Formally Modeling Microprocessor Caches and Branch Predictors

Hans Vandierendonck[1] Jean-Marie Jacquet[2] Bavo Nootaert[1] Koen De Bosschere[1]
[1]Dept. of Electronics and Information Systems
Ghent University
Belgium
{hans.vandierendonck,bavo.nootaert,kdb}@elis.ugent.be
[2]Institute of Informatics
University of Namur
Belgium
jmj@info.fundp.ac.be

Abstract: - Microprocessors are subject to many hard constraints related to performance, power consumption, worst-case execution time, reliability, dependability, etc. Prooving any of these properties is currently nearly impossible. We believe that such proofs could be made if formal models of microprocessors were available. For these reasons, we formally model the operation of caches and branch predictors. These are structures that are present in virtually all microprocessors and have a high impact on the above mentioned system properties.

Key-words: - operational semantics, microarchitecture, cache

1 Introduction

State-of-the-art processors feature several 100 millions of transistors, yielding an extremely high level of complexity. As processors are composed of semi-independent parts (e.g. pipelines, instruction queues, caches, branch predictors) gaining insight is facilitated by studying each component in separation. However, each of these components gains in complexity as the research field matures. E.g. very few people really understand all the intricate details of modern branch predictors [2, 3, 5]. This inherent complexity hampers insight and may slow new research findings as well as wide-spread adoption of these techniques.

With high degrees of complexity, it becomes difficult to make hard claims about the performance of the structure. Hard claims are essential when reliability, dependability or real-time constraints are concerned. In such cases it does not suffice to show an average performance. Rather, it is required that one proves that a minimum performance will be obtained with a minimum guaranteed probability. For these reasons, we develop formal models of processor components. The goal of these models is to reason about these components and to formally prove their properties.

This paper presents a formal model of caches specified using operational semantics. Caches are well understood, which is why our first attempt is applied to caches: we have sufficient knowledge to debug our models and to steer the construction of the model in the right direction. We claim without proof that this model can also be readily applied to branch predictors as well as other types of predictors.

2 Basic Model

Following [4], we shall formally describe the computation of a program as a structured combination of elementary steps. In this approach, steps are characterized as moves from snapshots of the execution to snapshots of the execution. Such a snapshot varies from one language to another as well as from what is observed. In our context of memory structures, we shall consider snapshots as pairs composed of the content of the memory structure and of the sequence of instructions accessing this structure. Formally, the set of situations $Ssit$ is defined as follows

$$Ssit ::= Stable \times Ssinst$$

where the set of memory structures $Stable$ and the set of instructions $Ssinst$ are themselves defined below.

2.1 Notations

In order to do so, we first introduce some auxiliary notations.

For a set E, we denote by $E^{<\omega}$ the set of finite sequences of elements of E. The empty sequence is denoted by λ. The sequence obtained by prepending element e to the sequence S is denoted as $e.S$.

The set of booleans will be denoted by $\mathbb{B}$ and the set $\mathbb{B}^{<\omega}$ by $\mathcal{B}$.

Note that numbers in a binary representation have their least-significant bits at the head of the sequence, i.e., the decimal number 13 is represented in 2-complement binary notation as $(1.(0.(1.(1.\lambda))))$.

2.2 The Trace of Instructions

The instructions in a program are executed sequentially: each instruction operates on the processor state as it is left by the instructions executed before it. The sequence of executed instructions characterises the execution of the program. The presented methodology departs from this sequence or *trace* of instructions. Each instruction is identified here by three items:

- the instruction address namely the value of the program counter,
- the argument address namely the memory to be addressed in a load/store instruction or the branch target address in a branch instruction,
- additional information to be specified in specific context (e.g., the true branch direction).

At most two of these items are required at the same time. E.g., a data cache is accessed with the memory address and an instruction cache is accessed with the program counter. It suffices to have two fields, namely an identifier field that identifies the cell in the table that will be accessed, and the data field that represents the data that will be stored there. Depending on the structure that is modelled, we place different information in the identifier and data fields. Both items are essentially sequences of bits. We can therefore define their sets as $\mathcal{B}$. As a result, the set of instructions $Sinst$ is defined as follows:

$$Sinst = \mathcal{B} \times \mathcal{B}$$

As we shall consider sequential programs only, the set $Ssinst$ can be defined as follows.

$$Ssinst = (\mathcal{B} \times \mathcal{B})^{<\omega}$$

2.3 Tables

Structures are modelled as tables. A table has S rows and A columns. Each element in the table is composed of an address argument and a prediction information relevant to the type of prediction performed. We shall formally define such a table as a function that given a row number returns a sequence of information about the cells on that row of the table. The cell information is defined as a pair of sequences of bits. To make the framework simple, we define the structure as

$$Crow = (\mathcal{B} \times \mathcal{B})^{<\omega}$$

$$Ctable = \mathbb{N} \rightarrow Crow$$

with the understanding that if the given row or column exceeds those indicated by S and A then the undefined value $\perp$ is returned. Moreover, by abuse of language, we use $\perp$ to denote the table with all cells undefined.

