University of Rhode Island
Dept. of Electrical and Computer Engineering
Kelley Hall
4 East Alumni Ave.
Kingston, RI 02881-0805, USA

Technical Report No. 0697-0001

Verification of ILP Speedups in the 10's for Disjoint Eager Execution

Augustus K. Uht
Department of Electrical and Computer Engineering
University of Rhode Island

Email: uht@ele.uri.edu
WWW: http://www.ele.uri.edu/~uht

June 24, 1997

Abstract

Disjoint Eager Execution (DEE) has demonstrated Instruction Level Parallelism (ILP) speedups of a factor of 30 in simulations[15]. Theoretical and practical arguments verifying these gains are presented. This includes the use of Amdahl's Law, analysis of the partial trace of a detailed simulation of a SPECint92 benchmark, and analysis of the effect on cycle time of the scheduling hardware of the proposed Levo DEE-realization prototype. The static tree heuristic used to economically realize DEE in Levo is also examined. In particular, the shape of the tree is verified via formal proofs, and the variations in the dimensions of the tree with respect to available resources and branch predictor accuracy are modeled and studied.

This work was supported in part by the Intel Corporation.
This paper has been submitted for publication to the 30th International Symposium on Microarchitecture (Dec. 1997). Versions of Figures 1 and 2 have previously appeared in [15] and [16].

1 Introduction and Background

Many *Branch Effect Reduction Techniques*, or BERTs, have been proposed to increase the amount of *Instruction Level Parallelism* (ILP) exploitable by uniprocessors[16]. One promising BERT is *Disjoint Eager Execution* (DEE), proposed and studied in [15]. Although common BERTs realize ILPs of factors of about 2 to 3 (performance speedup over sequential code), one version of DEE (DEE-CD-MF: DEE combined with minimal control dependencies) is claimed to realize ILP speedups of factors in the 10's. However, the initial DEE work made certain simplifying assumptions for its simulations that must be examined further in order to verify the large gains. This is done in this paper.

1.1 Review of DEE and its Realization

As shown in Figure 1, there are three basic forms of speculative execution of typical code. Single Path (SP) is at one extreme, and is the most commonly used. When a branch is encountered, its *sign* (taken or not taken) is predicted, and execution proceeds only down the predicted path. Eager Execution (EE) is at the other extreme, and is not commonly used. With EE, execution proceeds down both paths of a branch when the branch is encountered (there is no prediction).

As the depth of speculation (l) increases, the hardware cost of an SP-based processor grows linearly in l, which is good. However, as l increases, the overall likelihood of executing the code at the bottom of the overall SP path or tree goes to 0. The likelihoods or *cumulative probabilities cp* of all of the paths in the figure are indicated next to the paths.

Considering eager execution, its cost is prohibitive, rising exponentially with l. It has been shown both that unconstrained (infinite l) eager execution gives the best performance (same as an oracle), and also that the performance improvement is not great for even large finite values of l [8, 15].

The third form of speculative execution is Disjoint Eager Execution (DEE). As will be

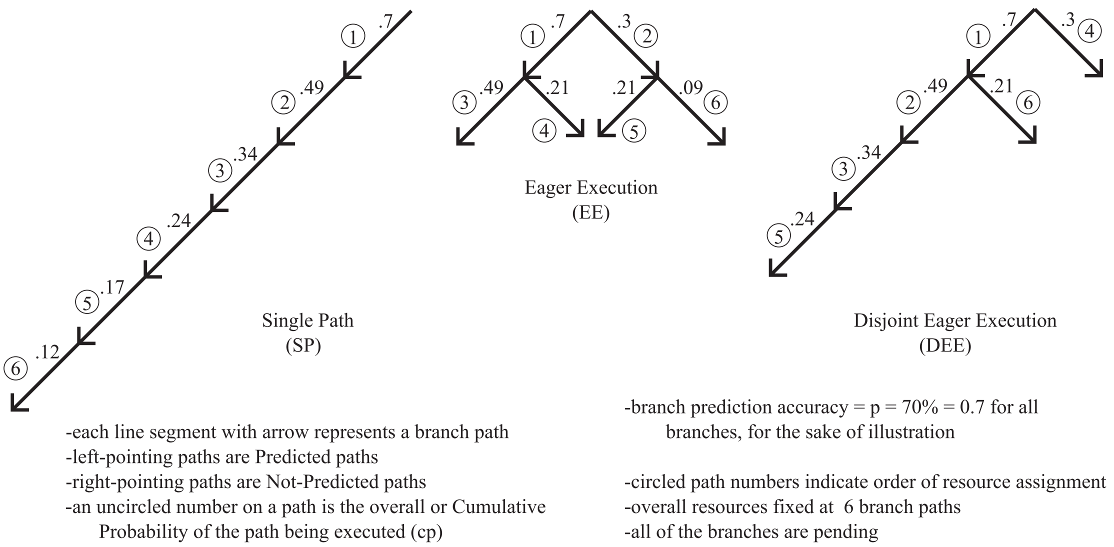

Figure 1: *Comparison of the three speculative execution strategies.* In each strategy, resources are assigned only to the paths shown. Note that the cumulative probabilities of the DEE paths are greater than the corresponding ones of both single path execution and eager execution: in DEE, the most likely code to be executed is given the resources.

shown later in this paper, it is actually a generalization of SP and EE, depending on its parameters. The key idea of DEE is to assign processing resources only to the most likely pending paths. This is shown in Figure 1; every DEE path is as or more likely to be executed than the corresponding paths in the SP and EE models. Thus, it should perform better than both SP and EE. Its cost is about the same as SP's.

Since it is both hard to dynamically compute cumulative probabilities, and to make completely general assignment of resources (such as Processing Elements), a *static tree* heuristic has been proposed[15] to implement DEE; see Figure 2. With the static tree model, resources are fixed to the pattern of paths that arises out of assuming that the branch predictor predicts every branch equally well. This is, of course, false, but still works well.

The combination of DEE with Minimal Control Dependencies (MCD)[12], called: DEE-CD-MF following Lam and Wilson's notation[5], is necessary for large performance gains. As an example of MCD, instructions after a forward branch's target are control independent of the branch; however, in the classical model, followed by virtually all commercial processors,

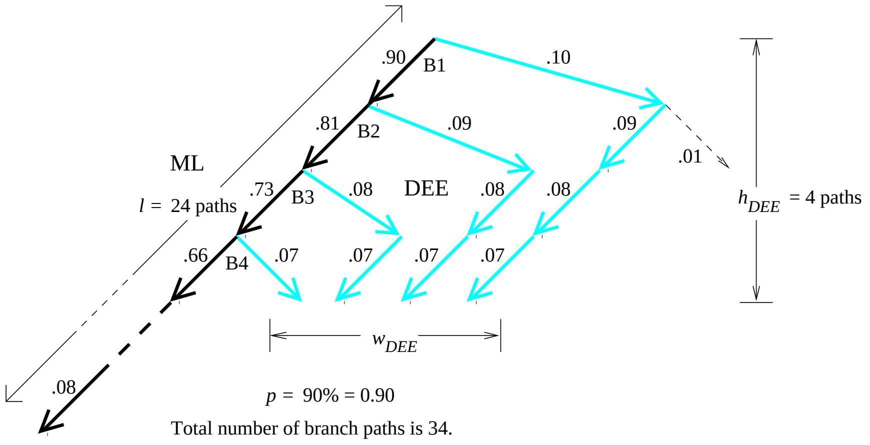

Figure 2: *Typical static DEE assignment tree.* ML is the *Main-Line path* or *region*. The execution of the ML path can be realized via SP execution. Each *composite DEE path* to the side (blue or shaded paths; each 1 to 4 branch paths long, in this example) can be treated with SP execution within itself. All of the composite DEE paths together form the *DEE region*.

all instructions after a branch are dependent on the branch.

