From the *Proceedings of the 28th International Symposium on Microarchitecture, IEEE/ACM, November/December 1995.*

Disjoint Eager Execution: An Optimal Form of Speculative Execution

Augustus K. Uht and Vijay Sindagi
Department of Electrical and Computer Engineering
Kelley Hall
University of Rhode Island
Kingston, RI 02881, USA
email: {uht,vijay}@ele.uri.edu

Abstract

Instruction Level Parallelism (ILP) speedups of an order-of-magnitude or greater may be possible using the techniques described herein. Traditional speculative code execution is the execution of code down one path of a branch (branch prediction) or both paths of a branch (eager execution), before the condition of the branch has been evaluated, thereby executing code ahead of time, and improving performance. A third, optimal, method of speculative execution, Disjoint Eager Execution (DEE)*, is described herein. A restricted form of DEE, easier to implement than pure DEE, is developed and evaluated. An implementation of both DEE and minimal control dependencies is described. DEE is shown both theoretically and experimentally to yield more parallelism than both branch prediction and eager execution when the same, finite, execution resources are assumed. ILP speedups of factors in the ten's are demonstrated with constrained resources.*

1 Introduction and background

The goal of this work is to improve the performance of all uniprocessors, from microprocessors to supercomputers, with an emphasis on general purpose or unpredictable-branch-intensive applications. The architectural enhancements proposed are instruction set independent; our microarchitecture is readily adapted to all machine codes. Therefore, so-called "shrinkwrapped" code, machine code for which no source code is available, will be able to be run on our machine; no recompilation will be necessary. Our work is applicable, in whole or in part, to most current research methods of enhancing performance with Instruction Level Parallelism (ILP): hardware-based, software-based (including VLIW [Very Long Instruction Word][1]), and mixtures of the two[4]. ILP as used herein refers to the parallelism that exists between two or more machine instructions in a program. Up to six instructions may be executed concurrently in current or announced machines, e.g., the IBM POWER2 [14], but there are severe limitations on the mix of simultaneous instructions allowable and the typical average performance gain due to ILP is only at most a factor of 2 or 3 better than an ideal sequential machine.

1.1 ILP enhancement techniques

Enhancing the independence of instructions so that more instructions can be executed in parallel is a key goal of ILP machine design. A *dependency* exists between two instructions if they must be ordered, or sequentialized, for correct program execution. After the earlier instruction executes the dependency is said to be *resolved*, and the later instruction may be executed. *Data dependencies* arise due to instructions having the same source and sink variables in certain combinations. With variable renaming, only *flow* dependencies need be enforced. A flow dependency occurs from one instruction to an earlier instruction if the later instruction has as an input the output or sink of the earlier instruction. *Minimal* data dependencies consist only of flow dependencies.

An orthogonal ILP enhancement technique for reducing the ill effects of branches is the reduction and minimization of *control dependencies* (also called branch or procedural dependencies) [2, 8]. This al-

The Intel Corporation has been most generous by supporting our work through a grant from the Intel Research Council. We also thank the University of Rhode Island Research Office for their support through a Proposal Development Grant. This work was also supported by the National Science Foundation through grant number: CCR-8910586.

Copyright © by the IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes of for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

lows the unconditional execution of some code past a branch, before the branch has executed. To illustrate the theory consider the example below:

```
1. IF (A==2)
2.     THEN Z=3
3.     ELSE Y=5
4. ENDIF
5. B=C+D
6. IF (Q==6) GOTO ---
```

In typical machines all instructions after the branch of instruction 1 are dependent on the branch, including instruction 5; this is a *restrictive* control dependency model. However, with a reduced *Control Dependency* (CD) model[3] instruction 5 can execute independently of, and concurrently with, the first IF statement (instructions 1-4). With the CD model, branches must still execute sequentially. If multiple branches are allowed to execute concurrently, the *Multiple Flow* (CD-MF) model[3] results. The CD-MF model is a minimal control dependency model[8]. In the example, with the CD-MF model instruction 6, another forward or backward branch, could potentially execute concurrently with (including before) the prior branch (instructions 1-4). Using this method with minimal data dependencies alone gives slight gains; however, when branch prediction is also used the gains are large[3].

ILP is enhanced greatly with the *speculative execution* of code: the execution of code after a branch before the branch's condition has been evaluated (before the branch has *resolved*). For any speculative execution technique, we define the *depth* of speculation as the greatest number of levels (l) of branches speculatively executed at a time. In other words, l is the maximum height of a technique's execution tree. For example, in Figure 1 $l_{\text{SP}} = 6$, $l_{\text{EE}} = 2$ and $l_{\text{DEE}} = 4$.

The most common technique used to alleviate the difficulties presented by branches is known as *branch prediction*, or following the single most likely path at a branch. We call this *Single Path* (SP) speculative execution herein, to distinguish it from the actual predicting of branches, which is also used in other speculative techniques. SP is very attractive from a cost viewpoint, having a cost that grows linearly with l, i.e., $\bigcirc(l)$. SP is limited, in that the deeper the speculation, the less likely the latest speculated code will be used; for example, in Figure 1, path 6 of the SP tree has an overall likelihood of execution of only 0.12.

There are two known alternatives to SP that also reduce the ill effects of branches. The first, *Eager Execution* (EE), executes the machine code on both paths of a branch, *bypassing* the branch, giving a branch time penalty of zero, the best performance; but it also has prohibitive cost, being exponential in the number of branches bypassed, $\bigcirc(2^{l+1})$.

Disjoint Eager Execution (DEE) [10] is the second alternative to SP, and is the more promising one. As will be seen, disjoint eager execution conceptually lies between SP and eager execution, but performs better than both of these models when resources are constrained. The basic idea of DEE is to only speculatively execute the code that is the most likely to be needed, over all pending code. *The combined use of DEE and minimal control dependencies can produce order-of-magnitude ILP speedups.* DEE's cost is also attractive: $\bigcirc(kl^2)$, where k is much less than one. This is typically slightly greater than SP's cost, but always much less than EE's cost.

DEE is applicable to more than just hardware-based ILP machines. In fact, as stated by Rau[4], VLIW machines can advantageously employ dynamic scheduling methods of hardware-based machines. For software-based machines, e.g., classic VLIW machines, DEE theory and heuristics indicate which code to execute speculatively. If an ALU is otherwise free in a cycle, DEE indicates which code to assign to it, for the best performance. Similarly, for multiprocessors, DEE can be used to assign spare processors to intelligently speculatively execute code, improving performance.

The primary goal of this paper is to establish the basic worth and performance of DEE. In the future, explicitly limited *Processing Elements* (PE's), non-unit latencies, and a suitable memory system will be studied.

1.2 Prior work

There is a large literature concerning the idealistic speedups of code. The classic study is by Riseman and Foster [5], demonstrating speedups of general purpose code of a factor of 25.65 (harmonic mean, infinitely many branches eagerly executed). This infinite resources case of this work has been repeated many times for different benchmarks, different machine assumptions, and different code preprocessing assumptions; see, for example: [3]. Most of them demonstrate speedups of general purpose codes, assuming hardware-based concurrency techniques with no software preprocessing, by a factor of 10's over sequentially executed code. For a more detailed comparison of many more of these studies, see [10].

