Contents

0-8493-0052-5/00/$0.00+$.50

Chapter 6

Multipath Execution

Augustus K. Uht
University of Rhode Island

6.1 Introduction

To branch or not to branch? Why not do both?

These are the essential questions Riseman and Foster asked in their classic paper on performance and cost limits on the execution of branchy code[1]. They had been stymied by the intractability of getting any kind of reasonable performance from branch-intensive code, and sought to find out just what the performance and hardware cost limits were.

Riseman and Foster performed many simulations on typical branch- or control-flow intensive code (SPEC was not then in existence) to study *eager execution*. In this scenario, whenever the program counter (PC) comes to a branch, execution continues speculatively down both sides or *paths* of the branch. The incorrect path's results are discarded when the branch condition is evaluated (the branch is *resolved*). This operation is repeated indefinitely (and recursively), limited only by the resources of the target machine.

Riseman and Foster determined that for unlimited resources, e.g., Processing Elements (PE), *pure* eager execution would completely eliminate branch delays and result in very high performance; in their measurements they found that the harmonic mean *Instruction-Level Parallelism (ILP)* of such eagerly executed general-purpose code was about a factor of 25. That is, an eager execution machine could execute 25 single-cycle instructions in one cycle.

However, the catch-22 of the situation was that the only way to get a sizeable fraction of this high performance was to use excessive resources. This is due to the construction of the execution *branch tree*. Starting from one unresolved branch, the number of active paths grows exponentially with the

0-8493-0052-5/00/$0.00+$.50

depth of branches encountered. The result is the well-known binary tree.

Eager execution's cost was so abysmal that it discouraged further investigations into reducing branch penalties for about a decade. Eager execution is the first known general form of multipath code execution. However, a limited form was implemented for instruction-fetching only, somewhat earlier, in 1967 in the IBM 360/91 [2].

In general, *multipath execution* is the execution of code down both paths of one or more branches, discarding incorrect results when a branch is resolved. There are many dimensions to the differences between the many types of multipath execution. This chapter explores these dimensions.

The remainder of this chapter is organized as follows. In Section 6.2 we further motivate our study of multipath execution, and present its basic concepts and terminology. Section 6.3 presents a taxonomy of the dimensions and types of multipath execution. This section also contains a description of the little multipath theory that exists, in part of subsection 6.3.1.3.1. Some of the known microarchitectural implementations of multipath execution are examined in Section 6.4. Key issues that multipath execution raises are discussed in Section 6.5. Section 6.6 suggests some areas for future multipath research, and presents our own personal multipath prognoses.

6.2 Motivation and Essentials

Most forms of multipath execution use some form of branch prediction, presented earlier in this book. In typical speculative execution a branch's final state, taken or not-taken, is predicted when the branch is encountered. Then execution proceeds *only* down the predicted path. This can be repeated for any number of dynamic branches, constrained only by machine resources. This is called single path or *unipath* execution; see Figure 6.1 tree (a).

Multipath execution differs from unipath execution in that both paths of a branch may be speculatively executed simultaneously. Thus, multipath execution is a branch predictor's way of hedging its bets: if the prediction is wrong, some or all of the other path has already been executed, so the machine can use the not-predicted path's results immediately. Thus, any branch *misprediction penalty* normally occurring in unipath execution is either reduced or eliminated, improving performance. Other branch-penalty reduction methods such as predication, discussed in a prior chapter, can also be combined with multipath execution, further improving performance.

Speculative code can consist of both multipath and unipath sections; that is, execution may proceed down one or both paths of a branch, and this situation may change over time. This may sound odd, but there are sound theoretical and practical reasons for doing this, as we will see.

Confidence estimation [3] of a branch prediction has played a key role in some multipath implementations. For example, a branch might only be eagerly executed (multipath execution) if its predictions are frequently wrong, that is, when there is little confidence in their accuracy; otherwise, the branch only executes down the predicted path (unipath execution).

We now briefly return to our discussion of multipath's performance potential; the latter can generally be summarized as the amount of *Instruction-Level Parallelism (ILP)* present in the typical code to be executed by a target machine. How well a microarchitecture exploits this potential parallelism is the number of *Instructions Per Cycle (IPC)* the microarchitecture actually executes and yields from the potential ILP. There have been numerous ILP limit studies over the years, e.g. [1, 4], demonstrating very high levels of ILP **if** the ill effects of control flow can be alleviated[5]. ILPs in the tens or even hundreds have been found.

A word or two on terminology. A *branch path* consists of the dynamic code between two conditional branches with no intervening conditional branches. This is similar to but different from the more common *basic block.* They are usually about the same size.

When a branch is *forked* execution proceeds down both paths of the branch at the same time. When the two paths begin execution at different times, then the later path is *spawned* from the branch. Lastly, a branch is *split* if either the branch is forked or spawned; it includes both possibilities.

6.3 Taxonomy and Characterization

We now present the various taxonomical dimensions of multipath execution and analyze the characteristics of each dimension's attributes. Sample multipath branch trees are shown in Figure 6.1. The taxonomy is summarized in Table 6.1; it includes example machines and methods and their classifications, as well as cross-references to both corresponding text sections and to representative trees in Figure 6.1. Referring to the Figure, a tree's *Mainline* path is the complete group of predicted paths starting at the root of the tree; in other words, it is the most likely path of execution. A group of paths split off of the Mainline path via one not-predicted branch is call a *Sideline* path.

In general, multipath execution can be applied to all types of branches: conditional, unconditional, calls, returns, indexed, indirect, etc. In practice, only two-way conditional branches are usually split. The other kinds of branches can either be treated as special cases of conditional branches or as unsplitable. N-way branches can sometimes be split by using the compiler to convert them to combinations of two-way conditional branches. An example is the binary code implementation of `switch` statements.

