The Dinamica EGO Virtual Machine

Bruno Morais Ferreira, Britaldo Silveira Soares-Filho, Fernando Magno Quintão Pereira

Universidade Federal de Minas Gerais – Avenida Antônio Carlos, 6627, 31.270-010, Belo Horizonte

Abstract

This paper describes DinamicaVM, the virtual machine that runs applications developed in Dinamica EGO. Dinamica EGO is a framework used in the development of geomodeling applications. Behind its multitude of visual modes and graphic elements, Dinamica EGO runs on top of a virtual machine. This machine - DinamicaVM - offers developers a high-level instruction set architecture, featuring elements such as map and reduce, which are typical in the functional/parallel world. Ensuring that these very expressive components work together efficiently is a challenging endeavour. Dinamica’s runtime addresses this challenge through a suite of optimizations, which borrows ideas from functional programming languages, and leverages specific behavior expected in geo-scientific programs. As we show in this paper some of these optimizations deliver speedups of almost 100x, and are key to the industrial-quality performance of one of the world’s most widely used geomodeling tools.

Keywords: Dataflow, Referential transparency, Geomodeling, Virtual Machine

1. Introduction

Dinamica EGO is a framework that supports the development of geomodeling applications [Soares-Filho et al. (2002, 2009, 2013)]. It was first released in 1998, and since then it has grown to enjoy international recognition as an effective and useful framework for geomodeling. It has been used to model carbon emission and deforestation [Carlson et al. (2012); Nepstad et al. (2009)], biodiversity loss [Pérez-Vega et al. (2012)], urbanization and climate change [Huong and Pathirana (2011)], emission reduction [Hajek et al. ]
and urban growth [Thapa and Murayama (2011)]. Testimony of Dinamica’s maturity are the intergovernmental collaborations where it is used. Among its application to public policies in collaboration with governmental institutions in Brazil and abroad we cite the World Bank and the United Nations Development Programme. For instance, the REDD project, which integrates state departments from Bolivia, Peru and Brazil, is using Dinamica to map the southwestern Amazon\(^1\). As another example of relevant use, Dinamica’s simulation of the environmental impact of the Santarém-Cuiabá Interstate (BR 163) has been key to lead the Brazilian government to create a national preservation area along this highway\(^2\). Finally, SimAmazonia, a large effort to model climate change in the Amazon Basin using Dinamica EGO [Soares-Filho et al. (2006)], is part of the IPCC\(^3\) that shared the Nobel Peace prize of 2007 with Al Gore.

Dinamica EGO was created as an assemblage of components implemented in C++, called *functors*, which represent typical *cartographic operations* [Tomlin (1990)]. Each functor has a number of inputs, and produces a number of outputs. The edges that interconnect these ports determine how data flows in a Dinamica’s application. The original Dinamica’s design had one fundamental disadvantage: functors were complex components implementing complete algorithms. Given this coarse granularity, whenever Dinamica’s users needed to implement new behaviors, they had to ask the developers of that framework to code new functors.

To circumvent this shortcoming, we have implemented DinamicaVM, a virtual machine designed to make the Dinamica framework more flexible. The goal of this paper is to describe this virtual machine and its companion programming environment. DinamicaVM contains an instruction set, a library of external components, a scheduler, a garbage collector and an optimizer. The instruction set is built around four functors: *Map*, *Reduce*, *Window* and *While*, plus functors for simple operations such as *And*, *Add*, *Mul*, etc. *Map* and *Reduce* are typical functional-oriented patterns, today heavily used in parallel programming [Dean and Ghemawat (2008)]. Henceforth, to avoid confusion with the maps used as Dinamica’s main data type, we shall call the *Map* functor *Apply*. *Window* returns a neighbourhood within a map.

\(^{1}\)http://csr.ufmg.br/map/

\(^{2}\)http://www.csr.ufmg.br/dinamica/applications/cuiaba-santarem.html

Figure 1: A typical screenshot of an EGO Script program. Components can be expanded in more complex views by the user.

Figure 2: The implementation of Conway’s Game of Life in Dinamica EGO. Letters in parentheses are not part of the original screenshot.

50 While receives a map, and a state, and return a new state which is a function of the input map.
51 Applications built on top of DinamicaEGO manipulate very large data:
maps having $70K \times 70K$ cells are not uncommon [Soares-Filho et al. (2014)]. However, Dinamica’s programming environment has been designed with an initial focus on expressivity, not efficiency. *EGO Script*, the graphical programming language that embodies this environment, ensures referential transparency; hence, fostering a functional – side effect free – user experience. In order to ensure that such abstractions can be implemented efficiently, Dinamica applies a number of optimizations onto chains of functors, after these elements are linked together, but before they are deployed in the runtime system. These optimizations boost Dinamica’s ability to deal with large datasets dramatically, and in some cases, can deliver speedups of over 100x, as we show in Section 4.3.