Lam and Wilson[3] simulated many abstract models of execution with unlimited resources, including the SP, CD and CD-MF models, as well as combinations of these, namely: SP-CD and SP-CD-MF. For comparison purposes, the SP variants are simulated herein, but with constrained resources.

Much work has also been done on branch prediction, including: [6, 15]. The current best methods have

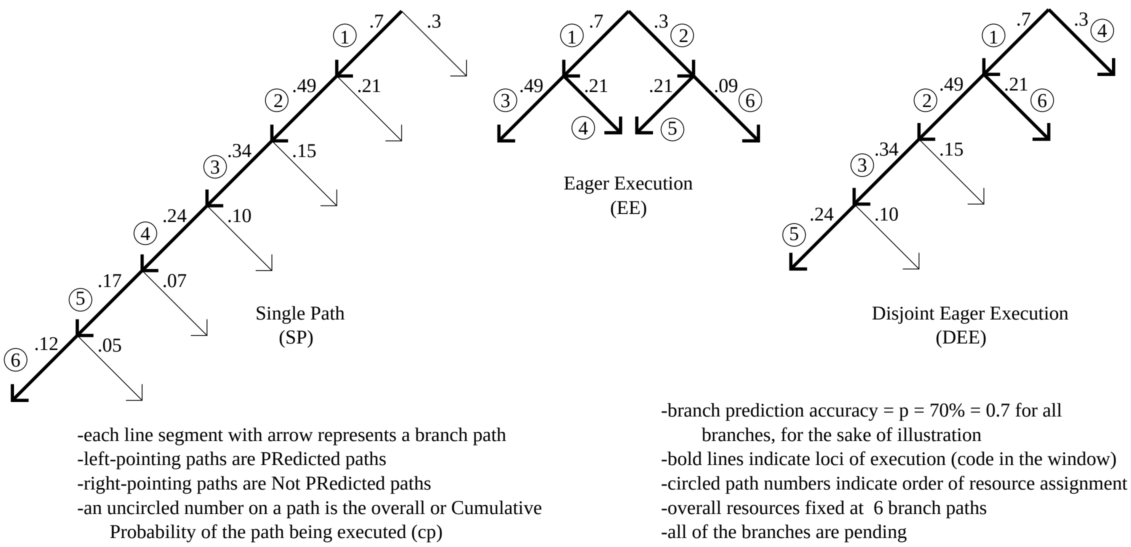

Figure 1: Comparison of the three speculative execution strategies.

prediction accuracies of 90 to 96%, depending on the amount of implementation hardware used.

Other branch effect reduction techniques have been developed. In [2], a method of minimal control dependencies, primarily for software, is developed, while in [8], a different model is presented, primarily for hardware. A machine model realizing both minimal register data and control dependencies is given in [9]. Eager execution is explored in [12].

1.3 Outline

The remainder of this paper is organized as follows. Section 2 derives DEE, proves its optimality and establishes its relationship with both SP and eager execution. A simple and practical form of DEE is described in Section 3. In Section 4, the basic microarchitecture of a machine model (Levo) realizing DEE and minimal control dependencies is presented. In Section 5, DEE and its variants DEE-CD (DEE with reduced control dependencies) and DEE-CD-MF (DEE with minimal control dependencies) are simulated, and are compared to other models. A Summary is given in Section 6.

2 Theory of Disjoint Eager Execution

The key to disjoint eager execution's efficacy lies in the *selection* of which paths to execute. In DEE, execution resources, e.g., Processing Elements (PE), are assigned to the most likely paths to be executed over all *unresolved* (issued but not executed) paths. (A PE, in our models, contains an integer and floating point ALU, branch execution unit and address translation hardware; much of a PE's hardware is shared among its functions.) Given a specific type of branch predictor, and constrained execution resources, DEE is optimal, giving the best performance of all speculative execution techniques. For an example, see Figure 1. In the SP and DEE parts of the figure, the left pointing paths correspond to paths predicted to be followed, while the right pointing paths correspond to the paths predicted not to be followed. The circled numbers identify the order of processing element assignment to paths. With DEE, after path 3 is assigned processing elements, the next most likely to be needed path is path 4, since its likelihood of execution or *cumulative probability* (cp) (.3) is greater than that of the alternative, path 5 (.24). It is demonstrated in Section 5 that DEE's selective path-taking results in disjoint eager execution exhibiting better performance than that of both SP and eager execution with constrained resources, without eager execution's high cost.

Prior to defining DEE and proving it to be stochastically optimal, some general terms are defined:

- A *branch path* consists of the dynamic code between branches, including the exit branch.
- ILP can exist within a branch path. Thus, branch paths can in general use multiple resources, e.g., PE's. A branch path is said to be *saturated* if it can productively use no more execution resources. In other words, assigning a branch path one more PE would not change its execution time, since the resultant number of PE's would be greater than the maximum number of instructions semantically executable in parallel in a single cycle.
- The probability of a path's being taken with re-

spect to its predecessor path (next highest in the tree) is the path's *local probability*. For example, in Figure 1 the local probabilities of all of the left-pointing paths, p, are 0.7, while the local probabilities of all of the right-pointing paths are 1-0.7=0.3.

- The overall probability of each path being taken is its cumulative probability cp. cp is the product of the path's local probability and the local probabilities of all of its predecessor paths, higher in the tree, up to the root of the tree. However the cp's are determined, only the individual cp's are important.
- It is desired to maximize performance P_{tot}, over the execution of the program. Initially, for Theorem 1, we assume that paths do not saturate. Thus, the more processors that can be applied to a given path, the better its performance.
- The execution resources applied to a particular path i are denoted: e_i. The total resources available are: $E_{tot} = \sum_{i=1}^{n} e_i$.
- The overall expected performance over all of the paths is: $P_{tot} = \sum_{i=1}^{n} cp_i e_i$. This is reasonable, since the quantity of busy resources used in a branch path, e.g., PE's, is an indicator of the speed of execution of that path. Also, weighting this indicator of performance by the corresponding path's cumulative probability indicates the overall likely benefit of assigning resources to that path.

Therefore, the problem is: what should the values of the e_i be so as to maximize P_{tot}, for given cp_i?

Theorem 1 *Given $cp_j = \max\{cp_i\}$, P_{tot} is maximized by placing all E_{tot} resources on branch path j.*

Proof Assume all resources are initially placed on path j; then $P_{tot} = cp_j E_{tot} = P_{tot_j}$. Can any other assignment do better?

Sort the cp_i according to their values; then cp_k is second largest, while cp_j is the largest. Take an increment $\triangle e_j = \triangle e$ from path j and put it on path k. Then compare the resulting performance increments. Since: $cp_k < cp_j$, then: $\triangle e \cdot cp_k < \triangle e \cdot cp_j$, and: $\triangle P_{tot_k} < \triangle P_{tot_j}$. Since $cp_i < cp_k$ for all i, not including j or k, then: $\triangle P_{tot_i} < \triangle P_{tot_k} < \triangle P_{tot_j}$, in other words, P_{tot} has decreased.

