*Speculative execution*, an orthogonal technique, has been used separately to great advantage [9, 11]. An instruction is executed speculatively if it is executed before a prior-occurring branch has executed, or in other words, before it is known for sure which path the branch will take. Traditionally, there are two forms of speculative execution: *branch prediction*, in which code execution proceeds past an unresolved or not fully executed branch on the most likely path; and *eager execution*, in which code execution proceeds past an unresolved branch on both paths. Branch prediction is the more efficient technique, the hardware cost growing linearly in the number of dynamic branches predicted at once; while eager execution is more costly, the hardware cost growing exponentially in the number of unresolved branches; but it is the best performing technique, since all possible desired computations are determined simultaneously with no branch penalty.

The first major objective of the proposed research is to establish the worth of a new method of speculative code execution, *Disjoint Eager Execution (DEE)*, and its combination with the reduced branch dependencies technique. DEE is derived in Appendix A; briefly, it combines branch prediction, or speculative execution down one path of a branch, with eager execution, or speculative execution down both paths of a branch. DEE assigns branch path execution resources in an optimal way, to the paths most likely to be taken. For example, consider the execution of a program, in which several branch predictions have been made, so there are many unresolved branches at once. Starting with the first branch, B1 (say its True path is the most likely); then,

[1] A branch's domain consists of the static instructions from the branch to the target; also see [18].

after some number of less than perfect accuracy branch predictions, the main path of execution, down the individually most likely paths of several branches, is no longer the most likely path overall; rather, the False path of branch B1 is now more likely, and execution resources should be assigned to it. Thus, B1 is now eagerly evaluated, or rather, more properly, DEE, since both its paths are taken, but starting at different times. In the proposed work, DEE will be characterized and its performance gains measured. It is hoped that it will combine the performance of eager execution with the efficiency of branch prediction.

As will be shown in the following section, the best results of branch prediction occur when many branches are predicted at once; the performance gain increases with the degree of speculative execution allowed. (This is also true of eager execution, not suprisingly.) In fact, to get anything like the large performance gains predicted in the literature, *massive speculative execution* must be employed. Realizing a machine capable of this is non-trivial: it is hard, in the conventional view, to think of multiple branches executing at once, much less predicted at once. Combining this with reduced branch dependencies has also been thought to be difficult.

## 2 Background and Motivation

In this Section the literature is reviewed, showing that substantial Instruction Level Parallelism exists in both scientific and general purpose codes. This was demonstrated by a number of researchers using one of two idealistic machine models as their base, namely:

**Oracle:** in which perfect branch prediction is assumed, so branch penalties are 0, and branches do not inhibit concurrency extraction in the code;

**Bypassing:** or eager execution, in which execution proceeds down both paths of a branch (the wrong path eventually being thrown away), thereby bypassing the branch, and reducing the branch penalty to 0; in this case, too, branches do not inhibit concurrency extraction.

Note that eager execution may be realizable[21, 23], whereas an oracle is not.

Once the potency of Instruction Level Parallelism is established, the question then becomes: *How do we get there? I.e., can a realistic machine model be devised to exploit this order of Instruction Level Parallelism?* It is our thesis that no model exists presently, but that one can be devised using new ideas developed in this proposal. In the remainder of this Section the questions are answered and our thesis is propounded.

### 2.1 Idealistic Speedups in the Literature

The studies considered are listed chronologically in Table 1, along with the harmonic mean of their benchmarks' speedups. The benchmarks are also described or named.

| *Study:* | *Benchmark code type:* | *Average Speedup (harmonic mean):* | *Notes:* |
|---|---|---|---|
| Riseman[12] | general purpose | 25.65 | |
| Nicolau[7] | scientific | 12.13 | Branching was predictable. |
| Kumar[5] | scientific | 500-3500 | Speedup is in FORTRAN lines (statements) on average. Simulations were done <u>without</u> assuming an oracle or eager execution. |
| Butler [2] | SPECmarks | 73.17<br>57.82 | Nasa benchmark not included.<br>Unpredictable benchmarks (see text). |
| Austin [1] | SPECmarks | 66.37<br>68.68<br>53.50 | All 10 benchmarks included. Conservative figure used.<br>Nasa benchmark not included. Conservative figure used.<br>Unpredictable benchmarks (see text). |
| Lam [6] | SPECmarks± | 158.26 | Some non-SPECmark codes used; not all SPECmark codes used. Non-numeric speedup figure used (conservative). |

Table 1: Idealistic speedup studies.