In Section 3 we introduce, informally, the semantics of each of the four core building-blocks of Dinamica’s applications. The architecture that these four components define finds no equal in other systems built with similar purpose, as we explain in Section 5. In Section 4 we describe the optimizations that ensure that these components run efficiently. Some of these optimizations are not new: they have already being implemented in functional languages [Wadler (1988)]. Nevertheless, we revisit them under the light of a virtual machine customized to handle maps and tables that represent geographic entities. For instance, even though cache optimizations are well studied, we claim the cache-related transformations from Section 4.2 as original contributions of this paper. Furthermore, our techniques to balance the tension between parallelism and memory allocation, which we address in Section 4.3, have not been described before.

This article reports results obtained in five years of implementation effort. During this period, previous work of ours has touched different aspects of the implementation of Dinamica EGO. In [Ferreira et al. (2012)] we discussed the problem of eliminating redundant copies of large data-structures. In Section 4.3 we revisit this optimization, this time equipped with a description of the operational semantics of Dinamica’s instruction set. This semantics lets us show that our implementation is correct. In [Ferreira et al. (2015)] we have described the four programming building blocks of DinamicaVM: *Apply, Reduce, Window* and *While*. This paper provides a more complete coverage of these components, substantiated with runtime numbers, including an empirical comparison with tools used with a similar purpose. In this particular regard, we emphasize that, even though Dinamica EGO is not the only environment used in the development of geoprocessing applications, to the best of our knowledge, it is the only one built on top of a virtual machine.
2. Dinamica in one Example

We illustrate the basic elements of Dinamica EGO via the implementation of Conway’s Game of Life [Gardner (1970)]. Figure 2 shows a screenshot of this implementation. The game happens on a two-dimensional grid of square cells. Each cell can be either active or inactive. The state of all the cells in a grid determine a generation of the game. Generation $g + 1$ is a function of generation $g$. The state of cell $i$ at generation $g + 1$ is determined by the state of this cell’s neighbours, at generation $g$. The neighbourhood of a cell $i$ is the $3 \times 3$ grid centered at $i$, excluding $i$ itself. If this neighbourhood contains 2 or 3 active cells, $i$ will be active in the next generation, otherwise, it will be inactive. Conway’s Game of life is the canonical example of cellular automaton. Dinamica uses, among other techniques, different cellular automata to model land evolution due to human occupation [Soares-Filho et al. (2002)].

The application in Figure 2 reads two inputs: an original map (a.1), plus an integer indicating how many generations of the game will be produced (a.2). Its output is the map after the final generation (a.3). A Repeat functor (b) produces the successive generations of the game. Repeat is a specialization of a more general functor called While, which we describe in Section 3.4. Repeat may either read a new map, or work on data that it sends back to its input port. This feedbacking is implemented by a functor called Multiplexer (b.1). Multiplexers are equivalent to the $\phi$-functions so ubiquitous in compiler analyses and optimizations [Cytron et al. (1991)].

We use an Apply functor (c) to produce generation $g + 1$ of the game, given generation $g$. This component applies some operation on each cell of the input map according to an iterating index (c.1). This operator does not work in-place, unless the runtime environment optimizes it, as we will see in Section 4.3. That is, Apply’s output is a copy of the input map. In this particular example, we are using each Apply’s index $i$ to derive a Window (e) of $3 \times 3$ cells centered at $i$. A Reduce operator sums up the number of active cells on each neighbourhood. If this neighbourhood contains 2 or 3 active cells, then $i$ will be active in the next generation, otherwise it will be inactive.

Some of the functors used in this example deal with data-structures. For instance, Window reads a map, plus an index, and returns a neighbourhood within the map. Other functors operate on individual data. For instance, the if container (f) implements a conditional expression made of several smaller
components, which we have not shown for the sake of readability. These components, e.g., Equal, And, Or and LessThan, implement unary and binary operations. Users program applications in Dinamica EGO by combining these operators. In particular, the control flow of a Dinamica EGO program is determined by how the different instances of Apply, Reduce, Window and While are interconnected. The next section provides more details about each of these components.

3. The Dinamica Virtual Machine

Dinamica EGO runs on top of a virtual machine called DinamicaVM. Figure 3 shows a schematic view of this virtual machine, including its programming environment. Dinamica provides its users with a Graphical User Interface, which is implemented in Java. It is also possible to load and run applications via a suite of command line tools. These applications are ensembles of functors. This virtual machine uses a set of functors, which include Apply, Reduce, While and Window. It also provides a suite of library components, which exist either due to efficiency reasons, or to keep compatibility with applications built prior to Dinamica 2.4.

The runtime environment of Dinamica EGO consists of an optimizer, and ahead-of-time compiler, a scheduler, an interpreter and a garbage collector.
The ahead-of-time compiler converts shading expressions, i.e., expressions that will be applied on every cell of a map or table, into binary code. These expressions are defined by Dinamica’s user through a syntax that we call EGO Script. Translation to binary works in two steps: first EGO Script commands are converted to C++ instructions. Then, these instructions are compiled into binary code by gcc. The scheduler sorts the functors topologically, and forwards this information to the interpreter. If the target machine has multiple cores, then the scheduler parallelizes the execution of functors according to their dependences. Automatic parallelization strategies Apply and Reduce are well studied in the literature [Da Mata et al. (2013); Dean and Ghemawat (2008)], and we shall not discuss them further. The memory occupied by data structures that are no longer used are reclaimed by a garbage collector, which is based on reference counting. Cyclic dependences are not a problem in Dinamica, as it is not possible to create circular structures in it. In this section, we shall explore the four key components that this virtual machine interprets: Apply, Reduce, Window and While.