Therefore, any other assignment of $\triangle e$ will lower P_{tot}, hence the best assignment is to place all of E_{tot} on the path with the largest cp_i, namely cp_j. □

But the situation is more involved than this; path saturation must be taken into account. What if, as Theorem 1 tells us, all resources are assigned to path 1, in Figure 1, but path 1 saturates?

Corollary 1 *If path j saturates, i.e., no more resources can be used, then the remaining resources are $E'_{tot} = E_{tot} - e'_j$, where e'_j are the number of busy resources on path j. E'_{tot} resources are assigned to the remaining paths, i, $i \neq j$, as in Theorem 1. This maximizes performance.*

Proof Path j saturating means that effectively $cp_j \rightarrow 0$, for E'_{tot} resources placed on it beyond saturation. Therefore, the incremental benefit of path j goes to 0.

If, referring to the Proof of Theorem 1, path j is now placed in the sorted list, it is at the bottom of the list (0), and will get no additional resources. The new assignment of resources is to a new maximum cp, cp_k, which maximizes performance over all cp_i. Therefore, assigning all remaining resources to path k is optimal, given path j has saturated. □

From Theorem 1 and Corollary 1 comes the following "rule" of Greatest Marginal Benefit for constructing DEE:

> Assign all remaining resources to the most likely idle path, overall, until the path saturates; repeat. This is Disjoint Eager Execution (DEE).

For example, resources are assigned to paths 1, 2, 3 and then, out of order, path 4, in Figure 1. Thus, DEE is optimal by construction, and subsumes both SP and eager execution. DEE becomes the same as SP as the branch prediction accuracy p approaches 1, and DEE becomes the same as eager execution as p approaches 0.5, for finite resources.

3 DEE in practice

In the DEE theory, it is assumed that the cp's are always known precisely. In reality, it is impossible to completely specify the cp's. If an estimate of cp is maintained, over which instances of the branch and its predictions should the estimate be made? An uncertainty principle is apparent: the smaller the window of an estimate, the less accurate the estimate. In the limit, if just the immediately prior execution of the branch and its prediction are examined, a branch prediction accuracy estimate of 0 or 1 is obtained, not very useful.

Assuming we could come up with a "good" set of past instances of branches to base branch prediction accuracy (p) estimates on, it is still difficult to compute cp's dynamically. This is because: 30-100 cp's must be maintained for a typical DEE tree; each cp is the product of many (possibly 10's) of potentially different local probabilities; the local probabilities are based on the branch prediction accuracy estimates and change dynamically; and branches higher in the DEE tree resolve as the program executes, thus changing the domain of the cp calculations. Therefore all of the cp's must be re-computed every cycle. Thus, hundreds or thousands of low-precision multiplications would have to be performed every cycle. Add to that the necessity of determining the largest cp's every cycle (sorting), and such an approach seems completely impractical. Even if we could do all this, the marginal performance gain over the following heuristic is not likely to be great; the heuristic already achieves about 59% of oracle performance (see Section 5).

3.1 The *static tree* heuristic

A method is now proposed that completely eliminates the necessity for dynamic cp computations, yet results in much ILP. In this *static tree* heuristic, the shape of the DEE tree is determined and fixed during the design of the CPU. (For a software-based ILP machine, or a multiprocessor machine, the tree could change from program execution to program execution.) The shape is determined with the following steps:

1. Measure the average or characteristic branch prediction accuracy, p, of the branch predictor to be employed by the machine by simulating the predictor on a representative group of benchmarks.

2. Assume that all branches to be executed in the target machine will be predicted with accuracy p.

3. Given the execution resources of the CPU E_T, and p, calculate the static DEE tree dimensions using the formulae presented later. The shape of the static tree is now fixed; for hardware-based ILP machines, it is never changed, and thus is constant during code execution.

4. The static tree defines the CPU's execution window. Execution resources, e.g., PE's, are made available for the execution of the dynamic code present in each branch path of the tree during program execution.

In the DEE part of Figure 1, the static tree includes the bold paths for a machine with 6 branch path resources. The tree for a more typical case is shown in Figure 2. Only dynamic instructions covered, or contained, by the tree are in the CPU's instruction window and are allowed to execute concurrently.

Effectively, in use, the tree "moves" down the dynamic code execution path. As branches resolve at the top of the tree, the tree moves down, freeing up resources from resolved-as not followed paths, and reassigning them to the leaves of the tree. Thus, new (dynamically later) code moves into the window, and is executed. Standard branch prediction methods are used for all branches in the tree, potentially all at once. Note that all of the branches in the tree may be unresolved at any given time.

The static tree characteristics are now briefly described and discussed. As is seen in Figure 2, for a typical branch prediction accuracy, p, the tree consists of two *regions*, the *Main-Line region* or path (ML) consisting of the l branch paths, and the *DEE region*, consisting of the remainder of the tree. Put another way, B1 through B4 are to be Disjoint Eagerly Executed, or *DEE'd*; they are called the *DEE branches*. A DEE branch's not predicted path, with its subsequent predicted paths, form a composite *DEE path*. All of the DEE paths taken together comprise the DEE region. The ML path is relatively long (e.g., 24 paths), compared to the DEE dimensions h_{DEE} and w_{DEE} (e.g., 4 paths) ($h_{DEE} = w_{DEE}$). The mathematical relationships amongst p, l, h_{DEE}, and E_T (the total number of branch paths in the tree), are seen to be:

$$
\begin{aligned}
E_T &= \log_p(1-p) + \frac{1}{2}h_{DEE}^2 + \frac{3}{2}h_{DEE} - 1 \\
h_{DEE} &= -\frac{3}{2} + \frac{1}{2}\sqrt{8E_T - \frac{8\log(1-p)}{\log p} + 17} \\
l &= h_{DEE} + \log_p(1-p) - 1
\end{aligned}
$$

These relations hold while $p^l > (1-p)^2$; for example, in Figure 2 this is until by DEE theory the path with $cp = 0.01$ is to be used in the tree. Also, $(1-p) > p^l$ must hold, i.e., there must be a non-empty DEE region.

The tree's structure is fortunate for implementation purposes. It suggests that a processor could execute the ML part using a standard branch prediction mechanism, and branch off DEE paths of execution, each of which also uses standard branch prediction. This is done in the Levo microarchitecture; it is discussed further in Section 4.3. Other considerations aside, traditional superscalar machines could be modified to do something similar.