The tables are operated by means of three functions. First an index function is used to determine which row of the table is affected by an instruction. Such a function is thus of type

$$Index = \mathcal{B} \rightarrow \mathbb{N}$$

Second, an output function determines the value read from the table based on the contents of the set:

$$Output = Crow \times Sinst \rightarrow \mathcal{B}$$

The output function either returns the value read from the memory or any value computed thereon (e.g., the prediction in case of a predictor). Third, an update function is used to modify the row as a result of the execution of the instruction. It is of type

$$\mathit{Update} = \mathit{Crow} \times \mathit{Sinst} \rightarrow \mathit{Crow}$$

Summing up, structures are characterized by six features: the number of rows (S), the number of columns (A), the contents of the cells (C), the index function (I), the output function (O) and the update function (U). This leads to the following formal definition of the set of structures *Stable*:

$$\mathit{Stable} = \mathbb{N} \times \mathbb{N} \times \mathit{Ctable} \times \mathit{Index} \times \mathit{Output} \times \mathit{Update}.$$

2.4 Operational semantics

Given the above formal definitions, the execution can be defined as sequences of small steps. The allowed small steps are characterized formally by the relation $\rightarrow$. Intuitively, $(X, R) \rightarrow (X', R')$ means that the computation moves from the state described by strucure X and sequence of instructions R to the new state described by structure X' and sequence of instructions R'. To capture misses, we introduce a label on the arrow: μ is used to denote a miss and ν to indicate no miss.

Formally, the relation $\rightarrow$ is defined as the smallest relation of

$$\mathit{Stable} \times \mathit{Ssinst} \times \{\mu, \nu\} \times \mathit{Stable} \times \mathit{Ssinst}$$

that satisfies the following properties:

$$\begin{aligned} &< (S, A, C, I, O, U), (\mathit{id}, \mathit{data}).R > \\ &\qquad \xrightarrow{\mu} \; < (S, A, C', I, O, U), R > \end{aligned}$$

$$\text{if} \left\{ \begin{array}{l} c = C(I(\mathit{id})) \\ c' = U(c, (\mathit{id}, \mathit{data}).R) \\ C' = C \text{ overridden with } I(\mathit{id}) \rightarrow c' \\ O(c, (\mathit{id}, \mathit{data})) = \mathit{data} \end{array} \right\}$$

$$\begin{aligned} &< (S, A, C, I, O, U), (\mathit{id}, \mathit{data}).R > \\ &\qquad \xrightarrow{\nu} \; < (S, A, C', I, O, U), R > \end{aligned}$$

$$\text{if} \left\{ \begin{array}{l} c = C(I(\mathit{id})) \\ c' = U(c, (\mathit{id}, \mathit{data}).R) \\ C' = C \text{ overridden with } I(\mathit{id}) \rightarrow c' \\ O(c, (\mathit{id}, \mathit{data})) \neq \mathit{data} \end{array} \right\}$$

The operational semantics

$$\mathcal{O} : \mathit{Ssinst} \rightarrow \{\mu, \nu\}^{<\omega}$$

is then defined as the sequence of labels produced during the computation: for any sequence of instructions R,

$$\mathcal{O}(R) = x_1 \cdots x_n$$

such that

$$<\perp, R> \xrightarrow{x_1} < CC_1, R_1 > \xrightarrow{x_2} \cdots \xrightarrow{x_n} < CC_n, \lambda >$$

3 Application to Caches

The model described in the previous section is general enough to define all caches and various predictors, such as branch predictors, value predictors, dependence predictors, etc. In this section, we apply the model to describe direct mapped and set-associative caches [6].

A cache is a small memory that holds the data or instructions that were most recently used by the processor. Because the cache is much smaller than the main memory, only part of the data can be stored in the cache at the same time. When a data item is requested by the processor, the address of the data is used to perform an associative search through the cache, i.e., the cache is searched for a *block* of data that is tagged with the requested address. Every block of data has the same size, namely B bytes, and its starting address is aligned (i.e., it is a multiple of B).

