Quantitative Static Analysis over semirings: analysing cache behaviour for Java Card

Pascal Sotin [1] David Cachera [2] Thomas Jensen [3]

Irisa, campus de Beaulieu
35 042 Rennes, France
Email: {sotin,dcachera,jensen}@irisa.fr

Abstract

We present a semantics-based technique for modeling and analysing resource usage behaviour of programs written in a simple object oriented language like Java Card byte code. The approach is based on the quantitative abstract interpretation framework of Di Pierro and Wiklicky where programs are represented as linear operators. We consider in particular linear operators over semi-rings (such as max-plus) that have proven useful for analysing cost properties of discrete event systems. We illustrate our technique through a cache behaviour analysis for Java Card.

Key words: Static analysis, resource usage, cache behaviour, linear operators, semirings

1 Introduction

This paper is concerned with the semantics-based program analysis of quantitative properties pertaining to resource usage. There exist numerous methods for estimating resource usage, ranging from monitoring or simulating [SC99] executions to the exact computation of the complexity of the program, passing by techniques for determining Worst Case Execution Time [PK89]. Our approach is based on a recent framework called quantitative abstract interpretation which leads to an elegant program model based on linear operators over vector spaces and which is able to deal with several kind of quantities. Quantitative semantic models, like their qualitative counterparts, are usually not computable, so we devise an abstraction technique for computing a correct approximation of the property of interest. In the quantitative case, this is usually an upper bound of the program consumption of a particular resource.

[1] Irisa/CNRS/DGA
[2] Irisa/ENS Cachan (Bretagne)
[3] Irisa/CNRS

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Di Pierro and Wiklicky [PW00] have proposed an abstract interpretation for a probabilistic semantics in which probabilities are attached to transitions. This naturally leads to a model where programs are modelled as linear operators represented by stochastic matrices. We follow up on the idea of expressing programs as linear operators but we are interested in estimating the resource consumption of a program where we can attach any numeric quantity to a transition and not just a probability between 0 and 1—this forcibly takes us beyond stochastic matrices. The costs of transitions can model for example stack height evolution, the number of calls to a given method or can represent benchmarked execution times. In this paper, we will focus on how to model cache misses.

We rely on a standard small-step operational semantics expressed as a transition relation $s \rightarrow^q s'$ between states $s, s' \in \mathit{State}$ with costs $q \in Q$ associated to each transition. There is a straightforward way to pass from this rule-based semantics to a matrix representation, associating a cost to a pair of states. We develop a technique for abstracting this semantics, in order to return a computable approximation of the overall program cost. To do so, we exploit the fact that the semantics of a program can be expressed as a linear operator on Q^{State}, where Q is the domain of the considered cost. We define two notions of cost for a program: the *global cost* from initial to final states, which is derived from the transitive closure of the semantics. When cycles exist in the underlying graph, this closure does not exist, but we are able to give the *long run cost* of the program, which is the average cost of a transition in the cycle. The computation of this cost relies on a variant of the Perron-Frobenius theorem on eigenvalues in idempotent semirings.

The paper is structured as follows. In Section 2, we define a quantitative semantics for the Java Card byte code language that explicitly models the (non-functional) cache behaviour of a Java Card program. In Section 3, we then define the corresponding linear operator semantics. For a given resource, we propose, in Section 4, to compute the global consumption of the resource by the program executing from the beginning to the end. It is sometimes impossible to compute this global cost, *e.g.* when the program is a reactive program, that by definition does not finish. In this case, we still give the average consumption on a long run execution. In Section 5, we then discuss how to approximate the overall behaviour of a program by projecting this operator onto a smaller state space.

2 Definition of a quantitative semantics

A quantitative semantics that describes non-functional properties still uses states that contain classical information on the execution in order to be able to compute the control flow. They also contains extra information, that does not affect the computation result, but may affect the property we deal with. The cache memory of a computer is a good example, because even if the

computation result and control flow do not change in presence of a cache, the execution time is strongly modified.

We give the semantics for the Java Card language. Java Card is a subset of Java, designed for programs embedded on smart cards. In this context of low memory available and of interaction with the customer, extracted information about the program on both memory and time is of interest. The mechanisms underlying Java Card are simpler than those underlying Java. The non-functional behaviour of the JVM is enhanced by features such as garbage collection, threads, and Just-In-Time compilers, which complicate the analysis of quantitative, non-functional properties. In comparison, Java Card mechanisms are fewer and simpler, so it is a better suited target for our analyses.

