

Avid Execution and Instruction Pruning in the Asynchronous Processor *Kin*

Rakefet Kol, Ran Ginosar¹ and Hisham Shafi²

VLSI Systems Research Center
Electrical Engineering Department
Technion—Israel Institute of Technology
Haifa 32000, Israel
rakefet@tx.technion.ac.il

¹ On Sabbatical leave at Intel, Hillsboro, OR

² Intel Corp., Haifa, Israel

Avid Execution and Instruction Pruning in the Asynchronous Processor *Kin*¹

Abstract

Avid execution prefetches and speculatively executes not only the predicted branches of a program but also a certain part of the non-predicted branches, in anticipation of misprediction. Once a misprediction occurs, the processor may immediately turn to one of the secondary paths and continue executing without interruption. The stale instructions are removed quietly without flushing the processor, by means of a *pruning* process. *Avid execution* employs computing resources which cannot be used for the mainline code, due to limited instruction level parallelism. It is intended for very large high performance processors, and is best used in asynchronous processors. The paper provides a full definition and mathematical analysis of *Avid execution*. A behavioral model of the *Kin* asynchronous processor has been simulated executing SpecInt95, and performance improvements of close to 100% are predicted thanks to *Avid execution*.

1. Introduction

As part of our research of whether asynchronous logic design is applicable to high performance general purpose processors, we have developed a novel method of multi-path speculative execution. The method, named *Avid Execution*, is applicable to synchronous processors as well. *Avid execution* addresses the question of how we can exploit massive resources, which will be made available by future technology, to enhance processor performance. The semiconductor industry predict that by the year 2012 we can expect one billion transistors per chip, operating at close to 10 Ghz [SIA97, Wei96]. We discuss the issue of how computer architecture may be modified in order to make it scalable for such vast resources.

Performance of present processors is limited by a number of factors, including true and false dependencies, limits to inherent instruction level parallelism in serial code, and pipeline stalls due to misprediction of branches. To achieve high performance, processors must run faster but also execute and successfully complete many instructions in parallel. Although many parallel execution units may be made available, data and control dependencies limit instruction level parallelism. To exploit such parallelism, the processor must search over a large window for instructions that can be executed. That window typically extends beyond multiple branches, by means of branch lookahead strategies. Those strategies are based on branch prediction algorithms, and on speculative execution beyond the predicted branches. This paper is not concerned with the branch prediction algorithm itself, but rather with the question of how it is used and which instructions are speculatively executed.

Future technology will most likely lead to asynchronous processors, due to the difficulty of maintaining very fast clocks and synchronized data transfer over very large chips. In fact, such very large processors may resemble large distributed networks, rather than tightly synchronized pipelines. Hence, we feel that aggressive architectures are better investigated in the context of

¹*Kin* was the God of Time of the Maya.

asynchronous processors. A large number of asynchronous processors have been previously designed [MBL+89, FDG+93, Pav94, Bru93, RB96, DGY93, NUK+94, SSM94, Dea92, Wol92, End95, Mar97]. Most of them have rather simple and straightforward architectures. None of them supports out-of-order execution, nor considers performance enhancement by branch prediction. All are targeted at current technology, and are not scalable to take advantage of the growing amount of resources promised by future technology.

Kin architecture comprises multiple fast self-timed units, interconnected over asynchronous channels, using handshake communication protocols. The asynchronous microarchitecture is described at a high level and allows flexible and robust implementation. Although *Kin* is designed as an asynchronous machine at the its top level, its individual modules may be implemented according to various timing disciplines (synchronous, asynchronous, or anything in between).

Avid execution prefetches and processes not only the predicted branches of the program but also a certain amount of the non-predicted branches, in anticipation of misprediction. Once a misprediction occurs, the processor may immediately turn to one of the secondary paths and continue executing without interruption. The stale instructions are removed quietly without flushing the processor, by means of a *pruning* process. Avid execution employs computing resources which cannot be used for the mainline code, due to inherent limitations on parallel execution. The paper provides a definition and mathematical analysis of Avid execution, and describes the encouraging results of SpecInt95 performance simulation of the model.

The novel architecture of *Kin* is described in Sect. 2. Avid execution is presented and analyzed in Sect. 3, and Sect. 4 explains instruction pruning. Modeling *Kin* and its performance simulation (using the SpecInt95 benchmark) are described in Sect 5.

2. *Kin* Architecture

2.1 General Description

Kin is a general purpose high performance asynchronous microprocessor that supports out-of-order and deep speculative (*Avid*) execution. It exploits massive parallelism and redundancy in order to execute hundreds or thousands of instructions simultaneously.

Kin comprises a distributed network of asynchronously interconnected modules, without any central control. The architecture is asynchronous at the top level, but modules can be implemented according to various timing disciplines. Each module operates at its own speed, and communicates with other modules over asynchronous FIFO channels. On average all modules are balanced, but asynchronous interconnects permit flexible work loads.

Instructions flow through *Kin* as self-identified packets carrying their own identity tags and all needed information, as in a *data flow* computer. They may leave some traces around such as instruction entries in the reorder buffer, but these eventually reunite with the instructions. Each module receives instruction packets, executes them at its own local rate, and forwards them to their next stops.

Kin (Fig. 1) combines known features (multiple execution units, out-of-order execution, and register renaming) with some novel ones (*Avid* Execution, Dynamic Instance Tagging, unified Multi-Execution, and Pruning). Multiple instructions are executed concurrently and out-of-order over multiple execution units. To preserve the serial nature of the code, instructions are committed (completed) in their original serial order, typically many of them at each time. Deep speculative execution is employed to avoid processor stalls; branches are predicted and code is prefetched from the more likely paths of the program.

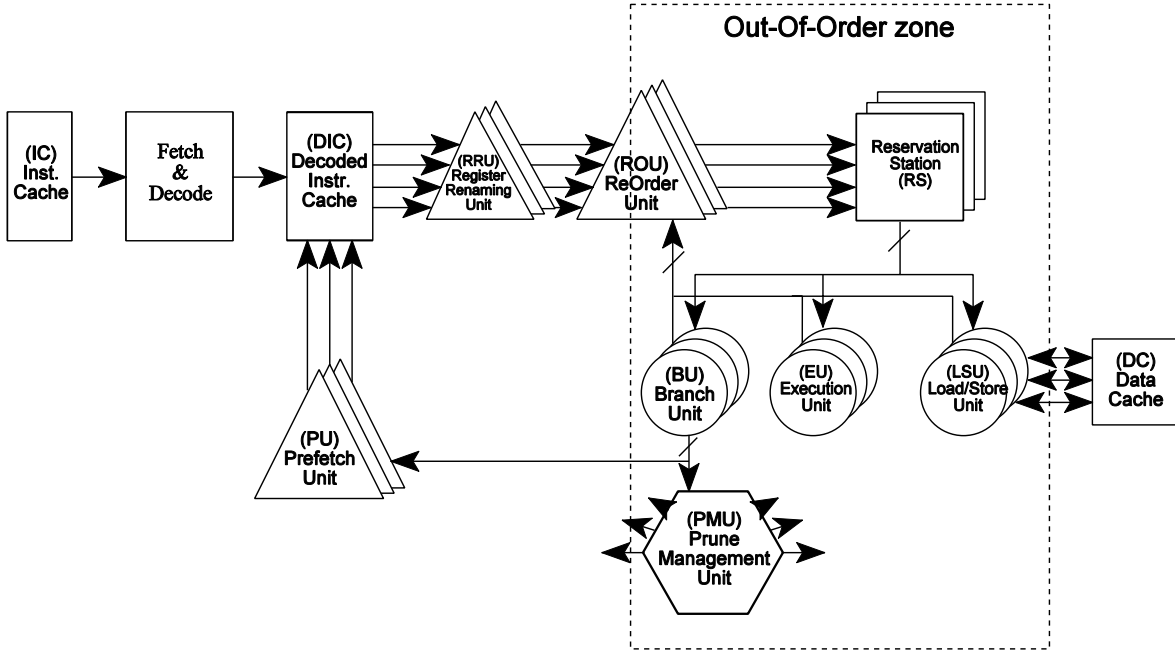


Figure 1: *Kin* asynchronous processor architecture.