4 Levo microarchitecture

Levo is a prototype computer embodying DEE being developed and designed at the University of Rhode Island. It is an extension of the CONDEL-2[9] machine model. In this section, a description of and rationale for using a static instruction window processor

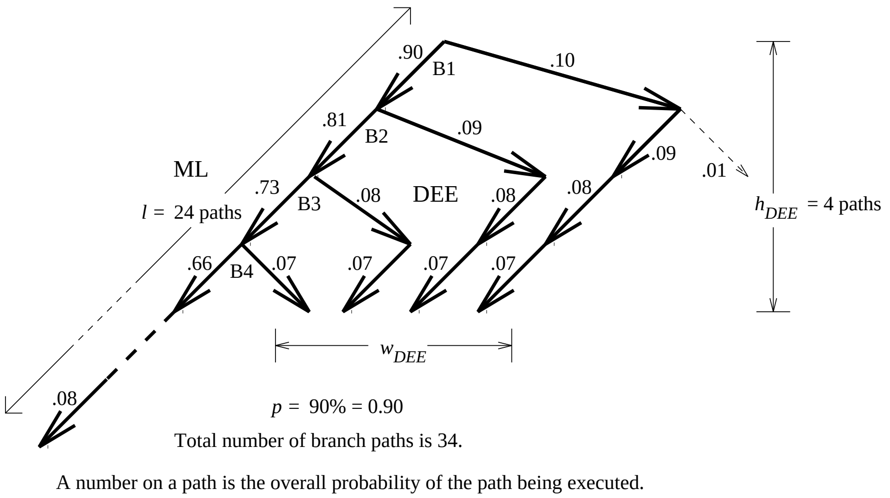

Figure 2: Static DEE assignment tree for $p = 0.90$, $E_T = 34$ branch paths.

are given. Next, the relevant elements and operation of CONDEL-2 are described, and then the Levo enhancements to the CONDEL-2 model are given.

4.1 Static instruction window benefits

CONDEL-2 and Levo have a fundamentally different microarchitecture than traditional RISC, CISC and superscalar machines. In traditional processors, the instruction window holds dynamic instructions. Mispredicted branches commonly cause the window to be flushed.

CONDEL-2 and Levo use a window called the *Instruction Queue* (IQ) [13] to contain the static instructions of the program being executed. Instructions are usually fetched into the IQ in the static program order. However, just as in dynamic window machines, the effective order of the actual execution of instructions is, and must be, the dynamic order.

Why bother with a static window? First, usually the contents of the IQ do not change upon a misprediction; many instructions are not squashed and may continue to execute, improving performance. Second, the IQ contents are usually invariant with respect to the direction of a branch's execution. This allows the realization of minimal control dependencies. As with mispredictions, other instructions not dependent on the branch, including other branches, continue to execute, improving performance.

The last major reason for using a static window is again concerned with speculative execution. With large ILP's, many, say 5-20 branches must be predicted, resolved and executed per cycle. But in a dynamic window processor, how can the 20th branch deep be predicted without knowing the predictions of the dynamically previous 19 branches? These 19 are <u>also</u> being predicted in the same cycle as the 20th. Consider: this problem is equivalent to the classic pointer-chasing problem that occurs with multiple levels of indirection, in this case with at least 5-20 levels of indirection through the Program Counter necessary, per cycle, with some of the indirections undefined. Altenatively, if branches are to be eagerly predicted, then there are approximately 2^{20} possible dynamic versions of the 20th branch necessary to be predicted. Should they all be predicted in the same cycle? Not likely.

Therefore a static instruction window is desirable.

4.2 The existing structure: CONDEL-2

CONDEL-2 is a *static instruction window* machine with an IQ holding n static instructions, conceptually arranged in a single column; see Figure 3. A mechanism is needed to translate the static order into a dynamic stream of instructions. This is done by keeping track of which instances of a static instruction have been executed using special bookkeeping hardware with special scheduling logic.

The major logical elements used in CONDEL-2, and hence in Levo, are shown in Figure 3. For Levo, the matrix dimensions $n \times m$ are targetted to be 32×8. For illustrative purposes in the following discussion, the dimensions used are: 8×4. The nominal dynamic order of the instances active in the CPU are shown by the folded-arrowed-line on the right-hand side of the figure, in the "dynamic instance order" box.

The major components of the bookkeeping hardware are the *Really Executed* (RE) and *Virtually Executed* (VE) $n \times m$ bit-matrices. Each ith row of a matrix corresponds to the ith static instruction in the IQ. The jth column entry of a matrix row corresponds to the jth instance of the row's static instruction. Altogether, the jth column of a matrix corresponds to the jth active iteration of a loop completely contained or *captured* in the IQ. Up to m instances of a static instruction may be in process at any given time. There is one entry in each matrix for each active instruction instance. There is also hardware employing $\bigcirc(n)$ comparators to determine and store the data and control dependencies among the instructions in the IQ.

In operation, instances execute as soon as their data and control dependencies have resolved, i.e., as soon as their depending instances have executed. Patented high-speed logic combines the RE, VE and dependency information every cycle to both determine if an instance is to execute, and to gate the instance's data sources to the instance's PE for instruction execution. Once an instruction instance executes, its RE bit is set.

Branches are executed as follows. A branch's RE bit is always set when it is executed. If a branch is not taken, nothing else is done. If a branch is taken, all of the instruction instances between the branch and its target are Virtually Executed by setting their VE bits.

Instances whose VE bits are set are ignored and not executed. Thus, VE bits perform the function of predicates or guards, but in a much more general way; also, the instruction set is not modified.

The results or *sinks* of instances are held in an $n \times m$ *Shadow SInk* (SSI) [13] matrix of word-length registers, with a corresponding $n \times m$ *Instruction Sink Address* (ISA) matrix of registers holding the instruction-set-architectural register or memory addresses of the sinks. Every (i, j)th sink and address corresponds to the (i, j)th RE and VE bits. The Shadow Sink registers are renaming registers, for both memory and register sinks. Minimal data dependencies (only flow dependencies) are thus realized for architectural registers, with somewhat more restrictive data dependencies for memory accesses. The static instruction window model realizes minimal control dependencies. Together, close to minimal semantic dependencies are obtained.

For each IQ element (each static instruction), there is a PE. Both the SSI and ISA registers are accessed in parallel by the PE's.

In order to make the CONDEL-2 microarchitecture physically realizable, and to allow for the requisite high SSI bandwidth required, all of the matrices and the IQ are replicated[11], once per PE, giving the structure shown in Figure 4-a). To ensure the coherency of the copies of the SSI and ISA matrices, each PE writes the same (i, j)th elements of these matrices in every copy of the matrices. By design each PE writes to a different row than the other PE's, and thus the writes are disjoint. Thus, all 32 PE's may write to the SSI copies at the same time, without fear of collision, and with a resultant high bandwidth. For the SSI copies to support this requires that they not be constructed as regular register files, but as assemblages of individual registers and busses. The accessing of each copy is controlled by the special gating logic mentioned before.

As part of this structure, the architectural registers are not physically realized in the classic register-file, but rather in higher-capacity, lower-speed memory, roughly equivalent to a cache (not shown). The performance impact of doing this has been demonstrated to be very low[7, 11]. There is one copy of each architectural register.