For processors appearing at the turn of the millennium, having sufficiently large numbers of resources, the simulations of [15] demonstrated speedups of the DEE-CD-MF model of a factor of about 26 (2,500%) over a sequential machine.

1.2 Other ILP Work

The ILP literature is vast, we mention only a small part of it here due to space limitations.

There are a variety of approaches to ILP enhancement and exploitation. Hardware methods include the dynamic execution model employed in the Pentium Pro[7]. Software methods include various VLIW-type approaches[2, 4]. There is also the multiscalar approach[10]. Recently, *Data Effect Reduction Techniques* (DERTs), such as data speculation[6, 9], have begun to be investigated. DEE is orthogonal to all of these methods, and can be used with any of them. It can be realized in hardware or software, and both within and among uniprocessors.

1.3 Overview of the Paper

There are three thrusts of the paper. The first two are concerned with verifying the demonstrated high performance gains of the DEE-CD-MF model; Section 2 considers the performance at the machine instruction level, while Section 3 examines both the effect on cycle time of large scheduling logics, and the effect of speed-of-light limitations on a single-chip implementation. The former section shows that the demonstrated speedups of DEE are real and intuitive; even large load instruction latencies do not significantly reduce performance. The latter section shows that the cycle time may be increased, but probably not by much; further, cross-chip latencies can be avoided. Thus, the order-of-magnitude speedups initially projected should be achievable.

The last thrust of the paper is to study DEE further from a theoretical perspective. In particular, in Section 4 the static tree shape is derived, and the changes in the shape with respect to both different branch prediction accuracies and varying levels of resources are plotted and discussed. Section 5 summarizes our findings and ends the paper.

2 The Large Magnitude of the Speedups of DEE

Prior work has demonstrated performance improvements of an order-of-magnitude over the state-of-the-art with the use of DEE. In this section we provide both theory-based arguments supporting these results, and concrete evidence of the size of the speedups, by way of an example of code execution drawn from one of the SPEC benchmarks simulated.

2.1 Theoretical Basis for the Large Speedups

Our theoretical justification for the large size of the speedups is based on Amdahl's Law: $S_{actual} = \frac{S_P}{1+f(S_P-1)}$, where f is the fraction of serial code in the program, S_P is the speedup of the parallel part of the program, and S_{actual} is the overall speedup of the program. From the Law it is seen that only a small value of f can result in a large decrease in S_{actual}, i.e., just a little sequential code will ruin net performance. Now, the measured SP numbers[15] are in agreement with [17], and the oracle speedups of [15] are in agreement with other researchers'[1, 5, 8]. The question is about the DEE-CD-MF model's speedups. We will show in a code fragment, in the next section, that the DEE-CD-MF model can reduce f significantly, dramatically increasing S_{actual}. This is due to the model's reduction of branch effects; eliminating the negative effects of just one branch can have a large effect on performance, when most of the rest of the code has been parallelized. Intuitively, the constraining effects of mispredictions on SP is understandable: a misprediction is like a barrier, with no code after the misprediction being executable with code before the branch; hence, sequentiality is introduced by the mispredictions. Remove the effect of the mispredictions, and you get oracle speedups.

2.2 Example of Code Execution

In Figure 3 we present an example of ILP execution of a real program, the kernel (main loop) of the `compress` SPECint92 benchmark. The static C source code appears first, followed by the dynamic trace of the assembly code executed over two iterations of the loop. The left-hand numbers on each assembly statement line are the line numbers of the corresponding source code. The address of the statement is in the next column, followed by its time (cycle) of execution in 4 different cases, corresponding to the SP, SP-CD-MF and DEE-CD-MF (two versions) models. The assembly statement follows. It is MIPS R4400 code, generated by an SGI C compiler. Renaming (minimal data dependencies) was assumed. Initially, each instruction executes in 1 cycle (non-unit latencies are addressed in Section 3.3). Note that this machine uses delayed branches, i.e., the instruction following the branch is executed independently of the branch. The branch sign follows each branch. If the branch was mispredicted, the notation "MIS" is appended. There were only two mispredictions in the example. A two-bit counter predictor was assumed. The scheduling of the execution for the SP and SP-CD-MF models was done by hand, while the DEE-CD-MF models' scheduling was automated using behavioral VHDL (C could also have been easily used).

We now analyze the executions. For the SP model, the mispredictions act as barriers, substantially hindering performance; the speedup is 8 (see the end of the trace for a speedup summary). When the minimal control dependency model is additionally employed (SP-CD-MF), the deleterious effect of the first misprediction is nullified: the misprediction no longer adds to the execution time. However, the second misprediction is still a barrier. The branch was predicted taken, and the wrong code was executed. The right code must execute after the branch resolves. Note that this was also true with the first misprediction, but there only one non-critical instruction had to wait for the branch to resolve. The rest of the code is control independent of the branch, and need not wait. With the second misprediction, the incorrectly skipped instructions are critical. The speedup is now 12.9. With the DEE-CD-MF model, again the first misprediction is not a problem, and now the second misprediction is not a problem. With DEE, the three branch paths (a DEE path) following the not-predicted path of the branch do not wait for the branch to resolve, and execute sooner. The DEE path code would have executed even sooner, but the resources for the path did not become available until time 8, by the static heuristic model. The speedup is now 16.8. If we were to extrapolate the last model's execution for more of the code's execution, we would see that the resources are released relatively sooner for succeeding iterations, until the resources are always ready when needed, and the second column DEE-CD-MF(+) execution times result. The (+) results are for the next two iterations; the same flow as for the first two iterations was assumed. Note that the fourth iteration ends at time 16, 6 cycles after the first two iterations completely executed. Each iteration of the kernel executes in about 3 cycles. The speedup is now 28. The second and last misprediction is effectively nullified, and the only concurrency inhibitors are data-flow ones, the classically minimal ones; hence, the speedup now is about the same as the oracle speedup.

All of these speedup numbers are in line with the experimental results of [15]. The example confirms our theory and intuition.

3 Realization and Implementation Issues

Many factors may reduce the beneficial effects of microarchitectural enhancements to the CPU of a computer. In this section we consider two of the most important: possible increased cycle time, and non-unit instruction execution latencies. Memory system accesses are included in the latter.

3.1 Cycle Time

Whenever microarchitectural performance enhancements to uniprocessors are contemplated, their effect on the cycle time of the computer must be examined: an excessive lengthening of the cycle time will negate the performance gain. Cycle time can be lengthened by either a larger number of gate delays in the critical path, or longer signal propagation transit times. In the first way, a large-scale ILP exploitation requires more scheduling logic, potentially having greater delay. In Levo[15], the critical path is through the SEN (Sink ENable) logic, which both schedules instructions for execution, and gates the proper source data from the Shadow SInk (SSI) matrix of renaming registers to the PE's for the executing (issuing) instructions. The basic form of the SEN logic for one source is described algebraically by the following Boolean expression[11, 13]:

$$SEN_{t,u} = DD_{t,u} \cdot RE_t \prod_{s=t+1}^{u-1} (\overline{DD}_{s,u} + VE_s)$$

Instances u are all of the instructions possibly requiring data (all instructions in the IQ). For each u there is a set of SEN outputs t for the possible SSI sources of the datum. The s instances are those between a t and a u. SEN will be true for exactly one t of a u, if u has a valid (computed) source. u will issue and be executed only if one of its SEN values is true; if none are true, the source datum has not yet been computed, and u will not issue. This logic is patented[14].

