Disjoint Eager Execution: What It Is / What It Is Not

Augustus K. Uht

***Abstract*— Disjoint Eager Execution (DEE) has been cited and described many times since its introduction in 1992, often incorrectly. This paper clarifies what DEE is and how it operates, as well as pointing out common DEE misconceptions.**

***Index Terms*— Microarchitecture, high-performance computing, control speculation, eager execution.**

I. Introduction

DISJOINT EAGER EXECUTION (DEE) was initially proposed[9, 10] to realize the performance promised by limit studies of Instruction Level Parallelism (ILP), in particular Riseman and Foster's work[4] and Lam and Wilson's work[3]. Both works' conclusions were that if branch effects[14] could be reduced or eliminated, much ILP could be extracted with an ensuing increase in performance. That was the good news; the bad news was that there was no known way of obtaining this ILP without a prohibitive hardware cost.

DEE was characterized by performance measurements [13] in 1995. Since then it has been gratifying to see it cited several times and spawn other work. It has been less gratifying to see that I did not do a good enough job of explaining DEE in that paper, with the end result that DEE is often misunderstood or incorrectly described. This has been true regardless of the experience of the citers.

The purpose of this paper is to briefly present an explanation of the DEE concept in order to clarify aspects of it that may have been misunderstood in the past.

II. What DEE Is

Several researchers have explored a variety of techniques to try and extract high ILP from general programs. Most of the existing techniques have focused on simple branch prediction and *Single Path (SP) speculation* (see Fig. 1). Riseman and Foster proposed *Eager Execution (EE),* perhaps the simplest form of multipath execution (see Fig. 1). In EE both paths of a branch are taken; if another branch is encountered before the first is resolved (executed), execution also proceeds down both paths of the second branch. The problem with SP is that the likelihood of needing the code at the end of multiple levels of predictions approaches zero, so the performance is not good. However, the hardware cost increases linearly with the number of branches predicted. The problem with EE is its exponential increase in cost with the number of branch levels eagerly executed. With infinite hardware, performance is the same as that with an oracle, and is very high. However, the cost is prohibitive even to get a small fraction of the peak performance.

So what should be done? Several approaches were investigated without success until the problem was more carefully examined. Where does one want to put one's resources to get the best performance? "Obviously," at the most likely code to be executed. But this is not SP, nor is it EE; it is something different. It is DEE.

Now let's take the name apart to see what it means:

1. "execution": speculative execution.
2. "eager": go down both paths of a branch.
3. "disjoint": go down each path of a branch at different times -- *when the path is the most likely to be executed among the paths that have not yet received resources.*

That last statement is true for all paths at all times. Put a slightly different way, the DEE rule is: *At all times, allocate resources to the most likely paths to be executed over the entire branch path space*.

Now, we still need a predictor to tell us the predicted paths. Furthermore, it is prohibitive to compute the path probabilities every cycle (but see [1] for a proposal to do that). So what do we do? The *static DEE tree heuristic*[13] says: assume every branch is predicted with the same prediction accuracy (that is, the average accuracy of the branch predictor as measured over relevant benchmarks); then construct a DEE tree assuming the average accuracy at each prediction point, and fix the tree (somehow) in the hardware. Then let the tree follow or "overlay" the dynamic code execution path, and assign resources, including to spawned paths, as dictated by the presence of the tree over the dynamic code.

See Fig. 2. for an illustration of the static DEE tree in action. Note how the bottom of the tree is the first part of the tree to encounter unexecuted code. Note further that if a branch is encountered, its direction is predicted and execution initially goes down only the predicted path, such as *P*. Then, as execution proceeds, the tree moves down the dynamic path, and eventually path *P* will be at a point in the tree (see C. in Fig. 2.) where both paths of a branch are followed. At this time, the likelihood of executing the not-predicted path of *P* is

This work was supported in part by the U.S. National Science Foundation under Grants Nos. CCR-8910586 and MIP-9708183, and in part by the University of Rhode Island Office of the Provost.

Augustus K. Uht is a Research Professor with the University of Rhode Island, Department of Electrical and Computer Engineering, 4 East Alumni Ave., Kingston, RI 02881 USA (telephone: +1-401-874-5431, e-mail: mailto:uht@ele.uri.edu).

greater than that of the current path just below the bottom of the static tree, *P'*. At this time, and at this time only, the branch associated with *P* spawns the branch's other, not-predicted and unexecuted path, which is then executed. As branches resolve, the resources of non-executed paths, with all of their children, are reallocated and the corresponding execution results are discarded.

This is what DEE is and how it is approximated with the static DEE tree heuristic.

III. What DEE Is Not

We now consider some of the misconceptions about DEE that have occurred over the past several years.

A. DEE means: when you come to a branch, go down both paths.

No, no, no and furthermore no. Remember, this is DISJOINT eager execution. The likelihood is that the program won't go down the not-predicted path for some time after initially encountering the branch and executing down its predicted path.