2.1 Modeling cache behaviour in a JCVM

The state in our semantics for Java Card is of the form:

$$< H, \ll f, m, ip, L, S \gg :: fr, C >$$

where

- H stands for the heap of objects;
- f is the frame identifier of the frame stack (i.e. procedure call stack) and fr is the remainder of this frame stack;
- m is the current method and ip the instruction pointer in it. The current instruction is given by the function $InstrAt(m, ip)$;
- L is an array containing the local variables of the frame;
- S is the operand stack of the frame;
- C is a set of *logical addresses*, representing which values are in cache at this point of the execution.

The set of logical addresses is managed similarly to the cache. For example, the maximum size of this set will fit the size of the physical cache, and the replacement policy will model the one done by the cache (e.g. LRU, FIFO). The function *update* models the cache behaviour: it takes as parameters the current state of the cache, and a list of typed accesses to logical addresses accessed by the program or instruction, the first element of the list being the first memory access. Logical addresses can be of three forms:

- `heap`.*reference.short* designates the field indexed by *short* of an object in the heap whose reference is *reference*.
- `local`.*frameId.short* designates the local variable, indexed by *short* of the frame *frameId*.
- `stack`.*frameId.int* designates the *int*-nth element in the operand stack zone of the frame *frameId*. For example, if t is the current size of the operand stack, t designates the element which is the current top of the stack, and

$t+1$ refers to the element one word over that top.

An access to one of these logical addresses can be either $read_\tau(address)$ or $write_\tau(address)$ which means that data of type τ is respectively read or written at this address. The type information is important to compute the size of the data items and how many of these items can be put simultaneously in the cache. Addresses can be expressed with a translation based on the data size, e.g. $write_\tau(address[+1])$ is equivalent to $write_\tau(address + 1 * sizeof(\tau))$.

For the semantics, we do not describe the 201 instructions of the JCVM [JCV], but we rely on the modeling of the JCVM instructions through the intermediate language Carmel [Mar01]. This language has a more manageable number of instructions, grouping together Java Card instructions with similar behaviour. Below, we give as an example the rule describing the `load` instruction. It corresponds to the Java Card set of instructions which loads a typed local variable, indexed by i, on the top of the operand stack (e.g. `iload`, `aload`).

$$\frac{\begin{array}{c} InstrAt(m, ip) = \texttt{load}\ \tau\ i \wedge L[i] = d \\ S' = d :: S \wedge size(S) = t \\ C' = update(C, [read_\tau(\texttt{local}.f.i); write_\tau(\texttt{stack}.f.t[+1])]) \end{array}}{< H, \ll f, m, ip, L, S \gg :: fr, C > \rightarrow^{q} < H, \ll f, m, ip+1, L, S' \gg :: fr, C' >}$$

In the example, q models the cost of the `load` transition. It needs to be instantiated with the considered quantity. For example, if we model the stack height evolution, q will be $q = +sizeof(\tau)$ for the `load` instruction. The value of q varies from one instruction rule to another, and can be a function of the state. We give below an instantiation of the quantities for all our semantics rules, dealing with the number of reading cache misses.

$$\text{Let } q = \begin{cases} nbRmiss(C, access) \text{ if the rule contains } C' = update(C, access) \\ 0 \text{ otherwise} \end{cases}$$

The function $nbRmiss(C, access)$ computes the number of reading cache misses generated by the list of memory accesses $access$ if the cache at the beginning of the instruction is C. This function has to be implemented in relation with the $update$ function, because its result depends on the cache policy for accesses beyond the first one. The computation of $nbRmiss$ proceeds as follows:

$$nbRmiss\ c\ [\,] = 0$$

$$nbRmiss\ c\ [a|r] = nbRmiss\ (update\ c\ a)\ r + \begin{cases} 1\ if\ \begin{cases} a = read\ m \\ m \in c \end{cases} \\ 0\ otherwise \end{cases}$$

This algorithm deals with cases where the first accesses change the presence or absence of the data for the remaining accesses, but it might be quite expensive. For implementation efficiency, it would be possible to make an approximation of *nbRmiss* independent from the *update* function. Such an approximation relies on the hypothesis that the cache does not unload values which will be used in the current instruction.

In the case of the `load` transition rule, both algorithms have the same result which is 1 if $\texttt{local}.f.i \in C$ and 0 otherwise. It is a particular case due to the fact that the only *read* access is the first of the access list so other accesses cannot have side-effects on it.