The SEN logic requires the equivalent of high fan-in gates. Students in our VLSI class

```
#kernel of compress.c
313:    #define htabof(i)       htab[i]         //hash table
314:    #define codetabof(i)    codetab[i]      //code table

785:    while ( (c = getchar()) != EOF ) {
786:            in_count++;
787:            fcode = (long) (((long) c << maxbits) + ent);
788:            i = ((c << hshift) ^ ent);      /* xor hashing */
789:
790:            if ( htabof (i) == fcode ) {
791:                ent = codetabof (i);
792:                continue;
793:            } else if ( (long)htabof (i) < 0 )      /* empty slot */
794:                goto nomatch;
795:            disp = hsize_reg - i; // secondary hash
                                      //     (after G. Knott)
796:            if ( i == 0 )
797:                disp = 1;
798:    probe:
799:            if ( (i -= disp) < 0 )
800:                i += hsize_reg;
801:
802:            if ( htabof (i) == fcode ) {
803:                ent = codetabof (i);
804:                continue;
805:            }
806:            if ( (long)htabof (i) > 0 )
807:                goto probe;
808:    nomatch:
809:            output ( (code_int) ent );
810:            out_count++;
811:            ent = c;
815:            if ( free_ent < maxmaxcode ) {
816:                codetabof (i) = free_ent++; /* code -> hashtable */
817:                htabof (i) = fcode;
818:            }
819:            else if ((count_int)in_count >=
                             checkpoint && block_compress)
820:                cl_block ();
822:    }
```

```
856:    /*****************************************************************
857:     * TAG( output )
858:     *
859:     * Output the given code.
860:     * Inputs:
861:     *      code:   A n_bits-bit integer.  If == -1, then EOF.
         *              This assumes
862:     *              that n_bits =< (long)wordsize - 1.
863:     * Outputs:
864:     *      Outputs code to the file.
865:     * Assumptions:
866:     *      Chars are 8 bits long.
867:     * Algorithm:
868:     *      Maintain a BITS character long
         * buffer (so that 8 codes will
869:     * fit in it exactly).  Use the VAX insv
         * instruction to insert each
870:     * code in turn.  When the buffer fills
         * up empty it and start over.
871:     */
872:
873:    static char buf[BITS];
874:
875:    char_type lmask[9] = {0xff, 0xfe, 0xfc, 0xf8, 0xf0, 0xe0, 0xc0,
                               0x80, 0x00};
876:    char_type rmask[9] = {0x00, 0x01, 0x03, 0x07, 0x0f, 0x1f, 0x3f,
                               0x7f, 0xff};
877:
878:    output( code )
879:    code_int  code;
880:    {
885:        /*
886:         * On the VAX, it is important to have
             * the register declarations
887:         * in exactly the order given, or the asm will break.
888:         */
889:        register int r_off = offset, bits= n_bits;
890:        register char * bp = buf;
896:
897:        if ( code >= 0 ) {
898:    /*
899:     * byte/bit numbering on the VAX is simulated
         * by the following code
900:     */
901:            /*
902:             * Get to the first byte.
903:             */
904:            bp += (r_off >> 3);
905:            r_off &= 7;
906:            /*
907:             * Since code is always >= 8 bits, only need to
908:             * mask the first hunk on the left.
909:             */
910:            *bp = (*bp & rmask[r_off]) | (code << r_off)
                                             & lmask[r_off];
911:            bp++;
912:            bits -= (8 - r_off);
913:            code >>= 8 - r_off;
914:            /* Get any 8 bit parts in the middle
                    (<=1 for up to 16 bits). */
915:            if ( bits >= 8 ) {
916:                *bp++ = code;
917:                code >>= 8;
918:                bits -= 8;
919:            }
920:            /* Last bits. */
921:            if(bits)
922:                *bp = code;
923:            offset += n_bits;
924:            if ( offset == (n_bits << 3) ) {
925:                bp = buf;
926:                bits = n_bits;
927:                bytes_out += bits;
928:                do
929:                    putchar(*bp++);
930:                while(--bits);
931:                offset = 0;
932:            }
933:
934:            /*
935:             * If the next entry is going to be too big for
936:             * the code size, then increase it, if possible.
937:             */
938:            if ( free_ent > maxcode || (clear_flg > 0))
939:            {
                    // ...
968:            }
984:        }
985:    }
```

```
SP, SP-CD-MF, DEE-CD-MF(2): iterations 1 & 2.  DEE-CD-MF(+): it. 3 & 4.

                  models:   SP  DEE-
                            CD  CD-MF
               address    SP MF (2)(+)           branch resolution
                          |  |  |  |
  [src 1: 786] 0x40100c:   1  1  1  1   lui     t6,0x1000
  [src 1: 786] 0x401010:   2  2  2  8   lw      t6,620(t6)
  [src 1: 787] 0x401014:   1  1  1  1   lui     t8,0x1000
  [src 1: 786] 0x401018:   3  3  3  9   addiu   t7,t6,1
  [src 1: 787] 0x40101c:   2  2  2  2   lw      t8,60(t8)
  [src 1: 786] 0x401020:   1  1  1  1   lui     at,0x1000
  [src 1: 788] 0x401024:   1  1  1  8   sllv    t6,s5,s2
  [src 1: 786] 0x401028:   4  4  4  10  sw      t7,620(at)
  [src 1: 788] 0x40102c:   2  2  2  11  xor     v0,t6,s0
  [src 1: 790] 0x401030:   3  3  3  12  sll     t7,v0,2
  [src 1: 787] 0x401034:   3  3  3  8   sllv    t9,s5,t8
  [src 1: 790] 0x401038:   4  4  4  13  addu    t8,s7,t7
  [src 1: 790] 0x40103c:   5  5  5  14  lw      a0,0(t8)
  [src 1: 787] 0x401040:   4  4  4  11  addu    v1,t9,s0
  [src 1: 787] 0x401044:   5  5  5  12  move    s6,v1
  [src 1: 790] 0x401048:   6  6  6  15  bne     v1,a0,0x401064   T
  [src 1: 788] 0x40104c:   3  3  3  12  move    s4,v0

  [src 1: 793] 0x401064:   6  6  6  15  bltz    a0,0x4010bc      T
  [src 1: 793] 0x401068:   1  1  1  11  move    a0,s0
  [src 1: 809] 0x4010bc:   1  1  1  1   jal     output
  [src 1: 809] 0x4010c0:   1  1  1  4   sw      s2,44(sp)
call (uncond. branch)
        output:
  [src 1: 879] 0x4007e4:   1  1  1  1   lui     t0,0x1006
  [src 1: 879] 0x4007e8:   1  1  1  1   lui     s3,0x1006
  [src 1: 888] 0x4007ec:   1  1  1  1   lui     s1,0x1000
  [src 1: 879] 0x4007f0:   2  2  2  2   addiu   s3,s3,29672
  [src 1: 879] 0x4007f4:   2  2  2  5   lw      t0,29664(t0)
  [src 1: 879] 0x4007f8:   1  1  1  3   addiu   sp,sp,-40
  [src 1: 888] 0x4007fc:   2  2  2  2   lw      s1,8752(s1)
  [src 1: 879] 0x400800:   2  2  2  4   sw      ra,20(sp)
  [src 1: 896] 0x400804:   2  2  2  12  bltz    a0,0x400a84      N
  [src 1: 889] 0x400808:   3  3  3  3   move    s0,s3
```