The Riseman and Foster study was the first of its kind, demonstrating that large speedups could be obtained if branches could be completely bypassed (eagerly executed). This aspect of their work has been repeated in kind by the other studies listed in the table. What they did do that has not been repeated was to characterize the speedups as a function of degree of branch bypassing in effect; we shall return to this subject later.

Nicolau's studies were based on a VLIW (Very Long Instruction Word) machine model, so are not completely applicable here. They demonstrate that even with a restricted model, such as VLIW, significant speedups are obtainable. It might also be said that perhaps for unpredictable branches, it might be best to look elsewhere for a suitable machine model. Although progress has been made here [3], such methods still cannot allow for "shrinkwrap" code, i.e., code that conforms to a common interface (machine language) but with different implementations.

Kumar's work demonstrates that there is much parallelism in scientific code. This is reasonable, as scientific code often manipulates large array structures in a parallel fashion, e.g., matrix multiplication. However, no oracle is used in the work, so the actual peak parallelism in the codes studied might be greater than that reported.

(Both the Austin and Butler studies assumed RISC machines, with comparable latencies, for their code interpretation engines. Butler did not include the Nasa SPECmark; the effect of leaving this out of the Austin data is minor and is shown in the table.)

The Austin/Butler experiments repeat the Riseman and Foster experiment for the complete branch bypassing case, but with different benchmarks. They demonstrate sizeable gains. It might be argued that the Austin/Butler figures include too many SPECmarks which are easily predictable and hence contain much "easy" concurrency. However, if we compute the speedups for the Austin/Butler case over just those SPECmarks which are relatively unpredictable (as given in Pan [8]: doduc, spice, fpppp, gcc, espresso, eqntott, and li), the speedups are still quite substantial (in the 50's, see Table 1).

The Austin/Butler speedups are comparable and what one expects with oracle performance on a system in which only hardware has been "modified" to take advantage of Instruction Level Parallelism. In other words, no software enhancements were used to augment the concurrency in the Austin/Butler numbers.

This is not true, however, with the Lam and Wilson experiments. In this study, all of the machine models simulated were with the benchmark codes enhanced. Specifically, the main differences between these numbers and the Austin/Butler numbers are likely due to the procedure inlining and perfect loop unrolling applied to the benchmarks in the Lam and Wilson study.

### 2.2 Determining the Requirements for Massive Instruction Level Parallelism

In this Section branch effect reduction techniques, both ideal and practical, are reviewed, particularly as to the insights they may give us in solving the concurrency extraction problem (the branch effect reduction problem).

In the Riseman and Foster work [12], the effects of restricting the number of branches that can be bypassed at any given time are determined. Perhaps the most telling number is the harmonic mean total speedup when 32 branches are bypassed: 8.77, vs. 25.65 with an infinite number of branches bypassed. If eager execution is used, this means that $2^{32} \simeq 4 \times 10^9$ paths must be maintained, all for a speedup of less than 10! Although it is likely that the number of paths need not be this high (see Uht[21]), it is likely to still be a large number.

This indicates three things: pure eager execution is not a desirable general execution model, many branches must effectively be bypassed to get substantial speedups, and, since this is key data, similar experiments should be run on current benchmarks, i.e., the SPECmarks.

The logical alternative to eager execution is *branch prediction.* In this scenario code is executed speculatively past a branch, before the branch executes, or is resolved. Once the branch resolves, either nothing happens (the prediction was correct), or the work done since the branch was predicted is discarded, and execution restarts at the branch point, down the not-predicted path[2]. But in order to get the same gains as eager execution, the predictions must be very good, in order to approximate 100% (perfect) prediction (recall that an oracle has the same gains as eager execution).

Seeing how good the branch prediction methods are, we note that the classic 2-bit counter techniques[13] do very well, especially considering their simplicity, having prediction accuracies on the unpredictable SPECmarks of about 90.3%[3][8]. Recently, some better prediction schemes have been proposed, with that of Yeh and Patt[25] having an accuracy of 96.0%, and a similar one, Pan's[8], having an accuracy of 94.0%.