2.2 *Kin* Architecture and Operation

Once fetched, instructions are decoded and stored in the multi-ported Decoded Instruction Cache (DIC, Fig. 1) where each cache line includes a single basic block (a sequence of instructions ending with a branch). The Prefetch Unit (PU) fetches multiple basic blocks from the DIC simultaneously, and tags each instruction with a unique Dynamic Instruction Tag (DIT, Fig. 2). The registers are renamed in the Register Renaming Unit (RRU), and the instructions are recorded in the ReOrder Unit (ROU), after which they are executed in the out-of-order zone. They enter one of the Reservation Stations (RSs) to wait for their operands, are processed in one of the Branch, Execute, or Load/Store Units (BU, EU, LSU), send results back to the RSs and return to the ROU for in-order commitment. Most instructions, however, never complete this cycle. Rather, they are *pruned* and discarded (Sect. 4).

Avid execution is fully explained in Sect. 3, but for clarity of the exposition it is described here in brief. The commonly used *single path speculative execution* employs branch prediction to decide which path to take following each branch; occasionally the prediction fails, the processor

is halted and flushed, and execution resumes along the correct path. In contrast, *Avid execution* also fetches and speculatively executes instructions along the non-predicted paths, so as to minimize the adverse effect of misprediction. Instructions which are found useless are pruned and discarded without preempting the processor.

The Prefetch Unit (PU) executes the branch prediction and *Avid* algorithms, and issues access requests to the DIC. It also generates *pathmarks*, which fully identify the path for each instruction (Sect. 4). The pathmark is attached to each instruction instance as it is fetched from the DIC, as part of a unique *Dynamic Instance Tag* (DIT, Fig. 2). The same basic block of code may be fetched simultaneously multiple times, e.g. as loops are unrolled. Each time, the loop is prefetched (and tagged) as a new instance, and must be treated separately by the rest of the machine (e.g., proper register renaming), regardless of the fact that it is the same original code. The instruction cache is multiported to provide simultaneous fetching of multiple cache lines, including multiple separate fetches of the same line. Access optimization techniques are employed to replace brute force multiple reads of the same line by a single access and intelligent duplication, but this is transparent to the PU. The PU, like other units in Fig. 1, is drawn as a triangle since it handles complex execution trees rather than just linear paths. Multiple triangles are drawn to symbolize multiple contexts.

Instruction		Dynamic Instance Tag (DIT)			
opcode	operands	root	path	context	pc
		pathmark			

Figure 2: Dynamic Instance Tag structure.

The Register Renaming Unit (RRU) maintains the renaming tables for the many possible execution paths avidly prefetched, to enable speculative out of order execution. The ReOrder Unit (ROU) maintains a binary tree of paths for each context, rather than just a linear sequence of instructions. The multiple Reservation Stations (RSs) store instructions from different paths and contexts together, and match operands to instructions based on physical register names. All memory access and bypass is handled in the Load/Store Unit (LSU), which manages the data cache and imposes ordering only when true dependencies are encountered: Store(*X*) instructions can bypass Load instructions, but the LSU keeps a record of the previous value of *X* until Store(*X*) commits, in case it is needed by an earlier Load(*X*). Load instructions can bypass Store instructions, except for Store to the same address, in which case the argument is forwarded from the Store instruction. The Branch Unit (BU) resolves branch instructions and returns the results to the ROU, the Prune Management Unit (PMU) and the PU. Upon receiving branch results, the PU updates the prediction algorithm and prefetches new instructions. The Pruning Management Unit (PMU) generates and distributes prune and ahead messages, to be described in Sect. 4 below.

The massively parallel *Kin* architecture assumes the availability of massive hardware resources, e.g., one billion transistors. However, as explained below, existing execution techniques do not scale well. *Avid* execution is designed to address this problem.

3. Avid Execution

3.1 Execution Modeling

In this section we model speculative execution with branch prediction, in order to explain the advantages of Avid execution.

Branch prediction and speculative execution are designed to reduce pipeline stalls due to conditional branches. On a misprediction, the pipeline is flushed, and new instructions are then fetched. **Misprediction Penalty** is defined as the time required for the pipeline to fill up after a flush, until instructions start committing again. This time depends linearly on the pipeline depth (the number of stages between the fetch and branch resolution stages).

Out-of-order execution takes advantage of Instruction Level Parallelism (ILP) and executes independent instructions concurrently. But as instruction rate increases, so does the rate of branch instructions (on average every fifth instruction is a branch [HP96]), and misprediction rate increases as well. This adverse effect is compounded by another setback: The deeper the pipeline and the higher the parallelism, the higher the penalty paid for misprediction.

Instruction execution rate is thus limited by several factors (misprediction rate, misprediction penalty, available ILP, and hardware resources--pipeline depth and width). Figure 3 defines the following execution model. If there were no mispredictions, the execution rate (presented by the dashed line) would be limited only by the available ILP and hardware resources. However, since mispredictions do happen, the processor is able to execute instructions only during n time units before a misprediction is encountered. During that time, E instructions (the shaded area in the graph) are committed. Following a misprediction, no instructions are committed for the next m time units (the misprediction penalty time). After that, instructions start to commit again. As depicted in the graph, misprediction might happen even before the maximum execution rate is achieved. This phenomenon repeats itself and the average execution rate R , presented by the horizontal dotted line in Fig. 3, is:

$$(1) \quad R = \frac{E}{n + m}$$

E , the number of instructions executed between mispredictions, is a function of the prediction accuracy of the branch prediction algorithm (Fig. 4). For an average basic block length B , and for a prediction accuracy p , misprediction occurs once every $E_{miss} = B/(1-p)$ instructions. Thus, for $B=5$ and $p=0.95$, misprediction is expected every 100 instructions.

Theoretically, in the absence of mispredictions, ILP depends on the instruction window size [HP96], and the window size depends on hardware resources such as the RS. ILP is defined as the number of independent instructions in the window that can be executed concurrently. Figure 5 presents the ILP of four benchmark programs [HP96], and their average ILP. Note, however, that for high ILP the typical window size exceeds typical values of E_{miss} (Fig. 4). Thus, in the presence of mispredictions, we must carry a careful transient analysis, as follows.

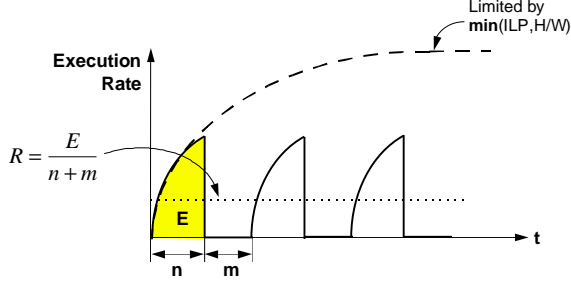


Figure 3: Instruction execution rate.