Either instructions or data can be fetched from the cache. Instructions are identified using the program counter, while data is identified using the memory address that is specified by the load/store instruction. Hence, the identifier field of the elements in the trace is either equal to the program counter (instruction fetches) or the data memory address (data loads and stores). The data field in the trace is always 1 to facilitate counting the number of hits and misses. The output function

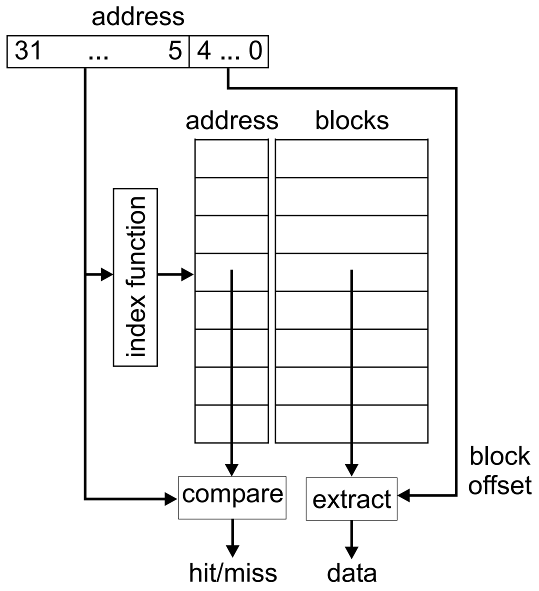


Figure 1. Indexing into a direct mapped cache with 32 byte blocks. It is assumed that addresses are 32 bits long.

returns a 1 on a cache hit and a 0 on a cache miss. This value is compared to the data field in the trace to select between a μ (cache miss) and ν (cache hit) transition.

3.1 Direct Mapped Caches

The direct mapped cache is organised as a table with S rows and one column. Every row can hold one block of data and also stores the address of that block (Fig. 1). To limit the search time, every block of data can be stored in only one row of the cache. This row is determined by the index function, that maps the address into the range $0 \ldots S-1$. When the row pointed to by the index function holds the requested block, then that block is read from the cache and the requested words are extracted from the block. If the data is not present in that row then the cache does not hold the data at all. It is subsequently fetched from the main memory and replaces the block in the designated row.

A direct mapped data cache with B-byte cache blocks and S sets is defined as:

$$C_{dm} = (S, 1, \perp, I_{cache}, O_{cache}, U_{dm})$$

It is assumed that S and B are powers of 2. The index function selects the row where the block is potentially stored. It has the responsibility to disperse active blocks of data equally over all sets of the cache, such that the cache is efficiently used. A commonly used index function selects the $log_2(S)$ lowest address bits of the block offset, i.e., the lowest $log_2(B)$ address bits are dropped and the next $log_2(S)$ bits are used as row selector. This method of indexing is so frequently used that we define the function $bitsel$ for convenience:

$$bitsel(id, blocksize, sets) = \rfloor id \div blocksize \lfloor_{log_2(sets)}$$

For numbers in a binary representation, dividing by a power of two corresponds to removing the $log_2(n)$ low-order bits, so we define $S \div n$ in terms of an operator on sequences. The auxiliary function $\rfloor S \lfloor_n$ truncates the sequence S to the first n elements. The indexing function is now given by Equation 1.

The output function returns a 1 on a hit and a 0 on a miss (Equation 2). The update function always overwrites the cell with the requested cache block. Hence, the row of the table contains the referenced cell (Equation 3).

3.2 Set-Associative Caches

Set-associative caches reduce miss rates by increasing the number of columns of the table. In order to keep the table size constant, the number of rows is proportionally decreased. This design introduces more freedom to place blocks: every block can be stored in every cell of the row indicated by the index function. The number of such cells is called the *degree of associativity* of the cache. When a block is loaded into the cache, then it has to be decided which cell in the target row will be overwritten with the new block. This task is the responsibility of the *replacement policy* and we model it here as part of the update function. The replacement policy is always kept simple: it is either a simplified variant of the LRU policy or a round-robin policy (similar to FIFO).