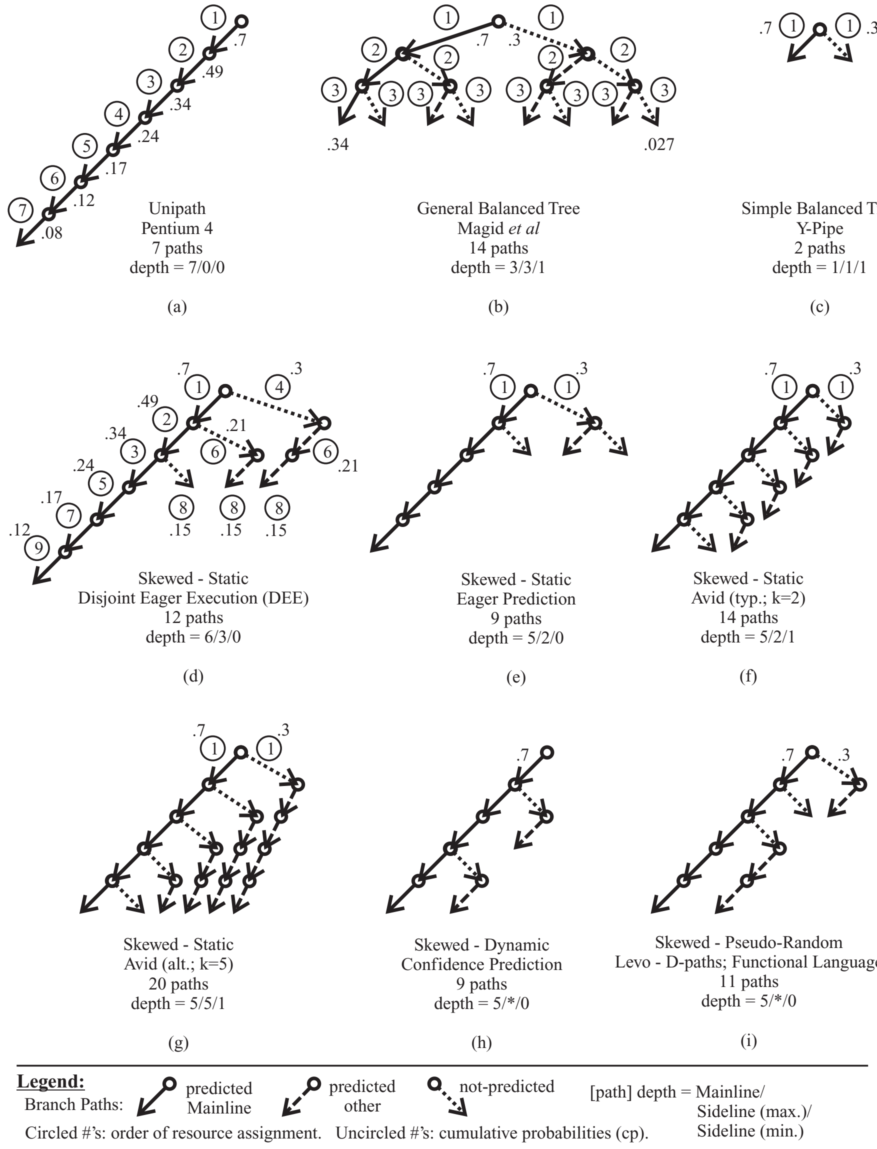


FIGURE 6.1: Some possible branch trees.
Notes: All of the trees are multipath trees except (a).

The text beneath a tree indicates the following:

Line 1 -	Tree name/classification
Line 2 -	Tree implementation example: machine or method
Line 3 -	Number of branch paths in the sample tree, which will vary with a machine's design or operation
Line 4 -	Depth three-tuple - See the right side of the **Legend**. These, too, may vary. ('*': dynamically variable).

TABLE 6.1: Multipath taxonomy and machine/model characteristics.

Section	Dimension/ Characteristic	Riseman *et al* [1], 1972	IBM 360/91 [2], 1967	IFetch-DEE [6], 2003	Y-Pipe [7], 1992	Eag. Prdct. [8], 1994	Sel. Dual [9], 1996	Lim. Dual [10], 1997	TME [11], 1998	PolyPath [12], 1998	PrincePath [13], 1998	Magid *et al* [14], 1981	DEE [15], 1992-95	ABT [16], 1998	Levo [17], 2000-03	Avid/Kin [18], 1997	DCE [19], 2001	CDE [20], 2003	DanSoft [21], 1997	Dyn. CMP [22], 2003	Slip. CMP [23], 2001	MP DEE [24], 1996	Prolog [25], 1995
	Section:	6.1	6.4.1	-	6.4.1.1	-	6.3.1.3.3	6.3.1.3.3	6.3.4.2	6.4.1.2	6.4.1.3	6.4.2.1	6.3.1.3.2	6.4.2.4	6.4.2.3	6.4.2.5	6.4.2.6	-	6.3.7	6.4.3.1	6.4.3.1	6.4.3.2	6.4.4
6.3.1	**Tree geometry**															f						?	
	Figure 6.1 tree:	b	c	h	c	e	h	b	h	h	h	b	d	h	i	g	h	?	h	h	c	d	i
6.3.1.2	*Balanced tree*	√	√		√							√											
6.3.1.3	*Skewed*																					√	
6.3.1.3.1	Theory-Based													√									
6.3.1.3.2	Heuristic:Static					√							√			√		√					
6.3.1.3.3	Heuristic:Dynamic confidence-based			√			√	√	√	√	√								√	√	√		
6.3.1.3.4	Pseudo-Random														√		√						√
6.3.2	**Path ID**	?															?	?				?	?
6.3.2.1	*Binary Tag*																						
6.3.2.1.1	Taken/Not-T.			√			√	√	√	√	√	√		√		√			√				
6.3.2.1.2	Predicted/Not-P.					√							√										
6.3.2.2	*Implied*		√		√										√					√	√		
6.3.2.3	*Other ID*																						
6.3.2.3.1	Thread								√							√							
6.3.2.3.2	Process								√							√							
6.3.3	**Phases of Operation**																						
6.3.3.1	*Pipe Stage*																	?					
	Split/Live/Prune/All																						
	Pre-Fetch															P							
	I-Fetch	S	A	A	S	S	S	S	S	S	S	S		S			S		A	S			
	Decode	L			P	L		?	L	L	L	L		L		A	L			L			
	Execute	P				P	P	P	P	P	L	P	A	P	A	P	P			P	A	A	S
	D-cache										P					P							P
6.3.3.2	*Disjoint*			√				√					√	√	√		?					√	
6.3.4	**Granularity**																						
6.3.4.1	*Fine-Instruction*	√	√	√	√	√	√	√		√	√	√	√	√	√	√	√		√	√	√		
6.3.4.2	*Thread*								√							√							
6.3.4.3	*Coarse-Procedure*																	√				√	
6.3.4.4	*Process*															√		√					
6.3.4.5	*Function*																						√
6.3.5	**Predication**												√		√		√		√			√	
6.3.6	**Data Speculation**														√		√						
6.3.7	**Compiler-Assisted**													√				√		√		√	
	Scheduling				√													?	?	√		√	
	Confidence Hints																	?	√				

6.3.1 Branch Tree Geometry

Perhaps the greatest differentiator of multipath execution schemes is the geometry of the 'live' branch tree. This geometry is formed by the pattern of unresolved conditional branches in the machine. Resolved conditional branches can be present in the tree, but they do not delineate branch paths in the tree; only unresolved conditional branches do that. Many branches can be live at the same time; exactly how many depends on the particular machine implementing the tree, and the code dynamics.

Branch trees fall into one of three broad categories: *unipath*, *balanced multipath* and *skewed multipath.* In the following discussion refer to Figure 6.1 for pictures of various styles of branch trees.

In the Figure an uncircled number next to a branch path is the path's *cumulative probability (cp)* of execution, formed by multiplying the corresponding branch's *Branch Prediction Accuracy (BPA)* by the cp's of other branch paths between it and the root of the tree. For the sake of illustration, a BPA of 70% was used for all of the branches; a typical BPA is much higher, say 90%.

A circled number next to a branch path in the figure is the branch path's priority for resources. A lower number is a higher priority.

6.3.1.1 Unipath Tree

The unipath *Single Path*[15]) tree is both a reference point and is frequently used as part of multipath trees. Unipath execution is just simple branch prediction: when a branch is encountered, its direction is predicted and execution proceeds <u>only</u> down the predicted path. This process is usually repeated.

The net result of the latter effect is to decrease the usefulness of instructions executed in lower paths of the tree (later in time). This is indicated by the rapidly decreasing cumulative probabilities of the branch paths as the depth of prediction or speculation increases.

6.3.1.2 Balanced Multipath Tree

Riseman and Foster's[1] eager execution tree, discussed in Section 6.1, is the canonical example of a balanced branch tree. Every branch encountered is *forked* to its two paths, one corresponding to the taken direction of the branch, and the other path to the not-taken direction of the branch.