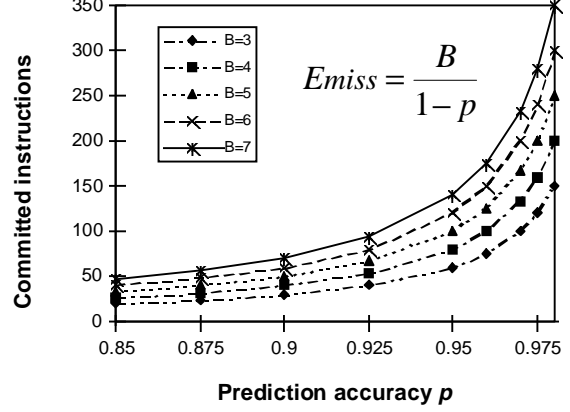


Figure 4: Number of executed instructions till misprediction.

Dynamically, the window size is the difference between the number of instructions that have entered and left the window, counting from the most recent flush. Let's assume that instructions enter the window at a constant rate of w instructions every cycle, where w represents the width of the fetch-rename-reorder pipeline. Each cycle, the number of instructions leaving the window (to be executed and committed) is assumed to be the ILP of that window size, as derived from Fig. 5. We apply curve fitting on the data to represent ILP functionally, and thus we need to solve the following system of recurrence equations:

$$(2) \quad \begin{aligned} ILP[Window(n)] &= a \ln[Window(n)] + b \\ E(n) &= \sum_{k=0}^n ILP[Window(k)] \\ Window(n) &= w \cdot n - E(n-1) \end{aligned}$$

To simplify the analysis, we define the average time unit required to complete one instruction at one processor stage as a 'cycle'. This need not necessarily imply that the processor design has either a synchronous or an asynchronous implementation. 'Cycles' should be thought of as time periods, which might be all equal, e.g., a clock cycle in a synchronous processor, or they might have an average length and variance in an asynchronous processor. The analyzed behavior, however, remains the same for the synchronous and asynchronous cases, since both are designed to be as balanced as possible. In Eq. 2, n is the cycle index, counting from the last misprediction.

Equation 2 is implicit and cannot be solved directly. Thus, we compute $E(n)$ as an iterative process, and the result is shown in Fig. 6 for various values of w .

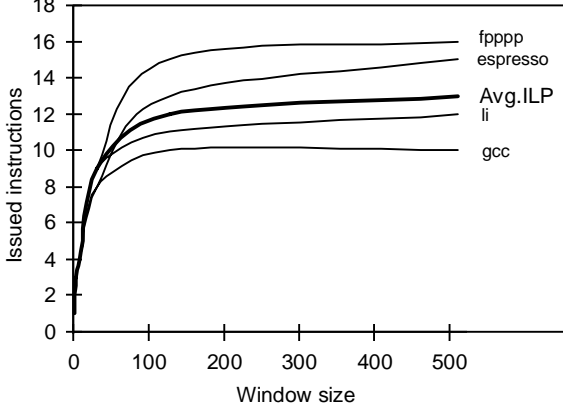


Figure 5: Studies of ILP as a function of window size.

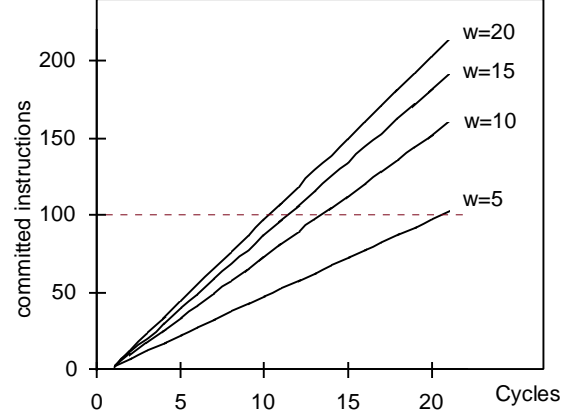


Figure 6: Cumulated executed instructions as a function of time cycles (assuming no misprediction), with hardware parallelism (w) as a parameter.

The horizontal dashed line in Fig. 6 marks $E_{miss}=100$ instructions (expected between mispredictions for $B=5$, $p=95\%$). As can be observed from Fig. 6, the higher the width, the higher the misprediction rate. Thus, there are diminishing returns to brute force investment in wider processors. The lines in Fig. 6 are clearly linear, and regression analysis yields the following empirical trends for $E(n,w)$:

$$(3) \quad \begin{aligned} E(n,w) &= \alpha(w) \times n + \beta(w) \\ \alpha(w) &= 4.1 \times \ln w - 1.5 \\ \beta(w) &= -4.7 \times \ln w + 5 \end{aligned}$$

Assigning $E=E_{miss}=B/(1-p)$ and Eq. (3) into Eq. (1) yields:

$$(4) \quad R = \frac{E_{miss}}{n+m} = \frac{E_{miss}}{\frac{E_{miss} - \beta(w)}{\alpha(w)} + m} = \frac{\alpha(w)}{1 + \frac{1-p}{B} \times [\alpha(w) \times m - \beta(w)]}$$

This expression is plotted in Fig. 7 for $B=5$ and $p=95\%$. Again, it can be seen in Fig. 7 that a high width w does not contribute to increase performance, and performance is actually limited by ILP. The effective window size (which affects ILP) cannot be increased, since misprediction occurs long before the window fills up. As the misprediction penalty m increases, the effect of a higher width w becomes negligible, since more time is spent incurring the misprediction penalty rather than doing computation. Thus, if the misprediction penalty is high, investing in higher parallelism in the processor does not improve the average execution rate and processor performance.

In summary: In processors employing branch prediction and speculative out-of-order execution, wider pipeline leads to a higher misprediction rate, and deeper pipeline leads to a higher misprediction penalty. Avid execution is designed to overcome this obstacle.

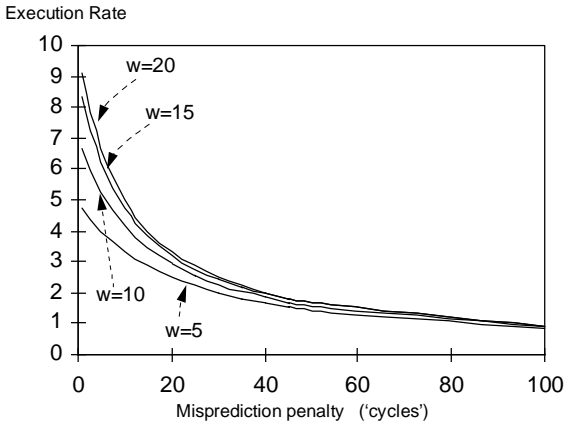


Figure 7: Hardware parallelism and misprediction penalty effect on execution rate.

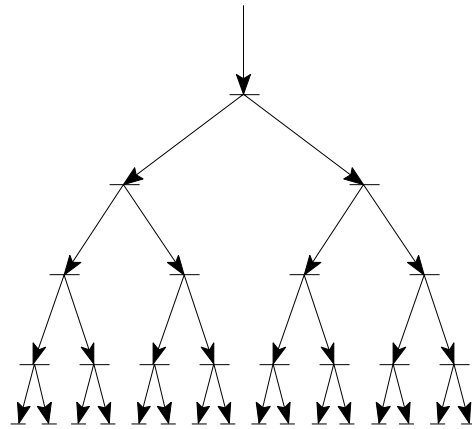


Figure 8: Tree of possible execution paths (vertices are branch instructions, edges are basic blocks).

3.2 Previous Models of Speculative Execution

To put *Avid Execution* in context, we briefly survey the following previous models of speculative execution: single path speculative execution, eager execution, multiple path exploration, and disjoint eager execution.

The tree of possible execution paths is shown in Fig. 8. The vertices of the tree are branch instructions. Each edge is a basic block, i.e., a sequence of nonbranch instructions terminated by a branch instruction.

Single Path Speculative Execution

Single path speculative execution (SPSE) was assumed in the analysis of the previous section. Branch prediction tags each branch as either taken or not-taken, based on its past history, and provides the branch target address [Cra92, HP96, LS84, YP92]. Instructions are fetched (speculatively) from the predicted branch target. On misprediction, the pipeline is flushed and the correct path is fetched. Misprediction rate and penalty were treated above.

