Kin[1]: A High Performance Asynchronous Processor Architecture

Rakefet Kol Ran Ginosar
VLSI Systems Research Center, Electrical Engineering Department
Technion - Israel Institute of Technology, Haifa 32000, Israel
rakefet@tx.technion.ac.il ran@ee.technion.ac.il

ABSTRACT

Kin is an asynchronous processor architecture designed for future technologies enabling one or more billion transistors per chip and extremely fast processing (e.g., as predicted for 2012). This huge resource is exploited for aggressive *avid execution*, where a large number of instructions (hundreds per cycle) are prefetched and executed speculatively, in order to reduce the penalty of stalls due to branch mispredictions and dependencies, and to yield a very aggressive rate of successfully completed instructions (tens of instructions every cycle). Unneeded instructions are removed efficiently and non-preemptively, under control of a *pruning* mechanism. A multi-ported, wide bandwidth decoded instruction cache, wherein each line is a program basic block, is employed to feed this voracious machine, and a multi-path prefetch unit generates multiple cache accesses each cycle. Instructions are fully identified with *Dynamic Instance Tags* and move about the processor as independent entities. *Kin* supports *multi-execution*, where multiple paths, threads and processes are all executed simultaneously out of order. The processor has been designed using statecharts, and has been simulated running the SpecInt95 benchmark. We conclude that such complexity, which seems necessary for very high performance computing, is best achieved with an asynchronous architecture.

Keywords

Asynchronous architecture, avid execution, pruning, dynamic instance tag, multi-execution

1. INTRODUCTION

Microprocessor performance has risen over the past 20 years from 500 KIPS to 300 MIPS. The industry plans to achieve 100 BIPS by the year 2012, through the integration of almost one billion transistors on a chip operating at close to 10 GHz [24, 30]. This explosive growth in performance has been made possible thanks to the rapid development of semiconductor technology [33], and improvements in architecture.

ICS 98 Melbourne Australia

Technological progress has contributed both to higher clocking frequencies and to growing levels of integration. As more transistors were integrated, more architectural features (pipeline, superscalar processing, out-of-order execution, caches, etc.) were introduced into microprocessors, contributing to their growing utility.

This impressive growth is expected to continue in the future [24], but designing a synchronous, single clock microprocessor will no longer be feasible: The basic axioms of synchronous design are intended for limited equipotential domains (where signal propagation times over all wires are negligible). In future large chips it will take any signal (clock or data) many clock cycles to propagate from one part of the chip to another. While the electromagnetic field travels in vacuum at the speed of light (c = 30 mm / 100 pSec, in VLSI terms), the electric signals inside chips progress about 10-100 times slower, depending on drive strength and on the capacitive load of the bus. Let's assume $c/20$ signal propagation speeds (clock and data); given a chip size of 30mm in 2012 technology [24], typical signals will require 3 nSec to cross the chip end-to-end. If the chip is clocked at 2 GHz, about 5-7 clock cycles may be required for signal propagation alone. As a result, it will no longer be feasible to separate the logical and physical design of the pipelines, as is done today; rather, today's wire buses will be transformed into explicit pipeline stages, whose only task is to move data around, and the number of stages per bus will depend strongly on where the various modules are placed on the VLSI chip. To make the situation even worse, the signal may arrive at the various receivers on multi-drop buses at different cycles. Other effects of technological progress on processor speed relate to clock distribution. Several cycles may be required to propagate a single clock transition over the entire chip, compared to less than a cycle today. A worse aspect of this is that many transitions will be present simultaneously on the clock distribution wires. While this wavefront superpipelining is not impossible, it is highly undesirable. Optical clock distribution employing pulse lasers and optical detectors/amplifiers distributed over the chip may provide a solution. Clock jitter and skew are also expected to present great difficulties. Skew is the result of in-die variations in physical parameters and jitter is caused by temporal variations in temperature and voltage and by crosstalk. Both types of variations may hamper correct operation and typically they are contained at the expense of power dissipation. Thus, the power dissipated for clock and data distribution alone in complex VLSI chips increases faster than the increase in clock frequency and integration levels [10, 11, 28]. In one reported case [1] over 40% of the power budget in an Alpha chip is dissipated by the clock distribution network in order to limit clock skew and jitter. This ratio is expected to grow even higher, when processors are predicted to dissipate more than 100W [24].