The major disadvantages of a balanced tree is its exponential growth in required execution resources as the branch depth increases, and its relatively low performance. The latter is due to the small cumulative probabilities of most of the tree's branch paths. Their results are not likely to be needed.

An advantage of a balanced tree is that since branches are always forked, it does not use a branch predictor. This is true only for balanced trees.

An example machine using a single-level balanced tree is the Y-Pipe machine[7]; see tree (c) in Figure 6.1. Only one branch is forked at a time. An example of a multi-level balanced tree implementation is Magid *et al* 's machine[14].

6.3.1.3 Skewed Multipath Trees

The latter two sections discussed the two extremes of branch tree construction. Both unipath execution and balanced trees have serious problems in high-performance microarchitectures: they both exhibit diminishing returns with increasing available resources.

Skewed trees are the middle ground of multipath trees; they make up the bulk of the trees realistically considered for multipath execution. We classify a skewed tree into one of the following categories: *Theory-Based*, *Heuristic:Static*, *Heuristic:Dynamic* or *Pseudo-Random*.

6.3.1.3.1 Theory-Based A theory-based machine's tree construction is based on some mathematically-based principle, e.g., a tree-construction theorem minimizing resource usage.

An example is the *dynamic DEE tree*. The idea is to assign branch path *resources*, such as Processing Elements (PEs) to the most likely-to-be-executed branch paths first, that is, those paths with the greatest cumulative probabilities. (An example of this construction, but with constant probabilities, is given in the next section). This assignment was proven to be theoretically optimal, giving the best performance for constrained resources[15], where the quantity of resources is directly proportional to the number of branch paths in the tree.

DEE is also shown to be theoretically optimal or close to optimal in related situations in [26]. Further, Gaysinsky *et al* [27] considered block fetching in file systems by looking at the theoretical implications of combined pipelining and caching. Depending on the assumptions made, DEE is shown to be either a theoretically-optimal or at least a near-optimal scheduling algorithm.

Theory-based machines tend to have dynamically-changing branch trees; see Figure 6.1 tree (h). The advantage of these machines is that they may operate beautifully[16], but the empirically-observed disadvantage is that they tend to be costly and complex.

6.3.1.3.2 Heuristic:Static Some possible heuristic:static configurations are shown in Figure 6.1, branch trees (d)-(g). These trees' shapes are based on some heuristic and fixed at machine design-time (static).

The *static DEE tree* (d) is a heuristic of the theoretically-optimal dynamic DEE tree discussed in the last section. The static DEE tree is a dynamic DEE tree having all of the branches' BPAs equal. In operation, at the root branch the predicted path ($0.7 > 0.3$) is assigned the first resources after the branch has been predicted; the cp of path one is the largest of the outstanding paths. Likewise, the next branch is predicted yielding a predicted-path having a cp of $0.7 * 0.7 = 0.49$ and a not-predicted path with a cp of $0.7 * 0.3 = 0.21$. Hence, the 0.49 path is the path with the second largest cp, so it gets the next set of resources, as indicated by its circled '2'. This process continues until we have exhausted the available resources. It becomes interesting after the fourth branch has been predicted, yielding the 0.24 cp predicted path. Now, in this case $0.24 < 0.3$ of the not-predicted path of the first branch, so the latter is assigned resources next, as indicated by the circled '4'. This is the time-'disjoint' resource assignment of disjoint eager execution.

In a real dynamic DEE tree the BPA of one branch is likely to be different from the BPA of another, so instead of the tree having the nice pseudo-symmetric shape shown in (d), a dynamic version of the tree might look something closer to tree (h). Tree (d) is suboptimal, BUT, it has the nice heuristic property that with the statically-fixed cp's the shape of the tree can be fixed at machine design time, and no dynamic cp calculations need be done. These calculations would otherwise entail many costly and slow multiplications whenever the tree is dynamically modified.

The 'statically-fixed shape' attribute is what makes heuristic:static trees so attractive, regardless of their actual shape. Both cost and time are saved, and operation is simplified. The static DEE tree of (d) is nice in particular since it is an approximation of a theoretically-optimal tree, and hence gives good performance[15]. An example of static DEE tree code execution is shown in Figure 6.2. Especially note the lack of a significant branch misprediction penalty.

6.3.1.3.3 Heuristic:Dynamic

In this category a branch tree is formed dynamically by some heuristic; see Figure 6.1 tree (h). This is perhaps the most common form of multipath execution.

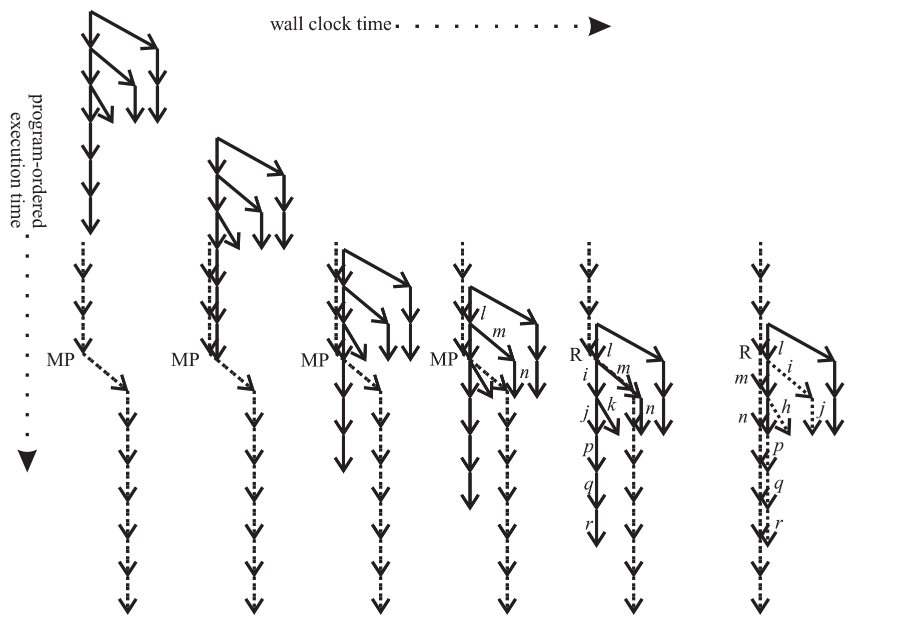
The most common operational example[9, 10, 12] is confidence-based forking. When a machine fetches a branch, not only is it predicted but the quality of the prediction is checked with a confidence estimator; see Section 6.2. If the branch confidence is low, the branch is forked and execution proceeds down both paths. Otherwise, unipath operation is assumed and the branch is executed in the predicted direction.

Either one[9, 10] or multiple[12] forked-branches can be live at the same time. In the former case, tree (h) would only show one forked branch. Single forked-branch machines are simpler and less costly than the multiple forked ones, but give lower performance.

6.3.1.3.4 Pseudo-Random

Pseudo-random trees are a by-product of physical constraints. For example, Levo's[17] branch tree is a physically-based heuristic on top of the static DEE tree heuristic; see Figure 6.1 tree (i) and Section 6.4.2.3. The branch tree then has a less obvious structure.

Since pseudo-random trees are tuned to a specific microarchitecture they may provide high performance with ease of operation. However, it may be hard to analyze such a tree to estimate a paper machine's performance.