An execution tree of depth n contains n edges for SPSE, so the cost is linear in the overall depth of prediction. However, the probability of correct prediction over n levels falls off exponentially as p^n . As explained above, misprediction is the primary limitation to enhancing processor performance.

Eager Execution and Multiple Path Exploration

In Eager Execution all paths are prefetched and (speculatively) executed. When a branch is encountered, execution proceeds down both paths of the branch. Once a branch is executed, its ‘losing’ sub-tree is aborted and disposed of. The principal benefit of eager execution is that

misprediction stalls are eliminated. However, eager execution is exponentially wasteful: Of the $2^n - 1$ edges of a n -level execution tree, only n edges are on the true path and eventually commit, while the remaining $2^n - 1 - n$ edges are discarded. Due to the enormous amount of resources required to implement eager execution, and the relative high accuracy of prediction algorithms available, eager execution is impractical and has not been implemented in any real processor.

Multiple Path Exploration limits eager execution to a tree depth of m levels [Mag80, MTM81]. Thus, only 2^m paths are explored simultaneously. The system contains 2^m path processors and one central processor which generates the instructions of each branch path from the program and issues them to the path processors. While the instructions of m branch levels are being executed, the controller generates 2^{2m} paths of the next m branch levels. Only 2^m of these paths are used and the rest are discarded once the valid path from the previous m levels block is determined. The results from the data cache of the selected processor are copied into the shared memory, and the processors are then assigned the next 2^m paths. Multiple Path Exploration requires approximately exponential hardware resources.

Disjoint Eager Execution

Disjoint Eager Execution calculates cumulative prediction probabilities, and fetches the most likely basic blocks [US95]. On misprediction the processor is flushed. When the prediction accuracy approaches 100%, disjoint eager execution practically converges into SPSE, since the first alternative path will be selected only after going n levels deep in the execution tree, such that $(1-p) > p^n$. Note, for instance, that for $p=90\%$ 22 levels are descended before the first non-predicted branch is taken. However, misprediction is expected already after ten branches. Thus, this model is inconsistent for typical levels of p . In addition, the model (as analyzed in [US95]) ignores not only misprediction rates but also misprediction penalties.

Avid Execution, defined in the following section, is designed to avoid the pitfalls of all these methods.

3.3 Concepts of Avid Execution

Avid Execution combines the benefits of both SPSE and eager execution, such that misprediction penalty is kept very low, while the exponential cost of eager execution is replaced by an approximately linear cost. Avid execution is basically an eager execution with limited eagerness, based on branch prediction. As in SPSE, the predicted path is prefetched and executed. In addition, for each branch encountered and predicted, certain parts of a k levels deep subtree which is predicted as not-taken are also fetched into the processor, and are speculatively executed.

The number k of prefetched levels in the non-predicted subtree is adjustable. Figure 9 shows two examples of Avid execution depths, for $k=2$ and $k=5$. The main predicted path is marked by solid arrows, while the extra (avid) paths are drawn as dashed arrows. Note that if $k=0$, Avid execution is reduced to SPSE. For $k=1$, about 50% of all instructions fetched will be pruned, since for every predicted basic block another basic block from the non-predicted path is also fetched. The price of exponential demand for resources in eager execution is avoided and is replaced by an approximately linear one: For Avid execution of depth k , the number of edges in an n levels deep

execution tree is $O(kn)$. Avid execution can produce instructions at a sufficient rate to reduce or even eliminate all stalls on misprediction, as analyzed in [Kol97]. The unneeded instructions are pruned asynchronously, without preempting continuous operation of the processor, as described in Sect. 4 below.

Avid depth can be selected either statically (e.g., all conditional branches have the same alternative path depth), or dynamically. Dynamic adjustment is preferred. It should be based on statistics collected at run time. If confidence is applied to prediction [JRS96, Smi81], the *Avid* depth can be adapted accordingly. When the prediction confidence level is low, a deeper *Avid* depth should be used, and for high confidence prediction a small *Avid* depth (or none at all) might be better. Obviously, $k=0$ for unconditional branches.

Observe that the first edge of each alternative path described in Fig. 9, originating from each branch instruction (a tree vertex), is the branch direction predicted as not being followed. The following edges of the alternative path are selected by branch prediction. Another option for selecting the alternative paths in *Avid* execution (instead of following ‘single path’ alternatives), is to span a (limited depth) sub-tree from the path predicted as not taken. *Avid* execution can be recursively applied to the alternative paths as well. Obviously, if the alternative sub-tree is as deep as the predicted one, and follows all possible paths, then *Avid* execution becomes eager execution.

As more alternative paths are fetched by *Avid* execution, more resources are required. Our simulations verify that the single path alternatives is quite adequate when prediction accuracies are very high. Spanning more alternative paths results in diminishing returns.

Consider the following example of performance improvement achievable by *Avid* execution. The pipeline depth (measured in the number of basic block stages that fit between instruction fetch and the branch unit) is $m=5$, $p=95\%$, and sufficient hardware resources are available so that execution is limited only by ILP and mispredictions. Analytical studies, described in the following section, show that performance can be improved by as much as 50% under these conditions.

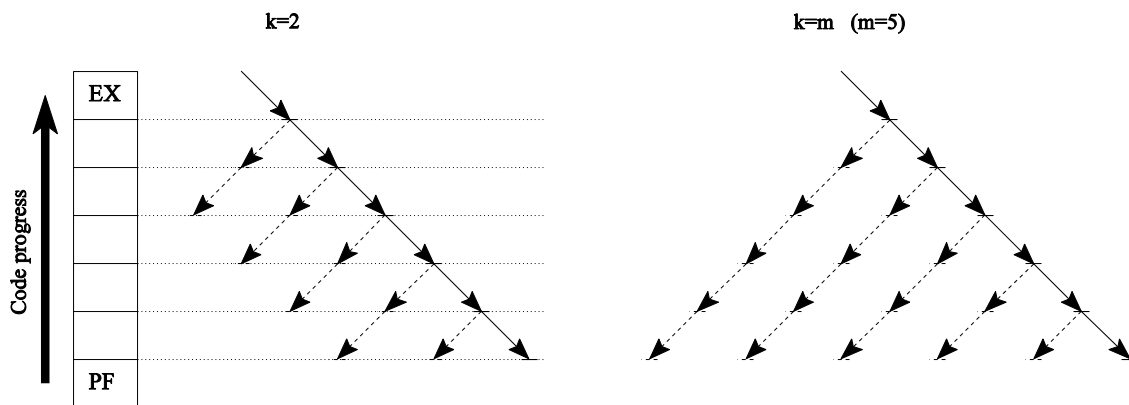


Figure 9: Examples of *Avid* Execution depth (k). m is the number of processor pipeline stages between prefetch (PF) and branch resolution (EX) stages.

3.4 Performance Analysis of Avid Execution

Assume there are m stages in the processor between the prefetch stage (marked PF in Fig. 9) and the branch execution stage (marked EX). Each processor stage must be able to concurrently handle instructions from $k+1$ basic blocks. Once the instructions of a basic block commit in-order, the entire subtree is shifted and progresses in the processor stages, while new paths are predicted and prefetched. For every branch on the predicted path, alternative paths (emerging from it) of length k are also predicted. When a branch is resolved at the execution stage (possibly out-of-order), the redundant path is pruned. Execution continues along the resolved path uninterrupted, regardless of whether it was predicted or not. The misprediction penalty of stalling all processor stages while waiting for the correct instructions to be fetched and handled is avoided, or reduced, as follows. Two cases of misprediction penalty for Avid execution are presented in Fig. 10.

