A Formal Model for Microprocessor Caches

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: - Contemporary processors have reached a bewildering level of complexity featuring multiple execution pipelines, out-of-order instruction issuing, speculative execution, various prediction components and cache memories. The performance of these components is sometimes not well understood. To facilitate the analysis of these components, we propose the use of formal models of these components. Hereby, we aim to lay a formal basis for reasoning on processor components and to formally proove their properties. In this paper, we develop an operational semantics of cache memories and show how it describes operational aspects of caches.

Key-words: - operational semantics, microarchitecture, cache

1 Introduction

State-of-the-art processor 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 peoply 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 prooves 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 proove 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 *Sinst* are themselves defined below.

2.1 Notations

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

Definition 1 *Let E be a set. We subsequently 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 $e.S$.*

Definition 2 *The set of booleans is subsequently denoted by $\mathbb{B}$. Moreover, we shall denote by $\mathcal{B}$ the set $\mathbb{B}^{<\omega}$.*

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. The additional information indicates whether a load or a store is performed. 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 cache 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, instructions will be taken in sequences. The set *Ssinst* can thus be defined as follows.

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

2.3 Tables

Caches 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 cache 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 cache 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

$$Update = Crow \times Sinst \rightarrow Crow$$

Summing up, caches 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 caches *Stable*:

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

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 cache X and sequence of instructions R to the new state described by cache X' and sequence of instructions R'. To capture cache misses, we introduce a label on the arrow: μ is used to denote a cache miss and ν to indicate no cache miss.

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

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

that satisfies the following properties:

$$< (S, A, C, I, O, U), (id, data).R > \stackrel{\mu}{\rightarrow} < (S, A, C', I, O, U), R >$$

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

$$< (S, A, C, I, O, U), (id, data).R > \stackrel{\nu}{\rightarrow} < (S, A, C', I, O, U), R >$$

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

The operational semantics

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

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

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

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 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 returns a 1 on a cache hit and a zero on a cache miss. This value is compared to the data field in the trace to select between a μ and ν transition. Thus, when a cache hit occurs, we take a ν transition and when a miss occurs, we take μ 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:

$$bitsel(id, blocksize, sets) = \lfloor id \div blocksize \rfloor_{log_2(sets)}$$

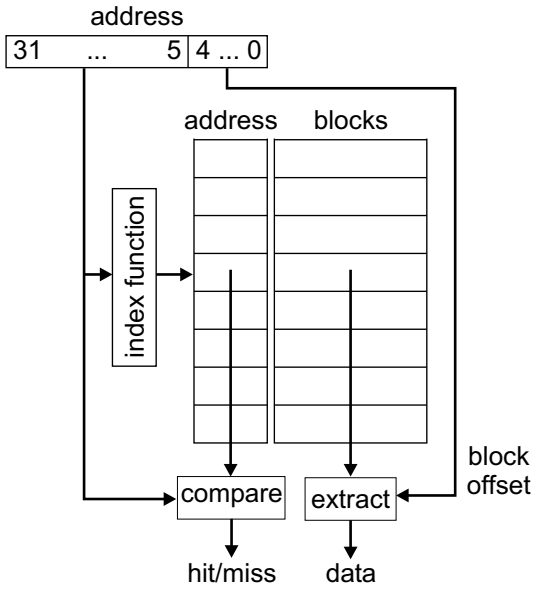

Figure 1. Indexing into a direct mapped cache with 32 byte blocks. It is assumed that addresses are 32 bits long.

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:

$$S \div n \quad = \quad drop(S, log_2(n))$$

where the auxiliary function $drop(S, n)$ drops the first n elements (i.e., the low-order bits) from the sequence S. The auxiliary function $\lfloor S \rfloor_n$ truncates the sequence S to the first n elements. The indexing function for a data access 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 blcoks: every block can be stored in every cell of the row indicated by the index function. The number of

The index function:

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

The output function:

$$\begin{array}{rcll} O_{cache}(\lambda, (id, data)) & = & 0 & \\ O_{cache}((sid, sdata).S, (id, data)) & = & 1 & \text{if } sid = id \div B \\ O_{cache}((sid, sdata).S, (id, data)) & = & O_{cache}(S, (id, data)) & \text{otherwise} \end{array} \quad (2)$$

The update function:

$$U_{dm}(C, (id, data).R) = (id \div B, \bot).\lambda \quad (3)$$

Figure 2. Operational semantics of a direct mapped cache

such cells is called the *degree of associativity* of the cache. Fig˙ 3 shows a two-way set-associative cache. Every block can be stored in either column 0 or column 1. 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 choice of replacement policy can have a large impact on the miss rate of the cache.

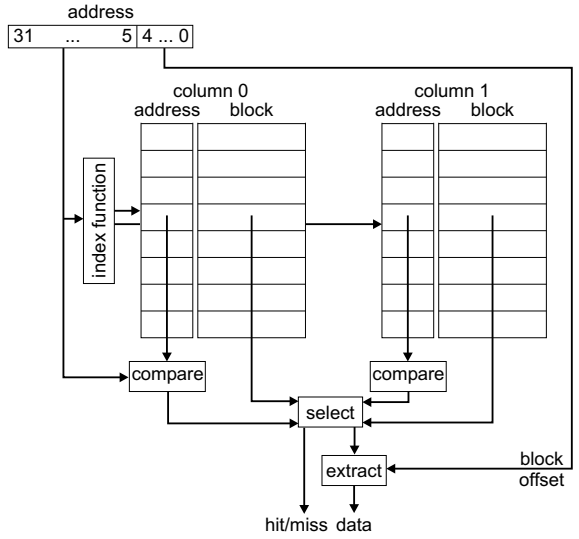

Figure 3. A two-way set-associative cache with 32 byte blocks.

In practice, desktop and high-performance processors contain level-1 caches in the range of 8 to 64kB, have 32 or 64-byte blocks and have a degree of associativity varying from 1 (direct mapped) to 4. 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:

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

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 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. The divider is the crucial part in

The update function for the LRU replacement policy:

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

The update function for the FIFO replacement policy:

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

Figure 4. Operational semantics of replacement policies for a set-associative cache.

their model and they analyze several alternatives for the divider.

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

5 Conclusion and Future Work

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

6 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.