A. Start: nothing executed.

B. Three branch paths being executed <u>The DEE-tree starts to spawn execution on select not-predicted paths.</u>

C. Mispre-diction (MP) reached. Lowest three DEE-tree paths now executing wrong code. <u>DEE tree now has complete spawned path results for three branhes.</u>

D. DEE-tree continues to execute predicted path.

E. The mis-prdicted branch MP resolves (R), but there is no stall and <u>little effective mispredition penalty</u> since two of the not-predicted paths, *m* and *n,* have already executed.

F. As a result of the R resolution, <u>the second spawned DEE path becomes the Mainline path</u>, and the dynamic code path 'straightens out,' as indicated by the position changes of the *m* and *n* paths. The dotted lines indicate new- (*h*), or re-execution (*i, j, p, q,* and *r*). *k* is no longer needed.

NOTES:

1. Each arrow is a branch path.
2. Branch targets are at the arrowheads.
3. Solid and dotted 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. A branch prediction accuracy of about 70% assumed for illustration purposes.
8. The dynamic code has one misprediction.
9. At F., DEE is two branch paths ahead in execution after the misprediction, as compared to a unipath execution. (Pipeline effects are not included.)

MORE NOTES:

a. Execution occurs only on the solid and dotted paths, that is, the DEE tree paths.
b. Correct execution occurs only where a solid (DEE tree) path is 'over' a dashed (actual) code execution path.
c. The *i, j, p, q,* and *r* results are corrected as needed, also reducing the effective misprediction penalty.

FIGURE 6.2: Static DEE tree[15] operating on a dynamic code stream.

6.3.2 Branch Path/Instruction ID

6.3.2.1 Binary Path Tag

In multipath execution a tag is used to uniquely identify an instruction's tree branch-path of origin (where it came from). This tag differentiates versions

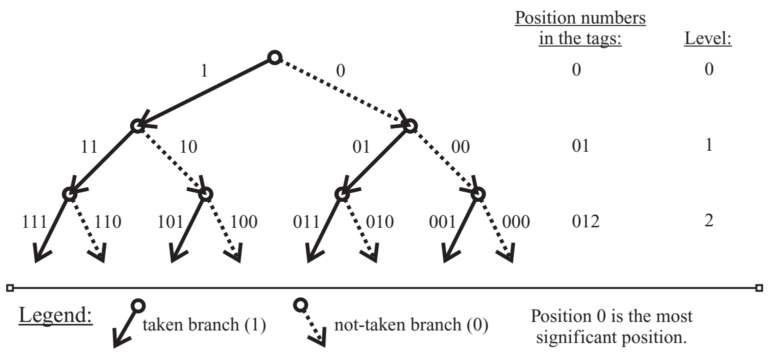


FIGURE 6.3: Typical Branch Path ID numbering scheme.

of the same static instruction and is used for pruning. Figure 6.3 shows the most common tag numbering schemes' foundation. Normally, branch paths are indicated by a *path ID tag*.

The path ID tags use the path labels shown in the figure, one ID position per level. These tags use the fewest number of positions needed to identify a path, i.e., one position for level 0, etc. For ease of use there may be as many positions as levels. A two-bit per position encoding can be used to distinguish among three possible cases: branch taken, branch not-taken and branch invalid ('X'), e.g., [12]. The 'invalid' case occurs if a branch has not yet been predicted or has been resolved. From the figure, an example of this 3-valued tag using all positions would be: '0XX' for the level 0 not-taken path.

For a pruning example, assume that the tree of Figure 6.3 is indicative of the current live branch paths throughout the processor. Now say the second branch at the top of level 1 is mispredicted not-taken. Then the path ID tag corresponding to the mispredicted path tag, '00X' is broadcast throughout the processor. Instructions compare their path ID tags to the broadcast tag, ignoring the 'X' positions. All instructions having matching values for level positions 0 and 1, '00*', know that they are on the mispredicted path or its descendents' paths and are pruned (their results are discarded and their resources re-allocated). Other paths' instructions continue execution.

Whenever the root branch resolves, all of the tags in the machine are shifted left by one position. This frees up ID space on the right of the tags for new branches on the new, lowest level. The elegance and ease of operation of the overall scheme explains why it is so widely used.

6.3.2.1.1 Taken/Not-Taken

Path ID's typically directly indicate branch execution direction. This can help with multipath operation.

6.3.2.1.2 Predicted/Not-Predicted

Considering branches as predicted or not-predicted abstracts away the actual direction of a branch, making such a scheme especially suitable for theoretical analysis and general multipath execution modeling, e.g., Figure 6.1.

6.3.2.2 Implied

Some machines use ID tags to indicate other speculative entities in the machine; thus, actual branch path ID's are *implied.* For example, Levo uses ID tags for speculative groups of instructions whether they share the same branch ancestors or not; see Section 6.4.2.3.

The decision to use explicit or implicit tags is mainly dependent on the details of a specific microarchitecture.

6.3.2.3 Other ID

The basic branch path ID tag can be augmented with tags for other purposes, e.g., to exploit other levels of parallelism. Different threads and processes may also be active in a machine at the same time, share resources, and thus append thread and process ID's to instruction path ID tags to completely differentiate instruction instances.

6.3.2.3.1 Thread The actual construction of a thread ID for a multipath machine is implementation dependent, but it is not hard[11, 18].

6.3.2.3.2 Process Process ID construction is also not hard[18]. It can be as simple as appending the process ID or equivalent from the Translation Look-Aside Buffer (TLB) to the path ID tag. In existing microarchitectures the process ID may already be part of an instruction tag.

6.3.3 Phases of Operation

In this section we mainly consider the three phases of multipath operation, namely when/where in a machine: a branch is *split*, multiple paths are *live* and operate, and path *pruning* is performed after a branch resolves.

6.3.3.1 Pipeline Stage(s)

Table 6.1 shows common pipeline stages and where multipaths Split, are Live, and are Pruned for the example machines. Note that a particular machine may have a multipath phase be active in more than one stage, be active throughout a stage, or be active just at the beginning or end of a stage.

Phase location affects the benefits a machine gets from multipath operation. Intuitively, the longer the multipaths stay speculative, the better the performance but the greater the cost and complexity of operation. However, the precise relationship of performance to phase operation or location is unknown.

In the table the execution stage is assumed to include memory loads, speculative or not. Stores in the execution stage are always speculative.

Speculative memory caches[28] allow multipath execution to go about as far into the memory system as possible, at least with near-in technology. However, this may be undesirable, since every version of a store may be sent to the speculative cache, increasing data bandwidth requirements.

Pruning can be done in a single location or be distributed among the sections of a machine, depending on the details of the machine. Classically-based (superscalar) machines such as PolyPath[12] often perform pruning in one location; this keeps the operation simple although it may create a bottleneck and lengthen cycle time. Non-classical machines such as Kin[18] may distribute pruning throughout the execution stage, avoiding a bottleneck.

6.3.3.2 Disjoint Splitting

The vast majority of multipath machines do not split paths at different times, i.e., they are not disjoint (they don't spawn paths), but always fork paths. Forking is a straightforward way to implement splitting.

However, DEE theory tells us that forking is usually suboptimal, and that spawning the second path at a different time may be preferable; it works for Levo (see Section 6.4.2.3). In the final analysis, whether or not spawning is worth doing likely depends on an individual machine's characteristics.