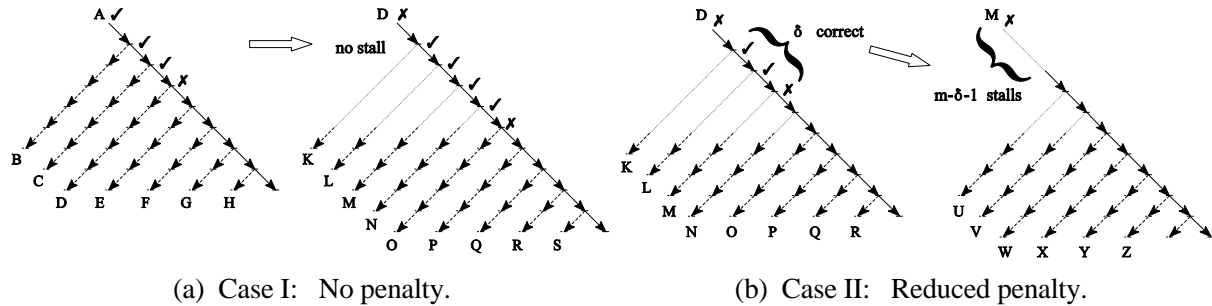


Figure 10: Misprediction penalties in Avid Execution.

In this example, the Avid depth is $k=m$. In Case I (Fig. 10(a)), execution follows the path marked A. Assuming there is no misprediction of several previous branches, the Avid subtree (including the alternative paths B through H) has been fetched and is handled in the various stages of the processor. The first two predictions are proven correct (checkmarked '✓'), execution continues along path A, and paths B and C are pruned. The next prediction is incorrect (marked 'X'), and the rest of path A is pruned along with all the alternative paths emerging from it (E through H). After the misprediction, execution continues along path D. Since path D has already been predicted, fetched, and partially handled for depth m , none of the processor stages is stalled.

Note that none of the alternative paths of the new main speculative path D has been prefetched, so they must now be predicted and fetched. These 'holes' in the Avid tree are presented by dotted lines. If there is no subsequent misprediction for at least m branches, then the Avid tree is completed again. If a misprediction happens at a later time, e.g., at the 'X' mark leading to path O, then again no misprediction penalty is incurred, since path O already has m basic blocks in the processor.

Figure 10(b) shows Case II. After switching to path D, a second misprediction occurs after δ correctly predicted branches. Execution should now follow path M, which has not yet reached the execution stage. Some of the processor stages are stalled while waiting for the M path to proceed through them. While path M moves forward towards the execution stage, its alternative Avid paths are predicted and prefetched to restore the proper Avid tree. Since part of the alternative

path M does exist in the processor at the time of the misprediction, only a reduced penalty is incurred. This is in contrast to what happens after misprediction on a SPSE processor, wherein the whole processor is flushed and stalled.

Average execution rate (as defined in Sect. 3.1) is used to measure the performance improvement that can be achieved by Avid execution (this measure is similar to the ‘instructions per cycle’ parameter used for synchronous processors performance). We identify the possible cases of mispredictions, and weigh the different misprediction penalties according to the associated probabilities. We focus on cases of two mispredictions separated by some correctly predicted branches; for reasonable prediction accuracies ($p > 80\%$), the probability of three successive mispredictions, $(1-p)^3$, is negligible.

If i is the number of consecutive basic blocks executed without a misprediction, then $i-1$ consecutive branches were correctly predicted. The misprediction penalty depends on the number of stages between the prefetch and execution stages (m), the Avid depth (k), and the number of basic blocks executed without a misprediction (i). The last two parameters affect the size of the ‘bubble’ in the processor. Thus, the misprediction penalty (in ‘cycles’) is given by

$$(5) \quad M_i = m - \min(i, k)$$

and the average rate R_i is defined as

$$(6) \quad R_i = \frac{E_i}{n_i + M_i}$$

where n_i is the number of cycles required to execute E_i instructions between the two mispredictions.

Mispredictions which occur once every m branches or more (i.e., $i \geq m$) have the same reduced penalty, which depends on k (if $k=m$, then there is no penalty, as explained above). The total average execution rate is defined as the weighted sum of execution rates:

$$(7) \quad R_{avg} = \sum_{i=1}^{m-1} (1-p)^2 p^{i-1} \times R_i + \left(1 - \sum_{i=1}^{m-1} (1-p)^2 p^{i-1} \right) \times R_m$$

where p is the prediction accuracy.

As an example of performance improvement achievable by Avid execution, consider a case of $w=20$, $m=5$, and sufficient hardware resources so that execution is limited only by ILP and mispredictions. As can be seen in Fig. 11, more than 100% increase in performance can be achieved by Avid execution, depending on Avid depth and prediction accuracy.

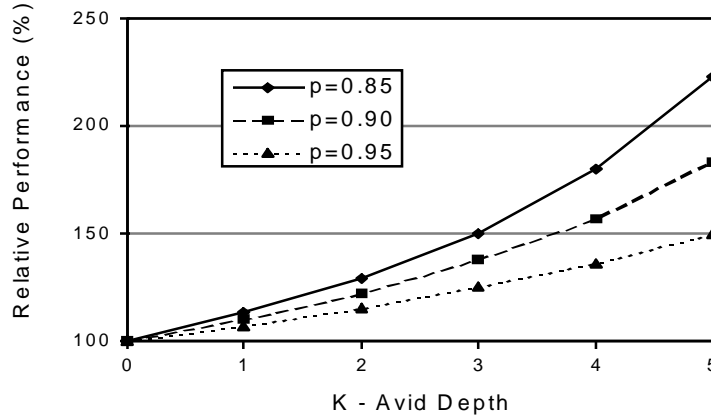


Figure 11: Average performance improvement achieved by various Avid depths (k) and prediction accuracy (p), for $m=5$, and $w=20$ (execution limited by ILP).

Obviously, Avid execution can contribute more to performance gain when misprediction penalty (m) is higher, since reducing that penalty has a larger effect on overall performance. The lower the prediction accuracy (p), the higher the increase in performance that is achievable by Avid execution, because mispredictions happen more often and Avid reduces the penalty paid.

On the other hand, although Avid execution can optimally use any ‘spare’ processing bandwidth which is not utilized due to limited parallelism in code, it might also hamper performance if there are insufficient resources. The deeper the avid depth, the more resources are required. Most of the instructions are pruned at early stages of the processor, but if the processor width (w) is not sufficiently wide, the extra instructions will slow the execution. Still, if the reduction of misprediction penalty increases performance more than the decrease in execution, the overall performance is increased. Some examples are presented and further explained by simulation results in Sect. 5.

4. Instruction Pruning

Avid execution employs pruning to remove unneeded instructions on the fly, without preempting execution, without stalling the processor, and without flushing the pipes. Pathmarks, which are part of the DIT (Fig. 2), distinguish alternative paths and identify the doomed instructions. Each edge of the execution tree is assigned a unique prefix pathmark: for an edge marked m , the sequentially following edge and the branch target edge are marked $m0$ and $m1$, respectively (Fig. 12). Pathmarks are generated dynamically during program execution and are affixed to each instruction at prefetch by the PU (Sect. 2).

Pruning removes entire subtrees per each resolved branch. Since the pathmarks of all instructions of the subtree share the same prefix, a single *prune()* message suffices. If a branch m was taken (not taken), the *prune(m0)* message (*prune(m1)*, respectively) is broadcast to the entire processor. Out-of-order pruning is possible and permitted. For instance, a branch with pathmark $m0k$ may execute before branch m ; the *prune(m0k1)* message may precede the *prune(m0)* message; the

latter will override the former. Pruning messages are generated and distributed by the PMU (Fig. 1).