A set-associative cache with S sets, B byte blocks and a degree of associativity equal to A is defined as:

$$C_{sa} = (S, A, \perp, I_{cache}, O_{cache}, U_{sa})$$

The index function:

$$I_{cache}((id, data)) = bitsel(id, B, S) \tag{1}$$

The output function:

$$\begin{aligned} O_{cache}(\lambda, (id, data)) &= 0 & \\ O_{cache}((sid, sdata).c, (id, data)) &= 1 & \text{if } sid = id \div B \\ O_{cache}((sid, sdata).c, (id, data)) &= O_{cache}(c, (id, data)) & \text{otherwise} \end{aligned} \tag{2}$$

The update function:

$$U_{dm}(c, (id, data).R) = (id \div B, \perp).\lambda \tag{3}$$

Figure 2. Operational semantics of a direct mapped cache

where I_{cache} and O_{cache} are defined above for direct mapped caches. U_{sa} can be defined in various ways for set-associative caches.

A common update policy is the *least recently used* replacement policy (LRU), overwriting the block that was least recently referenced. Hereto, we place all blocks in the same row in a sequence, with the most recently referenced block in the first position and the least recently referenced block in the last position. Thus, when the referenced block is present in the cache, then it is removed from the sequence and inserted again at the front. If the block is not present, it is simply inserted at the front. The LRU policy is defined by Equation 4 where the auxiliary $delete(a, S)$ deletes all occurrences of the element a from the sequence S, while leaving all other elements in their original order.

The *first-in first-out* policy (FIFO) overwrites the cell that was least recently loaded. The cells in the sequence for one row of the table thus matches the order that the blocks were loaded. If a block is referenced and it is present in the cache, then the order of the blocks is unchanged (Equation 5).

4 Application to Branch Predictors

The goal of branch prediction is to speed up the execution of a program by speculating on the outcome of a branch. The instructions that logically come after the branch are executed speculatively, i.e., it is not yet known whether they really do have to be executed. When the branch target was predicted correctly, a speedup is thus achieved. If the branch prediction is incorrect, then the speculatively executed instructions are discarded.

A distinction is generally made between *conditional branch prediction* and *branch target prediction*. A conditional branch instruction determines the next instruction to execute based on a condition, typically the result of a comparison (e.g.: is the loop counter less than 10?). Only the outcome of this comparison needs to be predicted to know the next instruction address. The second case is more general. Here, the next instruction address itself is predicted. This type of prediction is specifically important for branches that compute the their target at run-time (e.g., when the target is fetched from memory).

In both cases, the identifier field in the trace is the program counter of the instruction. The data field is the followed branch direction (taken or not-taken) in the case of conditional branch predictors and the branch target address (i.e., the program counter of the next instruction in the trace) in the case of unconditional branch prediction.

4.1 Bimodal Branch Predictor

A bimodal branch predictor is a conditional branch predictor and outputs for every branch a one when the branch is predicted taken and a zero when the prediction is not-taken. It is organised as a table of saturating up/down counters (Figure 4(a)), each b bits wide. The program counter of the branch instruction selects a counter in the table, whose value determines the prediction. The branch is predicted taken for high values ($\geq 2^{b-1}$), and not-taken for low values ($< 2^{b-1}$).

The counter is updated based on the actual branch outcome. It is incremented by 1 for taken branches, saturating at $2^b - 1$, and decremented by 1, saturating at 0, otherwise. Note that the counters are not tagged, i.e., all branches that map to the same row share the same counter.

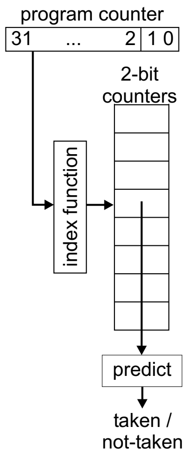
The update function for the LRU replacement policy:

$$U_{sa,LRU}(c,(id,data).R) = \rfloor(id \div B,\bot).delete((id \div B,\bot),c)\lfloor_A \quad (4)$$

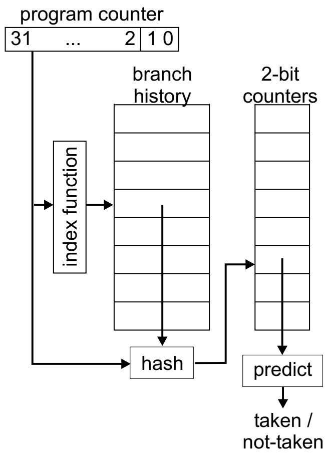
The update function for the FIFO replacement policy:

$$\begin{array}{lcll} U_{sa,FIFO}(c,(id,data).R) & = & c & \text{if } (id \div B,\bot) \in c \\ U_{sa,FIFO}(c,(id,data).R) & = & \rfloor(id \div B,\bot).c\lfloor_A & \text{otherwise} \end{array} \quad (5)$$

Figure 3. Operational semantics of replacement policies for a set-associative cache.



