Modeling Value Speculation

Yiannakis Sazeides
Department of Computer Science
University of Cyprus
yanos@ucy.ac.cy

Abstract

Several studies of speculative execution based on values have reported promising performance potential. However, virtually all microarchitectures in these studies were described in an ambiguous manner, mainly due to the lack of formalization that defines the effects of value–speculation on a microarchitecture. In particular, the manifestations of value–speculation on the latency of microarchitectural operations, such as releasing resources and reissuing, was at best partially addressed. This may be problematic since results obtained in these studies can be difficult to reproduce and/or appreciate their contribution.

This paper introduces a model for a methodical description of dynamically–scheduled microarchitectures that use value–speculation. The model isolates the parts of a microarchitecture that may be influenced by value–speculation in terms of various variables and latency events. This provides systematic means for describing, evaluating and comparing the performance of value–speculative microarchitectures.

The model parameters are integrated in a simulator to investigate the performance of several value–speculation related events. Among other, the results show value–speculation performance to have non-uniform sensitivity to changes in the latency of these events. For example, fast verification latency is found to be essential, but when misspeculation is infrequent slow invalidation may be acceptable.

1 Introduction

Performance has been a driving force for microarchitecture research since the advent of the computer. One of the popular approaches used to improve performance is the application of hardware and software transformations to a program to increase its Instruction Level Parallelism (ILP). Fundamentally, true dependences limit the amount of ILP that can be extracted from a program. Two instructions exhibit a true dependence, or simply are *dependent*, when an output operand of one instructions is an input of another.

Dependences are typically divided into control and data dependences. Probably, the most primitive type of control dependence is the program counter (PC) dependence because every instruction depends on its predecessor's output program counter. However, most instructions modify the PC in a trivial fashion, i.e. they increment it by the instruction length. This length is fixed for many architectures (e.g. RISC) and for others is a function of the instruction type. In either case, the instruction length can be determined prior to execution. Processors effectively deal with the trivial PC–dependences by employing instruction caches that allow multiple consecutive instructions to be fetched simultaneously. This leaves control transfer instructions as the remaining control dependence problem.

For a control transfer instruction determining the next PC requires the execution of the instruction and this leads to a serialization in the execution. Prediction and speculation [4, 10, 36] have been proposed as a means for alleviating the impediments of control dependences by predicting the next PC of control transfer instructions and speculatively executing the instructions that follow them.

In addition to control dependences through the PC, there is also a serialization of execution due to data dependences through register and memory locations. These dependences were shown to be predictable [1, 22, 25] and suited to drive speculative execution [11, 22, 25, 35].

The predictability of register dependence values (value prediction) - the motivation for this work - and of other program information types (such as control and memory dependences) may suggest the existence of fundamental transformations that can reduce/eliminate predictable computation. The identification of such transformations represent promising directions for research. However, until such transformations are discovered and understood the performance of future processors may benefit by *value–speculation*: use of value prediction to drive speculative execution.

There is a plethora of work that reports on the promising performance potential of value–speculation. Value–speculation has been proposed and investigated for dynamically [3, 6, 8, 11, 13, 14, 17, 18, 21, 22, 28, 31, 42] and

statically [7, 26] scheduled processors, for distributed microarchitectures [23, 27, 30, 43], and with compiler assistance [12, 16, 41].

Although the above and other related work represent a significant body of knowledge, its focus was mainly directed towards higher accuracy predictors and compiler support/transformations to facilitate more effective value prediction. We argue that there has been insufficient investigation of the microarchitectural implications of value speculation. This is evident by the imprecision with which microarchitectures are described in value–speculation work. This can be attributed to the lack of a formalization for describing such microarchitectures. What's more, the implications of value–speculation on the latency of microarchitectural operations such as releasing resources and reissue, are rarely addressed. This can be problematic because the obtained results may be difficult to reproduce and assess their significance. Although ambiguity is expected for a new research subject, the amount of research invested for value speculation justifies the need for formalization.

This paper introduces a model (the central theme of the paper) for a systematic description of dynamically–scheduled microarchitectures that use value–speculation. The model defines the design space of the microarchitecture influenced by value–speculation in terms of various *variables* and *latency events*. The model can therefore provide a method for accurate evaluation and comparison of the performance of value–speculative microarchitectures. The model parameters are integrated in a microarchitectural simulator to evaluate the performance with value–speculation for several latency events.

1.1 Motivation

The most problematic issue with value–speculation work is the timing of microarchitectural events. In general, to avoid unnecessary stalls in a pipelined processor, future events are scheduled based on (expected) deterministic latencies of currently executing events. For example, when a register–to–register add instruction is issued, the control logic of a processor **anticipates** that this operation will complete within a deterministic latency. Consequently, instructions dependent on the *add* can be scheduled (and possibly issued) prior to the completion of the *add*. The anticipation approach is employed almost in every stage of a pipelined processor. Meeting the timing constraints for anticipation at each stage is a critical design issue since uncertainty can compromise correct functionality and long paths can slow down clock. In practice, when the timing can not be met this may result in an additional pipe stage or a pipeline stall.

Value–speculation may increase the critical path of one or more anticipation mechanisms in a pipeline. Although one can be optimistic about how value–speculation influences these latencies, it is important to study the performance as the latencies change. This view is not adopted in most previous value–speculation work since nearly all latencies are fixed. For instance, most papers assume one cycle minimum latency between misprediction and reissue. Furthermore, vague explanations are given for the assumptions and implications of the one cycle latency.

We believe it is paramount to establish how sensitive value–speculation performance is to the latency of various events and then consider mechanisms to achieve or approximate the desired performance. For motivation consider the example in Fig. 1 showing the pipelined execution of three instructions - *1*, *2* and *3* - with correct and incorrect prediction (the example is discussed in more detail in Section 4). Three execution models are considered - *Super*, *Great* and *Good* - each with different latencies for various events. The (in)sensitivity to the changing latencies is evident. This underlines the need for a methodical investigation of the microarchitectural events influenced by value–speculation: wakeup, selection, verification, invalidation, resource releasing, branch resolution and memory resolution. This paper attempts to propose a systematic framework for doing such exploration and examines the effect on performance for most of these events.