But are these methods good enough? In the Lam and Wilson[6] work, speculative execution was also studied. Assuming prediction methods with an average prediction accuracy of 90.6%, the speedup of the speculatively executed codes alone was only 6.80; with reduced control dependencies this goes to 13.27. Recall that these are optimistic numbers, including software pre-processing. These numbers are also nowhere near the optimal, oracle, speedup. Although prediction accuracies have improved to 94%, it is doubtful that this accuracy is high enough to give significantly better speedups. It is desirable to repeat the speedup experiments of Lam and Wilson with better prediction mechanisms; however, it is demonstrated below, without going through the tracing mechanism, that the new accuracies are not likely to be high enough; this is just to get a feel for the situation.

It is desired to model the *cascaded* branch prediction accuracies as the depth of predictions increases. First, to be condervative, we assume a basic branch prediction accuracy of 94%. Second, we estimate a prediction reduction factor $f$ of 94%; this number reflects the growing uncertainty of the branch predictions on subsequent predictions of the same static branch, when prior dynamic versions of the branch are still unresolved, and the prediction mechanism must work with this questionable prediction data. In other words, as the predictions get deeper, e.g., with more iterations of a loop, and the number of unresolved branches increases, the quality and accuracy of the predictions declines.

The execution scenario is as illustrated in Figure 1. Five branches, dynamic, are shown on the left, with their individual branch probabilities, and on the right, with their cumulative cascaded probabilities. The terms are defined as follows:

$f$- probability reduction factor; decrease in accuracy of branch prediction at each level of prediction.

$fcum$- cumulative probability reduction factor; $fcum_j = \prod_{i=1}^{j} f_i$;

$p$- branch probability; this indicates the likelihood that the left path (most likely path) will be taken. Note that

[2] The ill effects of restarting can be ameliorated somewhat by using reduced control dependencies[6, 18].

[3] These prediction numbers were computed as the arithmetic means of the prediction accuracies on the 7 unpredictable benchmarks used in [8].

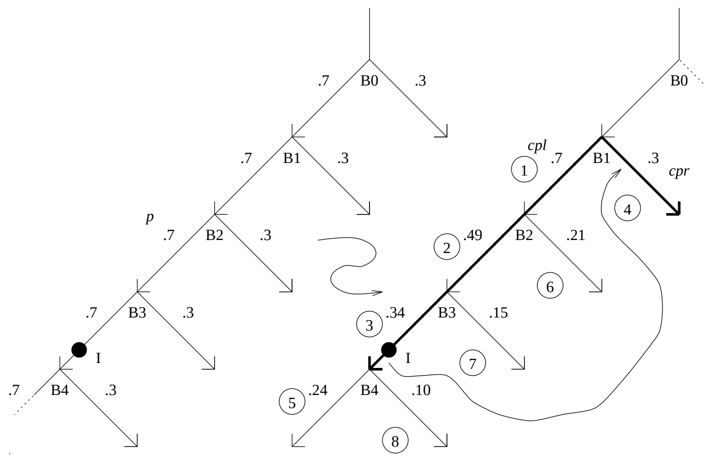


Figure 1: Evolution of Disjoint Eager Execution.

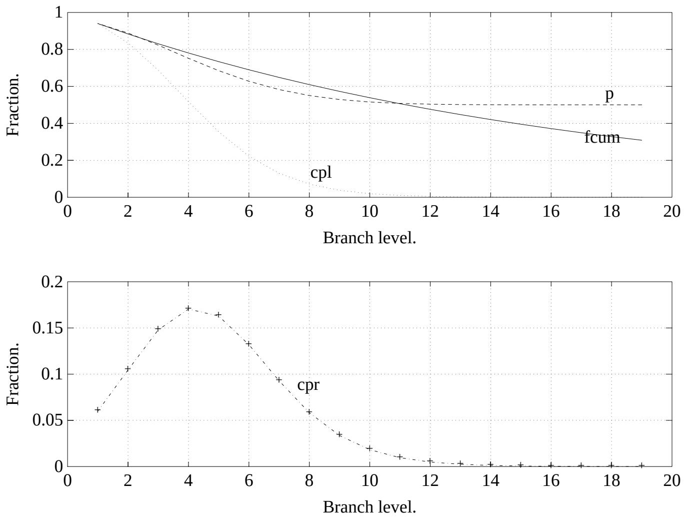


Figure 2: Branch and cascaded probabilities as a function of prediction depth, one branch loop.

either path may be the taken path, as long as the most likely path is on the left.