(a) Bimodal predictor



(b) Two-level predictor

Figure 4. The modeled branch predictors.

The bimodal branch predictor easily fits in our model, using the following definitions:

$$C_{bimodal} = (S, 1, \bot, I_{bimodal}, O_{bimodal}, U_{bimodal})$$

When instructions are 4 bytes long and are always aligned on 4-byte boundaries, then the index function ignores the lowest 2 bits from the program counter, as these bits are always zero:

$$I_{bimodal}((id,\bot)) = bitsel(id, 4, S)$$

The output function returns whether the branch direction is predicted taken, i.e. whether the counter is sufficiently large:

$$O_{bimodal}((\bot,s).c,(id,dir)) = s \geq 2^{b-1}$$

The update policy adjusts the counters to conform with the last observed branch direction:

$$U_{bimodal}((\bot,s).c,(id,dir).R) = (\bot,s').\lambda$$

where

$$\begin{array}{lcll} s' & = & max(s-1,0) & \text{if } dir = \text{not-taken} \\ s' & = & min(s+1,2^b-1) & \text{if } dir = \text{taken} \end{array}$$

4.2 Two-Level Branch Prediction

The direction of a conditional branch instruction depends on the direction observed for previous executions of the same branch and on the direction of other branch instructions. A two-level branch predictor uses this property to increase the prediction accuracy. It has two tables that are accessed successively.

The first-level table is accessed using the program counter and stores the history of directions of the branches that map to that row. The branch directions are read from the table and are combined with the program counter to index the second-level table, which stores saturating up/down counters just like the bimodal branch predictor. The selected counter is used to make a prediction.

To update the first-level table, the correct branch direction is appended to the branch history, which is then truncated to the right number of bits by discarding the oldest branch directions. The cell in the second-level table is updated similarly to the bimodal branch predictor.

We first define the first-level table. To simplify the mathematics, we place each bit in the history in a different column. This predictor remembers L bits of branch history:

$$C_{level1} = (S_1, L, \bot, I_{level1}, O_{level1}, U_{level1})$$

Assuming that instructions are 4 byte entities, the first-level table is indexed by the program counter modulo 4 (Equation 6). The output function outputs the contents of the cell, such that it can be used in the second-level table (Equation 7). The table with histories is updated by appending the last branch direction to the history and forgetting the oldest branch direction (Equation 8).

The second-level table is similar to the bimodal

The index function for the level-1 table:

$$I_{level1}((id,\perp)) = bitsel(id, 4, S_1) \tag{6}$$

The output function for the level-1 table:

$$\begin{aligned} O_{level1}(\lambda, (id, dir)) &= \lambda \\ O_{level1}((id, dir).c, (id_1, dir_1)) &= dir.O_{level1}(c, (id_1, dir_1)) \end{aligned} \tag{7}$$

The update function for the level-1 table:

$$U_{level1}(c, (id, dir).R) = \rfloor(\perp, dir).c\lfloor_L \tag{8}$$

The index function for the level-2 table with bitwise exclusive OR:

$$I_{level2}((id,\perp)) = bitsel(id, 4, S_2) \text{ bitwise-XOR } O_{level1}(C(I_{level1}((id,\perp))), (id,\perp)) \tag{9}$$

The index function for the level-2 table with concatenation:

$$I_{level2}((id,\perp)) = concatenate(bitsel(id, 4, 2^k), O_{level1}(C(I_{level1}(id)), (id,\perp))) \tag{10}$$

The index function for the level-2 table using only the history:

$$I_{level2}((id,\perp)) = O_{level1}(C(I_{level1}(id)), (id,\perp)) \tag{11}$$

Figure 5. Operational semantics of the two-level branch predictor.

branch predictor, except for the index function:

$$C_{level2} = (S_2, 1, \perp, I_{level2}, O_{bimodal}, U_{bimodal})$$

The index function combining the program counter and the branch history can be almost anything. Frequently, the bit-wise exclusive or of L bits from the program counter and the L history bits is computed (Equation 9). In this case, $S_2 = 2^L$. Alternatively, one can concatenate the L history bits with k bits from the program counter (Equation 10), so $S_2 = 2^{k+L}$. Or the hashing function can ignore the program counter and just copy the branch history (Equation 11).