6.3.4 Granularity

6.3.4.1 Fine-Instruction

The vast majority of past and present multipath schemes operate on individual instructions. This is quite natural, since the basic enabler of multipath execution is the branch instruction, a fine-grain element itself.

6.3.4.2 Thread

Early on, multithreaded-machine researchers realized that unused hardware thread-resources could, with a little extra effort, also be used for speculatively executing a Sideline path in other threads[11], that is, realize multipath execution. A mixture of threads from both the same and different programs can be active simultaneously, so that both throughput and single-program performance improve.

6.3.4.3 Coarse-Procedure

I know of only one study or machine using this level of granularity, that of Hordijk and Corporaal[24]. Briefly, they used a compiler to partition a program into tasks, where each task was a procedure. A procedure invocation corresponds to how we have been viewing branch splitting. They simulated the resulting code on a multiprocessor model, using a DEE construct, and obtained an ILP of about 65. To achieve this the DEE tree only needed to go three levels deep. (Presumably, this means three nested procedure calls.) It was primarily a limit study. We will return to this work in Section 6.4.3.2. It appears that there is a lot of potential here, and that it has implications for multipath execution on Chip Multiprocessors (CMP).

6.3.4.4 Process

Not much work has been done on multipath execution at the process level, to my knowledge. The closest work is the Kin processor[18], targeted to execute instructions from different processes simultaneously, but this is not necessarily executing processes down multiple paths.

6.3.4.5 Function

Functional-language and logic-language machines can benefit from a type of multipath execution called *eager evaluation.* (Unfortunately, over the years 'eager execution' and 'eager evaluation' have often been used interchangeably or with their opposite meanings. We use the current seemingly-accepted definitions; see Section 6.4.4.) A function's evaluation is the basic unit of granularity for such machines. Tubella and Gonzalez[25] show that judicious use of eager evaluation can lead to substantial performance gains.

6.3.5 With Predication

Predication exhibits synergistic gains with DEE[15]. Intuitively, predication should add to the cost and complexity of a machine. However, it could be argued that Levo's implementation of hardware-based predication is actually simpler and cheaper than a more classical approach to typical branch execution. Predication is uncommon in hardware-based multipath machines, while it is common in software-based multipath methods; see [29].

6.3.6 With Data Speculation

Levo uses a data speculation/multipath combination. While little performance gain has been realized to date, the combination has great potential; IPC speedups of a factor of 2-3 may be possible[30].

6.3.7 Compiler-Assisted

Little work has been done here. The Trace Scheduling-2 multipath compiler was proposed for VLIW machines, but the compiler complexity proved to be daunting[29]. The proposed DanSoft machine[21] uses a simple mechanism of statically-generated *confidence hints* from the compiler to help the processor determine whether or not to dynamically fork a branch for dual-path I-Fetch.

6.4 Microarchitecture Examples

The examples are categorized into four sections. *Hardware: Classically-Based* concentrates on multipath machines based on typical superscalar cores. Although they are easy to build, they are not likely to provide substantial performance gains. ("If you want big changes, you have to change a lot."- after Prof. Donald Hartill.) The *Hardware: Non Classically-Based* section looks at many multipath machines with radically new processor cores. In the *Multiprocessors* section we examine several approaches to multipath multiprocessors. We end with something completely different: *Functional or Logic Language Machines.*

6.4.1 Hardware: Classically-Based

The IBM 360/91 [2] is an early example of multipath execution; it kept dual paths for instruction fetch. While multipath execution restricted to instruction fetching or pre-fetching continues to be used in some current microarchitectures, this is done usually to keep such classical superscalar pipelines fed with instructions; thus, no big performance wins are likely.

However, an exception to this is the Y-Pipe machine[7]. We also discuss two high-end multipath machines based on superscalar cores.

6.4.1.1 Y-Pipe

The Y-Pipe machine[7] is one of the earliest and simplest of modern multipath machines. Y-Pipe assumes a simple five-stage pipeline (IF-ID-EX-M-WB), and the ability to fetch two instructions in one cycle. Y-Pipe achieves 0 cycle (no) branch penalty without a branch predictor as follows.

The code is scheduled so that every conditional branch will be dynamically immediately preceded by its associated compare instruction, and not another conditional branch. At run time, every conditional branch is forked at instruction-fetch. Its dual paths stay live until the end of the decode stage, when the branch's compare instruction has just executed, the branch is resolved, and the wrong path is pruned. Thus, there is no branch penalty! And without a branch predictor!

Y-Pipe may be ideal for embedded applications with its low cost, likely low-power, relatively high performance and deterministic branch penalty (0 cycles). The latter is a great attraction for embedded systems' real-time programming. The dual instruction-fetch per cycle requirement is less of an issue in an embedded environment.

6.4.1.2 Polypath

The advanced Polypath machine is an implementation of *Selective Eager Execution (SEE)* [12]. Polypath can simultaneously follow many more than two paths at one time. Branches can resolve out-of-order. Polypath's front-end uses a JRS confidence estimator[3] to make the fork/no-fork decision. While predictable branches stay unipath, unpredictable branches (low confidence) are forked and executed simultaneously.

Polypath uses a *context tag* to show instruction's branch history; this tag is a version of the path ID labeling scheme of Section 6.3.2.1. Every instruction proceeds through the execution stage with its tag. Thus, typical register operation structures can be used, suitably augmented to store context tags.

Functional units send forked branch resolution information to the instruction window on *branch resolution buses*, one bus for each resolution desired per cycle. If a mispredicted branch's path tag is received by the window, the mispredicted path and its descendents are pruned and their results squashed. Mispredicted low-confidence multipath branches exhibit no branch penalty since they are forked. Mispredicted high-confidence unipath branches have the same (high) penalty as a pure unipath processor.

The added hardware is somewhat costly and complex, though not a large fraction of the overall cost of the core. The added performance is modest. In simulations fetch bandwidth was kept the same for both uni- and multipath machines, perfect caches (no misses) and perfect memory disambiguation were assumed, and the processor was 8-way issue, with many functional units. The overall performance improvement was about 14%, from 3.85 to 4.4 IPC, from a unipath to a multipath model, respectively. The low accuracy common to confidence estimators likely contributed to wasted resources or missed forking opportunities and hence low performance gain. I-Fetch memory bandwidth requirements increased substantially.

6.4.1.3 PrincePath

PrincePath[13] [our name, for "Princeton multiPath"] is similar to the Polypath machine. PrincePath also uses confidence estimation[3] to select which paths to fork at I-Fetch time, and allows multiple forked paths to be live at any given time. The equivalent to context tags is also used. As is typical equal priority is given to the forked paths, that is, disjoint splitting is not used; see Section 6.3.3.2.

Unlike the Polypath machine, PrincePath uses a global ARB (Address Resolution Buffer) [28] in front of the memory system to hold speculative stores suitably tagged. Therefore, PrincePath is able to keep multiple paths alive for probably the longest time realistically possible. When pruning occurs if the resolving branch is the earliest unresolved branch in the machine, the wrong path store versions in the ARB are discarded and the non-speculative store is committed to the remainder of the memory system.

The simulations' benchmarks were similar to those used in the PolyPath simulations. The PrincePath results indicated that branches with a misprediction rate greater than 35% should be forked, with lower rate branches unsplit. PrincePath achieved similar speedups to PolyPath: about 15.5% over the baseline case. PrincePath is similarly likely constrained by classical superscalar assumptions. Speculative caching does not seem to me to help; this may be due to short-lived speculative paths. PrincePath did not experience a substantial increase in fetch bandwidth requirements.