The execution of loops with lengths less than that of the Instruction Queue can be enhanced by a machine-code to machine-code loop unrolling *filter* program, to achieve average loop sizes of about 3/4 the length of the Queue. Loops not captured by the IQ are helped in their execution, in Levo, via the operation of the machine in a special linear-code execution mode. A scan of several of the SPECint92 benchmarks indicates that more than 70% of the conditional-backwards-branch-formed dynamic loops' executions fit in an IQ of length 32. As hardware densities increase, allowing the IQ length to increase to, say, 64, almost all of these dynamic instances of the loops will fit in the Queue. Branches may execute concurrently. See [9, 11] for more detailed descriptions of the machine, and other code execution examples.

4.3 Levo, the prototype DEE machine

The major novel contributions of Levo are the incorporation of general branch prediction with minimal control dependencies, and the realization of DEE, and hence, DEE-CD-MF. The architecture and operation of these changes are described in this section. Also note that Levo has a data-flow like operation, but it uses standard instruction sets, and has no special array constructs as in typical data-flow machines (I-structures, etc.). Minimal data dependencies are realized for memory accesses, resulting in minimal semantic dependencies being realized.

General branch prediction: Branch prediction is realized in Levo by having one branch predictor of arbitrary type associated with each IQ row (with each static instruction). Of course, only the instructions that are branches use the predictors. It may be possible to reduce the number of predictors in the future, but since reducing them would complicate the design, and might slow down the machine, we assume one predictor per IQ instruction herein.

Total control dependencies are also maintained; they include both regular direct, and *indirect* or transitive control dependencies; e.g., given instruction I_3 is data

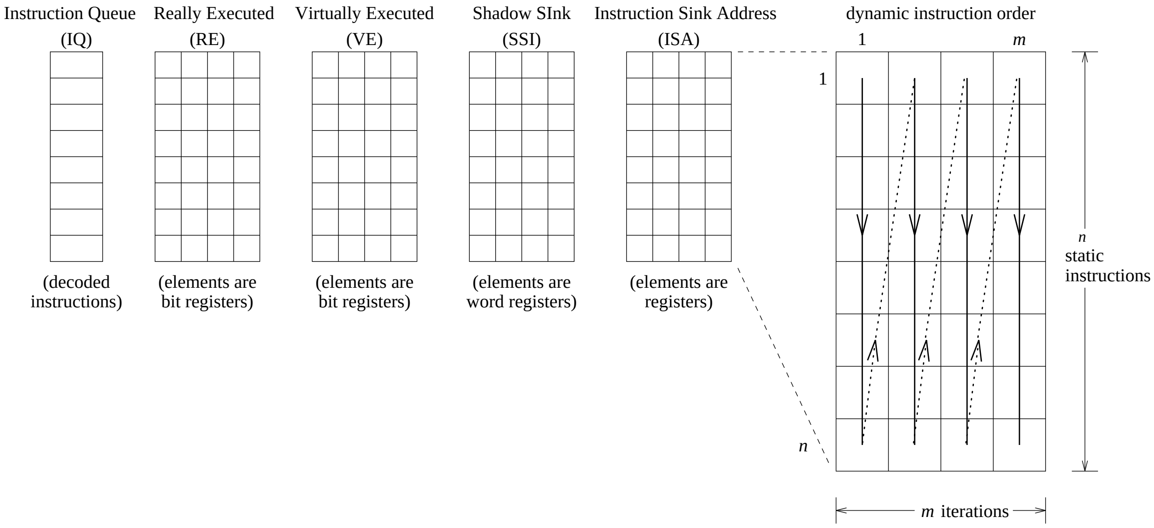

Figure 3: Major logical elements of CONDEL-2, also used in Levo.

or control dependent on I_2, and I_2 is control dependent on I_1, then I_3 is indirect and total control dependent on I_1. Total control dependencies are computed similarly to the other dependencies.

Instances now execute as soon as their operands are available, and independently of the state of execution of any branch. If a branch mispredicts, its total control dependent instruction instances are squashed, their sinks are not written to memory, and the instances re-execute, implicitly with new operands. Sinks are written to memory only after all of an instance's total control depending branches have been resolved.

In Levo many branches may be predicted (up to 32), resolved (up to 256) or executed (up to 256) per cycle. This is unrealizable in a dynamic instruction window machine using the classic Program Counter.

Although the standard 2-bit counter prediction method is desirable to be used in Levo, as it is in the DEE simulations of Section 5, it may not be possible. This is due to the high number of unresolved branches likely to exist, per static branch, at any given time. The counter method requires being updated with the actual direction taken of a branch before its next branch instance is predicted; thus a 90% prediction accuracy may not be realizable with the counter method.

However, if *PAp adaptive prediction* [15] is used, with history register lengths of 2 bits, and one pattern history table per row, the 90% prediction accuracy should be realizable. This is due to the speculative update of the predictor with the predicted directions of unresolved branches, allowing speculative predictions.

The design penalty for a misprediction of a branch is currently one cycle, but only for instances total control dependent on the branch. This may be reducible to 0 cycles. The penalty for an instance is only one cycle, total, for any number its total control depending branches resolved as mispredicted in the same cycle.

The realization of DEE (see Figure 4-b)): The structures described for CONDEL-2 constitute the MainLine (ML) section of Levo. This ML section is the implementation of the ML path or region of the static DEE tree (see Figure 2).

The only hardware essential to add to Levo to implement DEE is state (RE, VE), data (SSI) and address (ISA) hardware for each DEE path. In Figure 4-b) each SSI^{1-4} column implements a DEE path of the static DEE tree (RE, VE and ISA are not shown). Since theoretically DEE paths can be much shorter than the ML path, in Levo DEE paths need only have one or two iteration columns. In the example, there are 4 DEE paths, each of 1 column, and thus are 8 instructions long. Since a branch path contains about 5 instructions, an actual DEE path containing one column 32 instructions tall contains about 6 branch paths.

The DEE paths in Levo use the same PE, IQ and dependency hardware as that used for the ML path (the old CONDEL-2 hardware). Therefore, the extra hardware necessary to implement DEE is not necessarily that much. At this stage of Levo's development it is not clear whether or not one PE will suffice for all ML and DEE instances associated with the same static instruction (on the same row). This PE sharing may not inhibit performance since DEE branch paths do not typically execute in saturation, and many DEE and ML instances are "branched around" (virtually executed or disabled), thereby requiring fewer PE's. For example,