B. You can use DEE by itself.

True, but not if you want to get the big gains. As demonstrated in our 1995 paper[13], you also have to *minimize control dependencies* as discussed in [14], originally postulated by Tjaden[5] and formalized by Ferrante, et al [2] and this author[6-8].

C. DEE is only applicable to control speculation.

False. It is possible that it could also be applied to data speculation, although to my knowledge this has not yet happened.

D. Using a confidence predictor gives the same results as DEE.

Not very likely. First, confidence predictors aren't that good yet. Second, and more importantly, DEE with minimal control dependencies helps eliminate or at least dramatically reduce the slowdowns due to Amdahl's Law[11]. Only going down one path still leaves the non-zero possibility of a misprediction and its concomitant penalty cycles, and then speedup is substantially reduced due to the serialization associated with those penalties (Amdahl's law). That is, it always helps to do DEE, if it is done in the right way.

E. DEE can only be used with backward branches. OR: only with forward branches.

It makes no difference what kind of branches they are. This only depends on the implementation. Ours realizes DEE for both kinds of branches.

F. DEE dramatically increases the required instruction fetch bandwidth, hence reducing performance.

Not if you do it right: load instructions in the static order and use loaded instructions for both paths of the branches. See: [12, 13].

G. DEE leads to exponential use of resources and is highly wasteful of hardware.

No, this is true of EE, not DEE. DEE uses $O(kL^2)$ hardware resources, where L is the depth of speculation and $k < 1$.

H. DEE wastes resources because it uses them for not-predicted paths; they could be better spent going deeper with simple branch prediction (SP).

No-no. You didn't read the first part of this paper, or I'm still not explaining it well enough. The code below the bottom of the SP path is LESS likely to be executed than the higher-up not-predicted paths. Hence, going deeper with SP WASTES resources. This is true even with typical branch prediction accuracies of 90+%. Also, due to the Amdahl effect, doing DEE always helps.

References

[1] T. F. Chen, "Supporting Highly Speculative Execution via Adaptive Branch Trees," in *Proceedings of the 4th Annual International Symposium on High Performance Computer Architecture*: IEEE, January 1998, pp. 185-194.

[2] J. Ferrante, K. Ottenstein, and J. Warren, "The Program Dependence Graph and its Use in Optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 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*. Gold Coast, Australia: IEEE and ACM, May 1992, pp. 46-57.

[4] E. M. Riseman and C. C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps," *IEEE Transactions on Computers*, vol. C-21, no. 12, pp. 1405-1411, December 1972.

[5] G. S. Tjaden, "Representation and Detection of Concurrency Using Ordering Matrices," PhD thesis, The Johns Hopkins University, 1972.

[6] A. K. Uht, "Hardware Extraction of Low-Level Concurrency from Sequential Instruction Streams," PhD thesis, Electrical and Computer Engineering, Carnegie-Mellon University, Pittsburgh, December 1985.

[7] A. K. Uht, "An Efficient Hardware Algorithm to Extract Concurrency From General-Purpose Code," in *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*, January 1986, pp. 41-50.

[8] A. K. Uht, "A Theory of Reduced and Minimal Procedural Dependencies," *IEEE Transactions on Computers*, vol. 40, no. 6, pp. 681-692, June 1991. Also in the tutorial ``Instruction-Level Parallel Processors", Torng, H.C., and Vassiliadis, S., Eds., IEEE Computer Society Press, 1995, pages 171-182.

[9] A. K. Uht, "Extraction of Massive Instruction Level Parallelism," Department of Electrical Engineering, University of Rhode Island, Kingston, RI 02881, Technical Report 1292-0001, December 1992.

[10] A. K. Uht, "Extraction of Massive Instruction Level Parallelism," *ACM SIGARCH Computer Architecture News*, vol. 21, no. 2 and 3, pp. (June) 5-12, March and June 1993.

[11] A. K. Uht, "Verification of ILP Speedups in the 10's for Disjoint Eager Execution," Department of Electrical and Computer Engineering, University of Rhode Island, Kingston, RI, Technical Report 0697-0001, Rev. A, July 1997. Available via http://www.ele.uri.edu/~uht.

[12] A. K. Uht, A. Khalafi, D. Morano, T. Wenisch, M. d. Alba, and D. Kaeli, "Levo: IPC in the 10's via Resource Flow Computing," *in Work-In-Progress session, PACT-2001; appears in a special issue of IEEE TCCA News*, December 2001.

[13] 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)*. Ann Arbor, MI, November/December 1995, pp. 313-325.

[14] A. K. Uht, V. Sindagi, and S. Somanathan, "Branch Effect Reduction Techniques," *IEEE COMPUTER*, vol. 30, no. 5, pp. 71-81, May 1997.