1.2 Outline

The microarchitecture used in the paper is described in Section 2. Section 3 discusses the design space of value speculation and reviews related work. Section 4 introduces a model for describing value–speculative microarchitectures. The experimental framework is outlined in Section 5. Results are discussed in Section 6. The conclusions are presented in Section 7.

2 Microarchitecture

This section describes a base-microarchitecture and then introduces the value–speculation microarchitecture used in the paper.

2.1 Base Microarchitecture

For baseline microarchitecture we consider an out-of-order superscalar processor based on the Register Update Unit [39] which unifies issue resources (reservation stations [40]) and retirement resources (reorder buffer entries [37]). In the text will refer to the unified issue/retirement structure as the instruction window. An entry in the instruction window will be referred as reservation station(RS).

As instructions are fetched, they are assigned entries in the instruction window according to the dynamic program order and remain in the window until they can be retired.

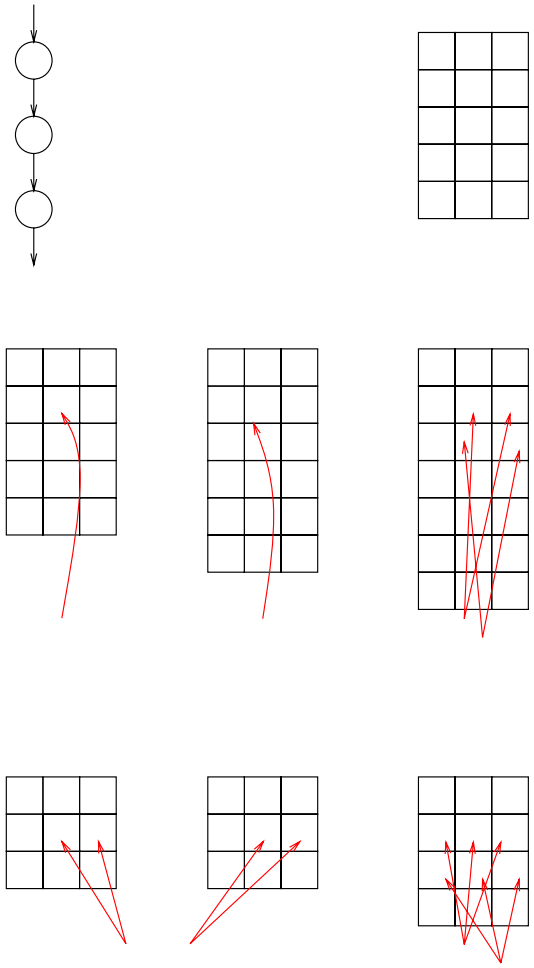

Figure 1. Execution example based on different Speculative Models

Values in this organization can exist in the following locations: register file, instruction window and functional units. The register file maintains architected state and is only updated when instructions retire. Instructions read any available input values from the register file before entering a reservation station in the instruction window. Instructions can also receive values through a bypassing network from predecessors in the window. Instructions with unresolved dependences monitor the results bus and capture their source operands as the producing instructions finish execution. The result of an executed instruction is also written in a field in its RS. When an instruction is the oldest in the window, it can commit its result in the register file or memory and release its entry in the window.
The important fields in a RS are shown below:

In1Ready	In1Tag	In1Value
In2Ready	In2Tag	In2Value
OutValue	Issued	Executed

The *ready* fields are used to indicate whether an input operand is available (*valid* or *invalid*). The *tags* are used to specify which results an instruction is waiting for. The *value* field holds the actual input value. The *outvalue* stores the output of an instruction that is eventually used for updating the processor state when the instruction retires. The two binary fields *issued* and *executed* are used to guide issue and retirement.

Wakeup and selection logic, in the instruction window, determines the instructions that get issued each cycle. *Wakeup* determines which instructions can be considered for issue in the next cycle. An instruction can *wakeup* when its *ready* fields are *valid* and has not issued already. *Selection* chooses which of these instructions to issue in the next cycle. The selection scheme used in this paper gives priority to branches and loads and then to the oldest instruction.

Memory instructions consist of two operations: address generation and memory access. Loads are executed when all preceding store addresses in the instruction window are known and hence no memory dependence violations can occur. A perfect load cache hit predictor is assumed, i.e. load dependent instructions are not issued when a load will miss in the cache. The only source of misspeculation in the base-processor is due to branch misprediction: when a branch is mispredicted all subsequent instructions in the window are squashed and fetching starts from the new target address.

2.2 Microarchitecture with Value Speculation

The base-processor is augmented with value–speculation as shown in Fig. 2. This is a similar microarchitecture to the one proposed in [21]. Essential to a processor supporting speculation are mechanisms that: (a) provide predictions (and confidence estimation for predictions), (b) verify predictions, and (c) invalidate misspeculated instructions. The integration of these mechanisms in a superscalar processor pipeline can lead to pervasive changes in the functionality and/or latency of different microarchitectural events.

One such change regards the handling of various values types. In particular, the use of value prediction introduces two additional types of values in the processor: *predicted* and *speculative*. A value is *predicted* if it is obtained directly from the value predictor, and is *speculative* if it is the result of computation(s) that included a predicted value. An input value is *valid* if it is read from the architected file or is the result of a computation that involved only valid inputs. Therefore, with value–speculation, an input operand may be: speculative, predicted, valid, and invalid.

Virtually all papers agree about the treatment of predicted, valid and invalid values, however there are two ap-

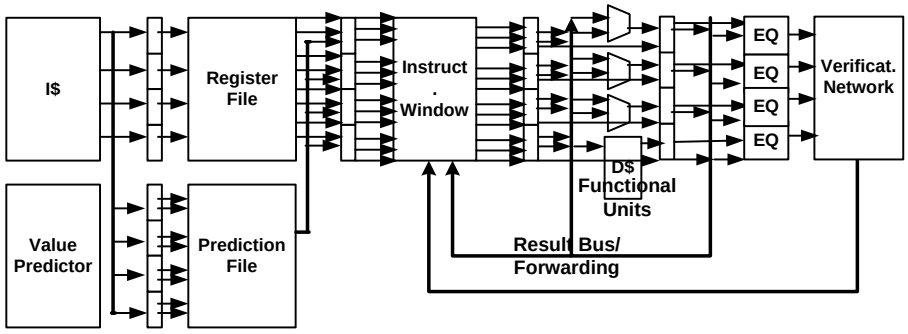

Figure 2. Pipeline With Value Prediction

proaches towards speculative values: one does not support forwarding of speculative values [31] whereas the other allows propagation of speculative values(the latter being the typical assumption in value speculation work). Although no forwarding can offer an implementation advantage, in this paper we choose to allow the forwarding of speculative values because it may represent the method with the highest potential.