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

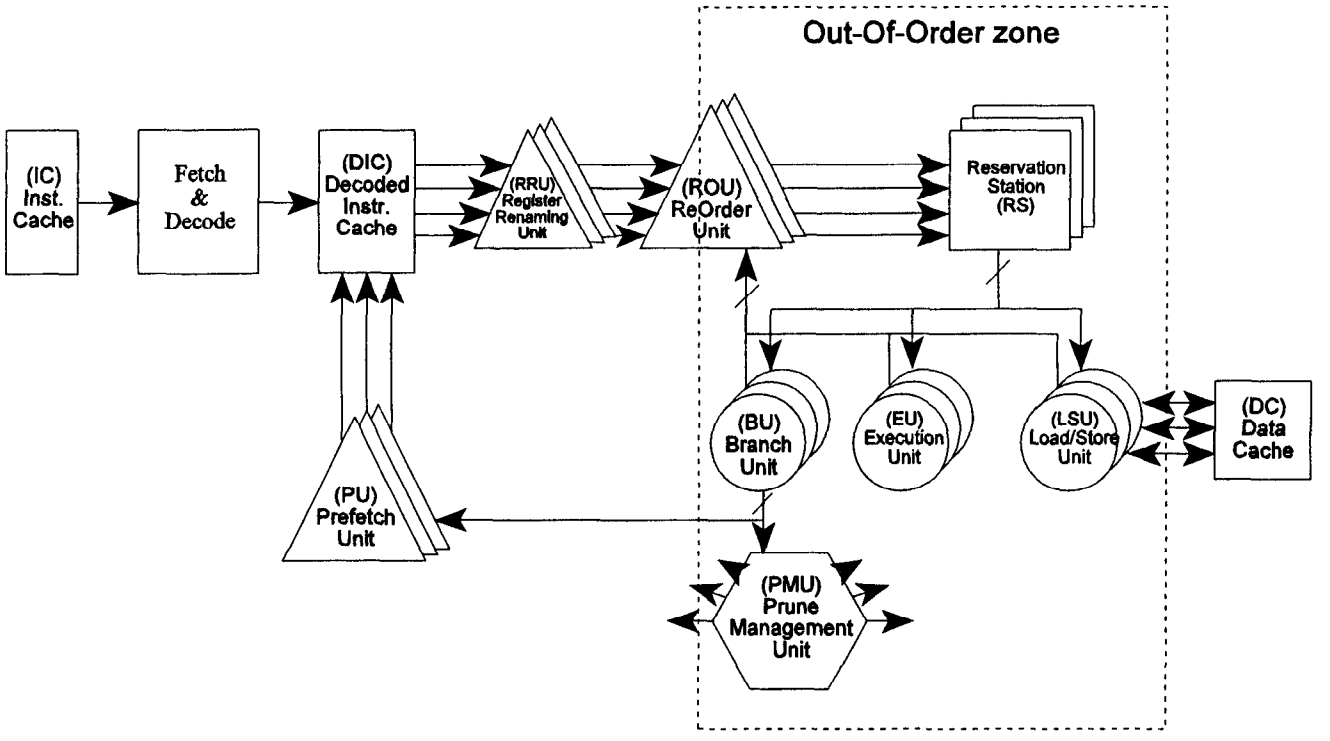

Figure 1: *Kin* asynchronous processor architecture.

In simple electrical engineering terms, the processors of the future will transcend from lumped systems into distributed ones. Modern processors have introduced some elements of distributed computing, such as decoupling modules with FIFO buffers and executing out-of-order. This trend is expected to continue towards more distribution of the various processor components. We have found that asynchronous architectures are a natural fit for such distributed systems.

This paper explains how an asynchronous processor architecture is most suitable for meeting the technological and architectural constraints of future technology, such as forecast for the year 2012 and beyond, when CMOS feature size is around 0.07μ, close to one billion transistors are integrated on a single chip, and the clock (if used) operates at close to 10 GHz. The paper describes a processor architecture for the asynchronous future, including a novel aggressive speculative execution method (necessary for high speed and suitable for asynchronous processors).

A large number of asynchronous processors have been previously designed [20, 7, 22, 2, 23, 4, 21, 27, 5, 31, 6, 19]. Most of them have rather simple and straightforward architectures. None of them supports out-of-order execution, nor considers performance enhancement by an advanced 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 top level, its individual modules may be implemented according to various timing disciplines (synchronous, asynchronous, or anything in between).

The novel architecture of *Kin* is described in Section 2. Avid execution is presented in Section 3, and Section 4 explains instruction pruning. Multi-execution is discussed in Section 5. Modeling *Kin* and its performance simulation (using the SpecInt95 benchmark) are described in Section 6. In Section 7 we argue that the complexity of *Kin*, as well as the technological constraints, call for an asynchronous architecture.

2. *KIN* ARCHITECTURE

2.1 General Description

Kin is a general purpose high performance 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. The instruction set combines both RISC and CISC instructions; each one is decomposed by the decoder into (one or several) internal simple micro-operations (μOps).