*cpl*- cascaded probability - left. This is the overall or total likelihood that the left path will be taken; it is equal to the product of all of the $p$'s to that point. Note that in the figure, branches B1-B4 are unresolved.

*cpr*- Cascaded probability - right. This is similar to *cpl*, but is on the right (not predicted) path, so instead of multiplying the last term by $p$, it is multiplied by $(1-p)$.

Note that all of the *cpr*'s, and the last (leftmost) *cpl* add up to 1.00; one of these paths must be taken.

Initially, it is assumed that the same static branch is predicted over and over, with no other intervening static or dynamic branches. This is like a loop with only the terminating branch in its body, no other branches. This results in the plots of the probabilities as a function of branch depth shown in Figure 2. For a discussion of the mathematics behind this figure, see Appendix B.

Going from the poor probabilities of Figure 1 to the realistic prediction probabilities of Figure 2, it is seen that the situation is not as good as might be anticipated from the high branch prediction accuracy. Due to the steady decrease in the probability reduction factor, the branch probability levels off rapidly to 0.5, after about 10 branches. In other words, after 10 branches have been predicted, and are unresolved, it is a toss-up as to which path to take.

The situation is not quite this bad since, extrapolating from [10] data, on average there are 3 forward branches per backward branch domain; this implies the cascaded probabilities of Figure 3. Although the branch probability does not decline as rapidly as in Figure 2, the *cpl* still declines quickly. The *cpl* is small after about 15 branches have been predicted. But, recalling the Riseman and Foster[12] study, it should be possible to go much deeper than 32 branches, eagerly executed, to get even small speedups. What is to be done?

The answer lies in examining the cascaded probabilities, as well as Figure 1. In Figure 1, note that after branch B4, the *cpl* is down to 0.24. In standard branch prediction scenarios, branches would continue to be predicted, and code would execute, down the left path. But this is not optimal! Now, the $cpl = 0.24$, but this path is less likely than the right path of branch B1, for which the path probability $(cpr)$ is 0.3! This means that code execution should proceed down the right path of B1 after going to B4; since both paths of B1 would thus be taken, we say this is a form of eager execution: Disjoint Eager Execution (DEE)[4].

DEE is not always this easy; return to Figure 2. The point at which DEE should begin is when $cpl_i < \max(cpr_j), \forall j$. In this case, it occurs at branch 7, at which $cpl_7 < \max(cpr_j) = cpr_4$. In the more general case of Figure 3, the DEE starting point is when the 16th branch has been predicted; then, $cpl_{16} = 0.080 < cpr_5 = 0.086$. The figure indicates that the optimum order of DEE, after regular prediction, are branches 5, 9, 6, 10, 7, 8, 11, then back to the main prediction line (*cpl*).

From the above discussion, the usefulness of DEE is seen. It is guaranteed to have better performance than pure branch prediction, although worse than pure eager execution, but with less hardware than eager execution.

Two points need to be addressed before utilizing speculative execution in a massive fashion is considered further:

1. What gains can be expected by DEE? What is the hardware cost/performance tradeoff? These will be addressed in the proposed research.

2. What is an appropriate machine model to implement DEE and/or massive speculative execution? This is partially addressed in the next section.

[4] The general derivation is given, and the optimality is shown, of DEE, in Appendix A.

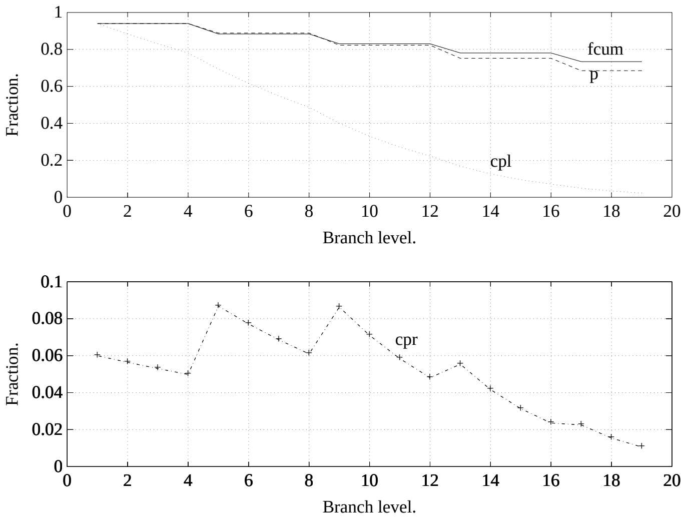


Figure 3: Branch and cascaded probabilities as a function of prediction depth, four branch loop.