To accommodate the new value types in the proposed microarchitecture, the ready fields in the RS are expanded to two bits. Another required change is for checking the correctness of output predictions: a field is added in the RS to indicate whether the output of an instruction is predicted and another field is used to contain the predicted output value. The modified RS incorporating the above changes is shown below (this is similar to the changes suggested by [31]):

In1Ready	In1Tag	In1Value
In2Ready	In2Tag	In2Value
OutValue	Issued	Executed
Predicted Value	*Predicted*	

The following section discusses the design space of various value speculation mechanisms and expands on the way the different RS flags can be used to guide various policies.

Although the rest of the paper emphasizes issues relevant to the microarchitecture outlined in this section, the discussion will often have a broader scope.

3 Design Space and Related Work

Value–speculation was proposed independently by Lipasti et al. and Gabbay and Mendelson [11, 22] as a performance enhancing method exploiting the predictability of values. Since then value–speculation has been a subject of a number papers. Although it is difficult to categorize published work, we divide value–speculation papers into two categories: those that address microarchitectural issues and those that do not. For example, a paper that introduces a value predictor may do so in a microarchitectural independent fashion. More relevant to this work are papers that made microarchitectural contributions or were conducive in better understanding the microarchitectural implications of value speculation.

The fundamental microarchitectural contributions related to value speculation were made by Lipasti et al. [20, 21, 22]. The authors introduced all basic microarchitectural functions required by value–speculation, namely: providing predictions, verifying correctness and invalidating in case of misspeculation. The notion of selective invalidation was also introduced in their work. In addition some of the effects of value–speculation on the microarchitecture were described.

Follow-ups to the above work helped define the design space of value speculation more accurately. However, the microarchitectures were described in an ad-hoc manner with little justification for the choice of various parameters. We realize that simulation models in previous work may have been detailed enough, what this paper contends is that their descriptions were not.

Statically scheduled processors can also employ value–speculation [7, 26]. And such microarchitectures may be less problematic to describe because value–speculation related operations can be done in software. Issues related to statically scheduled value–speculation are beyond the scope of this paper.

The following sections consider the design space of value–speculation. The purpose of the discussion is to elucidate possible misconceptions, identify subtle to describe/distinguish microarchitectural features and give direction for future research. Several issues related to prediction and predictors, such as prediction model, tables configuration, number of ports, hash functions and replacement are not discussed due to limited space. The interested reader can consult [2, 6, 18, 20, 31, 32].

3.1 Invalidation

Invalidation is responsible for informing the direct or indirect successors of a mispredicted instruction that they

have received incorrect operands. This is essential to recover from side effects caused by a misprediction. Invalidation latency can be crucial because it can determine how quickly a misspeculated instruction reissues. There are two basic invalidation models to consider: **complete** and **selective**.

Complete invalidation treats a value misprediction similar to a branch misprediction. Few papers [8, 41] compared the performance of selective and complete invalidation and observed smaller but still positive potential for complete invalidation. Complete invalidation may be practical to implement and beneficial in terms of performance if value mispredictions are rare and/or the potential of value–speculation is large. Value mispredictions can be made rare using confidence estimation[2, 8, 15].

Most papers adopt a hierarchical selective invalidation, that is an instruction can only invalidate its direct successors. The invalidated successors then invalidate their own successors. This process is repeated until all successors receive the invalidation. This may be build on top of the existing(or similar) tag broadcasting mechanism used to wakeup instructions[22, 30, 31, 42]. So far the only proposed design for selective invalidation requires instructions with speculative/predicted operands to remain in their reservation stations after they issue. This may be necessary as it is unclear how to perform selective actions in a pipeline otherwise. The following discussion assumes that invalidation is selective and all instructions with predicted/speculative operands remain in their RS after they are issued.

The invalidation mechanism with the highest performance potential is one that invalidates in parallel all direct and indirect successors. Selective parallel invalidation is effectively a flattened–hierarchical invalidation scheme. The model described in Section 4 considers invalidation as a distinct event that forces a misspeculated instruction to reissue.

3.2 Verification

When an instruction is predicted correctly, verification is responsible for informing the instruction's direct and indirect successors that their input operand(s) are valid. Since a verification mechanism has almost identical functionality with a selective–invalidation mechanism (Section 3.1), one mechanism may be sufficient to implement both.

Considering that an instruction may need to hold a resource until all its input operands become *valid*, then fast verification can be decisive for improving performance. Verification directly influences the release of *issue* resources (reservation stations) and *retirement* resources (reorder buffer entries).

Fast verification latency may also be relevant when it is desirable to resolve branch or memory instructions only with *valid* values. The problem with resolving branches using predicted/speculative values is that value prediction may be less accurate than branch prediction and hence may lead to additional branch mispredictions. Similarly, load instructions may be preferable to access memory with valid addresses to avoid store-load dependence violations. Slow verification may translate in a longer misprediction penalty for both branches and values.

There are at least four approaches for performing verification:

Hierarchical–Verification

With hierarchical–verification a correctly predicted instruction can validate only its direct successors. The verified successors will then verify their own successors. This process will be repeated until all successors get verified. Hierarchical–verification can be implemented using the existing, or similar, tag broadcasting mechanism used by processors to wakeup instructions. This verification approach can provide a performance improvement provided there are separate dependence chains in the instruction window. Otherwise, the increased execution parallelism from value–speculation will be nullified by the serialization in verification.

Retirement–Based Verification

It can be demonstrated that the retirement mechanism, used in dynamically–scheduled superscalar processor, can be used to *verify* in parallel multiple instructions. That is, verification can be overloaded to retirement. However, this approach may have two pitfalls:

(a) for each cycle only the w oldest instructions in the instruction window can be validated, where w is the retirement bandwidth of the processor. This may be undesirable if a younger instruction is otherwise valid but forced to hold needlessly a resource.