Kin comprises a distributed network of asynchronously interconnected modules, without any central control. Each module operates at its own speed, and communicates with other modules over asynchronous channels. FIFO buffers over those channels decouple the processor. On average all modules are balanced, but asynchronous interconnects permit flexible work loads.

While in contemporary (synchronous) processors the collective knowledge about an executed instruction is distributed among the pipeline stages, the controller, the buses and the registers, this location-dependent distribution of data and control information is unmanageable

in large distributed systems like *Kin*. Rather, instructions flow through the system as self-sustained packets carrying their own identity tags and all needed information. 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. This model resembles the *data flow* architectural concept.

The *Kin* architecture is described in Fig. 1. It combines many known features, like multiple execution units, out-of-order execution, and register renaming, and 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.

2.2 *Kin* Architecture and Operation

Once fetched, instructions are decoded and stored in the 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.

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

Figure 2: Dynamic Instance Tag structure.

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. 3). The pathmark is attached to each instruction as it is fetched from the DIC, as part of a unique ***Dynamic Instance Tag*** (DIT, Fig. 2). The same basic block of code (or part thereof) may be fetched simultaneously multiple times. Consider a simple loop which ends with a conditional branch. Each time we reach that branch, we should most likely prefetch the same loop again. 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 (Sect. 5).

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 renaming process replaces architectural register names by virtual ones, to filter out false dependencies. The condition codes are treated as one of the registers and are renamed accordingly. A new physical entry in the ReOrder Buffer (ROB) is allocated for each μOp destination (architectural) register. This entry number serves as the virtual name of the destination register. The μOp source registers are renamed according to the last name allocated to them on the same path, or their ancestor's path in the same execution tree.

The **ReOrder Unit** (ROU) manages out-of-order execution in *Kin*, and enforces in-order committing of instructions, whereby results are written back into architectural (real) registers and into memory. A ReOrder Buffer (ROB) is used in the ROU to keep track of the instructions from the many possible avid execution paths. The ROU maintains binary tree of paths rather than just a linear sequence of instructions. It also maintains the architectural registers, and employs them whenever possible to provide operands to fresh μOps. After commit, ROB instruction entries and RRU allocations are released.

Instructions wait in the several **Reservation Stations** (RSs) for their operands. Operand values may arrive from the various execution units or from memory. Once ready, instructions are routed by a scheduler to one of several **Execution Units** (EUs). Execution results are distributed to the ROB for committing, and to all RSs, wherein other instructions might be waiting for them.

The **Load/Store Unit** (LSU) handles memory access and bypass. It takes advantage of the locality of references of data access. While being similar to a data cache, it is designed as an independent smart associative table that tracks load and store operations. Ordering is enforced only when true dependencies are encountered, to guarantee correctness: For instance, 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). Similarly, Load instructions can bypass Store instructions, except for Store to the same address, in which case the argument is forwarded from the Store instruction. Thus, the LSU can return values even before they are physically written to memory or data cache. Giving higher priority to Loads over Stores can increase the issue rate of instructions, because Loads generate operands for successive instructions, while Stores can wait without stalling any other instructions. Loads can be executed speculatively without affecting correct operation. However, Stores can only be done at commit, at which time it is known that the Store is on the actual true path of the program.

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 behead messages, to be described in Sect. 4 below.

3. AVID EXECUTION

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 the instruction level parallelism. To exploit instruction-level parallelism in full, the processor must search over a large window for instructions that can be executed. That window typically extends beyond multiple branches, since on average every fifth instruction is a branch. This can be done by using various branch lookahead strategies. Most of them are based on certain branch prediction algorithms, and on speculatively executing instructions 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.

As explained in Sect. 1, the advent of technology is expected to permit the integration of huge resources on single chip processors. We propose to apply those resources, within an asynchronous architecture, to a dynamically adjustable, speculative *Avid Execution* in order to reduce the misprediction penalty. The principles will become clear after we survey briefly the existing approaches.

Traditionally (before speculative execution), conditional branch instructions stalled the processor pipeline, since it was unclear which instruction to fetch until the branch was executed. Early RISC processors employed delayed branches to avoid the stall, with limited improvements. Next, branch prediction has been invented, whereby each branch is predicted as either taken or not-taken, based on its past history [3, 9, 16, 32]. Instructions are fetched (speculatively) from the predicted branch target (**Single Path Speculative Execution**), before the actual branch has executed. If the prediction is correct, processing continues normally. On a misprediction, however, the pipeline is flushed and the correct path is fetched. *Misprediction Penalty* is the time required for the pipeline to fill up after a flush, until instructions start to commit again. This time depends linearly on the pipeline depth, measured in the number of stages between the fetch and branch resolution stages.