6.4.2 Hardware: Non Classically-Based

6.4.2.1 Magid *et al*

Little was published on this machine[14]. We know that it is a proposed implementation of the original Riseman and Foster eager execution model[1], using a multi-level balanced tree. Since the pure eager execution model's required resources are so large for so little gain, the Magid machine would not be likely to realize a high IPC. It may have been the first machine to use the elegant binary path ID scheme described earlier in Section 6.3.2.1.

6.4.2.2 Condel-2-Based - Early DEE

The Condel-2 -based static DEE tree implementation[15] was an early version of the Levo machine. Since this early version was costly, not scalable, cumbersome to operate, and has been superseded by the current Levo, we will not dwell on it. The machine was never simulated, although the static DEE tree concept itself was[15]. DEE with MCD gave an ILP of about 30.

6.4.2.3 Levo - Current DEE

It is my experience that it is very hard to implement a realistic multipath processor that follows the DEE theory closely. Thus, the Levo[17] implementation of multipath execution is a multipath heuristic on top of another multipath heuristic; to wit, the Levo microarchitecture uses a modified version of the standard static DEE tree, in particular a version that meshes well with the rest of the Levo microarchitecture. Even with this double DEE approximation, Levo benefits greatly from multipath execution. Levo is described in much greater detail in a later chapter of this book, so we will only describe its broad characteristics here.

Figure 6.1 tree (i) is an approximation of Levo's branch tree. It is pseudo-random because the Sideline path lengths are not ordered by their size, as is the case with the static DEE tree, tree (d). However, unlike tree (h), the unresolved split branches' predicted branch paths are all normally adjacent.

Unlike Levo, in most multipath machines a misprediction of a split branch causes all of the instructions following the Sideline path's instructions to be flushed, and their results squashed. In Levo the Mainline instructions after the mispredicted branch's spawned path are not flushed nor are their results discarded. Instead the spawned path broadcasts its result data forward to cause the re-execution of only directly and indirectly dependent instructions.

Levo uses implied path IDs; sections of instructions have an ID corresponding to either a part of the Mainline path or an entire Sideline path. Typically, only 8 partial-Mainline and 8 Sideline-path are present, so the implied ID tags are just a few bits long. Levo gave about 5 IPC with detailed simulations using realistic assumptions, with the potential to realize IPCs in the 10's.

6.4.2.4 Adaptive Branch Trees

The ABT machine proposal[16] is an implementation of the optimal dynamic DEE tree. The machine dynamically computes and updates the cumulative probabilities of all of the branch paths in its window, assigning execution resources to the most likely-to-be-used paths. The ABT machine does not use reduced control dependencies ala either the Minimal Control Dependencies of [15], or typical software or hardware-base predication.

ABT realization is difficult because of both the massive amount of computation necessary to keep the cumulative probabilities updated, and the difficulty of the calculations needed to dynamically sort the branch paths by their cumulative probabilities (to determine which path gets resources).

However, the ABT simulations partially verified the initial DEE work[15]. ABT with an optimal tree performed very well and better than the corresponding suboptimal static DEE tree. Also, ABT didn't perform substantially better, indicating the usefulness of the static DEE tree heuristic.

6.4.2.5 Kin and Avid Execution

Kol devised and studied a new form of multipath execution called *Avid Execution*, to be implemented in a novel asynchronous machine called 'Kin'[18]. A typical Avid branch tree model is shown in Figure 6.1 tree (f). The two key features of this tree are: every conditional branch is forked, and: the length of each forked path is the same for every branch (with the caveat that forked paths cannot exist past the end of the Mainline path).

Kol defines a parameter k equal to the length in branch paths of a forked path. In tree (f) $k = 2$. Tree (g) is a special Avid case in which the forked path depth is equal to the Mainline path length; in this case $k = 5$. We will concentrate on the most useful configuration, tree (f). Kin's multipath operation is similar to others previously mentioned, except that the multipath operation phases are organized a bit differently and are spread throughout most of the machine; see the 'Avid/Kin' column in Table 6.1.

Limit-type simulations on some of the SPECint95 benchmarks including `gcc` indicated speedups of about a factor of two for branch trees with $k = 1$ over $k = 0$. A $k = 2$ tree performed about the same as a $k = 1$ tree. These simulations used a branch predictor with about a 90% BPA. Also interesting, I-Fetch requirements actually DECREASED for trees with $k = 0$ to $k = 1$.

The results are quite remarkable: little multipath execution needs to be done for big gains. The DEE simulations gave similar results, and also showed that branches tended to resolve at or close to the root of the DEE tree. Combining the Avid and DEE results seems to indicate that less multipath execution than either Avid or DEE used is needed for good performance. Kol also suggests that a confidence estimator could be used to dynamically vary the forked path length to further improve performance and resource utilization; this was not simulated. Kol also states that the length could even go to zero; I must disagree with this; at some point, every conditional branch should be split. We'll come back to these results and analyses in Section 6.5.4.

6.4.2.6 Dynamic Conditional Execution (DCE)

In general-purpose code simulations dos Santos *et al* [19] found that most mispredictions are in *short* branches (little code between the branch and its target) and most branches are short branches. They argued that most short branches can fit in a reasonably sized cache line. So when a proposed machine fetches a line containing a short branch, the machine is also effectively fetching down both paths of the branch automatically. Multipath execution of these paths ensues. This is multipath execution with just about no extra I-Fetch bandwidth requirements (maybe 0), and on the cheap. Some SPECint simulations, including `gcc`, indicate that about a factor of two performance gain over a standard superscalar machine may be possible. The proposed machine is evolving, and the gains may increase. This idea is clever and most promising.

6.4.3 Multiprocessors

6.4.3.1 Chip MultiProcessors (CMP)

Chidester *et al* [22] proposed a fine-grain multipath CMP containing eight specialized processors. Compiler support is needed to insert special instructions for forking and other information into the machine's programs. A mapping approach that worked well used one processor for the Mainline path and the other processors for Sideline paths. Most of the performance gain was realized with a small degree of forking. Some of the SPECint benchmarks, including `gcc`, were simulated on the CMP; the average performance gain was 12.7%. The researchers found that the main inhibitor of performance was the on-chip interconnect's effect on the transmission latency of intermediate results from one processor to another.

In another approach, Sundaramoorthy *et al* [23] proposed a CMP version of confidence-estimation based dual-path execution. Two processors execute the same program, just a bit differently: each processor executes a different path of a low-confidence branch. When the branch resolves, only control-flow and data-flow dependent state are transferred from the correct path processor to the incorrect path processor. The transfer enables the processor on the incorrect path to correct its results and to catch up to the correct path's processor, ideally before the latter reaches the next low-confidence branch. While the design is elegant, the negative effect of the interconnect latency resulted in a small performance gain, about 5% on `gcc`.

6.4.3.2 Coarse-Grain

We now return to Hordijk and Corporaal[24]'s study, first examined in Section 6.3.4.3. To briefly review: they simulated a coarse-grain multiprocessor consisting of individual ILP processors. Procedure calls were the basic unit of granularity. The compiler did the partitioning. DEE execution was used.

Minimal Control Dependencies (MCD) (essentially predication) were used, but it did not provide much benefit. This is in contrast to our fine-granularity study[15] in which both MCD and DEE were needed to get the best performance. This raises interesting questions for future research.