## 2.3 Derivation of the Execution Machine Model, With Comparison to Other Work

This section begins with an overview of other work, and then argues for the use of our new machine, Levo.

One of the chief differences between our work and other work is the type of instruction stream order used. The dynamic instruction stream order is the order of instructions seen by the classic CPU, as indicated or followed by its Program Counter. The static order [15, 18, 19, 24] is the order of the code as it exists in memory, i.e., it is independent of the control flow, or branch executions, of the code.

Patt *et al*[9] first combined a classic data dependency reduction scheme, the Tomasulo algorithm[16], with branch prediction to yield the HPS model. Like most superscalar machines [e.g., Intel 586, Metaflow[11]] it processes only one branch per cycle at most, and uses the dynamic instruction stream as its execution model. Executing 1 branch per cycle is much less than what needs to be allowed for; consider Lam and Wilson's[6] peak speedup number of 138; assuming a branch frequency of 1 branch every 4 dynamic instructions, at least 14 branches must be executed every cycle; the average number of unresolved branches is liable to be greater.

Other techniques to reduce the ill effects of branches are briefly described and cited in [20]. None of them use the static instruction stream, and none come near to massive concurrency extraction.

Software can also be used to assist the hardware without explicit communication between the software and hardware[22]. In [17] such software, normally resident in the compiler, is described and its benefits quantified on the CONDEL-2 model. Although the machine instructions' order and selection were modified by the software, the instruction set was <u>not</u> modified, and no information other than typical machine instructions were passed to the hardware. Applying the software in a post-compilation method, to maximize performance amongst different implementations having the same instruction set architecture, is the subject of future work of a different proposal.

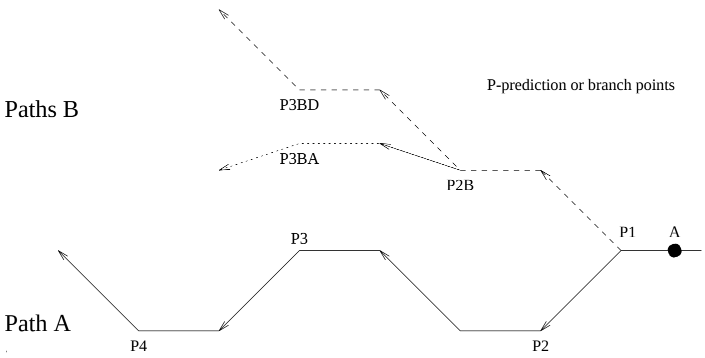


Figure 4: The sequentiallity problem with the dynamic instruction stream.

The work of Wang[23] is also significant in that it combines the use of branch prediction and eager execution for the first time, demonstrating substantial gains, although the machine methodology is somewhat ad hoc and not extendable. It also uses the dynamic instruction stream.

The only current machine model that does use the static instruction stream is CONDEL-2[18, 19]. The use of the static instruction stream allows the CONDEL-2 model to achieve minimal control dependencies, as well as a restricted form of eager execution. Detailed comparisons of CONDEL-2 with other machines may be found in [19]. The new Levo model, proposed herein, builds on a revised CONDEL-2 model.

The other striking aspect of using the static instruction stream is that not only is it desirable for massive concurrency extraction, it is <u>necessary</u>. Consider the dynamic instruction stream: it is necessary to predict all preceding branches before one can predict the current one; each branch is potentially different; <u>but</u> it is necessary to know the earlier prediction before it is known which path to go down, <u>or</u> it is necessary to go down (fetch) multiple paths simultaneously[4]. The latter does not always lead to path explosion[21]; but neither does it solve the problem, it just moves it from the fetch stage to the decode or execute stages.

This problem with the dynamic instruction stream and branch prediction is illustrated by Figure 4. Say code has executed to point A at time $t_A$; it is then desired to execute tens of instructions concurrently, so that instruction fetching and execution must go past, say, for argument's sake, 3 branches at once (4 basic blocks are examined), otherwise concurrency is reduced. Path A is the true path. Therefore, it is desired to predict P1-P3 in the the same cycle. Predicting P1 is straightforward; but it is harder after that. P2 cannot be predicted concurrently with P1 because the result of the P1 prediction is needed to determine if P2 or P2B should be the second branch predicted. Likewise, for P3 (or P3B, etc.), P2 or P2B, and P1 must be predicted first.