Out-of-order (*ooo*) execution allows the execution of later instructions if they are independent of former ones. Although not directly related to speculative execution, the latter helps increase the availability of instructions for *ooo* execution. On the down side, if all instructions are executed at a higher rate, so do branch instructions, and consequently the misprediction rate is also increased. This adverse effect is compounded by another setback: The deeper the processor and the higher the parallelism, the higher the misprediction penalty. Note that in order to analyze processor performance, both measures (misprediction rate *and* penalty) must be observed.

Current branch prediction algorithms are p=85-95% accurate [9]. For p=90%, every tenth branch is mispredicted. Since the average basic block length is five instructions, misprediction can be expected every 50 instructions. Single path speculative execution is highly sensitive to the quality of branch prediction and to pipeline depth. An execution tree of depth n contains n edges for single path speculative execution, so the cost is linear in the overall depth of prediction. However, the probability of correct prefetch over n levels falls off exponentially as p^n.

In **Eager Execution** all paths are prefetched and (speculatively) executed. When a branch is encountered, execution proceeds down both paths of the branch. Multiple resources are required to support the parallel prefetch and execution of multiple paths. Once a branch is executed, its 'losing' sub-tree may be aborted and disposed of, and the corresponding resources can be released. The principal benefit of eager execution is that misprediction stalls may be 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 should be discarded. If we consider an execution tree of depth n=5, then only about 25 instructions out of 155 will be committed, and this ratio grows exponentially. 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 [17, 18] is an attempt to implement eager execution with a limited tree depth. **Disjoint Eager Execution** [29] is another attempt to combine the benefits of eager execution and single path speculative execution methods. However, for high prediction accuracies it practically converges into single path speculative execution.

Avid Execution combines the benefits of both single path speculative and eager execution methods, such that the frequency of mispredictions 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 single path speculative execution, 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 3 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 single path speculative execution. 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 [15]. The unneeded instructions are pruned asynchronously, without preempting continuous operation of the processor, as described in Sect. 2 above.

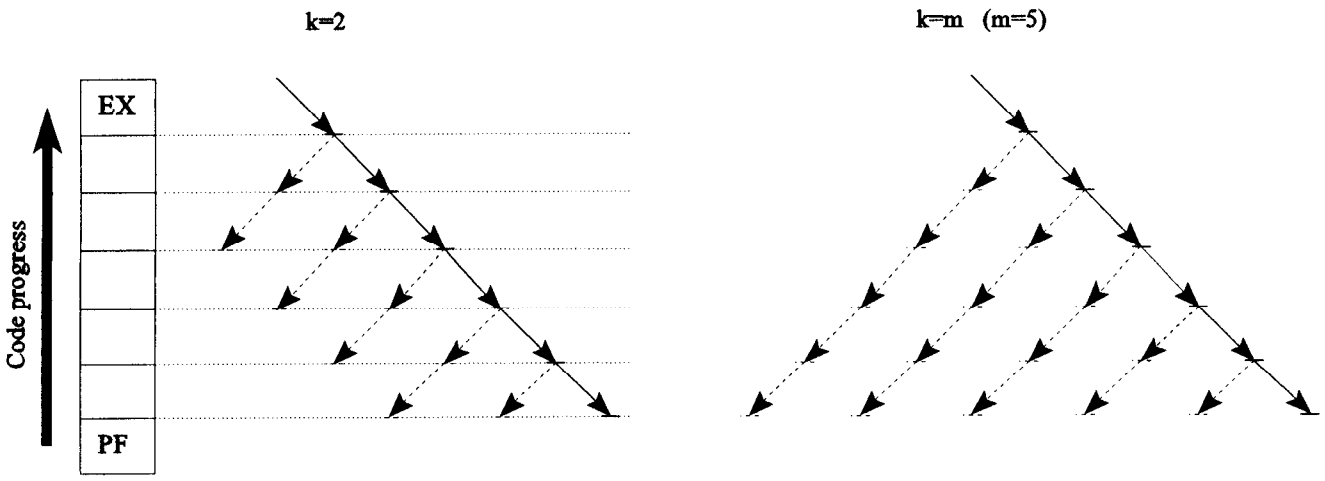

Figure 3: Examples of *Avid* Execution depth (k). m is the number of processor pipeline stages between prefetch (PF) and branch resolution (EX) stages.

Selecting the *Avid* depth can be done either statically (e.g., all conditional branches have the same alternative path depth), or dynamically. Dynamic adjusting of Avid depth can be done per each branch instruction, and can be based on statistics collected at run time. If confidence is applied to prediction [12, 26], 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 non at all) might be better. Obviously, k=0 for unconditional branches.