In the study, the basic assumption was that resources were infinite. (However, they found that the multipath level only had to be three procedures deep to get most of the performance gain; so resources were in some sense limited.) Data dependencies were honored, only RAW for register operands, and RAW, WAR and WAW for memory operands. Perfect disambiguation was assumed. A one-cycle communications latency was assumed; while this is a bit optimistic for a multi-chip multiprocessor, it is certainly in the realm of the possible for a CMP. Many SPECint benchmarks were simulated.

An oracle gave an average ILP of 66.7. The average ILP of the DEE approach without MCD was 63.0! `cpp` also did well with an ILP of 26.2.

In summary, this is a limit study with important significance for other multipath techniques, especially CMP. We have seen that most CMP approaches are aimed at the fine-grain or instruction level. This places great constraints on multipath CMPs' design and operation. The Hordijk results just presented argue strongly for coarse-grain multipath CMPs. This is intuitive: one wants as little communications as possible between the processors of a CMP at opposite ends of a chip. Much more work needs to be done to further characterize multipath CMP operation, especially at the coarse-grain level.

6.4.4 Functional or Logic Language Machines

Such machines typically execute programs using *lazy evaluation*, in which code is only executed non-speculatively, and only when its results are needed. This is a very safe approach, but it severely limits performance.

The alternative is eager evaluation, in which as many paths as possible execute code simultaneously, similarly to unlimited multipath execution on an imperative machine. The major problem with eager evaluation is that in operation the machine can completely run out of memory, then be unable to execute down the lazy (Mainline) path, and deadlock occurs.

Tubella and Gonzalez[25] proposed the parallel execution of Prolog via a compromise between lazy and eager evaluation. Breadth-first execution is done until resources are exhausted, then the mode of execution goes back to depth-first; when resources again become available, eager evaluation resumes. The tree shape can be dynamically changed for different program phases' execution. No compiler or interpreter modifications are needed. An SPMD machine model was used in the simulations. Speedups of x4-x63 over sequential execution were demonstrated for four Prolog benchmarks. The best results were for more non-deterministic programs (analogous to imperative programs with unpredictable branches).

6.5 Issues

6.5.1 Branch Prediction

Branch predictors are typically initially modeled and rated by themselves, with the predictor-state updated immediately after every prediction. In such a scenario predictors can give very high accuracies, say 95%. However, in a real machine predictions occur early in the pipeline, while branch results are not known until late in the pipeline. The resulting predictor-update delay can be 10-20 cycles or more for a Pentium 4 class processor.

For example, the PrincePath machine (Section 6.4.1.3) used a nominally accurate compound predictor having two two-level predictors, one local and one global. With the inherent update delays, it only gave an effective average BPA of 86.7% on the SPECint `go`, `gcc`, `compress`, `li` and `perl` benchmarks.

Since pipelines are getting deeper, effective BPAs will continue to decline, increasing the benefits of multipath execution.

6.5.2 Confidence Estimation

Delayed updating likely affects confidence estimators similarly. However, the situation is exacerbated since the confidence estimators start off providing rather low accuracy estimations. Therefore it will be even less advisable in the future to completely depend on confidence information for splitting decisions.

6.5.3 Pipeline Depth

Deeper pipelines not only have linearly increasing branch penalties with depth, but the negative performance effects of such increased penalties grow superlinearly with greater depth, due to the equivalent of Amdahl's Law. Multipath execution can greatly reduce the performance loss where unipath execution does not; see the next section.

6.5.4 Implications of Amdahl's Law - ILP Version

Amdahl's Law is:

$$S = \frac{1}{f_S + \frac{f_P}{P}}$$

Where: S is the net overall speedup, P is the speedup of the parallel code region, and f_S and f_P are the fractions of the Serial and Parallel code regions, respectively. As is well known, this expression tells us that a relatively small sequential region or f_S can result in a big drop in performance. Something similar happens in pipelined machines, where the effective branch penalty corresponds to Amdahl's sequential code. (These are, of course, just rough analytical models.)

Reformulating Amdahl's Law for ILP pipelined machines, we get:

$$R = \frac{1}{c_{MIS} + \frac{c_P}{P}}$$

Where: R is the net execution rate in IPC, c_{MIS} is the extra cycle fraction due to the misprediction penalty, c_P is the cycles per instruction due to normal non-mispredicted execution, and P is the Oracle parallelism, i.e., IPC with no mispredictions. We assume that P includes any parallelism bottlenecks in the machine, e.g., limited issue-width. Expanding:

$$R = \frac{1}{\frac{t_{MIS} * (1 - BPA)}{BPL} + \frac{1}{P}}$$

t_{MIS} is the *effective* misprediction penalty in cycles, BPA is the Branch Prediction Accuracy and BPL is the Branch Path Length in instructions.

Only one or two mispredictions can kill the overall performance of a machine. A misprediction gives worse than sequential execution for a period of time, it gives NO execution for that time. (Note that we are ignoring predication and MCD in this analysis.)

Seeing what the R expression means, take: $BPL = 6$ instructions; use PrincePath's $BPA = 0.867$; take: $P = 5$ IPC; and, assuming unipath execution, $t_{MIS} = 20$ cycles, similar to a common Pentium 4. Then: $R = 1.55$ IPC. This is just 31% of the peak performance.

Now, assume a PolyPath multipath implementation; not every branch is forked but the misprediction rate is still reduced by a factor of two, i.e., $(1 - BPA)$ is halved. Then: $R = 2.37$ IPC. This is much better, getting 47% of the peak performance, an improvement of 53% over the unipath case, but still not where we'd like to be.

Now consider Avid multipath execution with its fork depth $k = 3$ (see Section 6.4.2.5). Then the effective branch penalty is: $t_{MIS} = 20 - (k * BPL) = 2$ cycles. Second-order effects such as double mispredictions do not change this formulation significantly. Then: $R = 4.09$ IPC. This is 82% of the peak performance, and an improvement of 164% over the unipath case. Multipath execution <u>can</u> give great gains!

It is likely that all conditional branches should be forced to split. This is done in Avid, eager prediction and DEE, corresponding to the static trees (d)-(g) of Figure 6.1. Confidence estimation is not likely to get the big gains.

6.5.5 Memory Bandwidth Requirements

6.5.5.1 Instruction Fetch

Different studies give conflicting indications of the multipath effects on I-Fetch bandwidth. I believe that extra bandwidth is NOT necessarily needed.

Examining the static DEE tree model, we can readily see the possibilities. The same instructions fetched for the Mainline path can be used for the Sideline paths; no bandwidth increase here. Further, the Mainline path is much shorter than the unipath of a conventional machine, so fewer instructions need to be fetched there. Hence, multipath execution can actually lead to a decreased I-Fetch bandwidth requirement!

6.5.5.2 Data Accesses

The overall effect of multipath execution on data bandwidth is not clear. It has never been measured, to our knowledge. Intuitively multipath execution will increase the necessary load bandwidth. However, explicit or implicit pre-fetching, or load re-use, may even reduce load bandwidth requirements.

For in-order commit multipath machines store bandwidth should stay the same or even decrease, since stores may combine before an actual commit. For ooo commit, such as with speculative caches, store bandwidth may well increase, since many speculative stores will be sent to the speculative cache.

Therefore, multipath machines need not require greater memory bandwidth than classical unipath machines, and may even need less.