3.1. Apply

The Apply functor receives two inputs: (i) a map $m$, whose each cell has type $t$, e.g., $m : \text{Map}(t)$, and (ii) a function $f : t \mapsto t$, that transforms the contents of each cell. The functor then applies $f$ onto each cell of $m$, yielding a new map $m'$. Figure 4 (a) provides a visual representation of Apply when used in a program that increments every cell of a matrix of integers.

In Dinamica’s Jargon, Apply is a Container. A Container is a functor that may incorporate other functors. In terms of implementation, Containers follow the “composite” design pattern [Gamma et al. (1995)]. Containers, like every other functor, may have input and output ports. An Apply has only one input port, which receives the target map, and only one output port, which yields the new version of the map. However, contrary to regular functors, Containers have also internal ports, which are used to communicate with their components. Apply has only two internal ports. The first is Step, which returns the contents of a cell of the input map. This element keeps an internal state, in such a way that successive invocations of it return always different elements, which come from contiguous positions in a column-major traversal of the map. The second internal port is Set, which causes a value to be written in a position of the output map that corresponds to the last index visited by Step in the input map.
Figure 4: The four high-level functors in DinamicaVM’s instruction set architecture. (a) Apply; (b) Reduce; (c) Window; (d) While.

Apply is one of the most used components in the Dinamica’s ecosystem. Examples of its use include mapping coordinates into administrative regions such as countries, states and municipalities; mapping altitude into costs; mapping cells into slope values, which are calculated given these cells’s neigh-
bours, etc. Thus, it is very important that this component be implemented efficiently. Each iteration of Apply uses data that is completely independent from the data used by the other iterations. In the PRAM (parallel random-access machine) model, Apply can be implemented to run in $O(1)$. Thus, this functor is implemented to run in parallel.

3.2. Reduce

Reduce takes a map $m$ of type $\text{Map}(t)$, a binary operator $\oplus$, of type $t' \times t \mapsto t'$, and a seed $s$ of type $t'$. It then produces a single value $v$ of type $t'$, such that $v = s \oplus m[0] \oplus m[1] \oplus \ldots \oplus m[n-1]$. In this case, $m[0], \ldots, m[n-1]$ are all the cells in $m$, assuming that $m$ has $n$ cells. Figure 4 (b) shows an application that sums up all the elements in a map of integers, thus producing an integer as its result.

Like Apply, Reduce is also a Container. It has one internal input, Set, which bears the same semantics as the component of same name in Apply. It has one internal output port, Step, which delivers to the internal functors the current value of the iteration. A functor Mux performs the function of the accumulator used to keep track of the current value of a reduction. This functor, if applied on a $n_1 \times n_2$ map, runs sequentially in $O(n_1 \times n_2)$. We can parallelize it for a few operations, which are commutative and associative, such as summation, multiplication, minimum and maximum. In this case, it runs in $O(\ln(n_1 \times n_2))$ in the PRAM model.

3.3. Window

Several applications implemented in Dinamica use small neighbourhoods within a map: finding the average slope of a coordinate, with regard to its neighbours; detecting borders, smoothing images, applying convolutions, finding minimum/maximal cost paths, etc. Therefore, Dinamica provides users with an operator to find neighbourhoods in maps: the Window functor, whose inputs and output are represented in Figure 4 (c).

Window has three inputs ports, which receive a map, the size of a neighbourhood’s side and an anchor, e.g., the coordinate that is the center of the neighbourhood. It outputs a set of cells that constitute the neighbourhood. The vast majority of all the algorithms built in Dinamica use squared neighbourhoods whose sides contain an odd number of elements, and whose center point is the anchor. Because this setup is so common, it is heavily optimized, as we explain in Section 4.2.
3.4. While

Most of Dinamica EGO components are stateless. Data structures are usually copies, instead of being modified in place, for instance. However, there are cases when keeping track of state is desirable for efficiency reason. For instance, a stateless functor to model the movement of a ball, under the force of gravity only, when let loose onto an elevation map, could lead to a formidable number of copies of the target map. Dinamica avoids such situations by providing users with a statefull functor – the While iterator. The graphical representation of this element can be seen in Figure 4 (d).

An While has one input port, which receives an index set. An index set is a collection of sortable elements that index a data-structure: coordinates on a 2D or 3D map, points on a line, rows in a table, etc. The While has an internal Step port, which keeps track of the elements in the index set still to be processed. While also has an internal Set port, which may update the index set with new elements. Thus, in practice, While implements worklists: as long as the worklist is not empty, this functor perform an action. DinamicaVM uses While, for instance, to implement searches by depth and breadth in maps. A very common index set consists in contiguous sequences of integer numbers. This case is so common that we have a specialization of While – the Repeat functor – optimized to use it.