Observe that the first edge of each alternative path described in Fig. 3, originating from each branch instruction (a tree vertex), is the branch direction predicted as not being followed. The following edges of the alternative paths are selected by branch prediction. Other options are discussed in [15]. 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.

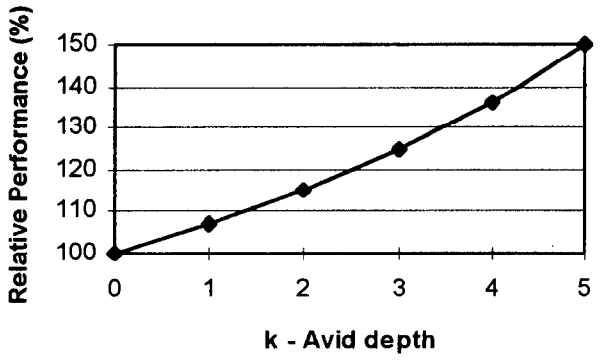

Figure 4: Average performance improvement achieved by various Avid depths (k), for m=5, p=0.95, and high bandwidth (execution limited by ILP).

Consider the following example of performance improvement achievable by Avid execution. The pipeline depth (measured in the number of stages, each stage handling a basic block, that fit between instruction fetch and the branch unit) is m=5, the accuracy of branch prediction is p=0.95, and sufficient hardware resources are available to execute all fetched execution paths, so that execution is limited only by ILP and mispredictions. Analytical studies [15] show that performance can be improved by as much as 50% under these conditions, as can be seen in Fig. 4.

4. INSTRUCTION PRUNING

Avid execution prefetches and executes both directions of each branch. Eventually, one of the two commits and the other must be pruned. As explained in Sect. 2, *Kin* performs the pruning clean-up tasks on the fly, without preempting execution, without stalling the processor, and without flushing the pipes. The pruning algorithm employs ***pathmarks*** to identify the doomed instructions.

Pathmarks are part of the DIT (Fig. 2) and distinguish alternative paths. Each edge of the execution tree is assigned a unique pathmark, based on prefix notation of binary trees. If an edge (a basic block, terminated by a branch instruction) is marked by m, then the sequentially following edge and the branch target edge are marked *m0* and *m1*, respectively (Fig. 5). The root is marked by the empty string. An instruction's pathmark is generated by accumulating these bits as a road map to follow from the root until the edge the instruction is on. Note that the marks of all edges in the (dashed) subtree of node n are prefixed by n. 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 the instruction of the subtree share the same prefix, a single *prune()* message suffices for the job. 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).

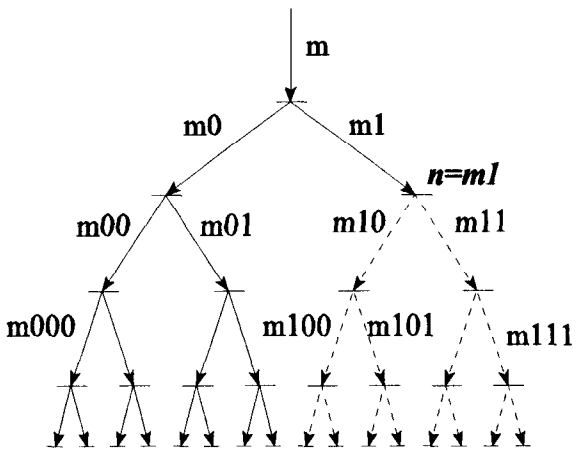

Figure 5: Pathmarks based on prefix notation.

The pathmark length grows very fast, as one more bit is attached on every branch. On the other hand, much of the information of the pathmark becomes irrelevant when processing progresses down the tree. Consider a mark *m*, L-bits long. Once a branch instruction marked *m* commits, the pathmarks of all useful subsequent instructions in the processor will be prefixed by *m*. Since the *m* prefix is now redundant, it is ***beheaded***. A distributed beheading algorithm is employed to contain pathmarks growth.

To behead prefixes, a *root mark R* is added to the DIT (Fig. 2). Once every *L* committed branches, a *behead(R,m)* message (*R*=path root, *m*=path prefix of *L* bits) is generated and distributed by the PMU. Following the receipt of such a message, each unit modifies each instruction it encounters as follows: If the instruction's DIT contains root mark *R* and pathmark prefix *m*, then the root mark is updated to *R+1* and the pathmark is left-shifted by *L* bits. In effect, linear pathmark growth is thus replaced by logarithmic growth of the root mark.