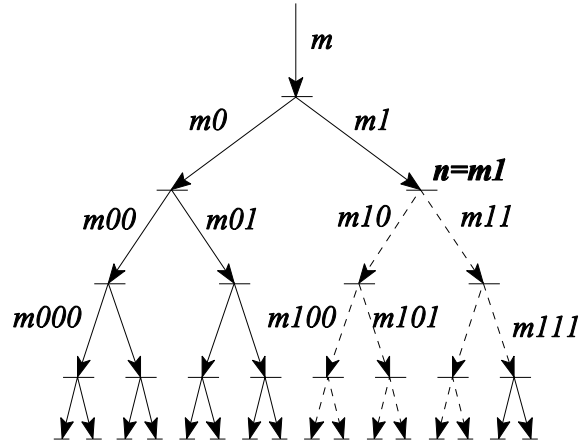


Figure 12: Pathmarks based on prefix notation.

The pathmark length grows very fast, as one more bit is appended on every branch. A mechanism for periodically *beheading* this length is described in [Kol97]. It effectively replaces linear pathmark growth by logarithmic growth of the *root* mark, which is added to the DIT.

5. Performance Simulations of Avid Execution

We have developed a software model of *Kin* and Avid execution, and simulated that model executing the SpecInt95 benchmark.

A standard branch prediction algorithm [YP92] was implemented. Various prediction accuracies were obtained by changing the size of the branch target buffer.

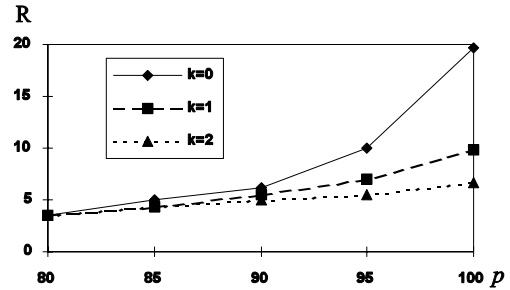
Kin was specified at the high level using statecharts and the iLogix Magnum tool [Har87, iLo96]. The internals of each module were specified as functions in C. This formal and operational specification of *Kin* has been used directly for event driven simulations, as well as for debugging, animation, and validation. Module delays are tunable in the simulations [KGS97]. We simulated SpecInt95 traces, and gathered information on average and worst case FIFO and table sizes, committing and pruning rates, and program execution times. Full data is given in [Kol97, Sha97].

Avid execution was simulated for three possible (fixed) Avid depths: $k=0$, 1, and 2. The processor pipeline width (the number of instructions that can be handled concurrently in each processor unit) was simulated at 20, 40 and 80 instructions. Avid execution spanning an ‘eager’ subtree for $k=2$ was also simulated, and demonstrated diminishing returns, as expected. The results were at best the same as those obtained by Avid execution spanning a ‘single path’ for $k=2$, and some times even worse, due to high prediction accuracies and contention for resources.

First, the simulation of a synthetic program is summarized in Fig. 13. The tables contain the

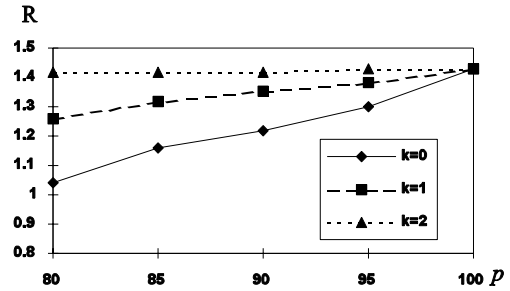
average execution rate R and relative performance of Avid execution (for $k=1,2$) compared to SPSE ($k=0$). When there are no dependencies (Fig. 13(a)) and no mispredictions ($p=100\%$), Avid execution reduces the execution rate (from 19.69, which is almost equal to the pipeline width $w=20$), to 50% and 33%, for $k=1$ and 2, respectively, as expected, since it uses resources that can be used for the ‘true’ path. Since there are no dependencies, the execution along the ‘true’ path can proceed with no stall, except when a branch is mispredicted. At $p=80\%$, Avid execution does not decrease the performance any more. With even lower prediction accuracies, the overall performance is increased with Avid execution.

p	80%		85%		90%		95%		100%	
	R	%	R	%	R	%	R	%	R	%
$k=0$	3.56	100	4.97	100	6.22	100	9.95	100	19.69	100
$k=1$	3.56	100	4.35	88	5.52	89	6.99	70	9.90	50
$k=2$	3.56	100	4.35	88	4.97	80	5.54	56	6.61	33



(a) Trace with No Dependencies.

p	80%		85%		90%		95%		100%	
	R	%	R	%	R	%	R	%	R	%
$k=0$	1.04	100	1.16	100	1.22	100	1.30	100	1.43	100
$k=1$	1.26	121	1.32	114	1.35	111	1.38	106	1.43	100
$k=2$	1.42	137	1.42	122	1.42	116	1.43	110	1.43	100



(b) Trace with Full Dependencies.

Figure 13 : Synthetic traces simulation results (for $w=20$).

When each instruction depends on the previous one (Fig. 13(b)), the available resources in the processor are not fully utilized, and execution is stalled due to data dependencies. When $p \leq 100\%$, Avid execution can exploit the ‘spare’ idle resources to work concurrently on instructions from alternative paths. Note how Avid execution becomes more effective as p decreases and k increases.

Next, we have simulated all SpecInt95 programs with several Avid execution depths ($k=0, 1, 2$) and various processor widths ($w=20, 40, 80$) [Sha97]. The simulation results for $w=40$ are presented in Fig. 14, and analyzed below. Varying degrees of prediction accuracies and basic block lengths were found (Tab. 1).

Program	Average Basic Block Length	Prediction Accuracy (%)
go	5.3	85
m88ksim	3.6	88
gcc	4.6	83
Compress	5.8	93
Li	3.8	93
Ijpeg	4.2	97
Perl	4.7	93
Vortex	4.2	95
Average	4.525	90.875

Table 1 : Properties of the SpecInt95 benchmark programs, as found in the simulation of *Kin*.

A first glance at Fig. 14 reveals that, over all benchmarks, the incremental improvement of $k=1$ over $k=0$ is more significant than the additional incremental improvement provided by $k=2$. The simulation of Compress95 (Fig. 14(a)) shows that for $k=2$, performance is best up to about $p=86\%$. Above $p=88\%$, $k=0$ is best. For Gcc (Fig. 14(b)), $k=2$ is better than $k=1$, up to $p=83\%$, where they become equal. At that point, they both give 11% higher performance than $k=0$. For Go program (Fig. 14(c)), $k=2$ shows better performance than either $k=1$, or $k=0$, up to $p=85\%$. Simulation of Ijpeg (Fig. 14(d)) resulted in highest performance for $k=2$ up to $p=77\%$, then $k=1$ gives better performance up to $p=97\%$. They are always better than $k=0$. Although we could expect Avid execution to be more beneficial for programs having lower prediction accuracy, it did prove useful even for Ijpeg, which shows the highest prediction accuracy in that benchmark. A similar behavior was seen for the Li program (Fig. 14(e)), where $k=2$ performs better than $k=1$, up to $p=92\%$. At $p=93\%$ they switch, but are still both better than $k=0$. M88ksim program (Fig. 14(f)) also behaves the same, but the switch in performance gain between $k=2$ and $k=1$ occurs at $p=76\%$. For Perl (Fig. 14(g)), $k=2$ is best up to $p=86\%$, and then $k=1$ is better but $k=0$ becomes the best. Vortex (Fig. 14(h)) always resulted in best performance for $k=2$ (up to $p=95\%$), while even $k=1$ was better than $k=0$.