3.5. Specific Components

The While functor seen in Section 3.4, plus the binary and unary operators of Dinamica EGO define a Turing complete language. Turing completeness comes from the fact that these functors subsume the While formalism, typically used to illustrate programming language semantics [Nielson and Nielson (1992)]. Nevertheless, there are applications that do not translate easily into amalgamations of these few elements. In particular, there exist behaviors that our optimizations from Section 4 do not derive automatically. Thus, Dinamica EGO provides a few specific – higher-level – components which are not implemented as combinations of the four previously described functors. These components are also necessary to keep compatibility with applications developed prior to Dinamica v2.4, which did not use the virtual machine that we describe in this paper.

For instance, Dinamica EGO contains a functor called CalcCostMap, which constructs cost-surface maps out of raster images [Eastman (1989)]. The cost calculation problem is very common in land use simulations. The problem has two inputs: a friction map, and a map of source points. The outcome
of a cost calculation is a map that tells us, for each cell, the minimum cost to reach one of the source cells. This problem emerges, for instance, whenever it is necessary to determine the paths that roads must traverse to link each interior city to a given set of harbours. The cost calculation problem is usually solved via chaotic iterations. We start with a solution map in which each cell is mapped to an infinitely large cost. Then, we iterate successive applications of the operator below, until a fixed point is reached:

\[
\text{cost}(x) = \min \left\{ \begin{array}{l}
\text{cost}(x) \\
\text{cost}(y) + \text{friction}(x) \\
\sqrt{2} \times (\text{cost}(z) + \text{friction}(x))
\end{array} \right. 
\]

We have implemented the chaotic iterations as successive applications of four loops, whose iteration space is given in Figure 5(a). Each of these loops is parallelized independently. Figure 5(b) shows the pattern of dependences in the first loop, which traverses the map from the upper-left corner towards the lower-right corner. The execution runtime has a predefined number of available workers. Each worker has a task queue and can run a single task at a time. Tiles that must be processed are organized as a digraph of pending tasks. Tasks become eligible to run after all their dependencies have been processed. If a thread is idle, then it reclaims a tile that has no pending dependencies. This pattern continues until all the tiles have been processed. If the task queue of a processor becomes empty, then it might steal work from the queue of other processor. If a thread cannot steal any task, then it votes for the end of the computation. The computation terminates when a consensus is achieved among all the workers.

4. Optimizations

In order to be accepted by its users, the Dinamica Virtual Machine had to be at least as efficient as the original implementation of Dinamica’s runtime, which was used until Dinamica v2.4, last released in 2014. The key to achieve this efficiency are optimizations. Not only the implementations of Apply, Reduce, While and Window are highly engineered, but also the way that these components interact is optimized. All the optimizations that we
describe here, except the prefetching from Section 4.2, are applied after an
application has been type checked, but before its modules start to run. EGO
Script’s type system is static, i.e., types are known before an application
starts running. Furthermore, this language does not support the dynamic
loading of components, like PHP or JavaScript do. Therefore, we know the
size of each map cell that is manipulated within an application, and we have
a complete view of the dependence graph between components. This knowl-
edge is important to generate code for the routines that read data, and move
data between different functors. In this section we briefly touch the most im-

---

4Prefetching is part of the implementation of Window; it does not requires any program transformation.
important transformations that DinamicaVM applies onto its building blocks before an application runs. All the numbers that we show alongside the description of the optimization have been obtained in an Intel Core i5 with clock of 2.67GHz and 8GB of RAM.

4.1. Fusion

Fusion is a transformation that we implement onto combinations of Apply + Apply, and Apply + Reduce. This optimization is common in functional languages [Wadler (1988)]. It consists in combining the operators used by different functors in the following way:

\[
\text{Apply } f \ (\text{Apply } g \ m) = \text{Apply } (f \circ g) \ m
\]

\[
\text{Reduce } s \ f \ (\text{Apply } g \ m) = \text{Reduce } s \ f' \ m
\]

where \( f = \lambda(x, y). x \oplus y \)

and \( f' = \lambda(x, y). g(x) \oplus y \)

Function fusion is not a new idea of ours. In fact, we are using a very limited form of fusion, as we only apply it to two combinations of functions. More extensive implementations have been described, for instance, by Jones et al. [Gill et al. (1993)]. Nevertheless, our simple implementation of function fusion is enough to speed up some of Dinamica’s applications dramatically.