(b) the additional functionality may stress the critical path of the anticipation mechanism for releasing retirement resources. In particular, the condition for committing a value–predicted instruction requires an additional compare operation. If the latency of the comparison is on the critical path then resources may be freed with at least an extra cycle delay.

Hybrid Retirement–Based and Hierarchical Verification

This approach attempts to build on the strengths of the two previous approaches. *Retirement–based* verification is used for releasing resources faster whereas *hierarchical* verification is intended for faster detection of mispredictions.

Flattened–Hierarchical Verification

In this scheme all direct and indirect successors of a correctly predicted instruction are validated in parallel. This is analogous to **flattened–hierarchical** invalidation. The *flattened–hierarchical* verification represents the verification method with the highest performance potential, however it is also likely to be the method with the highest implementation cost.

It is noteworthy that a microarchitecture may require a

different verification approach depending on: (a) whether the issue and retirement resources are unified, and (b) whether branch and memory instructions are resolved with speculative/predicted values. Due to space limitations we do not elaborate on this important topic.

This work considers a microarchitecture with unified issue and retirement resources. Branches and memory instructions are allowed to execute only with *valid* operands. For verification(and invalidation) the functionality of a flattened–hierarchical tag broadcasting scheme is assumed. It is also assumed that at any given point any number of instructions can verify/invalidate their successors. This is referred as the *verification network*.

Although the functionality of the verification network is intuitive to understand, its implementation may present a significant challenge. The objective of this work is to assess whether such mechanism is essential to achieve high performance with value–speculation. It is unclear if any previous work considers such a scheme. The proposed model (Section 4) will be used to determine how important fast verification(invalidation) is and how critical it is to inform quickly branch and memory instructions about the state of their input operands (e.g. when predicted or speculative operands become *valid*).

Parallel–verification was assumed in a number of papers, however, no previous work discussed it in detail. Hierarchical–verification is explained in [31]. The first work to explore the effects of value–speculation on branches was presented by Sodani and Sohi [38]. The authors compared the performance when branches are resolved with speculative/predicted values and when resolved only with valid values. That work also considered the effects of 0 and 1 cycle validation latency. However, it is unclear how the assumed latencies affect various processor events (such as releasing resources). Also, it is unclear whether additional latency, to the verification latency, is considered when branches are not allowed to be resolved with speculative/predicted values. In previous work [19, 21, 29, 30] branches were resolved out-of-order only when their operands were known to be non-speculative. A third option for dealing with branches is to combine the two approaches examined in [38] based on confidence [9]. A recent study [28] considered the combination of address, value and dependence prediction for resolving speculative loads.

The above suggest that evaluation and design of different verification mechanisms present interesting directions for future research.

3.3 Equality

Equality is responsible for determining whether a value is predicted correctly or incorrectly. Equality can be performed by comparing the predicted value against the actual value (**value–equality**). As far as we know, value-equality is the approach used for checking predictions by all proposals that rely on prediction and speculation. Alternatives that do not require strict equality have been suggested but have not been explored[20, 32]. In the proposed microarchitecture (see Fig. 2) value-equality is performed in the write/verification stage using comparators (denoted EQ).

The latency for performing value–equality can be on the critical path of many microarchitectural operations, such as recovery from misspeculation and releasing resources, and hence is an important performance parameter. The proposed model (Section 4) considers the effects with increasing value–equality latency.

3.4 WakeUp

Without value–speculation an instruction can wakeup when it has not issued already and has received the tags for all its input operands – indicated by the ready fields being *valid* in its RS.

The use of value–speculation introduces additional choices for instruction wakeup. Wakeup may be useful to be seen as a filter that selects instructions for speculative execution. In particular, the wakeup function can consider the following information for an instruction: (a) the ready state of its inputs (ready fields of a RS), (b) issued state (issued field of a RS), (c) predicted state (predicted field of a RS), and (d) the speculative "state" of its inputs (this is a value that comes from the verification network and *roughly* corresponds to the next ready state for an input operand).

The options described above imply a number of possibilities as to when to wakeup instructions. Only one wakeup function is considered in this paper and allows for an instruction to wakeup only when its inputs are either valid and/or speculative and the instruction has not yet issued. Due to space limitation we do not report the precise function used.

Wakeup serves another purpose: nullifying the effects of a misprediction. The semantics for nullification are: (a) remove the effects of previous execution of an instruction, and (b) enable a future wakeup of the instruction. This is achieved by resetting the issued field in the RS whenever an instruction gets *invalidated*.

Note that the above policies ensure that instructions without predicted or speculative operands can wakeup as fast as on the base-processor.

Sodani and Sohi [38] performed a comparison of two wakeup schemes: wakeup each time a new value is reaching an instruction[30] or limiting the wakeup of an instruction to at most two executions[22]. The two approaches have a subtle but important difference, the former effectively ignores speculative status of the operands and hence may reissue faster a misspeculated instruction. This also implies that instructions may issue needlessly when they are

not misspeculated.

Considering the numerous possibilities that exist for wakeup functions, future work should investigate their performance and design.

3.5 Selection

Selection with speculative execution presents new tradeoffs since instructions can be candidates for issue with: predicted, speculative and/or valid inputs. Therefore *selection* needs to consider additional information such as, how many speculative inputs an instructions has, confidence in those inputs and whether the instruction is on the critical path. Selection of which value–speculative instructions get issued based on confidence and critical path information was proposed by Calder et al.[8]. This was also addressed more recently by Larson and Austin[16].

A selection scheme combined with accurate confidence estimation can ignore the speculative state information of an operand since most of the time predictions will be correct. Alternatively, a selection with a poor confidence estimator may assign higher priority to all instructions with valid inputs and then consider instructions with speculative operands.

In this work we consider a selection scheme that assigns highest priority to branch and load instructions and prioritizes the rest based on dynamic program order - oldest first. Non-speculative instructions are preferred over speculative. Selection for speculative execution is an important research subject not explored in this paper.