As explained in Sect. 3.1, the average execution rate is affected by several parameters, namely the instruction level parallelism, the prediction accuracy, the processor width and the misprediction penalty (pipeline depth). The effects of varying prediction accuracies, Avid depth and different programs were shown in Fig. 14. To demonstrate the effects of the other parameters on the execution rate and performance gain of Avid execution, one program (M88ksim) was simulated with varying w , p , and k , as explained below.

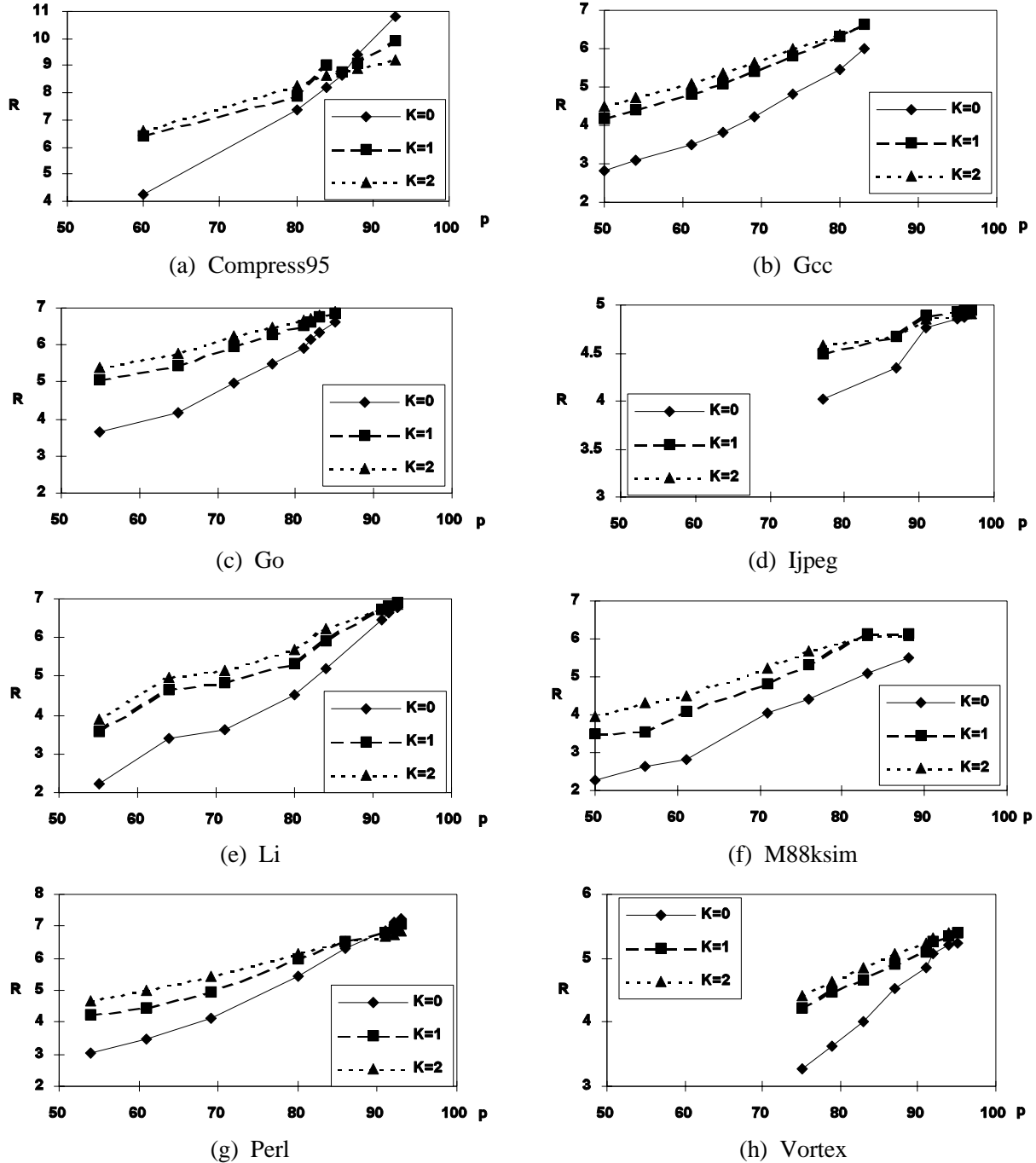


Figure 14: SpecInt95 simulation results (for $w=40$). The graphs describe the average execution rate (R) as a function of prediction accuracy (p), with Avid execution depth (k) as the parameter.

Figure 15 shows relative performance compared to $k=0$. Both graphs show that doubling pipeline width from $w=40$ to $w=80$ results in much less gain than when going from $w=20$ to $w=40$. As the width increases, more instructions are executed concurrently. This leads to higher branching rate, and consequently to higher frequency of mispredictions and the resulting penalty. Careful observation reveals that $k=2$ provides better performance than $k=1$ only up to $p=76\%$ for $w=20$ and $w=40$, but in the case of $w=80$, $k=2$ remains preferred up to $p=88\%$. Similar behavior was observed for the other traces simulated, where the ‘switch’ between the performance gains of

$k=0,1,2$ occurs at different prediction accuracies, depending on the processor width.

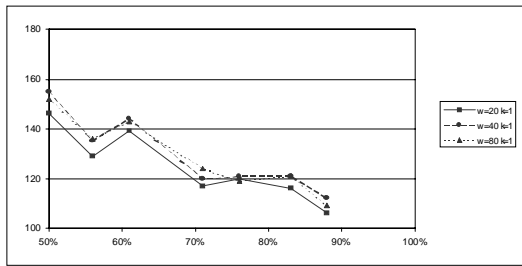


Figure 15(a) : M88ksim simulations with Avid depth $k=1$ and varying width w .

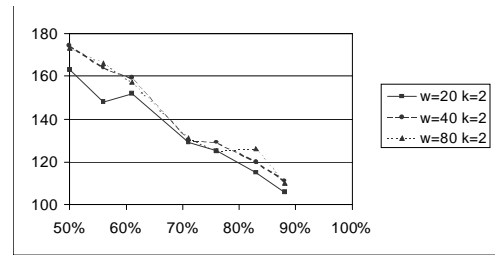


Figure 15(b) : M88ksim simulation with Avid depth $k=2$, and varying width w .

Kin does not have a pure pipeline structure, but the units in it may be viewed abstractly as pipeline stages. By changing relative timing of the units we could change the pipeline effective ‘depth’, and affect the misprediction penalty. Figure 16(a) shows the simulation of such a short pipeline (lower misprediction penalty). Although the execution rates increase because of the lesser stall on each misprediction, the performance increase gained by Avid execution remains relatively the same (comparing relative performance).

Due to some technical limitations of the simulator, we have simulated the effect of very deep Avid execution (high k) by using a very short pipeline (2 stages deep). Fig. 16(b) shows growing execution rates, thanks to reduced misprediction penalty, and Avid execution results in a substantial performance improvement of 25% at $p=88\%$, and up to 80% improvement for lower prediction accuracies.