Figure 6 illustrates some of these performance gains. In this example we are using three very simple instances of Apply and Reduce:

\[
\text{Inc } m = \text{Apply } (\lambda x. x + 1) \ m
\]

\[
\text{Div } m = \text{Apply } (\lambda x. x/2.17) \ m
\]

\[
\text{Sum } m = \text{Reduce } 0 \ (\lambda(x, y). x + y) \ m
\]

In the figure we use random square matrixes of integers having sides of 5.0K, 7.5K and 10.0K cells. Without fusion DinamicaVM takes 9.940 seconds to \text{Div} \circ \text{Inc} every cell of the \(10^3 \times 10^3\) matrix. Once fusion is activated, this time drops to 5.222 seconds. In the case of \text{Reduce}, gains are of similar nature. It takes us 7.294 seconds to \text{Sum} \circ \text{inc} the matrix with 10K rows without fusion, and 2.998 seconds if we use fusion. These gains are due to two factors: the elimination of intermediate data structures, and the improved locality. Concerning the first factor, fusion automatically eliminates the need to copy the map that the leftmost Apply produces. As for locality, the input map will be traversed only once instead of twice. Indeed, only one iteration is necessary for any sequence of applications of the Apply functor, e.g.:
Figure 6: Example of fusion. The Apply operator is always the increment function, and the Reduce operator is the sum of integers.

apply \( f_1 \ldots (\text{apply } f_n m) \ldots \) = apply \( (f_1 \circ \ldots \circ f_n) m \)

Fusion’s improvements are proportional to the complexity of the kernel operator used in Apply or Reduce. The more complex is the computation used inside these functors, less performance gains we shall observe. The composition below illustrates this trend:

Apply Normalize (apply calcSlope \( m \))

Normalize is a simple linear function of the input value, but the calcSlope operation is a substantially more complex functor present in the Dinamica EGO library. It applies a Reduce over the output of a Window for each index in the input map. For a 7500 × 7500 input map, fusion gives us 6% of speedup in this example.

4.2. Window Optimizations

Window is a heavily used functor; thus, it is natural that it be optimized. DinamicaVM applies two optimizations on Window: prefetching and unrolling. The latter is only applicable on 3×3 instances of Window. Prefetching avoids unnecessary trips to main memory in order to collect the pieces of a
Figure 7: The three-lines cache. The dashed arrows show line pointers in the previous iteration of Window. The solid arrows show the pointers in the current iteration. (a) Input map. (b) Cached lines. (c) Center of $3 \times 3$ window.

squared window view. Unrolling removes unnecessary control flow from the most common type of view that we have observed in Dinamica’s applications.

Prefetching. most of the applications that use Window slide it over an image in row-major order, that is, starting from the upper-left corner of an image, and going to its lower-right corner. This pattern is so common because it is the default order in which Apply and Reduce evaluate the elements of a map. Our optimizer ensures that each cell of a Window is read only once from main memory, if Window is used in that way. To ensure this property, we pre-fetch the lines that will be traversed by Window.

Figure 7 illustrates this approach for a $3 \times 3$ instance of Window. In this example, each time Window is called, it reads nine elements of the input map. Instead of fetching this data when Window is created, we pre-fetch three entire lines of the map, and let Window slide on these lines. Once Window reaches the rightmost border of the image, we discard the topmost line, and read one line more from main memory. If Window works with submatrices of $n$ rows, then we should, in principle, keep $n$ lines in cache. However, most of the applications available in the Dinamica’s ecosystem work with $3 \times 3$ windows. Thus, we chose to work with only three lines at a time. Consequently, larger instances of Window may lead to multiple trips to the main memory.

The prefetching is only necessary for maps that cannot fit entirely in the L0 cache. This is usually the case in Dinamica, as maps are very large, and each of their cells contains a non-negligible amount of data, which include
colour patterns and geographic information. In the absence of this optimization, a $n \times n$ Window causes each – non-border – map cell to be read $n^2$ times. Usually data in the same row of Window are fetched only once to the L0 cache; however, data may be fetched more times if it happens to be read as part of different rows of Window.

Performance Improvement due to Prefetching. Figure 8 shows the performance of three different instances of an image smoothing algorithm. The algorithm uses a $3 \times 3$ convolution matrix that does simple average to implement smoothing. The smoothing filter returns, for a given cell $i$, the average of all the immediate neighbours of $i$ plus $i$ itself. The three instances of the algorithm are:

- **Library** – the algorithm was implemented using a monolithic filter available in Dinamica’s library.
- **DVM - No Opt** – our algorithm, built as the following combination of functors:
  \[
  \text{Smooth } m = \text{Apply} \left( (\text{Reduce Average}) \circ \text{Window} \right) m
  \]
- **DVM - Prefetching** – the previous implementation, with prefetching enabled in DinamicaVM.

Figure 8 varies the number of times that the image is smoothed. Each of these times requires one application of the smoothing algorithm. As we
observe, our optimization speeded up Window by a factor that reached 3.9x for 40 applications of the smoothing algorithm. It even improved on the library component, which has a much more monolithic design. Our optimized version of image smoothing is 1.7x faster.

**Unrolling.** the most used type of Window is a $3 \times 3$ squared view of a map, with anchor in the center. Because this pattern is so common, we use a special implementation of it, which has no control flow. This implementation reads a chunk of memory that is large enough to fit each one of the nine indices to be processed. It then divides this memory into nine pieces, and fills up the positions in the map view with them. The size of memory that must be read is determined by the ahead-of-time compiler, before Window is invoked, but after the type of its input is already known.