3 Linear operator semantics

The small-step, quantitative operational semantics for Java Card induces a labelled transition system over $State$ with labels in Q and a transition relation $\rightarrow \subseteq State \times Q \times State$, written $s \rightarrow^q s'$. Such a transition states that a direct (one-step) transition from s to s' costs q.

These unitary transitions can be combined into big-step transitions, using two operators which will form a semiring on Q. Costs could have been defined in a more general way but this, arguably rather restricted, definition has interesting computational properties.

The operator $\otimes$ on Q defines the global cost of a sequence of transitions, $s \rightarrow^{q_1} \ldots \rightarrow^{q_n} s'$ simply as $q = q_1 \otimes \ldots \otimes q_n$. This is written $s \stackrel{x}{\Rightarrow}^q s'$ where x is the sequence of states.

(1) The operator $\otimes$ is associative and has a neutral element e. The quantity e represents a transition that costs nothing.

(2) The operator $\otimes$ comes with a function called the nth root, written $\sqrt[n]{q}$ or $q^{1/n}$, such that $\otimes_{i=1}^{n} \sqrt[n]{q} = q$. A sequence containing n transitions, each costing $\sqrt[n]{q}$, will cost q. If $\otimes$ stands for $\times$, $+$, `max` or $\cup$, the nth root of x will respectively be $\sqrt[n]{x}$, $1/x$, x and x.

When there exist several ways to reach a state s' from a state s, $X = \{s \stackrel{x}{\Rightarrow}^{q_x} s'\}$, the global cost between them is defined using the operator $\oplus$ on Q to be $q = \bigoplus_{x \in X} q_x$. This is written $s \Rightarrow^q s'$.

(3) The operator $\oplus$ is associative, commutative and $\perp$ is its neutral element. The quantity $\perp$ means the impossibility of a transition.

(4) $\otimes$ is distributive for $\oplus$ and $\perp$ is absorbant element for $\otimes$.

(5) $\oplus$ is idempotent i.e. $q \oplus q = q$, so if several transitions go from a to b for the same cost q, the global cost is also q.

Definition 3.1 A structure $(Q, \oplus, \otimes)$ that fulfills the conditions 1, 3 and 4 is a semiring. With the condition 5, it is called an idempotent semiring.

We work with structures fulfilling the five conditions, *i.e.* with idempotent

semirings equiped with an nth root operation, which we call semirings of costs. For instance, $(Time, \texttt{max}, +)$ is a semiring of costs and it leads to the definition of the Worst Case Execution Time.

When two states can be joined by several sequences of transitions which cost different times, the worst time is taken. To compute the cost of a sequence of transitions, we sum the costs of each transition.

Computations of costs, using an adequate semiring, are easily defined in terms of computation on matrices in this semiring. The set of one-step transitions can be equivalently represented by a matrix, called a transition matrix, defined by:

$$M \in \mathcal{M}_{State \times State}(Q)$$

$$s \rightarrow^q s' \Leftrightarrow M_{s,s'} = q$$

Here, $\mathcal{M}_{A \times B}(C)$ stands for the set of matrices with rows in A, columns in B and values in C.

This matrix may also be seen as a linear operator on the semimodule Q^{State}; a semimodule being for a semiring what a vector space is for a ring.

Relation with trace semantics

Even if computations with matrices/linear operators are elegant and sufficient, we give their equivalent computation, in term of a more classical trace semantics. For a program P, we consider its initial states, its final states and its trace semantics, $[\![P]\!]_{tr}$:

- The initial states I are the states in which the program can be started.
- The final states F are the states where the execution will be halted by the machine. *E.g.* a state reached after the `return` instruction of the `main` function of a Java program.
- $$[\![P]\!]_{tr} = \left\{ s_1 \rightarrow^{q_1} \dots s_{n-1} \rightarrow^{q_{n-1}} s_n \middle| \begin{array}{l} s_1 \in I, \\ s_i \rightarrow^{q_i} s_{i+1}, \end{array} \right\}$$

4 Computing global and long run costs

In this section, we express the cost of a given program in terms of matrix computations. Two kinds of costs can be computed:

- if the program terminates, we can compute a *global cost* that represents its cost from initial to final states.
- for a non-terminating program, we can compute a *long run cost*, that expresses an average cost over cycles of transitions.

In the following, we work with a cost semiring $(Q, \oplus, \otimes)$.