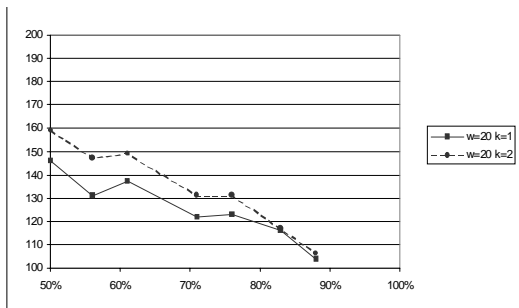


Figure 16(a) : M88ksim simulations with a short pipeline

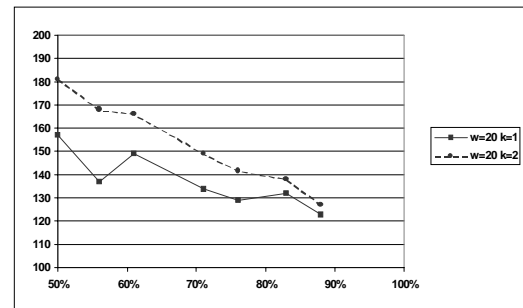


Figure 16(b): M88ksim simulations imitating high Avid depth (k) by means of very short misprediction penalty

Another interesting effect of Avid execution found in the simulations relates to the total number of instructions fetched from memory. In several cases $k=1$ actually resulted in less instructions being fetched, since $k=0$ had to flush many instructions. Thus, Avid execution not always results in an increased memory bandwidth.

We have only implemented and simulated a fixed Avid depth. It indicates however that better

performance can be achieved when an adaptive Avid depth is used, based on the prediction accuracy and confidence of each branch, as discussed in Sect. 3.3.

6. Conclusion

We have introduced, analyzed, and simulated Avid execution, which improves older methods of speculative execution. When more resources are available than can be effectively utilized to execute serial code, Avid execution prefetches and executes non-predicted paths, in preparation for any imminent misprediction. The depth of Avid execution may be adjusted dynamically according to prediction confidence. We have introduced the Dynamic Instance Tag (DIT) to uniquely define a path, and defined a set of operations on the DIT to insure that useful computation is executed and useless computation is discarded. Avid execution applies pathmarks and pruning to execute instructions from many paths as soon as their operands are ready, but stop executing the remaining instructions on a path as soon as it is known that it will not be taken. We have simulated a fixed Avid scheme, and have discussed other alternatives for future research. Simulations show that Avid execution can achieve performance improvement close to 100%, depending on many factors such as the accuracy of branch prediction and the instruction level parallelism inherent in the program. The simulations further validate the mathematical analysis.

Further discussion of Avid execution, its effect on computer architecture, and features such as extending it to multi-execution, are included in [Kol97].

Asynchronous architectures (such as *Kin*) are best suited for Avid execution, because of the complex design and the great variance of computation loads. Avid execution is designed in the context of very large processors, such as predicted for another 15 years.

References

- [Bru93] E. Brunvand, "The NSR Processor," *Proc. of the 26th Annual Hawaii Int. Conf. on System Sciences*, Vol. 1, pp. 428-435, 1993.
- [Cra92] H. G. Cragon, *Branch Strategy Taxonomy and Performance Models*, IEEE Computer Society Press, 1992
- [Dea92] M. E. Dean, *STRiP: A Self-Timed RISC Processor*, PhD thesis, Stanford Univ., 1992.
- [DGY93] I. David, R. Ginosar, and M. Yoeli, "Self-Timed Architecture of a Reduced Instruction Set Computer," in *Asynchronous Design Methodologies*, Furber S. and Edwards M. editors, IFIP Transactions Vol. A-28, Elsevier Science Publishers, pp. 29-43, 1993.
- [End95] P. B. Endecott, *SCALP: A Superscalar Asynchronous Low-Power Processor*, PhD thesis, Dept. of Computer Science, Univ. of Manchester, 1995.
- [FDG+93] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods, "A micropipelined ARM," *VLSI'93*, 1993.
- [Har87] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, **8**(3), pp. 231-274, 1987.
- [HP96] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd edition, Morgan Kaufmann, 1996.

- [iLo96] i-Logix, Inc., *Statemate MAGNUM documentation*, 1996
(See also: <http://www.ilogix.com>).
- [JRS96] E. Jacobsen, E. Rotenberg, and J. E. Smith, "Assigning Confidence to Conditional Branch Prediction," *Proceedings of the 29th International Symposium on Microarchitecture (Micro-29)*, pp. 142-152, Dec. 1996.
- [KGS97] R. Kol, R. Ginosar, and G. Samuel, "Statecharts Methodology for the Design, Validation, and Synthesis of Large Scale Asynchronous Systems," *IEICE Trans. on Information and Systems*, **E80-D(3)**, pp. 308-314, Mar. 1997.
- [Kol97] R. Kol, *Self-Timed Asynchronous Architecture of an Advanced General Purpose Microprocessor*, PhD thesis, Dept. of Electrical Engineering, Technion, Israel, 1997.
- [LS84] J. K. F. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer*, **17(1)**, pp. 6-22, Jan. 1984.
- [Mag80] N. F. Magid, *High Speed Computer Systems as a Result of Concurrent Execution of Sequential Instructions*, PhD thesis, Dept. of Electrical Engineering, Illinois Institute of Technology, Chicago, Illinois, 1980.
- [Mar97] A. Martin, *et al.*, "The Design of an Asynchronous MIPS R3000 Microprocessor," *Proc. Advanced Research in VLSI*, Sept. 1997.
- [MBL+89] A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, and P. J. Hazewindus, "The Design of an Asynchronous Microprocessor," Technical Report, Caltech-CS-TR-89-02, 1989.
- [MTM81] N. Magid, G. Tjaden, and H. Messinger, "Exploitation of Concurrency by Virtual Elimination of Branch Instructions," *International Conference on Parallel Processing (ICPP)*, pp. 164-165, Aug. 1981.
- [NUK+94] T. Nanya, Y. Ueno, H. Kagotani, M. Kuwako, and A. Takamura, "TITAC: Design of a Quasi-Delay-Insensitive Microprocessor," *IEEE Design & Test of Computers*, **11(2)**, pp. 50-63, Summer 1994.
- [Pav94] N. C. Paver, *The Design and Implementation of an Asynchronous Microprocessor*, PhD thesis, Dept. of Computer Science, Univ. of Manchester, 1994.
- [RB96] W. F. Richardson and E. Brunvand, "Fred: An Architecture for a Self-Timed Decoupled Computer," *2nd Int. Symp. on Advanced Research in Asynchronous Circuits and Systems (Async'96)*, pp. 60-68, Mar. 1996.
- [Sha97] H. Shafi, *Avid Execution and Instruction Pruning in the Asynchronous Processor KIN*, MSc thesis, Dept. of Electrical Engineering, Technion, Israel, 1997, in preparation.
- [SIA97] Semiconductor Industry Association, *Semiconductor Technology Roadmap*, 1997 edition. Note to the referees: We use the numbers of the yet-unpublished 1997 SIA Technology Roadmap. That edition will become public by Dec. 97.
- [Smi81] J. E. Smith, "A Study of Branch Prediction Strategies," *Proc. Eighth Symposium on Computer Architecture*, pp. 135-148, May 1981.
- [SSM94] R. F. Sproull, I. E. Sutherland, and C. E. Molnar, "The Counterflow Pipeline Processor Architecture," *IEEE Design & Test of Computers*, **11(3)**, pp. 48-59, Fall 1994.
- [US95] A. K. Uht and V. Sindagi, "Disjoint Eager Execution: An Optimal Form of Speculative Execution," *Proceedings of the 28th International Symposium on Microarchitecture (Micro-28)*, pp. 313-325, Nov. 1995.
- [Wei96] U. Weiser, "Future Directions in Microprocessor Design," Invited lecture, presented at *2nd Int. Symp. on Advanced Research in Asynchronous Circuits and Systems (Async'96)*, Mar. 1996.
- [Wol92] T. L. Wolf, *The A3000: An Asynchronous Version of the R3000*, MSc thesis, Dept. of Computer Science, Univ. of Utah, 1992.
- [YP92] T.-Y. Yeh and Y. N. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction," *The 19th International Symposium on Computer Architecture (ISCA), ACM SIGARCH Computer Architecture News*, **20(2)**, pp. 124-134, May 1992.