Figure 9 shows the performance gains obtained due to unrolling and prefetching when applied on the implementation of Conway’s Game of Life. This application was discussed in Section 2. To provide some perspective to the reader, we show the runtime of the implementation of Conway’s automaton in Dinamica 2.4, before the virtual machine was released. In this case,
the game is implemented with a set of functors from the library. The series “VM Base” shows our application running on the virtual machine without either prefetching or unrolling. In this case, DinamicaVM is 59% slower than Dinamica v2.4’s implementation of Conway’s game. However, once we turn on optimizations, we see substantial gains: Unrolling already puts DinamicaVM’s times on pair with v2.4’s results. And the combination of unrolling and prefetching makes us 57% faster than the old version of Dinamica. In other words, the two optimizations makes our virtual machine 3.6x faster.

4.3. Copy Elimination

Dinamica EGO, like usual dataflow programming languages [Johnston et al. (2004)], provides to users the notion of referential transparency. In other words, the output produced by a functor depends only on its inputs. Referential transparency facilitates the development of models in Dinamica, because it ensures that the output of functors is always deterministic. This determinism helps developers to reason about the runtime behavior of programs, as it makes the evaluation order of independent functors immaterial. We say that two functors, \( f_0 \) and \( f_1 \), are independent if neither of them reads data modified by the other. As a consequence of this functional programming style, functors can be easily reused as black-box components.

A typical way to ensure referential transparency is to rely on immutable data structures [Sondergaard and Sestoft (1990)]. If the contents of a data structure must be updated, then the whole data is copied into a new memory location. However, this alternative is too expensive in Dinamica, because geoscientific data structures – maps and tables – tend to be very large. Therefore, a copy minimization algorithm is essential to allow these applications to scale. In this section we describe the algorithm that is implemented by Dinamica EGO to eliminate copies.

Dinamica’s main data structures are maps (images) and tables. If a functor updates a data-structure we call it a copy writer, \( \text{CWriter} \), because it produces a new version of the data-structure that it updates. DinamicaVM v1.9 introduces the notion of an in-place writer, or \( \text{IWriter} \) for short. Functors of this type do not copy entire data-structures; instead, they update individual map cells or table entries with new information. Replacing \( \text{CWriters} \) by \( \text{I Writers} \) is safe, as long as this replacement does not introduce conflicts. If functor \( g \) reads data produced by functor \( f \), then we say that \( f \) depends on \( g \). These dependence relations determine the graph that guides DinamicaVM’s scheduler. The scheduler is the part of DinamicaVM responsible for deciding
Figure 10: (i) Program with write-write conflict. (ii) Program without conflicts. (iii) Program with potential conflict, depending on the scheduling. (iv) Program after elimination of conflict in (i). (v) Program after elimination of conflict in (iv).

the order in which functors are fetched and executed. We say that a conflict starts at a functor \( f \) whenever the application’s graph contains two paths leaving \( f \), and at least one of these paths ends in a IWriter without passing first through a CWriter. The last condition, no CWriter on the path, is necessary to provoke the conflict, otherwise the data would be copied, before being modified in-place.

Figure 10 (i) illustrates a conflict. Independent on which functor is scheduled to run first, (b) or (c), one of these IWriters will update the data that the other reads. Because (b) and (c) are writers, we call the conflict write-write. The program in Figure 10 (ii) is conflict-free: functor (b) copies the input
data, before modifying it. Hence, functor (c) reads the original data that (a) produces. Notice that the absence of conflict depends on (b) running before (c). To ensure this property, we have added a fake dependence edge from (b) to (c). A fake dependence edge from functor $f_0$ to $f_1$ tells DinamicaVM's scheduler to execute $f_0$ before $f_1$; however, there is no actual flow of data from the former into the latter.

We solve conflicts by transforming one of the writers in a copy-writer, giving preference to the nodes outside loops. In Figure 10(ii), we have transformed $b$ in this way. We have also created a fake dependence edge $\overrightarrow{bc}$, to ensure that $c$ will not read an updated instance of the output that (a) produces. In general, if functor $f$ is the source of $n$ paths that end in writers, and $f_1, \ldots, f_n$ are the writers at the end of these paths, we must transform $n - 1$ of them in C Writers, and must schedule the unchanged node to execute first. We ensure this scheduling by inserting new edges in the dependence graph. If it is not possible to force a scheduling without creating cycles in the dependence graph, then every writer is transformed. For instance, in Figure 10(iv) we have a conflict between functors (c) and (d). It would be cheaper to transform (d) into a C Writer, as the copy would happen outside a loop. However, to do this transformation, we would need to insert a fake edge from (d) to (c). This edge would create a cyclic dependence in our dependence graph. Such a problem does not happen if we convert (c) into a C Writer. This solution, adopted in Figure 10(v), is the only possible way to solve a conflict in this example. In the rest of this section, we describe the algorithm that is used in Dinamica EGO since version 1.9.