Let M be a matrix representing a quantitative semantics for a program P. M contains the transition costs induced by one step of the operational

semantics. M^k, where the product of matrices is taken in the considered semiring, summarizes the transition costs of all paths of length k. Global and average costs are defined starting from this idea, by computing the successive iterates of the transition cost matrix until a fixpoint is reached. If such a transitive closure M^+ exists, it contains all the transitions costs from any state to any state.

$$M^+ = \bigoplus_{i=1}^{\infty} M^i$$

If the transition graph associated to the semantics contains cycles or infinite paths, this transitive closure might not be defined. In this case, we will preferably refer to an average cost as defined below.

Definition 4.1 Let I be the initial states of the program i.e. entry points and F be the exit states of the program. The global cost of P is defined as

$$gc(P) = \bigoplus \{M^+_{i,f} | i \in I, f \in F\}$$

The global cost is related to the standard trace semantics by:

Theorem 4.2

$$gc(P) = \bigoplus \{\bigotimes_{j=1}^{f-1} q_j | \sigma_1 \rightarrow^{q_1} \ldots \rightarrow^{q_{f-1}} \sigma_f \in [\![P]\!]_{tr}, \sigma_f \in F\} \tag{1}$$

Proof

$$\begin{aligned}
&\bigoplus \{\bigotimes_{j=1}^{f-1} q_j | \sigma_1 \rightarrow^{q_1} \ldots \rightarrow^{q_{f-1}} \sigma_f \in [\![P]\!]_{tr}, \sigma_f \in F\} \\
&= \bigoplus \{\bigotimes_{j=1}^{f-1} q_i | \sigma_1 \rightarrow^{q_1} \ldots \rightarrow^{q_f} \sigma_f \in (\rightarrow^{\cdot})^+, \sigma_1 \in I, \sigma_f \in F\} \\
&= \bigoplus_{n=1}^{\infty} (\bigoplus \{\bigotimes_{j=1}^{n} q_j | \sigma_1 \rightarrow^{q_1} \ldots \rightarrow^{q_n} \sigma_f \in (\rightarrow^{\cdot})^n, \sigma_1 \in I, \sigma_f \in F\}) \\
&= \bigoplus_{\substack{\sigma_1 \in I \\ \sigma_f \in F}} (\bigoplus_{n=1}^{\infty} (\bigoplus \{\bigotimes_{j=1}^{n} q_j | \sigma_1 \rightarrow^{q_1} \ldots \rightarrow^{q_n} \sigma_f \in (\rightarrow^{\cdot})^n\}))
\end{aligned}$$

We derive the other definition in a similar form:

$$\begin{aligned} gc(P) &= \bigoplus \{M^+_{\sigma_1,\sigma_f} | \sigma_1 \in I, \sigma_f \in F\} \\ &= \bigoplus_{\substack{\sigma_1 \in I \\ \sigma_f \in F}} M^+_{\sigma_1,\sigma_f} \\ &= \bigoplus_{\substack{\sigma_1 \in I \\ \sigma_f \in F}} \bigoplus_{n=1}^{\infty} M^n_{\sigma_1,\sigma_f} \end{aligned}$$

Equality (1) is verified under the following condition, which is easily proved by induction on n:

$$M^n_{a,b} = \bigoplus \{ \bigotimes_{j=1}^{n} q_j | a \rightarrow^{q_1} \ldots \rightarrow^{q_n} b \in (\rightarrow^{\cdot})^n \}$$

□

Long run cost

If the transitive closure M^+ does not exist, we will consider the eigenvalue of the matrix. In the case where M is irreducible, this value is unique and it is defined by:

$$\rho(M) = \bigoplus_{k=1}^{\infty} (\mathtt{tr}\ M^k)^{1/k}$$

where $\mathtt{tr}\ M = \bigoplus_1^n M_{i,i}$. This follows from the Perron-Frobenius theorem (more precisely, its instantiation to idempotent semirings, as developed in [Gau99]) which states that the spectral ray of an irreducible matrix A is the eigenvalue of A and that the associated eigenspace is a one-dimensional vector space generated by a vector with strictly positive components.

In the case where the matrix is reducible, we can partition it in the form of a triangular, by blocks, matrix. With this matrix, we can obtain in a similar way a unique eigenvalue. In the case where all maximal traces of program P are finite, the notion of long run does not make sense. In this case, the above formula yields an eigenvalue equal to $\bot$.

The average cost is related to the trace semantics in the following way: ρ is the sum (under $\oplus$) of the geometric average of all cycles appearing in a trace of P. For instance, if we work in the semiring $(Time, \mathtt{max}, +)$, $\rho(M)$ is the maximal average of time spent per instruction, where the average is computed on any cycle by dividing the total time spent in the cycle by the number of instructions in this cycle.