3.6 Confidence

Accurate confidence estimation[15] may be necessary to reduce value misspeculation. A number of interesting possibilities for confidence estimation for value prediction are discussed in [2, 8]. Calder et al.[8] explored the use of confidence levels (low, medium and high) for resetting counters. They also proposed propagation of confidence to dependent instructions. They have shown that speculating on predictions with low confidence that lie on the critical path can improve performance. Bekerman et al.[2] suggested to associate with a mispredicted instruction part of the control flow history that lead to it. In the case of future match, a prediction is assigned low confidence. In this paper we compare the performance with oracle and realistic confidence (Section 6).

4 Model for Value Speculative Microarchitectures

The norm in previous work is to describe ambiguously the microarchitectural implications of value–speculation. It is also common not to state assumptions or offer rationale for a value of a design parameter. The discussion in Section 3 illustrated that value–speculation may cause pervasive changes on a microarchitecture. Furthermore, subtle to describe but distinct microarchitectural features - with possibly different cost and complexity - may represent quite different or similar performance points (Section 6). We interpret this as a call for more formalized approach for describing value speculation microarchitectures.

We propose the combining of different microarchitectural mechanisms(variables) that influence speculative execution under a single model called *speculative-execution model*. Consequently, when describing a speculative execution the following information should be provided:

- a specific list of variables and their values, and
- manifestations of speculative execution in terms of latency between different microarchitectural events.

Below we summarize the design space of speculative execution models in terms of model variables and some of their possible values. The value(s) determine a method for implementing the mechanism corresponding to a variable.

Model Variable	**Values**
WakeUp	ready fields, issue flag, predicted-state, confidence
Selection	ready fields, instruction type, confidence
Branch Resolution	speculative or valid operands
Memory Resolution	speculative or valid operands
Invalidation	complete, selective (hierarchical or parallel)
Verification	hierarchical, parallel (based on retirement or dedicated network)

A model manifests itself in terms of at least the following **latency variables** that describe the latency required between microarchitectural events influenced by speculative execution. The latency variables are defined from the end of the first event to the end of the second event and should be given in terms of cycles:

Execution – Equality, latency required to determine if the prediction and a computed value are equal assuming equality is performed immediately after an instruction finishes execution.

Equality – Invalidation, latency required for an instruction to be invalidated after it was determined that a predecessor's prediction was incorrect. This may not be a fixed number of cycles (for example, the latency of selective serial invalidation is a function of the dependence chain invalidated).

Equality – Verification, similar to the previous but for correct predictions.

Verification – Free issue resource, latency after an instruction is verified before it can release its reservation station

entry.
Verification – Free retirement resource, latency after an instruction is verified before it can release its reorder buffer entry.
Invalidation – Reissue, latency after an instruction is invalidated before it can reissue.
Verification – Branch, latency after verification of the inputs of a branch before a branch can issue.
Verification Address – Memory Access, latency after verification of a speculative address generation before issue to memory.

These latencies are not all relevant to every speculative execution model. The latency for branch and memory instructions is pertinent if these instruction types are not allowed to be resolved based on speculative values. The verification – free issue resource latency is not relevant when instructions issued with predicted and/or speculative values do not retain their reservation station. We note that the latencies for the events need not be given separately, for example instead of reporting Execution – Equality and Equality – Verification with two separate values they can be combined as Execution – Equality – Verification and described by a single value.

It is worthwhile to note that no previous work identified all of these microarchitectural events. To our best knowledge, this is the first paper that breaks misspeculation into three events: Execution – Equality, Equality – Invalidation, Invalidation – Reissue. Typically, misspeculation was treated as a single event with one cycle latency. In general, previous work may have overlooked value speculation events that may be performance critical.

Although we do not claim that the above are complete for describing speculative execution precisely, we believe that this is a more systematic approach that can mitigate the problems mentioned before. In the next section, we present several speculative execution models and vary some parameters to illustrate how they influence execution.

4.1 Example Speculative Execution Models

This section considers few speculative execution models with the following values for the model variables: instructions can wakeup based on id-tags and state-tags; selection is based on instruction types and dynamic order and considers speculative state (i.e. considers whether operands are predicted/speculative or valid); branches are always resolved based on valid values; memory instructions are not allowed to access memory with speculative addresses; verification/invalidation is based on the verification network. The specific choices were described with more detail in Section 3.

Three models, denoted **super**, **great**, and **good** are considered and defined as follows:

Latency Variable	**Super**	**Great**	**Good**
Execution – Equality – Invalidation	0	0	1
Execution – Equality – Verification	0	0	1
Verification – Free Issue Resource	1	1	1
Verification – Free Retirement Res.	1	1	1
Invalidation – Reissue	0	1	1
Verification – Branch	0	1	1
Verification Address – Mem. Access	0	1	1

When computation does not include predicted values, all models have behavior identical to the base-processor. Recall that because we have a unified issue/retirement structure, the latency to free(release) issue and retirement resources is the same. Also note that in the proposed microarchitecture, resources cannot be free earlier than a cycle following the completion of an instruction.

The *super* model is the most optimistic and the *good* model the most pessimistic. The difference between the *good* and the *great* is in verification/invalidation latency, from one to zero cycle. The *super* model has zero cycle verification/invalidation, zero cycle reissue latency - that can enable mispredicted instructions to execute early - and zero cycle latency to inform branch and memory instructions when their inputs become valid.

Fig. 1 illustrates the pipelined execution of three instructions using the three speculative execution models with correct and incorrect prediction. The figure also shows the execution without value speculation. For all seven scenarios the common initial condition is that the three instructions are in the instruction window. The various pipeline latency events are defined in the figure (EX for execution, W for write to RS etc). The three instructions, labeled 1, 2 and 3, form a dependence chain: 2 depends on 1 and 3 depends on 2.

The base processor requires 5 cycles to retire all instructions. For the misprediction scenario it is assumed that the outputs of 1 and 2 are mispredicted. The figure shows that the more optimistic a model is the more activities are packed in a cycle. For instance, for the Super model, at the beginning of cycle t+1 it is detected (effectively with zero latency) that the outputs of instructions 1 and 2 were mispredicted. At the same time the successors of the two instructions get invalidated (that will be instructions 2 and 3). Also at t+1, instruction 2 that consumes the output of instruction 1 is scheduled for reissue and starts executing. Instruction 3 wakes-up at t+1 and is scheduled during cycle t+1 to execute at t+2. In contrast, the *good* model detects the mispredictions early in cycle t+1 and instructions 2 and 3 get invalidated by the end of the cycle. During t+2 is determined that instruction 2 can reissue. Instruction 2 gets executed during cycle t+3. At t+3 instruction 3 wakes up and is scheduled to execute at t+4.