Figure 3: *Example of DEE-CD-MF, SP-CD-MF and SP execution of* `compress`. The C source code is first, followed by the dynamic assembly code. (Continued on next page.)

```
  [src 1: 903] 0x40080c:   3  3  3  6    sra     t6,t0,3
  [src 1: 904] 0x400810:   3  3  3  6    andi    v0,t0,0x7
  [src 1: 909] 0x400814:   1  1  1  1    lui     t8,0x1000
  [src 1: 903] 0x400818:   4  4  4  7    addu    s0,s3,t6
  [src 1: 909] 0x40081c:   4  4  4  7    addu    t8,t8,v0
  [src 1: 909] 0x400820:   5  5  5  8    lbu     t8,656(t8)
  [src 1: 909] 0x400824:   5  5  5  9    lbu     t7,0(s0)
  [src 1: 909] 0x400828:   4  4  4  12   sllv    t6,a0,v0
  [src 1: 909] 0x40082c:   6  6  6  10   and     t9,t7,t8
  [src 1: 909] 0x400830:   1  1  1  1    lui     t7,0x1000
  [src 1: 909] 0x400834:   4  4  4  7    addu    t7,t7,v0
  [src 1: 909] 0x400838:   5  5  5  8    lbu     t7,644(t7)
  [src 1: 910] 0x40083c:   5  5  5  8    addiu   s0,s0,1
  [src 1: 909] 0x400840:   6  6  6  13   and     t8,t6,t7
  [src 1: 909] 0x400844:   7  7  7  14   or      t6,t9,t8
  [src 1: 909] 0x400848:   8  8  8  15   sb      t6,-1(s0)
  [src 1: 911] 0x40084c:   1  1  1  1    lui     t9,0x1000
  [src 1: 911] 0x400850:   2  2  2  2    lw      t9,8752(t9)
  [src 1: 911] 0x400854:   1  1  1  1    li      t7,8
  [src 1: 911] 0x400858:   4  4  4  7    subu    v1,t7,v0
  [src 1: 911] 0x40085c:   5  5  5  8    subu    s1,t9,v1
  [src 1: 914] 0x400860:   6  6  6  9    slti    at,s1,8
  [src 1: 914] 0x400864:   7  7  7  10   bne     at,zero,0x40087c T
  [src 1: 912] 0x400868:   5  5  5  12   srav    a0,a0,v1

  [src 1: 920] 0x40087c:   6  6  6  9    beq     s1,zero,0x400888 N MIS
  [src 1: 922] 0x400880:   1  1  1  1    lui     t8,0x1000

SP: time >= 7

  [src 1: 921] 0x400884:   7  7  6  13   sb      a0,0(s0)

SP-CD-MF & DEE-CD-MF: Following is executed earlier, since
        it is not in domain of 87c.

  [src 1: 922] 0x400888:   7  2  2  2    lw      t8,8752(t8)
  [src 1: 923] 0x40088c:   7  1  1  1    lui     at,0x1006
  [src 1: 922] 0x400890:   8  3  3  6    addu    t0,t0,t8
  [src 1: 923] 0x400894:   8  3  3  3    sll     t6,t8,3
  [src 1: 923] 0x400898:   9  4  4  7    bne     t0,t6,0x400950   T
  [src 1: 926] 0x40089c:   7  1  1  1    lui     t7,0x1006

  [src 1: 937] 0x400950:   7  1  1  1    lui     t8,0x1000
  [src 1: 937] 0x400954:   7  1  1  1    lui     t9,0x1000
  [src 1: 937] 0x400958:   8  2  2  5    lw      t9,72(t9)
  [src 1: 937] 0x40095c:   8  2  2  2    lw      t8,8756(t8)
  [src 1: 931] 0x400960:   7  1  1  1    lui     s2,0x1000
  [src 1: 937] 0x400964:   9  3  3  6    slt     at,t8,t9
  [src 1: 937] 0x400968:   10 4  4  7    bne     at,zero,0x400984 N
  [src 1: 931] 0x40096c:   8  2  2  2    addiu   s2,s2,1844

  [src 1: 937] 0x400970:   7  1  1  1    lui     t6,0x1000
  [src 1: 937] 0x400974:   8  2  2  2    lw      t6,148(t6)
  [src 1: 937] 0x400978:   7  1  1  1    lui     at,0x1006
  [src 1: 937] 0x40097c:   9  3  3  3    blez    t6,0x400b40      T
  [src 1: 937] 0x400980:   10 4  4  7    sw      t0,29664(at)

  [src 1: 984] 0x400b40:   7  3  3  5    lw      ra,20(sp)
  [src 1: 984] 0x400b44:   7  2  2  4    addiu   sp,sp,40
  [src 1: 984] 0x400b48:   8  4  4  6    jr      ra
  [src 1: 984] 0x400b4c:   7  1  1  1    nop
return
  [src 1: 810] 0x4010c4:   7  1  1  1    lui     t7,0x1000
  [src 1: 810] 0x4010c8:   8  2  2  5    lw      t7,624(t7)
  [src 1: 809] 0x4010cc:   7  1  1  1    lui     v1,0x1000
  [src 1: 815] 0x4010d0:   7  1  1  1    lui     t9,0x1000
  [src 1: 809] 0x4010d4:   8  2  2  5    lw      v1,72(v1)
  [src 1: 815] 0x4010d8:   8  2  2  2    lw      t9,64(t9)
  [src 1: 810] 0x4010dc:   7  1  1  1    lui     at,0x1000
  [src 1: 810] 0x4010e0:   9  3  3  6    addiu   t8,t7,1
  [src 1: 810] 0x4010e4:   10 4  4  7    sw      t8,624(at)
  [src 1: 809] 0x4010e8:   7  1  1  1    lui     s1,0x1004
  [src 1: 809] 0x4010ec:   7  2  3  5    lw      s2,44(sp)
  [src 1: 815] 0x4010f0:   9  3  3  6    slt     at,v1,t9
  [src 1: 809] 0x4010f4:   8  2  2  2    addiu   s1,s1,22624
  [src 1: 815] 0x4010f8:   10 4  4  7    beq     at,zero,0x401128 N
  [src 1: 811] 0x4010fc:   7  1  1  8    move    s0,s5

  [src 1: 817] 0x401100:   7  4  4  13   sll     t6,s4,1
  [src 1: 817] 0x401104:   8  5  5  14   addu    t7,s1,t6
  [src 1: 817] 0x401108:   9  6  6  15   sh      v1,0(t7)
  [src 1: 818] 0x40110c:   7  4  4  13   sll     t8,s4,2
  [src 1: 818] 0x401110:   8  5  5  14   addu    t9,s7,t8
  [src 1: 817] 0x401114:   10 3  3  6    addiu   v1,v1,1
  [src 1: 818] 0x401118:   9  6  6  15   sw      s6,0(t9)
  [src 1: 819] 0x40111c:   7  1  1  1    lui     at,0x1000
  [src 1: 819] 0x401120:   7  1  1  1    b       0x40127c