5 Abstraction of a quantitative semantics

The transition matrix representing a program is in general of infinite dimension, so neither transitive closure nor eigenvalues can be computed in finite time. To overcome this problem, we define an abstract matrix—smaller than the concrete one and preferably finite—that can be used to approximate the computations with the infinite original matrix. We consider the necessary conditions so that approximation is correct. *E.g.*, if we compute the minimum memory needed to run a program, a correct approximation of this quantity must be greater than the effective minimum.

Abstraction

Let M be the linear operator on the concrete domain C, over the idempotent semiring $(Q, \oplus, \otimes)$, corresponding to the transition system of the program. The zero of the semiring is written $\perp$ and the unit is written e.

$$M \in \mathcal{M}_{C\times C}(Q)$$

Given an abstraction function from concrete states C to a set of abstract states D, we can lift this function to a linear abstraction operator $\alpha : Q^{D\times C}$ on the semiring as follows.

$$\alpha_{d,c} = \begin{cases} e \text{ if } \alpha(c) = d \\ \perp \text{ otherwise} \end{cases}$$

Let $M^\sharp \in \mathcal{M}_{D\times D}(Q)$ be a linear operator in the abstract domain D. The correctness condition on the abstraction α is:

$$\alpha \circ M \leq_D M^\sharp \circ \alpha$$

where $\leq_Q$ is the ordering induced by the semi-ring operation $\oplus$ ($q_1 \leq_Q q_2 \Leftrightarrow q_1 \oplus q_2 = q_2$) and $\leq_D$ is its matrix extension ($N \leq_D P \Leftrightarrow \forall i, j. N_{i,j} \leq_Q P_{i,j}$).

Correctness

The correctness condition implies that the computation of the global and long run cost is correct, i.e., is an overapproximation of the concrete cost. It derives from the linearity of α, M and $M^\sharp$.

Theorem 5.1 *If M^+ and $(M^\sharp)^+$ exist,* i.e. *converge in finite time, then*

$$\alpha \circ M \leq_D M^\sharp \circ \alpha \Rightarrow \alpha \circ M^+ \leq_D (M^\sharp)^+ \circ \alpha \tag{2}$$

If $\rho(M)$ and $\rho(M^\sharp)$ exist, i.e. *if M and $M^\sharp$ are not reducible, then*

$$\alpha \circ M \leq_D M^\sharp \circ \alpha \Rightarrow \rho(M) \leq_Q \rho(M^\sharp) \tag{3}$$

Proof We prove (2) by first showing that:

$$\forall n \geq 1, \alpha \circ M \leq_D M^\sharp \circ \alpha \Rightarrow \alpha \circ M^n \leq_Q (M^\sharp)^n \circ \alpha \tag{4}$$

It is proved by induction on n. The case where $n = 1$ is trivial. We then assume that $\alpha \circ M \leq_D M^\sharp \circ \alpha$ and $\alpha \circ M^n \leq_D (M^\sharp)^n \circ \alpha$. We have:

$$\begin{aligned}
\alpha \circ M^n &\leq_D (M^\sharp)^n \circ \alpha \\
(\alpha \circ M^n) \circ M &\leq_D ((M^\sharp)^n \circ \alpha) \circ M && (5)\\
\alpha \circ M^{n+1} &\leq_D (M^\sharp)^n \circ (\alpha \circ M) && (6)\\
\alpha \circ M^{n+1} &\leq_D (M^\sharp)^n \circ (M^\sharp \circ \alpha) && (7)\\
\alpha \circ M^{n+1} &\leq_D (M^\sharp)^{n+1} \circ \alpha
\end{aligned}$$

(5) is correct because $\oplus$ and $\otimes$ preserves the order $\leq_Q$ defined as $a \leq_Q b \Leftrightarrow a \oplus b = b$, respectively thanks to associativity and idempotency of $\oplus$, and to distributivity of $\otimes$ over $\oplus$. Going from (6) to (7) uses the first hypothesis. We then sum (4) for n in one to infinity, and factorise by α on both sides.

$$\begin{aligned}
\alpha \circ (M^1 \oplus M^2 \oplus \ldots) &\leq_D ((M^\sharp)^1 \oplus (M^\sharp)^2 \oplus \ldots) \circ \alpha \\
\alpha \circ M^+ &\leq_D (M^\sharp)^+ \circ \alpha
\end{aligned}$$

We now prove (3). Let c and d be such that $\alpha(c) = d$. We first show that $M_{c,c} \leq_Q M^\sharp_{d,d}$.