The most important observation from the example is that execution behavior appears to be sensitive to the model event-latencies. Note that for the *good* model, unlike *super* and *great* models, instructions with predicted outputs, but

not predicted or speculative inputs, still need to go through verification.

The models represent a spectrum of designs with variable degree of optimism regarding the different latencies and are only a *few* of numerous possible models. We believe that exploring the design space of different speculative execution models and understanding their requirements is essential for: (a) better comprehending how to design value–speculative processors, and (b) focusing research effort on performance critical issues. We evaluate the performance of the *super, great* and *good* models in Section 6.

5 Simulation Methodology

5.1 Parameters

To evaluate the performance of the different speculative execution models, a simulation study was performed for the microarchitecture presented in Section 2. The various model events were integrated in an out-of-order simulator. Provided the simulator is accurate, it offers the means to evaluate and compare the performance for various speculative execution models.

The simulator used in this work is a modified version of the out-of-order simplescalar simulator[5]. A gshare branch predictor[24] is used that hashes 16 bits of global branch history with the 16 lower bits of the branch PC to index a 64K prediction table. The branch predictor is updated with correct information following each prediction. Unconditional and direct jumps are always predicted correctly. Conditional branch targets are assumed to be predicted correctly as long as the branch direction is correct. The L1 instruction cache contains 64KB of instructions, with 32B per block, is 4-way associative and a hit requires 1 cycle. An ideal fetch engine is assumed: provided instruction references hit in the cache and branches are predicted correctly, then the fetch engine can read and align from multiple basic blocks in the same cycle. The L1 data cache has the same configuration as the instruction cache, however, has as many ports as half the issue width of the processor under consideration and its hit time is 2 cycles. A unified L2 cache that can hold 1MB of data and instructions is used. This L2 cache is 4-way associative, with 64B per block, 12 cycle hit and 36 cycle miss time. A load/store queue with size equal to the instruction window is used. Loads can receive a value from a preceding store in the queue in a single cycle. Wrong path instructions are executed and their side effects are modeled. There are no resource constraints except limited number of data cache ports. All simple integer instructions require one cycle to execute. Complex integer operations and floating point operations, depending on the type, require from 2 to 24 cycles.

Simulations were performed for all integer SPEC95 benchmarks(Table 1). The benchmarks were compiled using the simplescalar *gcc* compiler with *-O3* optimization. Speedup was calculated as a ratio of the performance of a configuration with value prediction to an identical configuration without value prediction. For average speedup calculation harmonic mean was used. Arithmetic mean was used for reporting average prediction rates so each benchmark effectively contributes the same number of predictions.

Benchmark	Input Flags	Dynamic Instr (mil)	Instructions Predicted (%)
compress	400000 e 2231	103	70.5
gcc	gcc.i	203	67.3
go	9 9	132	78.7
ijpeg	specmun.ppm	129	82.0
m88ksim	scrabbl.in	120	70.6
perl	modified train	40	63.9
vortex	modified train	101	61.9
xlisp	7 queens	202	61.7

Table 1. Benchmark Characteristics

5.2 Value Predictor and Confidence Estimation

This work considered the performance of value–speculation with a context-based value predictor[33, 34]. The predictor uses two tables. The first level (or history table) is indexed with the PC of the predicted instruction. An entry in the history table maintains the context - a hash of the most recent 4 values produced by the instructions that map to the entry. The context is used to index into the second table - prediction table - and read out a 32 bit prediction. We used direct mapped 64K entry history table and a 64K entry prediction table. Entries in the history table are always updated whereas the prediction table uses a one bit counter to guide replacement.

In addition to the tables used for prediction, a table is used for providing confidence estimation. A confidence table is indexed using the PC of the predicted instruction and contains resetting counters that are incremented by 1 on correct predictions and reset to 0 on incorrect predictions. A prediction is considered confident when the confidence value is at maximum. In the simulations we compare the performance of real confidence based on a confidence table and that of an oracle confidence. When real confidence is employed we assume a table with 64K entries with 3 bit resetting counters in each entry. No attempt is made to optimize the realistic confidence mechanism[2, 8], the intention is to observe general trends when in use. Future work should consider confidence in more detail.

One other predictor dimension considered is the effects of update timing on prediction. Results are presented when the value predictor is update immediately (I) after prediction

with the correct value, or delay updated at retirement (D). When delayed updating is used, the history table of the predictor is updated speculatively with the prediction.

6 Results

This section reports on the performance of three speculative execution models: *good*, *great* and *super* discussed in Section 4.1. The performance of the models was measured for three processor configurations with issue width/window size: 4/24, 8/48 and 16/96. Each configuration was studied for real (R) and oracle (O) confidence using delayed (D) and immediate (I) update timing (D/R, I/R, D/O and I/O). We report averages and do not show the individual benchmark behavior due to space limitations - the individual benchmark behavior is similar to the overall.

Fig. 3 shows the average speedup for the various models and different configurations. As expected, and shown in a number of previous studies, value speculation has the potential to improve performance. The benefits are increasing with larger issue width and window size. As it was argued in[13], wider processors expose more dependences and hence increase the potential of value–speculation.

Several important observations can be made: (a) the *good* model behaves significantly worse as compared to the *great* and *super* models; in some cases having worse performance than the base configuration, (b) there is no significant difference between the *great* and *super* models, (c) performance is much more sensitive to confidence than to the timing of updates.

The first observation underlines the importance of fast verification latency. The verification latency for the *good* model was 1 cycle whereas for the *great* and *super* models was 0. If 0 cycle verification is infeasible, is imperative to explore speculative execution models where verification can be varied between 1 and 0. The criticality of fast verification latency is underlined by the fact that even under immediate update and oracle confidence the performance can be lower than the base.