4.4. A Simplified Semantics Model of EGO Script

In order to describe the copy minimization problem, we define an abstract machine that subsumes the core aspects of EGO's semantics. This machine is given in Figure 11. The relation $\rightarrow$, defined by Rules PRGN and PRGX, defines the evaluation of a program. Programs are formed by a list of functors, e.g., $[F|S]$, where $F$ is the first functor in the list, and $S$ is the rest of it. We call this list a scheduling of components. A program is evaluated in an environment $\Gamma$, that maps names of functors to indexes that these functors use in a store $\Sigma$. In our simplified model, the store maps these indexes to integer numbers; in the actual implementation, the store contains large data-structures. The $\Gamma$ environment is not fixed: we can imagine that all the names in Dinamica are pointers to data in the store, and during the execution of a program these pointers, i.e., the bindings in $\Gamma$, might change.
The result of evaluating a program is a new environment, a new store, and the bindings produced in the output environment $\Theta$. We use the letter $N$ to denote the name of a functor, $I$ to denote an index in the store, and $V$ to denote a storable value, i.e., an integer in our case. Thus, we have $\Gamma : N \mapsto I$, $\Sigma : I \mapsto V$, and $\Theta : N \mapsto V$.

The relation $e \rightarrow$ defines the evaluation of individual functors. Figure 11 defines five different types of functors: loader, reader, writer, container and cwriter. We call this last functor a copy-writer. The functors reader, writer and cwriter are bound to a name $N$ plus a list of predecessors $P$. The list of predecessors in a functor $f$ contains names of functors that produce data that $f$ consumes. The predecessor relation defines the control flow graph, a directed graph that determines how data flows from one component to the other. Loaders are the source of new data in our formalism. The evaluation of loader($N$), in Rule Load, stores 0 at $\Sigma[\Gamma[N]]$. The evaluation of a reader or writer starts by computing the maximum value among all the storage cells in predecessors of the functor. We call the predecessor functor that has the largest value the producer. Writers dump this value, augmented by one, in the store, whereas readers dump it, without modification, in the output channel. The evaluation of writers or readers might change the environment $\Gamma$. The index $\Gamma[N]$ associated with a functor $f$ that is either a reader($N, P$) or a writer($N, P$), is always $\Gamma[p]$, where $p \in P$ is the current producer. Rules MaxN, MaxL and MaxG show how this index is computed. The evaluation of cwriter creates new indices. This functor simulates the copy of data-structures in our mini-language. As we see in Rule CWRT, a cwriter always produces a new index in the store. Old indices, even if no longer necessary, are not recycled. In practice, memory is recycled by Dinamica’s garbage collector, which is based on reference counting.

Loops, in Dinamica EGO, are produced by containers. The functor container($K, N, In, Fb, i|s, S$) evaluates $K$ times the sequence of functors in the list $S$. Each container is bound to a special functor, the multiplexer, which we represent as a four-elements tuple ($N, In, Fb, i|s$). This functor chooses an input either from a producer $In$ outside the container, if its state is $s$ (Rule ConS), or from a feedback producer $Fb$, in the container, if its state is $i$ (Rule ConI). Because these nodes are the entry points of loops, they are equivalent to the $\eta$ functions used in the Gated Single Assignment program representation Ottenstein et al. (1990); Tu and Padua (1995). An EGO Script program is well-formed if the only cycles that its flow graph contains are due to back-edges reaching multiplexers. Figure 12 illustrates
\[
\text{max}(\lfloor N \rfloor, \Gamma, \cdot) \rightarrow \Gamma[N]
\]

\[
\text{max}(R, \Gamma, \Sigma) \rightarrow I_0 \quad \Gamma[N] = I_1 \quad \Sigma[I_0] = V_0 \quad \Sigma[I_1] = V_1 \quad V_0 < V_1
\]

\[
\text{max}(\lfloor N \rfloor, R, \Gamma, \Sigma) \rightarrow I_1
\]

\[
\text{max}(R, \Gamma, \Sigma) \rightarrow I_0 \quad \Gamma[N] = I_1 \quad \Sigma[I_0] = V_0 \quad \Sigma[I_1] = V_1 \quad V_0 \geq V_1
\]

\[
\begin{align*}
\Gamma[N] = I & \quad \Sigma_1 = \Sigma_0[I \rightarrow 0] \\
\text{(loader}(N), \Gamma, \Sigma_0, \Theta) & \rightarrow (\Gamma, \Sigma_1, \Theta)
\end{align*}
\]

\[
\text{max}(P, \Gamma_0, \Sigma) \rightarrow I \quad \Sigma[I] = V \quad \Gamma_1 = \Gamma_0[N \rightarrow I] \quad \Theta_1 = \Theta_0[N \rightarrow V]
\]

\[
\begin{align*}
\text{(reader}(N, P), \Gamma_0, \Sigma, \Theta_0) & \rightarrow (\Gamma_1, \Sigma, \Theta_1)
\end{align*}
\]

\[
\text{max}(P, \Gamma_0, \Theta) \rightarrow I \quad \Sigma_0[I] = V \quad \Sigma_1 = \Sigma_0[I \rightarrow V + 1] \quad \Gamma_1 = \Gamma_0[N \rightarrow I]
\]