We have $\alpha \circ M \leq_D M^\sharp \circ \alpha$
in particular $(\alpha \circ M)_{d,c} \leq_Q (M^\sharp \circ \alpha)_{d,c}$
that rewrites into $\bigoplus_{c' \in C} (\alpha_{d,c'} \otimes M_{c',c}) \leq_Q \bigoplus_{d' \in D} (M^\sharp_{d,d'} \otimes \alpha_{d',c})$

We decompose both summations, this yields:

$$\begin{aligned}
\bigoplus_{c' \in \alpha^{-1}(d)} (\alpha_{d,c'} \otimes M_{c',c}) \oplus \bigoplus_{c' \notin \alpha^{-1}(d)} (\alpha_{d,c'} \otimes M_{c',c}) \\
\leq_Q \bigoplus_{d' = \alpha(c)} ((M^\sharp_{d,d'}) \otimes \alpha_{d',c}) \oplus \bigoplus_{d' \neq \alpha(c)} ((M^\sharp_{d,d'}) \otimes \alpha_{d',c})
\end{aligned}$$

Given the properties of α, that simplifies into:

$$\bigoplus_{c' \in \alpha^{-1}(d)} M_{c',c} \bigoplus_{c' \notin \alpha^{-1}(d)} \bot \leq_Q M^\sharp_{d,d}$$

This concludes the proof. Thanks to idempotency, we deduce that for any d,

$$\bigoplus_{c \in \alpha^{-1}(d)} M_{c,c} \leq_Q M^\sharp_{d,d}$$

By summing over d, we finally get:

$$\bigoplus_{c \in C} M_{c,c} \leq_Q \bigoplus_{d \in D} M^\sharp_{d,d}$$

Similarly, since for any $k \geq 1$, we have $\alpha \circ M^k \leq_D (M^\sharp)^k \circ \alpha$, we show that:

$$\begin{aligned}
\bigoplus_{c \in C} M^k_{c,c} &\leq_Q \bigoplus_{d \in D} (M^\sharp)^k_{d,d} \\
trM^k &\leq_Q tr(M^\sharp)^k \\
(trM^k)^{1/k} &\leq_Q (tr(M^\sharp)^k)^{1/k} \\
\bigoplus_{k=1}^{\infty} (trM^k)^{1/k} &\leq_Q \bigoplus_{k=1}^{\infty} (tr(M^\sharp)^k)^{1/k} \\
\rho(M) &\leq_Q \rho(M^\sharp)
\end{aligned}$$

□

This concludes the proof of Theorem 5.1.

Given a correct abstraction based on M, $M^\sharp$ and α, we can classify it according to the existence of M^+ and $(M^\sharp)^+$.

	M^+ exists	M^+ does not exist
$M^{\sharp +}$ exists	Overapproximation of the global cost	$\emptyset$
$(M^\sharp)^+$ does not exist	Approximation giving information on the global cost by a long run cost	Overapproximation of the long run cost

5.1 An example of abstraction

Consider the following fragment of a program in JavaCard bytecode:

```
1: iload x
2: istore t
3: iload y
4: istore x
5: iload t
6: istore y
7: ...
```

The quantity we deal with is the maximum number of read misses in the cache. Managing the whole state and whole cache would in general make the exact computation too costly, so we abstract the state and cache to only contain the instruction pointer and last data accessed, and write the abstract

state as (ip, var) where ip is the instruction pointer and var is the logical address of the last data accessed. We model the maximum number of read misses using the semiring $(\mathbb{N} \cup \{\bot\}, \mathtt{max}, +)$. If n is the effective maximum number of cache misses, a correct abstraction delivers $n^\sharp$ such that $\mathtt{max}(n, n^\sharp) = n^\sharp$, i.e., $n < n^\sharp$. In this case, we say that the abstract semantics over-approximates the concrete one.

To construct the abstract transition matrix, we compute the transitions costs in the abstract semantics. For example, we can compute $q^\sharp$ of the transition $(1, y) \rightarrow^{q^\sharp} (2, 1)$ by the following case analysis.