The small performance difference between the *super* and *great* models indicates that a cycle delay (a) for informing speculative branches and memory instructions that their inputs are (not)valid, and (b) for reissuing following misspeculation, are not critical to performance. Recall that branch and memory instructions are not allowed to resolve speculatively. The reason that the cycle delay may not be so detrimental is that when these types of instructions are predicted correctly they enable useful speculative execution of other instructions and hence their additional delay is not usually exposed. Also, the real confidence method used in this study, as we show next, allows very few misspeculations. Thus the quick reissue provided by the *great* model is underutilized. An interesting direction of future research

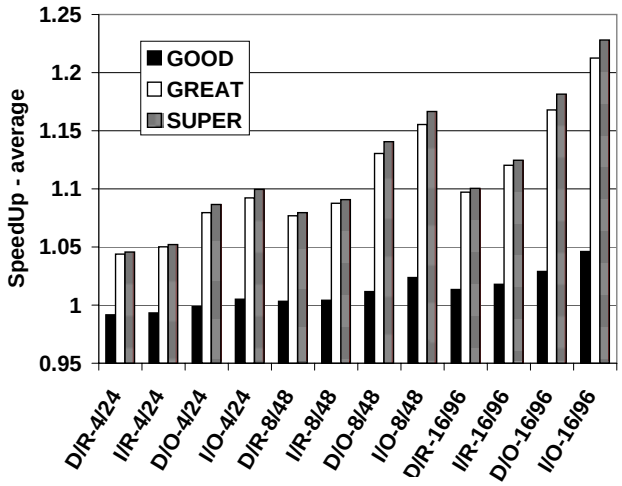

Figure 3. Speculative Execution Models Average Speedup

will be to study the above models with different confidence mechanisms. Specifically, we expect that with more frequent misspeculations, the relative difference of the *great* model will be more significant.

Another way to interpret the small performance difference between the *great* and *super* models is that accurate confidence can reduce the need for fast (and possibly complex) mechanisms for informing quickly speculative branch and memory instructions about the validity of their inputs. Recall that one of the reasons for using the verification network was to communicate quickly to the branch and memory instructions the speculative state (state-tag) of their inputs. The results indicate that this may not be important to performance. This fact, however, does not demonstrate that the verification network is not needed at all as it is also used to perform selective invalidation. An interesting direction of future research will be to investigate the relative importance of different verification schemes and determine how much additional benefit is provided by the verification network. Examples of schemes that can approximate the verification network include (in)validating successors up to a certain dependence chain depth, limiting the number of instructions that participate in a given (in)validation transaction, or limiting the period an instruction is not verified.

The data seem to support that confidence is a very significant performance parameter because moving from real confidence (X/R) to oracle confidence (X/O) provides a large performance increase (higher than the improvement achieved by immediate over delayed updating). This may indicate that either a lot of incorrect predictions are assigned high confidence and hence a lot of misspeculation, or many correct predictions are assigned low confidence and performance opportunity is lost. Prediction accuracy is considered

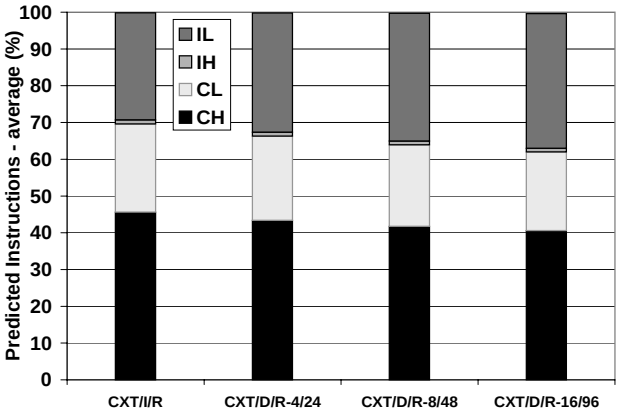

Figure 4. Average Prediction Accuracy

subsequently and reveals that the scheme based on 3–bit resetting counters performs poorly. This also suggests that one of the keys for realizing the potential of value prediction is accurate confidence predictors.

As for the effects of delayed updating on performance, the results suggest that more performance is lost, as compared to immediate updates, with increasing width/window. The reason is reduced prediction accuracy (discussed next).

Fig. 4 shows the average prediction accuracy for the great model. Predictions are divided into four sets, the set of predictions that were correct with high confidence (CH), correct predictions with low confidence (CL), incorrect predictions with high confidence (IH), and incorrect predictions with low confidence (IL). The total number of correct predictions is the sum of sets CH and CL. The results show that on the average 63% – 71% of the predictions are correct depending on the timing model and configuration used. The results suggest that context-based prediction is sensitive to timing and width/window size. Also it can be observed that with delayed updates and increasing width/window size prediction rate decreases.

One of the main reasons for lower accuracy, with increasing window size, is less constructive sharing of the prediction table among multiple instructions. With immediate updating, when two or more instructions produce identical sequences, one of the instructions can be mispredicted but is able to train the predictor immediately and, as a result, the other instructions get predicted correctly. Another reason for the sensitivity to update timing is any instruction that produces "almost" repeating sequences. Many of the correct predictions using context-based predictors are caused by instructions that are not 100% predictable. Immediately updating a predictor in the case of such instructions enables the context to point back to the correct sequence faster.

More interest to this work is the observation that the confidence method is successful in minimizing misspeculation (IH size is less than 1%), however this is done at the expense of a large set of correct predictions with low confidence (CL size is 20%–25% depending on the timing and configuration). This explains the large performance difference between real and oracle confidence and reinforces the importance of accurate confidence estimation for value prediction.

7 Conclusion

In this paper we argued that previous work did not describe systematically the effects of value speculation on a microarchitecture due to the lack of a formalized framework. We offer a discussion on the design space of value speculation to distinguish between subtle but possibly important design options, clarify misconceptions and provide research directions. The discussion is also used to underline the pervasive changes value speculation may require when integrated in a microarchitecture.

A model was introduced for a methodical description of microarchitectures that use value–speculation. The model isolates the parts of the microarchitecture that may be influenced by value speculation: wakeup, selection, verification, invalidation and resource releasing. The model describes the effect of value speculation on these parts of a microarchitecture in terms of various microarchitectural operations.