Therefore the dynamic instruction stream introduces a sequentiallity in the prediction of branches, since the dynamic order is branch dependent. This is undesirable.

For massive concurrency extraction, it must be possible to accomplish concurrent fetching, prediction, and resolving of much more than one branch at a time. The Dollas technique[4] handles the fetching part of the troika, showing the same convergence of control-flow as first demonstrated in [21]. For branch prediction, however, the problem remains as shown in Figure 4, in that the predictions must be done sequentially in the number of branch levels, at the least, since it is not known which branches to be predicted are on any path without going down that path.

Resolving can happen concurrently, given the predictions in the path are correct. With mispredictions, the earliest misprediction rules.

Therefore, the inherent sequentiallity of the branch predictions in a dynamic instruction stream process renders use of the dynamic instruction stream undesirable. Instead, the static instruction stream is proposed. In this model, instructions in the code window have a constant order, independent of the control flow. Thus, branch predictions occur concurrently, as well as resolvings (fetchings are not an issue); hence massive concurrent code execution is allowed in the static instruction stream. CONDEL-2 uses the static instruction stream model, as will Levo.

### 2.4 Summary of the Problems

There are currently three main impediments to massive concurrency extraction:

1. Lack of an adequate speculative execution strategy. There is insufficient knowledge characterizing the various speculative execution strategies.

2. Branch prediction techniques are inherently sequential with a dynamic instruction stream.

3. A machine model is lacking to realize the derived structures above.

Note that item 3 is relevant regardless of the outcome of solving 1, since eager execution can't be used, and branch prediction is needed in both DEE and pure branch prediction execution strategies.

## 3 Conclusions and Conjecture

The impediments to achieving massive concurrency extraction were reviewed and solutions proposed. A new kind of code execution, Disjoint Eager Execution, was developed and shown to be optimal.

At this time one might "speculate" on what the performance of such a massively concurrent machine as Levo might be. This is difficult but an estimate can be made by comparing it to the models in Lam and Wilson. Their highest performing model, SP-CD-MF, achieves a speedup of 39.62. Levo will have SP, speculative execution (branch prediction), with better prediction accuracy. Levo will also realize the minimal CD's, control dependencies, of CONDEL-2, which are equal or better than the CD model. Levo implements, as does CONDEL-2, a restricted form of MF, multiflow, execution (more than one program counter), as a byproduct of reduced control dependencies. The other factors necessary to compare Levo with SP-CD-MF are the procedure inlining, which Levo does a restricted form of, infinite size code window, which Levo does not have, although it is likely that its window will be large enough to get much of the concurrency[5], and perfect loop unrolling, which Levo can achieve to a large extent. The last factor is the speculative execution model employed; assuming it is warranted, Levo will have DEE, having an as yet unknown, potentially large, gain. This would move Levo closed to the Lam and Wilson Oracle performance, since that is the same as eager execution performance.

Putting the above together, it seems likely that the performance of Levo will be greater than that of the Lam and Wilson SP-CD model, speedup=13.27, and probably better than the SP-CD-MF model, speedup=39.62, closer to the Oracle performance, speedup=158.26. Therefore, we believe Levo will achieve massive instruction level parallelism.

## 4 Comments and Caveat

Comments on this paper are welcome and desired. Physical mail should be sent to the address on the title page. Electronic mail should be sent to: uht@ele.uri.edu.

There was not enough room in this paper to properly cite much relevant work; my apologies to the authers. A more extensive bibliography may be found in [20].

## 5 Acknowledgements

M. Saks helped with the math background of Appendix A, while M. Paturi likewise helped with Appendix B; I am indebted to them both.

## A Derivation of Disjoint Eager Execution

The problem is:

> Given a traditional sequential imperative instruction stream, and concurrent execution of same, how can finite resources be assigned to the differing speculative paths of execution, so as to minimize the likely overall execution time?

[5] The limited window results of Austin and Sohi[1] were combined with the oracle results of Lam and Wilson[6] to make an estimate; concurrency levels like those desired here (speedup factors in the tens) should be supportable by a window of 256 instructions, which is on the low end of what Levo is likely to have.