- For $ip = 1$ we have $InstrAt(m, ip) = \mathtt{load}\ \tau\ x$.
- For any transition $s \rightarrow^q s'$ such that $ip = 1$ in s, such that the last element accessed in the cache is y and such that $\mathtt{l}.f.x \in C$, we can show that $q = 0$ and that s (resp. s') is abstracted by $(1, y)$ (resp. $(2, 1)$).
- For any transition $s \rightarrow^q s'$ such that $ip = 1$ in s, such that the last element accessed in the cache is y and that $\mathtt{l}.f.x \notin C$, we can show that $q = 1$ and that s (resp. s') is abstracted by $(1, y)$ (resp. $(2, 1)$).
- The value $q^\sharp$ such that $(1, y) \rightarrow^{q^\sharp} (2, 1)$ is defined by $\bigoplus\{q | s \rightarrow^q s' \wedge \alpha(s) = (1, y) \wedge \alpha(s') = (2, 1)\}$ and so is equal to $0 \oplus 1 = \mathtt{max}\{0, 1\} = 1$.

The matrix $M^\sharp$ representing the abstract semantics is given below. Values are computed similarly to the above example.

$$M^\sharp = \begin{pmatrix}
 & \dots & 2,1 & \dots & 3,t & \dots & 4,1 & \dots & 5,x & \dots & 6,1 & \dots & 7,y & \dots \\
1,\emptyset & & 1 & & \bot & & \bot & & \bot & & \bot & & \bot & \\
1,x & & 0 & & \bot & & \bot & & \bot & & \bot & & \bot & \\
1,y & & 1 & & \bot & & \bot & & \bot & & \bot & & \bot & \\
1,t & & 1 & & \bot & & \bot & & \bot & & \bot & & \bot & \\
1,1 & & 1 & & \bot & & \bot & & \bot & & \bot & & \bot & \\
\vdots & & & & & & & & & & & & & \\
2,1 & & \bot & & 0 & & \bot & & \bot & & \bot & & \bot & \\
3,t & & \bot & & \bot & & 1 & & \bot & & \bot & & \bot & \\
4,1 & & \bot & & \bot & & \bot & & 0 & & \bot & & \bot & \\
5,x & & \bot & & \bot & & \bot & & \bot & & 1 & & \bot & \\
6,1 & & \bot & & \bot & & \bot & & \bot & & \bot & & 0 & \\
7,y & & \bot & & \bot & & \bot & & \bot & & \bot & & \bot &
\end{pmatrix}$$

This matrix contains many occurrences of the value $\bot$. Furthermore, columns and rows not shown are only $\bot$, so sparse matrix algorithms can be used to compute the transitive closure of $M^\sharp$ in $(\mathbb{N} \cup \{\bot\}, \mathtt{max}, +)$: From this matrix it is possible to extract the global cost of transition, cg, between the states with $ip = 1$ and those with $ip = 7$ by the following matrix operation:

$$cg = I.(M^\sharp)^+.F = 3 \oplus 2 \oplus 3 \oplus 3 \oplus 3 \oplus \bot \oplus \dots \oplus \bot = 3$$

where

- I is the row vector with value 0 for all abstract states having 1 as its instruction pointer and $\bot$ otherwise.
- F is the column vector with value 0 for all abstract states having 7 as its instruction pointer and $\bot$ otherwise.

The given abstraction offers good results for computing the maximum read misses, *i.e.* returns a value reasonably close to the effective maximum read misses. It is due to the fact that most of the time, the result of one instruction is immediately used by the next instruction.

$$(M^\sharp)^+ = \begin{pmatrix} & \dots\ 2,1\ \dots\ 3,t\ \dots\ 4,1\ \dots\ 5,x\ \dots\ 6,1\ \dots\ 7,y\ \dots \\ 1,\emptyset & 1 & 1 & 2 & 2 & 3 & 3 \\ 1,x & 0 & 0 & 1 & 1 & 2 & 2 \\ 1,y & 1 & 1 & 2 & 2 & 3 & 3 \\ 1,t & 1 & 1 & 2 & 2 & 3 & 3 \\ 1,1 & 1 & 1 & 2 & 2 & 3 & 3 \\ \vdots & & & & & & \\ 2,1 & \bot & 0 & 1 & 1 & 2 & 2 \\ 3,t & \bot & \bot & 1 & 1 & 2 & 2 \\ 4,1 & \bot & \bot & \bot & 0 & 1 & 1 \\ 5,x & \bot & \bot & \bot & \bot & 1 & 1 \\ 6,1 & \bot & \bot & \bot & \bot & \bot & 0 \\ 7,y & \bot & \bot & \bot & \bot & \bot & \bot \end{pmatrix}$$

6 Conclusion

We have proposed an extended operational semantics for the Java Card language, that models transition costs between states. This semantics is defined in a generic way, in order to express various kinds of costs. The semantic domains and transitions integrate a cache model, well suited to evaluate costs that do not only depend on the input-output behaviour, in particular execution time. As an example of instantiation, we give the costs attached to the computation of cache misses.