  [src 1: 819] 0x401124:   11 4  4  7    sw      v1,72(at)
uncond. branch
  [src 1: 822] 0x40127c:   7  1  1  1    lui     t9,0x1000
  [src 1: 822] 0x401280:   8  2  2  2    lw      t9,8564(t9)
  [src 1: 822] 0x401284:   7  1  1  1    lui     t6,0x1000
  [src 1: 822] 0x401288:   9  3  3  3    beq     t9,zero,0x4012a0 T
  [src 1: 822] 0x40128c:   7  1  1  1    lui     a0,0x1000
```

```
  [src 1: 822] 0x4012a0:   8  2  2  8    lw      t6,1828(t6)
  [src 1: 822] 0x4012a4:   7  1  1  1    lui     at,0x1000
  [src 1: 822] 0x4012a8:   9  3  3  9    addiu   t7,t6,-1
  [src 1: 822] 0x4012ac:   10 4  4  10   bgtz    t7,0x4012d0      T
  [src 1: 822] 0x4012b0:   10 4  4  10   sw      t7,1828(at)

  [src 1: 822] 0x4012d0:   7  1  1  1    lui     t6,0x1000
  [src 1: 822] 0x4012d4:   8  2  2  8    lw      t6,1832(t6)
  [src 1: 822] 0x4012d8:   7  1  1  1    lui     at,0x1000
  [src 1: 822] 0x4012dc:   9  3  3  9    lbu     v0,0(t6)
  [src 1: 822] 0x4012e0:   9  3  3  9    addiu   t7,t6,1
  [src 1: 822] 0x4012e4:   7  1  1  1    b       0x401324
  [src 1: 822] 0x4012e8:   10 4  4  10   sw      t7,1832(at)
uncond. branch
  [src 1: 822] 0x401324:   10 4  4  10   move    s5,v0
  [src 1: 822] 0x401328:   7  1  1  1    li      at,-1
  [src 1: 822] 0x40132c:   11 5  5  11   bne     s5,at,0x40100c
  [src 1: 822] 0x401330:   7  1  1  1    nop

Second iteration:

  [src 1: 786] 0x40100c:   7  1  1  1    lui     t6,0x1000
  [src 1: 786] 0x401010:   8  5  5  11   lw      t6,620(t6)
  [src 1: 787] 0x401014:   7  1  1  1    lui     t8,0x1000
  [src 1: 786] 0x401018:   9  6  6  12   addiu   t7,t6,1
  [src 1: 787] 0x40101c:   8  2  2  2    lw      t8,60(t8)
  [src 1: 786] 0x401020:   7  1  1  1    lui     at,0x1000
  [src 1: 788] 0x401024:   11 5  5  11   sllv    t6,s5,s2
  [src 1: 786] 0x401028:   10 7  7  13   sw      t7,620(at)
  [src 1: 788] 0x40102c:   12 6  6  12   xor     v0,t6,s0
  [src 1: 790] 0x401030:   13 7  7  13   sll     t7,v0,2
  [src 1: 787] 0x401034:   11 5  5  11   sllv    t9,s5,t8
  [src 1: 790] 0x401038:   14 8  8  14   addu    t8,s7,t7
  [src 1: 790] 0x40103c:   15 9  9  15   lw      a0,0(t8)
  [src 1: 787] 0x401040:   12 6  6  12   addu    v1,t9,s0
  [src 1: 787] 0x401044:   13 7  7  13   move    s6,v1
  [src 1: 790] 0x401048:   16 10 10 16   bne     v1,a0,0x401064   N MIS
  [src 1: 788] 0x40104c:   13 7  7  13   move    s4,v0

SP: time >= 17

SP-CD-MF: time >= 11

DEE-CD-MF(2): 3 branch path long DEE path at time 8
        (resources available at end of time 7,
        after 0864 (bne) executes, releasing all
        resources prior to path just above).

DEE-CD-MF(+): 1 branch path long DEE path at time 13
        (resources available at end of time 12,
        after 0804 (bltz) executes, releasing all
        needed resources).

  [src 1: 791] 0x401050:   17 11 8  14   sll     t9,s4,1
  [src 1: 791] 0x401054:   18 12 9  15   addu    t6,s1,t9
  [src 1: 791] 0x401058:   19 13 10 16   lhu     s0,0(t6)
  [src 1: 792] 0x40105c:   17 11 8  13   b       0x40127c
  [src 1: 792] 0x401060:   17 11 8  13   nop
uncond. branch
SP-CD-MF and DEE-CD-MF: would rejoin after 1048 T path to 1064.

  [src 1: 822] 0x40127c:   17 1  1  1    lui     t9,0x1000
  [src 1: 822] 0x401280:   18 2  2  2    lw      t9,8564(t9)
  [src 1: 822] 0x401284:   17 1  1  1    lui     t6,0x1000
  [src 1: 822] 0x401288:   19 3  3  3    beq     t9,zero,0x4012a0 T
  [src 1: 822] 0x40128c:   17 1  1  1    lui     a0,0x1000

  [src 1: 822] 0x4012a0:   18 5  5  11   lw      t6,1828(t6)
  [src 1: 822] 0x4012a4:   17 1  1  1    lui     at,0x1000
  [src 1: 822] 0x4012a8:   19 6  6 12    addiu   t7,t6,-1
  [src 1: 822] 0x4012ac:   20 7  7 13    bgtz    t7,0x4012d0      T
  [src 1: 822] 0x4012b0:   20 7  7 13    sw      t7,1828(at)

  [src 1: 822] 0x4012d0:   17 1  1  1    lui     t6,0x1000
  [src 1: 822] 0x4012d4:   18 5  5  11   lw      t6,1832(t6)
  [src 1: 822] 0x4012d8:   17 1  1  1    lui     at,0x1000
  [src 1: 822] 0x4012dc:   19 6  6  12   lbu     v0,0(t6)
  [src 1: 822] 0x4012e0:   19 6  6  12   addiu   t7,t6,1
  [src 1: 822] 0x4012e4:   17 1  1  1    b       0x401324
  [src 1: 822] 0x4012e8:   20 7  7  13   sw      t7,1832(at)
uncond. branch
  [src 1: 822] 0x401324:   20 7  7  13   move    s5,v0
  [src 1: 822] 0x401328:   17 1  1  1    li      at,-1
  [src 1: 822] 0x40132c:   21 8  8  14   bne     s5,at,0x40100c  T
  [src 1: 822] 0x401330:   17 1  1  1    nop

(Start of Iteration 3.)

Speedups        - 168 dynamic instructions -
    SP:                   21 cycles -> S= 8.0
    SP-CD-MF:             13 cycles -> S=12.9
    DEE-CD-MF:
        (2 iterations): 10 cycles -> S=16.8
        (>= 3 iterat.):  6 cycles -> S=28.0
```