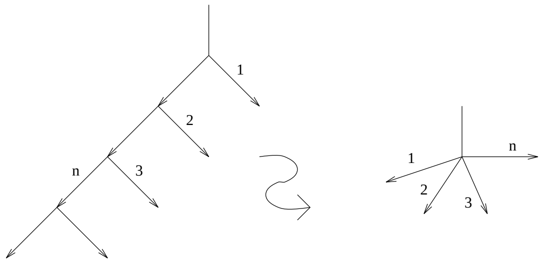


Figure 5: Branch equivalence.

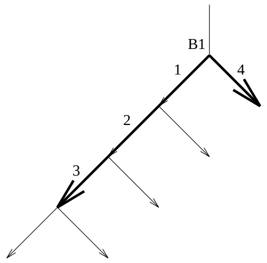


Figure 6: Exposition of Disjoint Eager Execution.

The situation is as shown in the left side of Figure 5. The figure shows some of the potential paths the program might take during execution; it is not known ahead of time which paths to take. The left hand side of the figure can conceptually be thought of as the same as the right side of the figure, a single multiway branch.

Now, note that each path has a certain likelihood of being taken, its path probability; and all the path probabilities add up to 1. Maximizing the performance of the whole execution is like maximizing benefit (performance) over the whole execution. But this is just a well-known math maximization problem; the answer is to put all of the resources on the most likely path. Thus, in computer terms, the solution to maximize performance is to do pure branch prediction, going down the most likely path.

The problem is a little more involved than that, though. Examining the reconstructed paths picture (see Figure 6), as execution begins at branch B1, path 1 is more likely than path 4, so path 1 is taken; all available resources are applied to the computation of path 1. Now, if the computation on path 1 *saturates*, i.e., not all computation resources can be used, there are extra resources available; where should they be put, which path? Since we now have a sub-problem which is just like the beginning problem, we know to assign resources to the most likely paths other than path 1; this is path 2. And so on until path 3 saturates, and then the result of our "rule" of Greatest Marginal Benefit is that path 4 is assigned the spare resources; this is now an instance of *Disjoint Eager Execution (DEE)*.

The rule of DEE (or Greatest Marginal Benefit) may be more concisely stated as follows:

> Assign all remaining resources, up until saturation, to the most likely idle path overall.

Thus, DEE is optimal by construction, and subsumes both branch prediction and eager execution. In Figure 7 the different possibilities are shown, where the bold line indicates execution.

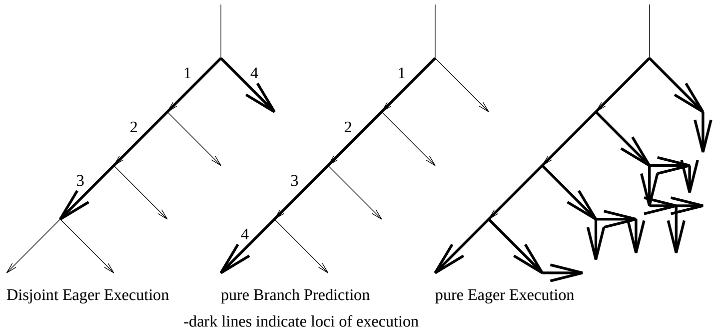


Figure 7: Comparison of the three execution strategies.

# B Mathematics of the Cascaded Probabilities

The recurrence relations of the branch cascaded probabilities model of Figures 1 and 2 are shown below:

$$p_i = f p_{i-1} \quad (1)$$

$$cpl_i = p_i cpl_{i-1} \quad (2)$$

$$cpr_i = (1 - p_i) cpl_{i-1} \quad (3)$$

These may be solved to form the $cpr_i$ function shown below ($\min(i) = 0$):

$$\begin{aligned} p_i &= f^i p_0 \\ cpl_i &= cpl_0 p_0^i f^{\frac{i(i+1)}{2}} \\ cpr_i &= (1 - p_i) cpl_{i-1} \\ &= (1 - f^i p_0) cpl_0 p_0^{i-1} f^{\frac{i(i-1)}{2}} \end{aligned} \quad (4)$$

Equation 4 is the same as that for a gaussian distribution, which is why the lower plots of Figures 2 and 3 have typical bell-shaped curves, although this equation only strictly applies to the first figure, and with a constant probability reduction factor $f$.