The model was integrated in a simulator that was used to investigate the performance of value–speculation. The results show value–speculation performance to have non-uniform sensitivity to changes in the latency of some events. For example, fast verification latency is found to be essential, but when misspeculation is infrequent slow invalidation may be acceptable.

8 Acknowledgments

James E. Smith is credited for supervising part of this work. Eric Rotenberg is acknowledged for influencing the author's views on microarchitectural issues. The original inspiration for this work was provided by Stamatis Vassiliadis. We thank the anonymous referees for their useful suggestions and constructive critique. Finally, the author is indebted to Pedro Trancoso and Toni Juan for proof–reading this manuscript and Francisco Jesus Sanchez for his comments on an earlier draft of the paper.

References

[1] J. L. Baer and T. F. Chen. An Effective on-chip Preloading Scheme to Reduce Data Access Penalty. In *Proceedings of Supercomputing*, pages 176–186, November 1991.

[2] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser. Correlated load-address predictors. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, June 1999.

[3] B. Black, B. Mueller, S. Postal, R. Rakic, N. Utamaphethai, and J. P. Shen. Load Execution Latency Reduction. In *Proceedings of the 12th International Conference on Supercomputing*, June 1998.

[4] W. Buchholz. *Planning a Computer System*. McGraw-Hill Book Company, NY, 1962.

[5] D. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-96-1308, University of Wisconsin-Madison, July 1996.

[6] M. Burtcher and B. G. Zorn. Exploring Last n Value Prediction. In *International Conference on Parallel Architectures and Compilation Techniques*, 1999.

[7] M. J. C. Fu, S. Larin, and T. Conte. Value Speculation Scheduling for High Performance Processors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[8] B. Calder, G. Reinman, and D. Tullsen. Selective value prediction. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, June 1999.

[9] Eric Rotenberg and Quinn Jacobson and James E. Smith. A study of control independence in superscalar processors. Technical Report CS-TR-98-1389, University of Wisconsin-Madison, December 1998.

[10] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[11] F. Gabbay and A. Mendelson. Speculative Execution Based on Value Prediction. Technical Report (Available from http://www-ee.technion.ac.il/fredg), Technion, November 1996.

[12] F. Gabbay and A. Mendelson. Can Program Profiling Support Value Prediction? In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 270–280, December 1997.

[13] F. Gabbay and A. Mendelson. The Effect of Instruction Fetch Bandwidth on Value Prediction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 272–281, June 1998.

[14] J. Gonzalez and A. Gonzalez. The Potential of Data Value Speculation. In *Proceedings of the 12th International Conference on Supercomputing*, pages 21–28, June 1998.

[15] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning Confidence to Conditional Branch Predictions. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 142–152, December 1996.

[16] E. Larson and T. Austin. Compiler Controlled Value Prediction using Branch Predictor Based Confidence. In *Proceedings of the 33nd Annual ACM/IEEE International Symposium on Microarchitecture*, 2000.

[17] S. Lee, Y. Wang, and P. Yew. Decoupled Value Prediction on Trace Processors. In *HPCA9*, 2000.

[18] S. Lee and P. Yew. On Some Implementation Issues for Value Prediction on Wide-Issue ILP Processors. In *International Conference on Parallel Architectures and Compilation Techniques*, 2000.

[19] M. Lipasti. *personal communication*, 1999.

[20] M. H. Lipasti. Value Locality and Speculative Execution. Technical Report CMU-CSC-97-4, Carnegie Mellon University, May 1997.

[21] M. H. Lipasti and J. P. Shen. Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 226–237, December 1996.

[22] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value Locality and Data Speculation. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, October 1996.

[23] P. Marcuelo and A. Gonzalez. Value Prediction for Speculative Multithreaded Architectures. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, 1999.

[24] S. McFarling. Combining Branch Predictors. Technical Report DEC WRL TN-36, Digital Western Research Laboratory, June 1993.

[25] A. Moshovos, S. E. Breach, T. J. Vijaykumar, and G. Sohi. Dynamic Speculation and Synchronization of Data Dependences. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 181–193, June 1997.

[26] Y. Nakra, R. Gupta, and M. L. Soffa. Value prediction in vliw machines. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, June 1999.

[27] J. Parcerisa and A. Gonzalez. Reducing Wire Delay Penalty through Value Prediction. In *Proceedings of the 33nd Annual ACM/IEEE International Symposium on Microarchitecture*, 2000.

[28] G. Reinman and B. Calder. Predictive Techniques for Aggressive Load Speculation. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 127–137, December 1998.

[29] E. Rotenberg. *personal communication*, 1999.

[30] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith. Trace Processors. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 138–148, December 1997.

[31] B. Rychlik, J. Faistl, B. Krug, and J. P. Shen. Efficacy and Performance of Value Prediction. In *International Conference on Parallel Architectures and Compilation Techniques*, October 1998.

[32] Y. Sazeides. An Analysis of Value Predictability and its Application to a Superscalar Processor. *PhD Thesis, University of Wisconsin-Madison*, 1999.

[33] Y. Sazeides and J. E. Smith. Implementations of Context–Based Value Predictors. Technical Report ECE-TR-97-8, University of Wisconsin-Madison, Dec. 1997.

[34] Y. Sazeides and J. E. Smith. The Predictability of Data Values. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 248–258, December 1997.

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

[36] J. E. Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 8th International Symposium on Computer Architecture*, pages 135–148, May 1981.

[37] J. E. Smith and A. R. Pleszkun. Implementing Precise Interrupts in Pipelined Processors. *IEEE Transactions on Computers*, 37(5):562–573, May 1988.

[38] A. Sodani and G. S. Sohi. Understanding the Differences between Value Prediction and Instruction Reuse. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 205–215, December 1998.

[39] G. Sohi. Instruction Issue Logic for High Performance, Interruptible, Multiple Functional Unit, Pipelined Computers. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.

[40] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11:25–33, January 1967.

[41] D. M. Tullsen and J. S. Seng. Storageless value prediction using prior register values. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, June 1999.

[42] G. Tyson and T. Austin. Improving the Accuracy and Performance of Memory Communication through Renaming. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, December 1997.

[43] Y. Wu, D. Chen, and J. Fang. Better exploration of region-level value locality with integrated computation reuse and value prediction. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, May 2001.