Figure 3: (continued) *Example of DEE-CD-MF, SP-CD-MF and SP execution of* `compress`*.*

designed a 4-input OR gate with less delay than a 3-input NAND. Assuming about 64 possible outputs (t) per u, the use of both the special OR gates and other *simple* gates (2-3 input NAND or NOR gates with fan-out of 2-3 and low RC delays), and including the other critical path logic, a nominal cycle time for a DEE-CD-MF computer was computed. For comparison purposes, the students also laid out and simulated a string of 9-10 simple gates in the same technology, indicative of the critical path delay (with latch) of the Pentium Pro[3, 7]. This gave us a base cycle time.

The DEE-CD-MF model's cycle time is slightly longer than the base machine's cycle time, giving potentially about a 30% loss in performance in the new model (only considering cycle time), or a net speedup of a factor of 26.0(10/13)=<u>20.0</u>. This is still about an order of magnitude better than current machines. Further, given that the full logic design efforts of a company such as Intel have not yet been applied to the logic, it may be possible to reduce the critical path delay through the logic later on.

3.2 Signal Propagation Delays

Another factor in cycle time determination is the distance signals in the critical path travel on the chip (we assume that the final realization of the uniprocessor is on a single chip, i.e., it is a microprocessor). This is due to the high frequencies of operation of current (200 MHz for the Intel Pentium Pro, 500 MHz for the latest DEC Alpha chip) and future microprocessors (1 GHz operation is planned by most microprocessor manufacturers). This became a key issue in the design of the Intel Pentium Pro[3]. Our initial estimate for Levo, assuming a very unoptimized rectangular floorplanning of the chip, was that one chip-crossing will be required per instruction execution, going to and from the Processing Elements and the renaming registers. The chip-crossing time may be partially compensated for by the increased cycle time. However, it may still be significant.

We investigated the effect of chip-crossing time, or more generally the *data travel* time, on the execution of the `compress` code with the DEE-CD-MF model above, by increasing the latency of every instruction from 1 to 2 cycles. Since there is at least one limiting data-dependency cycle in the code, the execution time doubled from 6 to 12 cycles (performance was halved). The code was then modified to break the dependency cycles of the loop counters; the execution time improved to 10 cycles. However, this is still a large performance drop.

Examining the floorplanning of the chip more carefully, and using our estimates of the hardware requirements of the individual sections of the prototype, Levo, we propose bringing the core elements of the critical data path together at the center of the chip. Thus, the PE and renaming register interconnect will be at the very center, with the PE's and registers fanning out radially from the center. (This is somewhat like the Cray-1.) With this arrangement, we estimate that a half-chip crossing or less will be needed per instruction execution. The net effect of the data travel time should thus be low.

Therefore we do not expect a major adverse impact on Levo's cycle time from either its larger amount of logic or the data travel time.

3.3 Instruction Latencies

The prior DEE-CD-MF simulations[15] assumed 1 cycle instruction latencies. It has been suggested that real-world instruction latencies will severely reduce the performance figures shown for DEE-CD-MF. For the `compress` benchmark above, all of the arithmetic and logic statements in the kernel have unit latencies in the MIPS R4400; longer latency instructions such as integer multiplies do not appear in the kernel. The use of low latency assignment instructions is typical of the SPECint92 benchmarks, and non-scientific user and system code in general. Branch latencies are not a large factor, due to the DEE-CD-MF model.

Loads, however, may have higher latencies. To investigate this, load latencies were varied using the automated scheduler. 2 cycle loads did not affect the execution time of the DEE-CD-MF(+) code (6 cycles). Each additional latency cycle added 1 cycle to the execution time of the two iterations, e.g., a latency of 4 gave an execution time of 8 cycles, still an ILP speedup of 21. Thus, latencies are not likely to be a major issue.

4 DEE Theory and Characteristics

4.1 Theory

DEE is an optimal form of speculative execution, when resources are constrained[15]. In [15], a *static tree* heuristic was proposed, and demonstrated to yield substantial performance gains, especially when combined with Minimal Control Dependencies. Characteristics of this tree are now discussed, and its shape is explained, through the presentation and proof of several theorems. The formulae used to compute the tree's dimensions are also derived.

4.1.1 Notation

We assume an arbitrary tree of branch paths BP, as shown in Figure 4, and make the left-pointing paths the predicted paths, and the right-pointing paths the not-predicted paths, as before. The individual paths are then indicated by BP_k, where k is a variably-lengthed string of 1's and 0's. A branch path's k string has one character for each pending ancestor branch of the path. The character is a 1 if the ancestor took its predicted path to BP_k, and 0 if the not-predicted path was taken. The leftmost character corresponds to the root branch, with succeeding characters referring to branches of successively lower levels of the tree. For example, BP_{110} refers to the branch path arrived at by following the predicted path of the root branch, the predicted path of the next lower branch, and the not-predicted path of the next lower branch from that. k may be composed using certain operators: "$j\{c\}$" means concatenate a group of j characters c together, while a comma "," is the concatenation operator. An example is: $BP_{3\{1\},0,2\{1\}} = BP_{111011}$

A branch path in the DEE region may also be referred to as: $DEE_k = BP_k$. A branch path in the MainLine (ML) region or path may also be referred to as: ML_l, where l is the depth of branching of the path (number of levels); note that this is a <u>different</u> numbering convention from that just described. The associated cumulative probability of a branch path is indicated by: $cpBP_k$, $cpML_l$ or $cpDEE_k$.

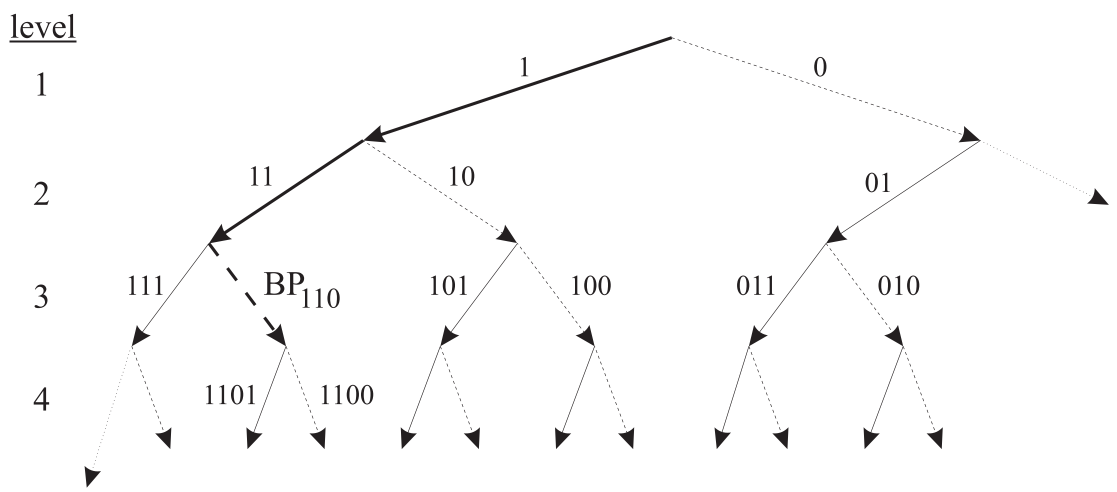

Figure 4: *Notation for branch path labeling.* An arbitrary tree of branches (vertices) and branch paths (edges) is shown, with the labeling convention. Left-pointing (solid) paths are predicted paths, while right-pointing (dashed) ones are not-predicted paths.

4.1.2 Proof of the Static Tree Shape

Our goal is to show that the shape of the static tree is as shown in Figure 2 on page 4, for fixed assumed branch prediction accuracy p and branch path resources E_T. Starting from no branch path resources, adding resources initially forms a tree composed of a straight line, equivalent to Single Path (SP) execution (see Figure 1 on page 3). Theorem 1 establishes at what point in the addition of resources a DEE region begins to be formed.

Theorem 1 *Given a branch prediction accuracy p the first branch path in the DEE region is created upon the addition of one more branch path resource to the CPU when the cumulative probability of the path just beyond the tail of the tree $cpML_{l+1}$ is less than the cumulative probability of the first not-predicted path $cpBP_0$ (see Figure 4). Mathematically, this is when:*

$$cpML_{l+1} = p^{l+1} < 1 - p = cpBP_0$$

where, in this case, $l = E_T = l_0$. l_0 is the value of l at which DEE paths begin to appear.

Proof By the rule of Greatest Marginal Benefit[15], resources are assigned next to the path with the highest cumulative probability. Since the cumulative probability of the first not-predicted path $(1-p)$ must always be greater than that of subsequent not-predicted paths, off of the Main-Line region, as well as being greater than any child paths of its own, it will be the first path to be assigned resources when speculation on the ML path has gone sufficiently deep. Since cumulative probabilities along the ML path are monotonically decreasing (no unconditional branches), the condition will be reached when the cumulative probability of the tail path has become low enough, which is when it is lower than the first not-predicted path's cumulative probability.