To compute Figures 2 and 3, the recurrence relations 1-3 are used directly; for Figure 2 $f$ is constant, while for Figure 3 $f$ varies, being 1 in the middle of a "loop", and less than one at the transition points (end of a loop).

# References

[1] T. M. Austin and G. S. Sohi. Dynamic dependency analysis of ordinary programs. In *19th Annual Inter-*

*national Symposium on Computer Architecture, Gold Coast, Australia*, pages 342–351. IEEE and ACM, May 1992.

[2] M. Butler, T-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow. Single instruction stream parallelism is greater than two. In *18th Annual International Symposium on Computer Architecture, Toronto, Canada*, pages 276–286. IEEE and ACM, May 1991.

[3] K. Ebcioglu. A compilation technique for software pipelining of loops with conditional jumps. In *Proceedings of the Twentieth Annual Workshop on Microprogramming (MICRO-20)*, pages 69–79. Association of Computing Machinery, December 1987.

[4] R. F. Krick and A. Dollas. The evolution of instruction sequencing. *IEEE Computer*, 42(4):5–15, April 1992.

[5] M. Kumar. Measuring parallelism in computation-intensive scientific/engineering applications. *IEEE Transactions on Computers*, 37(9):1088–1098, September 1988.

[6] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *19th Annual International Symposium on Computer Architecture, Gold Coast, Australia*, pages 46–57. IEEE and ACM, May 1992.

[7] A. Nicolau and J. A. Fisher. Measuring the parallelism available for very long instruction word architectures. *IEEE Transactions on Computers*, C-33(11):968–976, November 1984.

[8] S.-T. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV), Boston, Mass.*, pages 76–84. ACM and IEEE, October 1992.

[9] Y. Patt, W. Hwu, and M. Shebanow. Hps, a new microarchitecture: Rationale and introduction. In *Proceedings of MICRO-18*, pages 103–108. ACM, December 1985.

[10] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufman Publishers, Inc., San Mateo, California, 1990.

[11] V. Popescu, M. Schultz, J. Spracklen, G. Gibson, B. Lightner, and D. Isaman. The metaflow architecture. *MICRO, IEEE Computer Society*, 11(3), June 1991.

[12] E. M. Riseman and C. C. Foster. The inhibition of potential parallelism by conditional jumps. *IEEE Transactions on Computers*, pages 1405–1411, December 1972.

[13] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 135–148. ACM-IEEE, 1981.

[14] J. E. Thorton. Parallel operation in the control data 6600. In *Proceedings of the Fall Joint Computer Conference*, pages 33–40. AFIPS, 1964.

[15] G. S. Tjaden and M. J. Flynn. Representation of concurrency with ordering matrices. *IEEE Transactions on Computers*, C-22(8):752–761, August 1973.

[16] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal*, pages 25–33, January 1967.

[17] A. K. Uht. *Desirable Code Transformations for a Concurrent Machine*, chapter 25. Research Monographs in Parallel and Distributed Computing. The MIT Press, 1990.

[18] A. K. Uht. A theory of reduced and minimal procedural dependencies. *IEEE Transactions on Computers*, 40(6):681–692, June 1991.

[19] A. K. Uht. Concurrency extraction via hardware methods executing the static instruction stream. *IEEE Transactions on Computers*, 41(7):826–841, July 1992.

[20] A. K. Uht. Extraction of massive instruction stream parallelism. Technical Report 1292-0001, Department of Electrical Engineering, University of Rhode Island, Kingston, RI 02881, December 1992.

[21] A. K. Uht. Requirements for optimal execution of loops with tests. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):573–581, September 1992.

[22] A. K. Uht, C. D. Polychronopoulos, and J. F. Kolen. On the combination of hardware and software concurrency extraction methods. In *Proceedings of the Twentieth Annual Workshop on Microprogramming (MICRO-20)*. Association of Computing Machinery, December 1987.

[23] S. S. H. Wang and A. K. Uht. Ideograph/ideogram: Framework/architecture for. In *Proceedings of the 23rd Annual Symposium and Workshop on Microprogramming and Microarchitecture, MICRO-23.* ACM and IEEE Computer Society, November 27-29 1990.

[24] R. G. Wedig and M. J. Flynn. Concurrency detection in language-oriented processing systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 805–810. IEEE, October 1982.

[25] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *19th Annual International Symposium on Computer Architecture, Gold Coast, Australia*, pages 124–134. IEEE and ACM, May 1992.