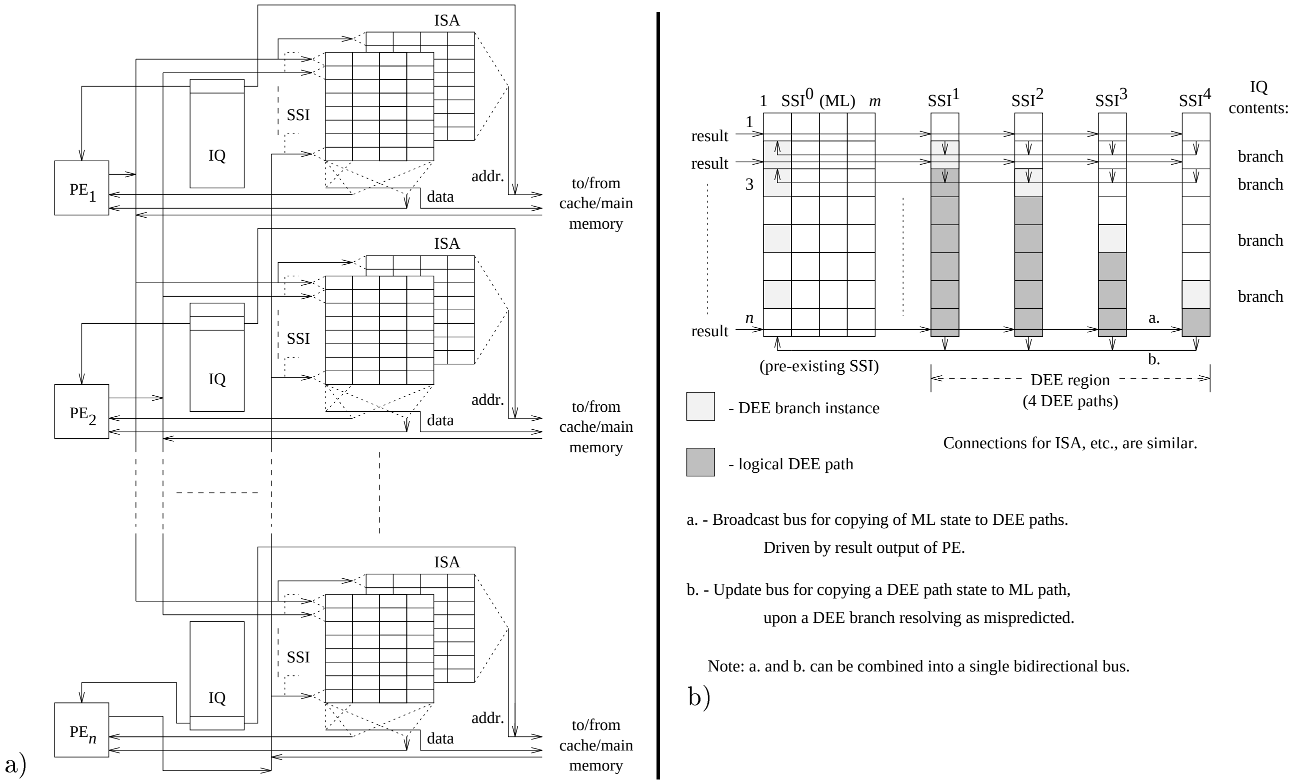

Figure 4: a) CONDEL-2 central CPU data path. b) Levo: DEE additions and modifications to CONDEL-2.

if a forward branch in the ML path is predicted taken, then the instances that are virtually executed do not require PE's. The DEE path corresponding to this branch can then use the PE's thus freed for the instances enabled for execution (by the forward branch being assumed not taken in the DEE path), without inhibiting performance. Also, dynamically later DEE instances are less likely to be ready for execution (data dependencies resolved), also reducing PE demands.

The assignment of a PE to an instance on its row is as follows. Typically, the ML path has top priority. The priority amongst the DEE paths is arbitrary, as same-level (same-row) instances are equally desirable to be executed (from DEE theory; see Figure 2).

The basic architecture and operation of DEE in Levo are now described; see Figure 4-b). A single DEE path is considered first. Referring to the DEE static tree model (Figure 2), the DEE and ML paths have the same state at the corresponding DEE branch, where the DEE path splits from the ML path. Therefore, in the actual hardware, the ML state is copied into the DEE state, for all instances dynamically before (above) the DEE branch, as the state is generated, by having the DEE path pick the SSI data off of the corresponding PE result bus (a.), at the same time that the data are being sent to the ML path. The DEE branch instance in the DEE path is executed oppositely to the way it was predicted in the ML path. SP-CD-MF execution down the DEE path, as well as the ML path, ensues normally. This is how a DEE branch is actually DEE'd.

Once a DEE branch resolves one of two actions is taken. If the branch was correctly predicted, the ML path stays as is, and the corresponding DEE path is ignored, and given no priority for execution resources (PE's), from then on. The time penalty here is 0 cycles. If the DEE branch was mispredicted, then the corresponding DEE path state must be copied into the ML path, and the ML state dynamically later than the end of the DEE path must be cleared. The state copying is accomplished by placing the specific DEE state on return busses (b.) and reading them into the ML path state hardware in parallel. For example, in Figure 4-b) if the DEE branch corresponding to SSI(3,1) is mispredicted, then the contents of SSI^0 (the ML path) columns 2-4 are flushed (actually, just the corresponding RE and VE bits are cleared, implicitly forcing these instances to re-execute); at the same time, DEE path 2 (RE^2, VE^2, SSI^2 and ISA^2) is copied into the first column of the ML path [$SSI^0(*,1)$]; execution then resumes. The time penalty for one or more DEE branch mispredictions, determined in the same cycle, is one cycle. It is conceivable that this penalty may be reducible to 0 cycles. Single column DEE paths' branches can use the same predictions as their corresponding ML branches.

Preliminary hardware cost estimates indicate that a Levo multi-chip prototype could be built with current CMOS technology. A single-chip version with 11 2-column DEE paths (E_T = 100 branch paths) could be built in about 5 years, assuming current CMOS transistor density increase trends continue. (It is expected that 50-100 million-transistor processor chips will be available in the year 2000.) About 40% of the CPU and on-chip cache hardware is concurrency-detection/scheduling-hardware and multiple-state-copies overhead. About 18% (resp. 3%) of the Levo hardware is used to realize DEE, assuming 11 2-column-wide DEE paths (resp. 3 1-column DEE paths [E_T = 32]). Therefore, the marginal cost of DEE is low. Each additional 1-column DEE path uses about 1 million transistors. Preliminary versions of all of the new logic needed for Levo and Levo's operation have been designed. The logics operate in parallel, are regular, straightforward, have relatively small delays (not overly impacting the cycle time), and are moderate in size or cost.

We believe that the proposed Levo implementation of DEE is superior to software-based approaches, as the dynamics of code execution can best be handled at run time. Also, unlike most software-based approaches, Levo works well on all captured loop structures, and tolerably well on other code. Its concurrency mechanisms are transparent to the machine code user.

5 Experimental results

5.1 Methodology

DEE was simulated on five of the six (integer) programs of the SPECint92 benchmark suite. The `sc` benchmark was not included as it was significantly more predictable than the others. The inputs used to the benchmarks were: `1explow.i` for `cc1`; `in` for `compress`; `int_pri_3.eqn` for `eqntott`; `bca`, `cps`, `ti`, & `tial` for `espresso` (their harmonic mean was used for each overall `espresso` datum); and `li-input.lsp` (9 queens) for `xlisp`. The MIPS R3000 instruction set was assumed, but with single cycle (unit latency) instruction execution. The heuristic static tree pattern method of Section 3 was used to give a better idea of likely performance gains to be experienced by a real machine, rather than by using a purely theoretical DEE model. Except for the Oracle simulations, the number of branch path resources was constrained. This implicitly limited the number of PE's, but not explicitly. The maximum number of PE's used at any time during the simulations is likely to be less than 200 (for 100 branch paths), with the average being much lower, due to the high density of branches and hence small size of the average branch path.