6.6 Status, Summary and Predictions

High-performance microprocessors are running out of options for improved performance. There is little headroom for further substantive increases in clock frequency. Increasing cache sizes is giving diminishing returns.

There is only one option: exploit more ILP. However, unipath methods cannot do this, e.g., branch prediction is becoming a dead-end for greater effective improved accuracies. We need multipath execution.

Fortunately, there is renewed interest in multipath execution[20]. Also, existing research efforts are continuing at both our lab[17] and at others.

There are many promising areas of future multipath research, such as: combining data speculation with multipath execution, deriving other optimal forms of multipath execution, investigating new and existing multipath heuristics, and studying coarse(r)-grain CMP machines. Multipath techniques may also be applicable both to other areas of computer science and engineering, such as database engines, and even possibly even to other disciplines.

Multipath execution has come a long way since the IBM 360/91, and it's going to go farther. It will be standard in high-performance architectures.

My apologies to those whose work could not be included due to space limitations, or whose work I missed or misinterpreted; I did the best I could in more than the time allotted.

References

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

[2] Anderson, D. W., Sparacio, F. J., and Tomasulo, R. M. The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling. *IBM Journal of Research and Development*, 11(1):8–24, January 1967.

[3] Jacobsen, E., Rotenberg, E., and Smith, J. E. Assigning Confidence to Conditional Branch Prediction. In *Proceedings of the 29th International Symposium on Microarchitecture (MICRO-29)*, pages 142–152. IEEE and ACM, December 1996.

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

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

[6] Valia, S. K., Koblenski, S. A., and Janes, D. R. Multipath Execution Using Dynamic Relative Confidence. Technical report, Department of Computer Sciences, University of Wisconsin, Madison, Madison, WI, USA, Spring 2003. Instructor: Prof. David Wood.

[7] Knieser, M. J. and Papachristou, C. A. Y-Pipe: A Conditional Branching Scheme Without Pipeline Delays. In *Proc. of the 25th International Conference on Microarchitecture*, page 125128. ACM/IEEE, 1992.

[8] Messaris, S. A. Combining Speculative with Eager Execution to Reduce the Branch Penalty on Instruction-Level Parallel Architectures. Mas-

ter's thesis, Department of Computer Science, Michigan Technological University, Houghton, Michigan, USA, 1994.

[9] Heil, T. H. and Smith, J. E. Selective Dual Path Execution. Technical report, Department of Electrical and Computer Engineering, University of Wisconsin-Madison, Madison, WI, USA, November 8, 1996.

[10] Tyson, G., Lick, K., and Farrens, M. Limited Dual Path Execution. Technical Report CSE-TR-346-97, Dept. of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, USA, 1997.

[11] Wallace, S., Calder, B., and Tullsen, D. M. Threaded Multiple Path Execution. In *25th Annual International Symposium on Computer Architecture*, pages 238–249. ACM, June 1998.

[12] Klauser, A., Paithankar, A., and Grunwald, D. Selective Eager Execution on the PolyPath Architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture, Barcelona, Spain*, pages 250–259, June 27 - July 01, 1998.

[13] Ahuja, P. S., Skadron, K., Martonosi, M., and Clark, D. W. Multipath Execution: Opportunities and Limits. In *Proceedings of the 12th International Conference on Supercomputing*. ACM, July 1998.

[14] Magid, N., Tjaden, G., and Messinger, H. Exploitation of Concurrency by Virtual Elimination of Branch Instructions. In *Proc. of the 1981 Intl. Conference on Parallel Processing*, pages 164–165, Aug. 1981.

[15] Uht, A. K. and Sindagi, V. Disjoint Eager Execution: An Optimal Form of Speculative Execution. In *Proc. of the 28th International Symposium on Microarchitecture, Ann Arbor, MI*, pages 313–325, Nov./Dec. 1995.

[16] Chen, T. F. Supporting Highly Speculative Execution via Adaptive Branch Trees. In *Proc. of the 4th International Symposium on High Performance Computer Architecture*, pages 185–194. IEEE, Jan. 1998.

[17] Uht, A. K., Morano, D., Khalafi, A., and Kaeli, D. R. Levo - A Scalable Processor With High IPC. *The Journal of Instruction-Level Parallelism*, 5, August 2003.

[18] Kol, R. *Self-Timed Asynchronous Architecture of an Advanced General Purpose Microprocessor.* PhD thesis, Dept. of Electrical Engineering, The Technion - Israel Institute of Technology, Haifa, Israel, Nov. 1997.

[19] dos Santos, R., Navaux, P., and Nemirovsky, M. DCE: The Dynamic Conditional Execution in a Multipath Contol Independent Architecture. Technical Report UCSC-CRL-01-08, University of California, Santa Cruz, Santa Cruz, CA, USA, June 2001.

[20] Acosta, C., Vajapeyam, S., Ramrez, A., and Valero, M. CDE: A Compiler-driven, Dependence-centric, Eager-executing Architecture for

the Billion Transistor Era. In *Proceedings of the Workshop on Complexity Effective Design (WCED'03), at the International Symposium on Computer Architecture (ISCA'03)*, July 2003.

[21] Gwennap, L. DanSoft Develops VLIW Design. *Microprocessor Report*, 11(2), February 17, 1997.

[22] Chidester, M. C., George, A. D., and Radlinski, M. A. Multiple-Path Execution for Chip Multiprocessors. *Elsevier Journal of Systems Architecture: the EUROMICRO Journal*, 49(1-2):33–52, July 2003.

[23] Sundaramoorthy, K., Purser, Z., and Rotenberg, E. Multipath Execution on Chip Multiprocessors Enabled by Redundant Threads. Technical Report CESR-TR-01-2, Center for Embedded System Research (CESR), Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC, USA, October 23, 2001.

[24] Hordijk, J. and Corporaal, H. The Impact of Data Communication and Control Synchronization on Coarse-Grain Task Parallelism. In *Proceedings of the 2nd Annual Conference of ASCI, Lommel, Advanced School for Computing and Imaging, Delft, The Netherlands*, January 1996.

[25] Tubella, J. and Gonzalez, A. Exploiting Path Parallelism in Logic Programming. In *Proceedings of the Euromicro Workshop on Parallel and Distributed Processing*, pages 164–173, January 25-27, 1995.

[26] Raghavan, P., Shachnai, H., and Yaniv, M. Dynamic Schemes for Speculative Execution of Code. In *Proc. of the Sixth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), Montreal, Canada*, July 1998.

[27] Gaysinsky, A., Itai, A., and Shachnai, H. Strongly Competitive Algorithms for Caching with Pipelined Prefetching. In *Proc. of the 9th Annual European Symposium on Algorithms, Aarhus*, pages 49–61. Springer-Verlag Lecture Notes In Computer Science, August 2001.

[28] Franklin, M. and Sohi, G. S. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[29] Fisher, J. A. "Global Code Generation for Instruction-Level Parallelism: Trace Scheduling-2". Technical Report HPL-93-43, Computer Research Center, Hewlett-Packard Laboratories, Palo Alto, CA, USA, June 1993.

[30] Gonzalez, J. and Gonzalez, A. Limits on Instruction-Level Parallelism with Data Speculation. Technical Report UPC-DAC-1997-34, Department Architectura de Computadores, Universitat Polytechnica Catalan, Barcelona, Spain, 1997.