Since the static tree assumes a constant p, the tail's cumulative probability is p multiplied by itself once for every additional level in the tree, or: p^{l_0}. The cumulative probability of the path just after the tail is the latter times p again, or: p^{l_0+1}.

Combining the above, the DEE region starts when: $p^{l+1} < 1 - p$ □

Corollary 2 *The DEE region is empty while:* $p^l > 1 - p$

Proof Follows trivially from Theorem 1. □

Theorem 3 *The shape of the static tree when E_T increases further is as shown in Figure 2.*

Proof The proof is by induction.

The induction is anchored by Theorem 1 and Corollary 2. After the l_0th resource ML_{l_0} has been added, the next added resource creates the first row of the DEE region, consisting of one branch path DEE_0.

The inductive premise on i is:

> With the last added resource assigned to the end of the ML path, as resources increase, branch paths are added alternately to the ML and DEE regions, adding a path to the tail of the ML region and then all of the next row (down) of branch paths in the DEE region.

That is, if the static tree has depth $l = l_0 + i$, and the DEE region height is i, implying that the last DEE row contains i branch paths, then the next added $1 + (i + 1) = i + 2$ resources will create branch paths ML_{l_0+i+1} and $DEE_{j\{1\},0,(i-j)\{1\}}$ ($j = 0$ to i), the latter forming row $i+1$ of the DEE region. The ML path is lengthened first.

This is now shown to imply that for $i + 1$ the added $(i + 2) + 1$ branch paths are $ML_{l_0+(i+1)+1} = ML_{l_0+i+2}$ and $DEE_{j\{1\},0,(i-j)\{1\}}$ ($j = 0$ to $i + 1$). We begin by reformulating the anchor and the premise.

First, Corollary 2 implies that $p^{l_0} > 1-p$. Further, Theorem 1 shows that $1-p > p^{l_0+1}$ for a DEE region to exist. Therefore: $p^{l_0} > 1 - p > p^{l_0+1}$. Multiplying this by p gives: $p^{l_0+1} > (1 - p)p > p^{l_0+2}$. Combining the last two compound relations gives:

$$p^{l_0} > 1 - p > p^{l_0+1} > (1 - p)p$$

This is another form of the induction's anchor.

Now multiplying again, this time by p^i, gives the inductive premise:

$$p^{l_0+i} > (1 - p)p^i > p^{l_0+i+1} > (1 - p)p^{i+1}$$

Multiplying once more by p gives the implied case:

$$p^{l_0+i+1} > (1 - p)p^{i+1} \quad > \quad p^{l_0+i+2} > (1 - p)p^{i+2} \tag{1}$$

It remains to map this expression to branch paths, which we now do. The first and third expressions of equation 1 are equal to $cpML_{l_0+i+1}$ and $cpML_{l_0+i+2}$, respectively, which in turn correspond to branch paths ML_{l_0+i+1} and ML_{l_0+i+2}, respectively.

To see how the second and fourth expressions of equation 1 correspond to DEE branch paths, we derive a simplified expression for the cumulative probability of an arbitrary

DEE branch path. For i equal to the level of the path minus 1, and for $j = 0$ to i:

$$\begin{aligned} cpDEE_{j\{1\},0,(i-j)\{1\}} &= p^j(1-p)p^{i-j} \\ &= (1-p)p^j p^{i-j} \\ &= (1-p)p^i \end{aligned}$$

This implies that the cumulative probabilities of all DEE branch paths on the same row are the same, and thus have equal priority for being assigned resources.

Further, and more importantly, the second and fourth expressions of equation 1 are now seen to be equal to: $cpDEE_{j\{1\},0,(i+1-j)\{1\}}$ and $cpDEE_{j\{1\},0,(i+2-j)\{1\}}$, respectively. Combining the direct correspondence of these to the DEE branch paths, with all four of the expressions of equation 1 gives the following order of resource assignment:

$$ML_{l_0+i+1}, \text{ then } DEE_{j\{1\},0,(i+1-j)\{1\}}, \text{ then } ML_{l_0+i+2}, \text{ then } DEE_{j\{1\},0,(i+2-j)\{1\}}$$

Pictorially, this gives the shape of the static tree as shown in Figure 2. □

4.1.3 Derivation of the Static Tree's Dimension Formulae

The formulae relating p, l, h_{DEE} and E_T were originally presented without formal justification in [15]. We now derive them.

The total number of resources in the static tree is the sum of those in the MainLine region and those in the DEE region:

$$E_T = E_{ML} + E_{DEE} \tag{2}$$

From Theorem 3, since the width of the DEE region increases by 1 branch path for every new row of branch paths, the shape of the DEE region is that of a lower-right triangular matrix of branch paths, with side dimensions of $h_{DEE} \times h_{DEE}$. Thus, the resources in the

DEE region are:

$$E_{DEE} = h_{DEE}(h_{DEE}+1)/2$$

To determine the resources in the MainLine region, we note that there is an approximate equivalence between the cumulative probability of the last path in the MainLine region and the cumulative probabilities of the branch paths in the last row of the DEE region:

$$\begin{aligned} cpML_{E_{ML}} = cpML_{\,l} &\approx cpDEE_{j\{1\},0,(h_{DEE}-j-1)\{1\}},\ \forall j \mid 0 \leq j \leq h_{DEE}-1 \\ p^{E_{ML}} &\approx (1-p)p^{h_{DEE}-1} \\ l = E_{ML} &\approx \log_p(1-p) + h_{DEE} - 1 \end{aligned} \tag{3}$$

Substituting into equation 2:

$$E_T \approx \frac{1}{2}h_{DEE}^2 + \frac{3}{2}h_{DEE} + \log_p(1-p) - 1 \tag{4}$$

Solving for h_{DEE} with the quadratic formula:

$$h_{DEE} \approx -\frac{3}{2} + \frac{1}{2}\sqrt{8E_T - 8\log_p(1-p) + 17} \tag{5}$$

Equations 3, 4 and 5 thus show the approximate relationships among p, l, h_{DEE} and E_T.

Observation: The static tree actually represents a continuum between Single Path (SP) execution, when p approaches 1 (h_{DEE} goes to 0, from Corollary 2), and at the other extreme Eager Execution (EE), when p approaches 0.5 (h_{DEE} goes to l, from equation 3).

4.2 Characteristics

Plots of the h_{DEE} and l dimensions of the static tree are shown in Figure 5 for different combinations of total resources E_T and branch prediction accuracy p. The plots were computed using equations 3, 4 and 5 of Section 4.1.3. We call the point at which increasing E_T creates the first path of the DEE region the *DEE threshold*. The decreasing values of p, except for $p = 50\%$, are created from exponentially increasing values of the misprediction rate $(1 - p)$.

From the figure, as expected, until the DEE threshold is reached, l rises linearly with E_T (it is equal to it). Above the threshold, as l (E_{ML}) becomes a smaller part of E_T, h_{DEE} rises slightly faster than as the square root of E_T.

The DEE threshold increases with increasing branch prediction accuracy p. From the lower-left plot, the rate of logarithmic increase in the DEE threshold for exponentially decreasing values of the misprediction rate is constant.

From the plots, and given both that branch prediction accuracies are not likely to exceed around 93% on real system (OS) and application code on real machines, as well as the presence of increasingly large number of resources in computers, DEE should be used for high performance.

5 Conclusions

In this paper we have examined the DEE-CD-MF execution model in depth, and found the previously determined order-of-magnitude performance gains[15] to be justified.