The Lam and Wilson simulator[3] was modified to give the *DEE simulator*, which was used to obtain the results. For all three speculative execution models, SP (Single Path), EE (Eager Execution), and DEE (Disjoint Eager Execution), an appropriately shaped static tree pattern (see Figure 1) was superimposed on the dynamic execution trace of a benchmark. Code execution was only allowed where the tree was, in other words, a limited code execution window was explicitly assumed. The tree could only move farther along in the dynamic trace when its earliest branch resolved, and the instructions along its branch path had fully executed. At such time the tree was moved down one or more branch paths, allowing new code at the tree's leaves to execute concurrently. A branch resolving in the remainder of the tree, i.e., not the first branch, has no effect on tree movement, until the branches dynamically prior to this later branch are resolved. Thus, a branch resolving in the later or lower part of the tree does not release resources to be reallocated, until everything above it has fully executed. This is very similar to the Levo machine model.

The branch prediction method used was the classic 2-bit saturating up/down counter method[6]. All of the counters were initialized to the non-saturated taken state. Because of time constraints, the branch prediction method was not upgraded to a two-level adaptive predictor. For reasons stated earlier in the paper, we believe that the accuracy of the counter method without speculative prediction is likely to be about the same as an adaptive method with both speculative prediction and constrained hardware. There is a tradeoff between predictor accuracy and its cost versus degree of DEE realization and its cost, for the same performance. The data suggest that some use of DEE is likely to be beneficial, regardless of the predictor accuracy.

Each benchmark was simulated for up to 100 million instructions, or the end of the benchmark, whichever came first. Most of the benchmarks executed to or near this limit. Due to a problem with the simulator, the `cc1` benchmark was simulated for only a total of 3.5 million instructions for each of the ILP models. Having examined `cc1` and its input, we conclude that these instructions are representative of its entire execution. The `cc1` data is retained in the summary data since it was the worst performing of the benchmarks, keeping our results conservative.

Although the simulator does not actually execute a benchmark in parallel, we attempted to ensure the veracity of the results by exhaustive testing, including: in-depth examinations of the static tree dynamics during simulations, at many points in many simulations, for all of the ILP models; exercising of the simulator with synthetic test cases; and gathering other supporting data, some of which will be described later.

5.2 The ILP models considered

The following ILP models were simulated. Except for the Oracle simulation, all models had constrained branch path resources. Minimal data dependencies were assumed.

- EE: Eager Execution. - for comparison.
- SP: Single Path execution. - for comparison.
- DEE: Disjoint Eager Execution alone, with restrictive control dependencies.
- SP-CD: SP with reduced control dependencies; branches serialized. - for comparison.
- DEE-CD: DEE with reduced control dependencies; branches serialized.
- SP-CD-MF: SP with minimal control dependencies; branches execute in parallel. - for comparison.
- DEE-CD-MF: DEE with minimal control dependencies; branches execute in parallel.
- Oracle: EE with unlimited resources; branches do not constrain the parallelism in any way. Not realizable.

5.3 Discussion of the data

The results are shown in Figure 5. The "Harmonic Mean" data points are the harmonic means of the corresponding five benchmarks' data points. On all of the graphs, the vertical axis is the speedup factor of a concurrent model over a sequentially executing model. The horizontal axis is the total number of branch paths, or resources, allowed to be active or used at a time (E_T) in a simulation.

Referring to the summary Harmonic Mean graph, SP's performance effectively stops improving at resources of 16 paths; above this point, adding incremental resources to the simple branch prediction model results in little or no incremental performance gain, verifying our analysis. Also above this point, The DEE-CD-MF model's incremental performance gain rises much faster than that of EE.

The most striking thing about the results are the high speedups of DEE-CD-MF above 16 path resources. Although this possibility is hinted at in the original Lam and Wilson simulations, it was still surprising. Minimal control dependencies are necessary so that branches may execute in parallel. SP-CD-MF does not show such great gains, since it is still constrained by the diminishing returns nature of single path execution. The other key to DEE-CD-MF's large gains are the extra side (DEE) paths in the static tree. To examine the dynamics further, statistics were gathered of the locations in the DEE static tree where mispredicted branches resolve. As it turns out, most of the resolving is done at the root of the tree, accounting for around 70-80% of the resolved mispredictions. Thus, the longest DEE path is often taken upon a misprediction, dramatically reducing the penalty of most mispredictions. The root location of the resolvings is reasonable, upon reflection, in that the higher a branch is in the tree, the more likely its dependencies are resolved.

DEE-CD and DEE-CD-MF are seen to be uniformly better than both SP and EE above 16 branch path resources. At and below this point, the DEE tree is the same as that of SP, since there all potential DEE cumulative probabilities are less than the last ML region's path's cumulative probability, so there DEE and SP have the same performance. There is an inflection point in DEE's performance at 16 resources most likely due to the fact that a DEE *heuristic* is being used, not the theoretically perfect form of DEE. By definition about half of the branches have prediction accuracies less than the average accuracy of 90.53%; performance would be improved if these branches were DEE'd earlier, at lower levels of E_T branch path resources. This implies that DEE paths could be usefully employed with many fewer than 32 branch path resources.

The performance of the DEE models is quite good. The number of branch path resources in Levo is targeted to be the equivalent of $E_T = 100$. From the graph, at this level of resources DEE-CD-MF is better than SP (plain branch prediction only) by a factor of 5.8, and better than EE (Eager Execution) by a factor of 4.0. At this point, DEE-CD-MF exhibits a speedup over the sequential execution of code of a factor of 31.9, or 3,190%. It is also seen that DEE-CD-MF with 8 branch path resources has the same performance as EE with 256 branch path resources. We also note that overall, DEE-CD-MF achieves about 59% of oracle performance, the theoretical maximum; this has not been done before.