FIFOs can be used for storing prune and behead messages. New messages push older ones out, so that old and redundant messages are automatically discarded. Other issues regarding prune and behead messages, such as races, overflow, and resource allocation, are treated in [15].

5. MULTI-EXECUTION ON *KIN*

The mechanisms built into *Kin* to support concurrent superscalar execution of multiple paths (such as the DIT and *ooo*) are also directly applicable to concurrent execution of multiple threads and multiple processes. All four paradigms are unified under a single *multi-execution* model.

The DIT fully identifies, for each instruction, to which path, thread, and process it belongs. The various units of the *ooo* zone treat all instructions equally, regardless of their context. The only exception is the RS, which matches instructions to operands based on the DIT. The units outside the *ooo* zone, on the other hand, must treat each context separately. Consequently, *Kin* contains multiple copies of the PU, RRU, and the ROU (the DIC may or may not be divided by context). Each copy is dedicated to one context (thread and process), and the several contexts intermingle only upon entering the *ooo* zone. Each unit does handle all paths of the same context (execution tree), and hence they are drawn as triangles in Fig. 1. They employ associative memories to efficiently handle many instructions in parallel.

6. *KIN* MODEL AND PERFORMANCE SIMULATIONS

We have developed a software model of *Kin* and Avid execution, and simulated that model executing the SpecInt95 benchmark. A standard branch prediction algorithm [32] was implemented. Various prediction accuracies were obtained by changing the size of the branch target buffer (implemented as a 1-way set associative). The pathmarks were limited to 32 bits, and 16 bits were allocated for the root mark (of which at most 5 were used during the simulations). Beheading was issued every two branch commits.

The *Kin* model was specified at the high level using statecharts [8]. The internals of each module were specified as functions in C. At the higher level, statecharts control the *Kin* model and activate the C functions as needed. This formal and operational specification of *Kin* has enabled us to execute event driven simulations, which are suitable for asynchronous design. Modules react to messages arriving at their inputs, process the data and generate proper outputs. Handshake protocols and mutual exclusions are controlled and executed by statecharts. The interface between the statechart model and the C functions is based on handshaking protocols and regards the programs as self-timed modules. Each program may be assigned a (variable) delay at real-time [14].

Kin's model has a (partly) synthesizable specification: The parts defined by statecharts can be synthesized automatically into VHDL programs, and may be converted to asynchronous implementation afterwards [13, 14].

We have made extensive use of *Kin*'s model for debugging and performance evaluations of the architecture, and specifically for simulating the avid execution concept with various depths. Animation of the model helped us identify deadlocks, races and bottlenecks in earlier versions of the architecture. We used SpecInt95 traces for the simulations, and gathered information on average and worst case FIFO and table sizes, committing and pruning rates, and program execution times.

Avid execution was simulated for three possible (fixed) Avid depths: k=0, 1, and 2. The processor hardware 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. Some of the results [15, 25] are shown in Fig. 6.

It can be seen in Fig. 6 that R improves with k, although the improvement diminishes at very high values of p. Note that R is limited in these simulations by the limited ILP inherent in the SpecInt95 benchmark. All simulated results agree with our analytical analysis. A first glance at Fig. 6 reveals that, over all benchmarks, the incremental improvement of k=1 over k=0 is more significant than the additional