4.3 Branch Target Buffer

A branch target buffer (BTB) predicts the full branch address of the following instruction. The BTB is organised in rows and columns, much like a set-associative cache. The program counter of the branch instruction is used to select one row of the BTB and an associative search is performed to find a cell that is tagged with the current program counter. If such a cell is found, then the associated branch target address is used as prediction. When the program counter is not found in the BTB, then the next sequential instruction is fetched.

A A-way set-associative branch target buffer with S sets is defined as:

$$C_{btb} = (S, A, \perp, I_{bimodal}, O_{btb}, U_{btb})$$

The BTB predicts the branch target address read from the table in the case of a hit, and the program counter incremented by 4 in case of a miss (Equation 12).

The BTB is only updated if the branch is taken. When the branch is not found in the BTB, the prediction for next program counter equals the current program counter incremented with 4 (the size of an instruction). For the branches that are stored in the BTB, the LRU replacement policy is applied, but any other policy can be used as well (Equation 13).

5 Related Work

Young, Gloy and Smith [8] present a formal model of branch predictors. They split a stream of (address, branch direction) pairs into substreams and predict each substream by a single 2-bit saturating counter. They analyze several alternatives for the divider.

Another formal approach to branch prediction was made by Emer and Gloy [1]. They formally

The output function of the branch target buffer:

$$
\begin{array}{rcll}
O_{btb}(\lambda, (id, target)) & = & id + 4 & \\
O_{btb}((sid, starget).S, (id, target)) & = & starget & \text{if } id = sid \\
O_{btb}((sid, starget).S, (id, target)) & = & O_{btb}(S, (id, target)) & \text{otherwise}
\end{array}
\qquad (12)
$$

The update function of the branch target buffer:

$$
\begin{array}{rcll}
U_{btb}(S, (pc, target).R) & = & S & \text{if } target = pc + 4 \\
U_{btb}(S, (pc, target).R) & = & \rfloor (pc, target).delete((pc, target), S) \lfloor_A & \text{otherwise}
\end{array}
\qquad (13)
$$

Figure 6. Operational semantics of the branch target buffer.

model components typically used in branch predictors and specify a language to combine these components. Finally, they use a genetic optimization algorithm to find the best branch predictor for a particular trace. They find several weird branch prediction structures that may be more cost-effective than commonly used structures.

Weikle *et al.* [7] develop the TSPec formal specification language to specify memory address traces. Furthermore, they view caches as filters on traces, i.e., the trace of cache misses is simply a subset of the original trace. Cache miss rates can be computed by computing the filtered trace of misses and then counting its length.

6 Conclusion and Future Work

This paper presents a formal model of caches and branch predictors as they are typically used in computer architectures. The formal model unambiguously describes the behavior of these components. Future work is to use these models to formally predict their properties. We believe that these formal models will aid in analyzing the behavior and performance of these and more complex hardware components, which is relevant to reliability, dependability and real-time execution constraints.

7 Acknowledgements

This research is sponsored by the Flemish Institute for the Promotion of Scientific-Technological Research in the Industry (IWT), by the Fund for Scientific Research-Flanders (FWO), Ghent University and by the European Network on Excellence on High-Performance Embedded Architecture and Compilation (HIPEAC).

References

[1] J. Emer and N. Gloy. A language for describing predictors and its application to automatic synthesis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 304–314, 1997.

[2] H. Gao and H. Zhou. Adaptive information processing: An effective way to improve perceptron predictors. In *1st Journal of Instruction-Level Parallelism Championship Branch Prediction*, page 4 pages, Dec. 2004.

[3] D. Jiménez. Piecewise linear branch prediction. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 382–393, June 2005.

[4] G. Plotkin. A Structured Approach to Operational Semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[5] A. Seznec. Analysis of the O-GEometric History Length branch predictor. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 394–405, June 2005.

[6] A. J. Smith. Bibliography and readings on CPU cache memories and related topics. *ACM Computer Architecture News*, Jan. 1986.

[7] D. A. B. Weikle, S. A. McKee, K. Skadron, and W. A. Wulf. Caches as filters: A framework for the analysis of caching systems. In *Third Grace Hopper Celebration of Women in Computing*, Sept. 2000.

[8] C. Young, N. Gloy, and M. D. Smith. A comparative analysis of schemes for correlated branch prediction. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 276–286, June 1995.