The major limiting factor of the machine, the limited-size window on the code, is included in the DEE

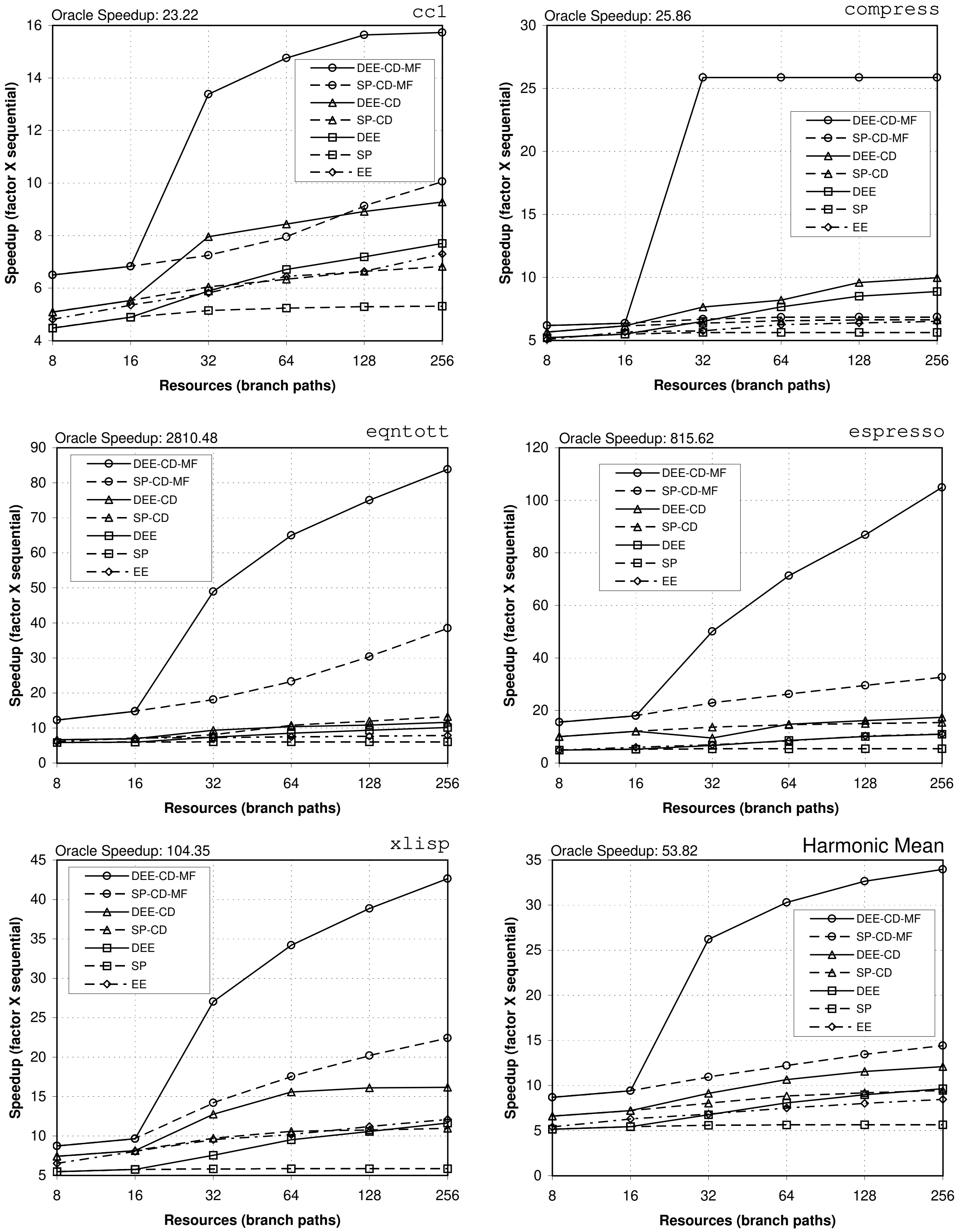

Figure 5: DEE simulation results on five of the six SPECint92 benchmarks.

simulator, and in all of the simulations made; therefore, performance reductions due to hardware constraints have largely been taken into account. It is not yet clear what the net effect of assuming non-unit latencies on the DEE-CD-MF model will be. On one hand, in other studies of other ILP models, the performance of the models decreased significantly. On the other hand, concurrent instructions in the DEE-CD-MF model may exhibit much more overlap than in these other cases. It may also be the case that Levo could be built with only 32 branch path resources (3 DEE paths), since the data shows that the speedup at that level of DEE-CD-MF is still quite high (a factor of 26).

DEE performs very well in both relative and absolute senses, greatly improving the performance of typical general purpose unpredictable-branch-intensive codes. DEE-CD-MF's speedup of such code by a factor of 31.9 is the best of any realistic ILP model considered anywhere to date.

6 Summary

Disjoint Eager Execution was shown to be an optimal form of speculative execution, performing better than both SP or eager execution, for the same resources. A DEE heuristic was described, having a structure lending itself to ready implementation. A DEE implementation, Levo, was presented, using the heuristic, and including minimal control dependencies. The basic worth of the DEE models was verified by the experimental results. Even allowing for a performance reduction of Levo arising from other realization constraints, order of magnitude ILP speedups are likely to be achieved by Levo.

Acknowledgements

We are indebted to Mike Saks for his help with the math background for the theorem. Many thanks to Monica Lam and Robert Wilson of Stanford University for making their simulator available to us, and helping us get up to speed. Sridhar Mahankali helped with his work on early versions of the simulator. We are also grateful to the anonymous referees whose keen comments helped (hopefully) to improve this paper. Lastly, but not in the least, we thank Laurette Bradley for her comments on a later draft of the paper, as well as much insight into the workings of compilers, and most importantly, her constant support to the first author.

References

[1] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. A VLIW Architecture for a Trace Scheduling Compiler. *IEEE Transactions on Computers*, C-37(8):967–979, August 1988.

[2] J. Ferrante, K. Ottenstein, and J. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

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

[4] B. R. Rau. Dynamically Scheduled VLIW Processors. In *Proceedings of the Twenty Sixth Annual International Symposium on Microarchitecture (MICRO-26)*, pages 80–92. IEEE and ACM, December 1993.

[5] E. M. Riseman and C. C. Foster. The Inhibition of Potential Parallelism by Conditional Jumps. *IEEE Transactions on Computers*, C-21(12):1405–1411, December 1972.

[6] 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.

[7] S. Somanathan. Performance Enhancement of a Concurrent Processor. Master's thesis, University of Rhode Island, May 1995.

[8] A. K. Uht. A Theory of Reduced and Minimal Procedural Dependencies. *IEEE Transactions on Computers*, 40(6):681–692, June 1991.

[9] A. K. Uht. Concurrency Extraction via Hardware Methods Executing the Static Instruction Stream. *IEEE Transactions on Computers*, 41(7):826–841, July 1992.

[10] A. K. Uht. Extraction of Massive Instruction Level Parallelism. *ACM SIGARCH Computer Architecture News*, 21(2 and 3), March and June 1993.

[11] A. K. Uht and D. B. Johnson. Data Path Issues in a Highly Concurrent Machine. In *Proceedings of the Twenty-Fifth International Symposium on Microarchitecture (MICRO-25)*, pages 115–118. ACM-IEEE, December 1992.

[12] S. S. H. Wang and A. K. Uht. Ideograph/Ideogram: Framework/Architecture for Eager Evaluation. In *Proceedings of the 23rd Annual Symposium and Workshop on Microprogramming and Microarchitecture (MICRO-23)*, pages 125–134. ACM and IEEE Computer Society, November 27-29 1990.

[13] R. G. Wedig. *Detection of Concurrency in Directly Executed Language Instruction Streams.* PhD thesis, Stanford University, June 1982.

[14] S. Weiss and J. E. Smith. *POWER and PowerPC.* Morgan Kaufman Publishers, Inc., San Francisco, California, 1994.

[15] T.-Y. Yeh and Y. N. Patt. A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 257–266. IEEE and ACM, May 1993.