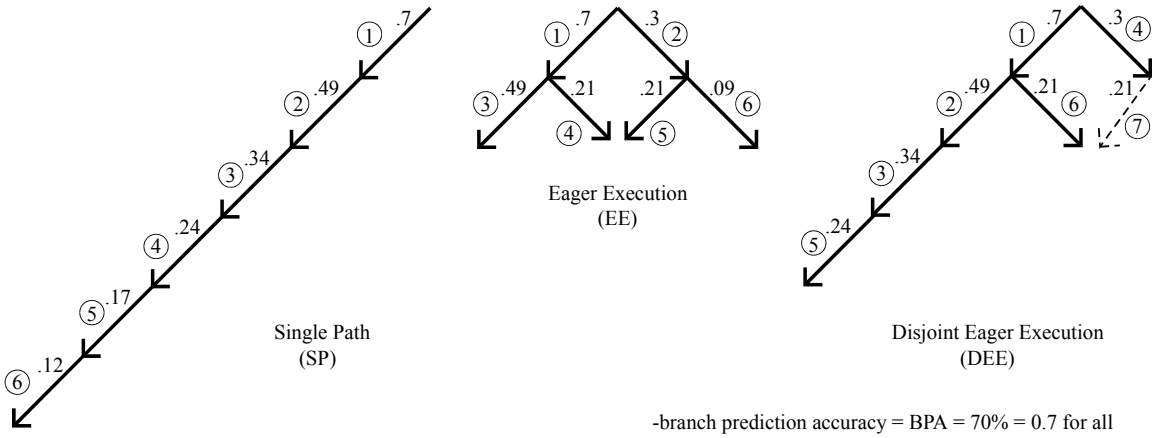
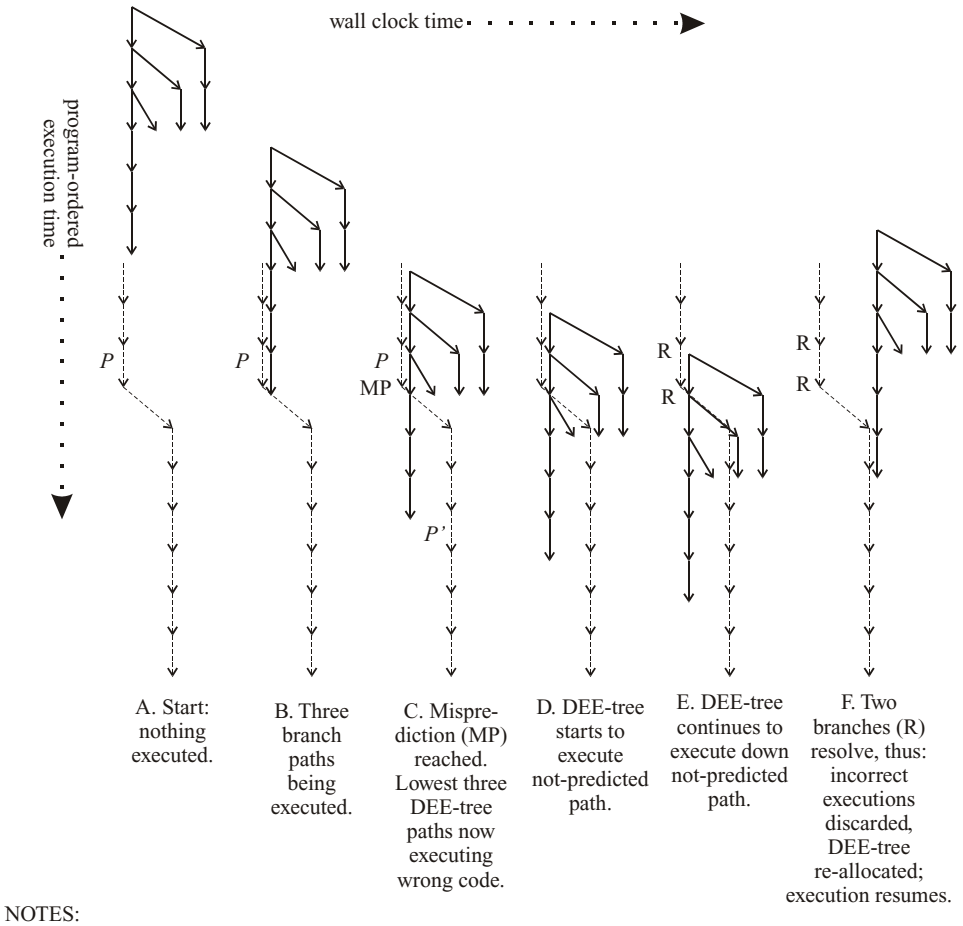


Fig. 1. Three types of control speculation. (Note: this is a modified version of figures published previously by the author.)



NOTES:

1. Each arrow is a branch path.
2. Branch targets are at the arrowheads.
3. Solid arrows comprise the static DEE tree.
4. Dashed arrows comprise the dynamic code execution.
5. Vertical arrows (pointed down) are *predicted* paths.
6. Angled arrows (to the right) are ***not-****predicted* paths.
7. Branch prediction accuracy of about 70% assumed for illustration purposes.
8. Dynamic path has one misprediction.
9. At F., DEE is two branch paths ahead in execution after the misprediction, as compared to how SP would have performed. This does not include pipeline effects.

Fig. 2. Static DEE tree in motion. (*P* and *P'* are referred to in the text.)