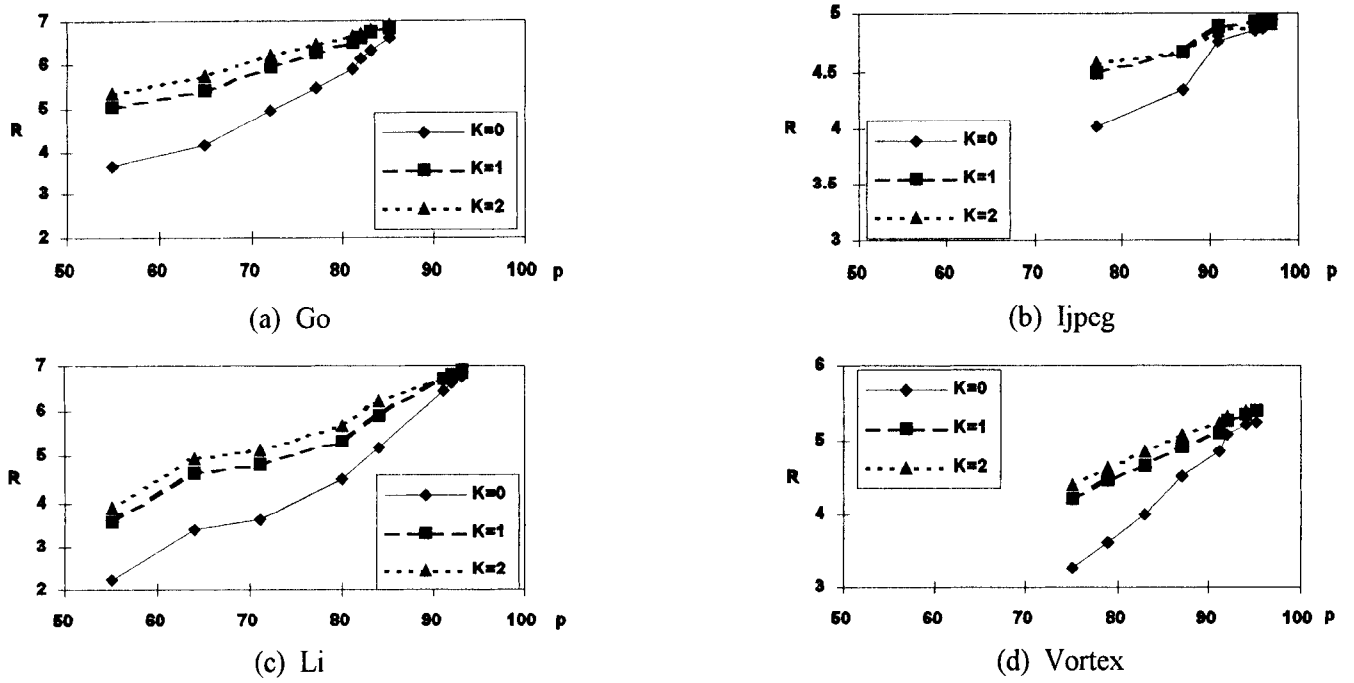

Figure 6: SpecInt95 simulation results (for w=40). The graphs describe the average execution rate (R) as a function of prediction accuracy (p), with Avid execution depth (k) as the parameter.

incremental improvement provided by k=2. The simulation of Go program (Fig. 6(a)), k=2 shows better performance than either k=1, or k=0, up to p=85%. Simulation of Ijpeg (Fig. 6(b)) 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. 6(c)), 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. Vortex (Fig. 6(d)) always resulted in best performance for k=2 (up to p=95%), while even k=1 was better than k=0.

7. WHY ASYNCHRONOUS *KIN* ?

The lessons of our study provide two principal reasons why *Kin* should be an asynchronous processor, one regarding the architecture and the other concerning technology constraints.

On the architectural side, *Kin* is a very complex and distributed processor, where the different parts perform very different tasks and their workloads vary significantly from one moment to another. *Kin* behaves more like a network than a well synchronized machine. Thus, although the individual modules may be implemented as synchronous circuits, at the high level the modules should be decoupled. As long as they are decoupled, they should be treated as components in an asynchronous system.

Avid execution is based on the premise that different components can efficiently process varying workloads. The dynamic instruction fetch process produces varying numbers of instructions each cycle. Unlike conventional ROUs, *Kin*'s must handle sparse lists of committing instructions, since the many instructions that are pruned leave 'holes' in the ROB. Self-timed modules tend to vary their speed depending on the data, and this flexibility must be accommodated for by other modules. Prune and behead algorithms are inherently distributed and operate with large variances, and do not need to be synchronized with normal processing. Variations are the results of varying program conditions such as changing branch prediction accuracies and switching contexts. A distributed asynchronous processor is clearly simpler to design and operate under such conditions.

On the technological side, *Kin* architecture is designed with very large chips in mind. The large size and the very high speed dictate that, if we were to employ a global clock, its wavelength would have been a fraction of the chip size. This is analogous to the operating conditions of any distributed computer network; *Kin* simply applies at the chip level the same solutions that are used for computer networks. This aspect has been further elaborated in the introduction.

8. CONCLUSION

We have described a very aggressive computer architecture and have explained that it should be designed as an asynchronous processor. The architecture is planned for future technologies enabling one billion transistors per chip and, if a clock were used, up to 10 GHz clocks (as projected by the SIA for the year 2012). A novel method of speculative *Avid* execution was introduced, which exploits massive parallelism to speed processing and bypass control and data dependencies. Asynchronous processing has led to smooth and efficient disposal of redundant computations (through pruning). Instructions are dynamically tagged and traverse the processor as independent entities in a data flow manner, leading to unification of several multi-context methods. High performance requirements were described for cache memory to support such architectures. A software model was constructed, and used for performance evaluation using the SpecInt95

benchmark.

ACKNOWLEDGMENT

This research has been funded in part by a research grant from Intel Corporation.