Specifically, Amdahl's Law was used to illustrate how DEE works: by eliminating branch effects, the fraction of serial code is reduced, and thus performance is dramatically improved. Further, a code fragment from a real application was examined and analyzed in detail, verifying the large speedups of DEE-CD-MF. The critical path of a DEE-CD-MF prototype machine was analyzed and shown not to excessively increase cycle time, maintaining the large performance gains. An initial examination of latency effects was made, showing that they are likely not to be a significant factor. However, the effects must be studied more thoroughly before a definitive statement can be made on this account.

The characteristics of the DEE static tree were examined in depth. The shape of the

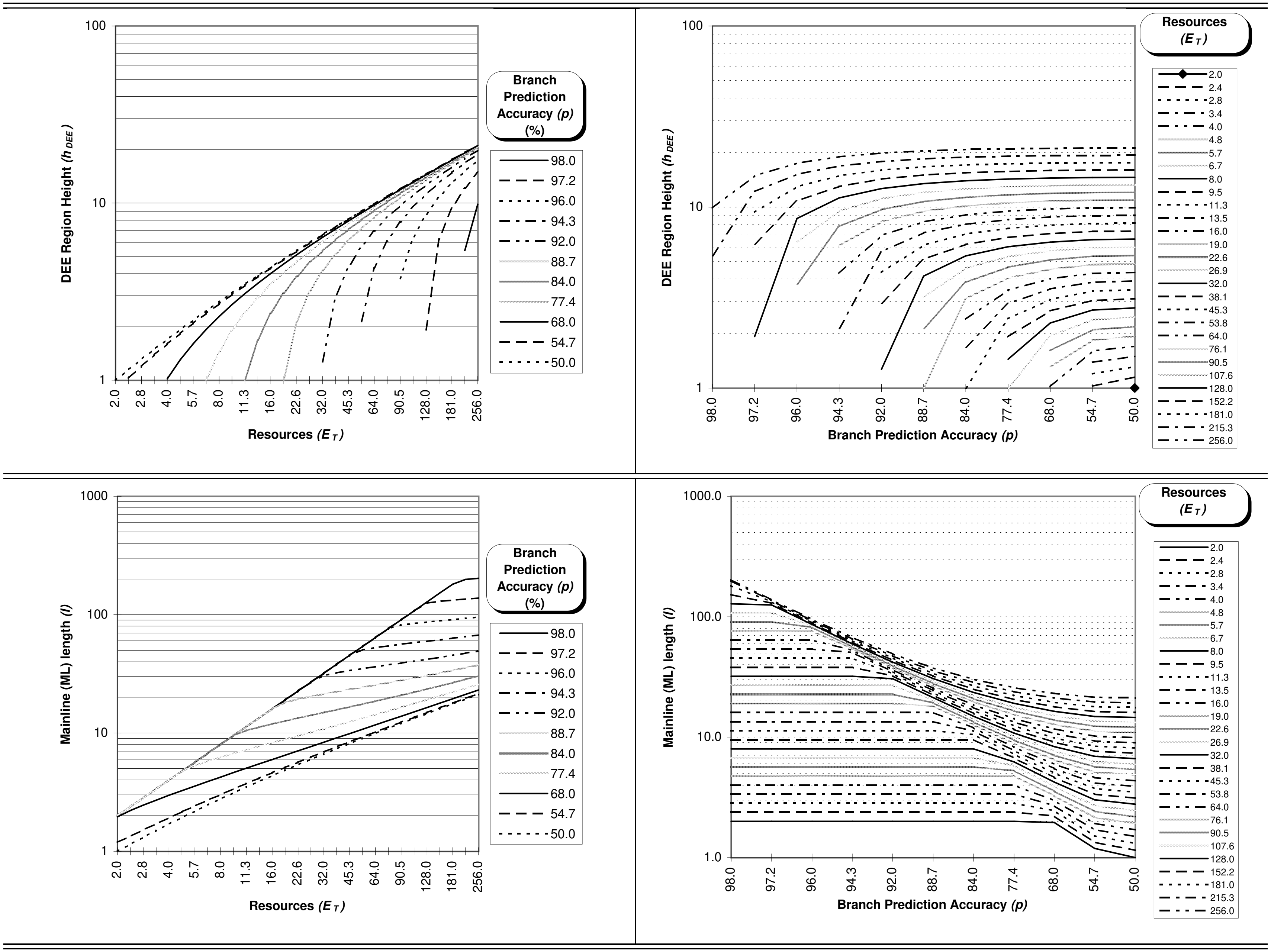

Figure 5: *Variation of static tree DEE region height h_{DEE} and MainLine path length l with resources E_T and branch prediction accuracy p.*

tree was derived, and the relationships between the tree dimensions, available resources, and branch prediction accuracy were derived, plotted and analyzed. DEE is shown to be valuable for high-performance computers (at the least).

Therefore we conclude that the DEE performance effects are real, will not disappear in implementation realities, and are desirable for high-end machines. Of course, much work remains to be done. We are currently constructing a detailed functional simulator for a DEE-CD-MF prototype, so that we may further investigate DEE, and study and solve the problems arising from such high performance CPUs (such as potentially high memory bandwidth).

References

[1] T. M. Austin and G. S. Sohi. Dynamic Dependency Analysis of Ordinary Programs. In *Proceedings of the 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia*, pages 342–351. IEEE and ACM, May 1992.

[2] P. Chang, S. Mahlke, W. Chen, N. Warter, and W. Hwu. IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors. In *Proceedings of the 18th Annual International Symposium on Computer Architecture, Toronto, Canada*, pages 266–275. IEEE and ACM, May 1991.

[3] R. P. Colwell, December 12-17, 1996. Personal communications.

[4] K. Ebcioğlu and E. R. Altman. DAISY: Dynamic Compilation for 100% Architectural Compatibility. IBM Research Report RC 20538, IBM Research Division, August 5, 1996.

[5] 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.

[6] M. Lipasti and J. Shen. Exceeding the Data Flow Limit via Value Prediction. In *Proceedings of the 29th International Symposium on Microarchitecture (MICRO-29)*, pages 226–237. IEEE and ACM, December 1996.

[7] D. B. Papworth. Tuning the Pentium Pro Microarchitecture. *MICRO*, 16(2):8–15, April 1996.

[8] 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.

[9] Y. Sazeides, S. Vassiliadis, and J. E. Smith. The Performance Potential of Data Dependence Speculation & Collapsing. In *Proceedings of the 29th International Symposium on Microarchitecture (MICRO-29)*, pages 238–247. IEEE and ACM, December 1996.

[10] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. IEEE and ACM, June 1995.

[11] A. K. Uht. *Hardware Extraction of Low-Level Concurrency from Sequential Instruction Streams*. PhD thesis, Carnegie-Mellon University, December 1985. Available from University Microfilms International, Ann Arbor, Michigan, U.S.A.

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

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

[14] A. K. Uht. System for Extracting Low-Level Concurrency from Serial Instruction Streams, April 6, 1993. U.S. Patent No. 5,201,057. This patent covers the CONDEL-2 machine.

[15] A. K. Uht and V. Sindagi. Disjoint Eager Execution: An Optimal Form of Speculative Execution. In *Proceedings of the 28th International Symposium on Microarchitecture (MICRO-28)*, pages 313–325. IEEE and ACM, November/December 1995.

[16] A. K. Uht, V. Sindagi, and S. Somanathan. Branch Effect Reduction Techniques. *COMPUTER*, 30(5):71–81, May 1997.

[17] D. Wall. Limits of Instruction-Level Parallelism. WRL Research Report 93/6, Digital Western Research Laboratory, Palo Alto, California, November 1993.