Expressing the semantics as a linear operator on semimodules allows to compute it through matrix operations. We have defined two distinct notions of cost attached to a program: whenever possible, the global cost from initial to final state is computed using the transitive closure M^+ of the semantics. If the underlying graph of transitions contains cycles, we are able to define a long run cost that gives an average of cost along transition cycles, using a variant of the Perron-Frobenius theorem for idempotent semirings.

Most of the time, the matrix defined by the operational semantics is of infinite dimension. To overcome this problem, we haved defined a framework to

abstract this semantics into a computable one. A correctness relation between concrete and abstract semantic matrices ensures that the result computed from the abstract semantics is an overapproximation of the concrete one.

Finally we have presented an example of abstraction that computes for a given program a safe approximation of the number of cache misses.

Related work

The present work is based on the quantitative abstract interpretation framework developed by Di Pierro and Wiklicky [PW00]. We have followed their approach in modeling programs as linear operators over a vector space, however, we have generalised this to consider operators that are semimodules over semirings. The reason for this generalisation is that such structures naturally arise in cost analyses. Another difference with respect to the body of work by Di Pierro and Wiklicky is that we consider a low-level object-oriented programming language rather than the idealized declarative languages (probablistic concurrent constraint programming and the lambda calculus). This allows us to study a variety of (low-level) quantitative properties such as cache behaviour but requires the incorporation of state abstractions that differ from the kind of abstraction used for analysing declarative languages.

Alt, Ferdinand, Martin, and Wilhelm [AFMW96] have proposed a cache behaviour prediction by abstracting interpretation. We work at a different level, given that their paper is centered on modeling the cache and abstract it properly. In our proposition, all the cache model is hidden behind the function *update*, which still has to be defined. There are three points of their work that we could use almost directly in our framework: the various cache models (*e.g.* direct-mapped, A-way associative) to implement our update function, their abstract domain, in order to design our quantitative abstractions and their observations about caches and writing, in order to develop an accurate model.

Future work

The example computations of costs given in the paper have been done by hand. Future work includes the implementation of the operators of transitive closure and eigenvalue with lazy computation in sparse matrices, which will allow an effective and efficient computation of program cost.

Computing a correct abstraction is an issue, as it is in general for quantitative abstract interpretation. We need to develop a framework that allows to define abstractions on states (either by classical abstraction functions or by equivalence relations) and then automatically obtain the finite, abstract matrix. The definition of abstraction using the Moore-Penrose pseudo-inverse has appealingly strong theoretical foundations but its use in actually computing an abstraction needs to be studied.

Another issue for further work is to relax the correctness criterion so that the abstract estimate is "close" to (but not necessarily greater than) the exact

quantity. This is possible since we have a metric on the abstract property space and hence can estimate the distance of the concrete and the abstract operator. Furthermore, for evaluating the impact *e.g.* of a program transformation, this kind of information would appear to be sufficient.

Finally, it would be worth investigating how to integrate our framework with the notion of resource algebra as defined by Aspinall *et al.* [ABH+].

Acknowledgement

Thanks are due to Guillaume Dufay for his thorough comments on an earlier draft of this paper, and to the referees for their valuable feedback.

References

[ABH+] D. Aspinall, L. Beringer, M. Hofmann, H-W. Loidl, and A. Momigliano. A program logic for resources. *Theoretical Computer Science*, to appear.

[AFMW96] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache Behavior Prediction by Abstract Interpretation. In *SAS'96, Static Analysis Symposium*, volume 1145 of *Lecture Notes in Computer Science*, pages 51–66, September 1996.

[Gau99] Stéphane Gaubert. *Introduction aux systèmes dynamiques à évenements discrets*. INRIA Rocquencourt, 1999.

[JCV] Virtual Machine Specification for the Java Card Platform. http://java.sun.com/products/javacard/specs.html.

[Mar01] R. Marlet. Syntax of the JCVM Language To Be Studied in the SecSafe Project (v1.7). Technical report, April 2001.

[PK89] Peter Puschner and Christian Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Journal of Real-Time Systems*, 1(2):159–176, Sep. 1989.

[PW00] Alessandra Di Pierro and Herbert Wiklicky. Concurrent constraint programming: towards probabilistic abstract interpretation. In *Principles and Practice of Declarative Programming*, pages 127–138, 2000.

[SC99] Timothy Sherwood and Brad Calder. Time Varying Behavior of Programs. Technical Report CS99-630, UCSD, August 1999.