REFERENCES

[1] W. J. Bowhill, et. al., "Circuit Implementation of a 300-MHz 64-bit Second-generation CMOS Alpha CPU," *Digital Technical Journal*, 7(1), pp.100-115, 1995.

[2] E. Brunvand, "The NSR Processor," *Proc. of the 26th Annual Hawaii Int. Conf. on System Sciences*, Vol. 1, pp. 428-435, 1993.

[3] H. G. Cragon, *Branch Strategy Taxonomy and Performance Models*, IEEE Computer Society Press, 1992

[4] I. David, R. Ginosar, and M. Yoeli, "Self-Timed Architecture of a Reduced Instruction Set Computer," in *Asynchronous Design Methodologies*, S. Furber and M. Edwards editors, IFIP Trans. Vol. A-28, Elsevier Science Publishers, pp. 29-43, 1993.

[5] M. E. Dean, *STRiP: A Self-Timed RISC Processor*, PhD thesis, Stanford Univ., 1992.

[6] P. B. Endecott, *SCALP: A Superscalar Asynchronous Low-Power Processor*, PhD thesis, Dept. of Computer Science, Univ. of Manchester, 1995.

[7] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods, "A micropipelined ARM," *VLSI'93*, 1993.

[8] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, **8**(3), pp. 231-274, 1987.

[9] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd edition, Morgan Kaufmann, 1996.

[10] M. Horten, "The Hot New Star of Microchips (Pentium Microprocessor)," *New Scientist*, **138**(1871), pp. 31-34, May 1993.

[11] Intel Corporation, "Pentium Family User's Manual," 1994.

[12] E. Jacobsen, E. Rotenberg, and J. E. Smith, "Assigning Confidence to Conditional Branch Prediction," *Proc. of the 29th Int. Symp. on Microarchitecture*, pp. 142-152, Dec. 1996.

[13] R. Kol and R. Ginosar, "A Doubly-Latched Asynchronous Pipeline," ICCD'97.

[14] 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.

[15] R. Kol, *Self-Timed Asynchronous Architecture of an Advanced General Purpose Microprocessor*, PhD thesis, Dept. of Electrical Engineering, Technion, Israel, 1997.

[16] 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.

[17] N. F. Magid, *High Speed Computer Systems as a Result of Cuncurrent Execution of Sequential Instructions*, PhD thesis, Dept. of Electrical Engineering, Illinois Institute of Technology, Chicago, Illinois, 1980.

[18] N. Magid, G. Tjaden, and H. Messinger, "Exploitation of Concurrency by Virtual Elimination of Branch Instructions," *Int. Conf. on Parallel Processing (ICPP)*, pp. 164-165, Aug. 1981.

[19] A. Martin, *et al.*, "The Design of an Asynchronous MIPS R3000 Microprocessor," *Proc. Advanced Research in VLSI*, Sept. 1997.

[20] A. J. Martin, *et al.*, "The Design of an Asynchronous Microprocessor," Caltech-CS-TR-89-02, 1989.

[21] 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.

[22] N. C. Paver, *The Design and Implementation of an Asynchronous Microprocessor*, PhD thesis, Dept. of Computer Science, Univ. of Manchester, 1994.

[23] 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*, pp. 60-68, Mar. 1996.

[24] Semiconductor Industry Association, *The National Technology Roadmap for Semiconductors*, 1997. *http://www.sematech.org*

[25] H. Shafi, *Avid Execution and Instruction Pruning in the Asynchronous Processor Kin*, MSc thesis, Dept. of Electrical Engineering, Technion, Israel, 1998, in preparation.

[26] J. E. Smith, "A Study of Branch Prediction Strategies," *Proc. 8th Symp. on Computer Architecture*, pp. 135-148, May 1981.

[27] 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.

[28] D. Strassberg, "Cooling Hot Microprocessors," *EDN (European Edition)*, **39**(2), pp. 40-44,46,48, Jan. 1994.

[29] A. K. Uht and V. Sindagi, "Disjoint Eager Execution: An Optimal Form of Speculative Execution," *Proc. of the 28th Int. Symp. on Microarchitecture*, pp. 313-325, Nov. 1995.

[30] U. Weiser, "Future Directions in Microprocessor Design," Invited lecture, presented at *2nd Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, Mar. 1996.

[31] T. L. Wolf, *The A3000: An Asynchronous Version of the R3000*, MSc thesis, Dept. of Computer Science, Univ. of Utah, 1992.

[32] T.-Y. Yeh and Y. N. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction," *The 19th Int. Symp. on Computer Architecture*, pp. 124-134, May 1992.

[33] A. Yu, "The Future of Microprocessors," *IEEE Micro*, **16**(6), pp. 46-53, Dec. 1996.