\[
\begin{align*}
\text{(writer}(N, P), \Gamma_0, \Sigma_0, \Theta) & \rightarrow (\Gamma_1, \Sigma_1, \Theta)
\end{align*}
\]

\[
\begin{align*}
K > 0 \quad \Gamma_0[In] = I & \quad \Gamma_1 = \Gamma_0[N \rightarrow I] \\
\text{(container}(K', N, In, Fb, i, S), \Gamma_2, \Sigma_1, \Theta_1) & \rightarrow (\Gamma_3, \Sigma_2, \Theta_2)
\end{align*}
\]

\[
\begin{align*}
K' = K - 1 \quad \text{(container}(K', N, In, Fb, s, S), \Gamma_0, \Sigma_0, \Theta_0) & \rightarrow (\Gamma_3, \Sigma_2, \Theta_2)
\end{align*}
\]

\[
\begin{align*}
K > 0 \quad \Gamma_0[Fb] = I & \quad \Gamma_1 = \Gamma_0[N \rightarrow I] \\
\text{(container}(K', N, In, Fb, i, S), \Gamma_2, \Sigma_1, \Theta_1) & \rightarrow (\Gamma_3, \Sigma_2, \Theta_2)
\end{align*}
\]

\[
\begin{align*}
K' = K - 1 \quad \text{(container}(K', N, In, Fb, s, S), \Gamma_0, \Sigma_0, \Theta_0) & \rightarrow (\Gamma_3, \Sigma_2, \Theta_2)
\end{align*}
\]

\[
\begin{align*}
(F, \Gamma_0, \Sigma_0, \Theta_0) & \rightarrow (\Gamma_1, \Sigma_1, \Theta_1) \\
(S, \Gamma_1, \Sigma_1, \Theta_1) & \rightarrow (\Gamma_2, \Sigma_2, \Theta_2)
\end{align*}
\]

\[
([F]S, \Gamma_0, \Sigma_0, \Theta_0) \rightarrow (\Gamma_2, \Sigma_2, \Theta_2)
\]

Figure 11: The operational semantics of EGO Script.
a) \( \Gamma[a] = 0, \Sigma[0] = 0 \)
b) \( \Gamma[b] = 0, \theta[b] = 0 \)
c) \( \Gamma[c] = 1, \Sigma[1] = 1 \)
d) \( \Gamma[d] = 1 \)
e) \( \Gamma[e] = 1 \)
f) \( \Gamma[f] = 1, \Sigma[1] = 2 \)
g) \( \Gamma[g] = 1, \Sigma[1] = 4 \)
d) \( \Gamma[d] = 1 \)
e) \( \Gamma[e] = 1 \)
f) \( \Gamma[f] = 1, \Sigma[1] = 6 \)
g) \( \Gamma[g] = 1, \Sigma[1] = 7 \)
h) \( \Gamma[h] = 1, \theta[h] = 7 \)
i) \( \Gamma[i] = 2, \Sigma[2] = 8 \)
j) \( \Gamma[j] = 2, \Sigma[2] = 8 \)
k) \( \Gamma[k] = 2, \theta[2] = 8 \)

Figure 12: Evaluation of an EGO program. Functors are scheduled in top-to-bottom order. At each execution cycle we show, for the evaluated functor, the mappings that have been modified. We have underlined the creation of new indices. Initially we have \( \Gamma[a] = 0 \).

the evaluation of a mini-EGO program.

Correct evaluation of EGO programs. We say that an EGO program is canonical if, and only if, it does not contain functors of the type \texttt{writer}. In other words, every functor that dumps a value different than zero in the store \( \Sigma \) must be a \texttt{cwriter}. If we remove back-edges, then any well-formed EGO program is a diagraph; hence, it can be sorted topologically. The execution
of a canonical program guarantees that any possible topological ordering will always give the same result. In this case, referential transparency holds because data is copied whenever it reaches a new writer. We say that a program \((S, \Gamma, \Sigma, \Theta)\) is correct if it produces the same \(\Theta\) as an equivalent program \((S', \Gamma, \Sigma, \Theta)\), in which every \texttt{writer}(N, P) \in S has been replaced by \texttt{cwriter}(N, P). The program in Figure 12 is correct. In this example, if we replace any \texttt{cwriter} by a \texttt{writer}, then the program will no longer be correct. We are interested in minimizing the number of copy-writers in the program. Definition 1 states this problem.

**Definition 1.** COPY MINIMIZATION WITH OPEN SCHEDULING \([\text{CMOS}]\)

*Instance:* a set of functors \(F\), plus a natural \(K\).

*Problem:* find a scheduling \(S\) of the functors in \(F\) that allows one to replace at least \(K\) writers in \(F\) by copy-writers, preserving correct execution.

**Theorem 1.** \(\text{CMOS}\) is NP-complete.

**Proof:** The proof consists in reducing this problem to the coloring of circular-arc graphs \([\text{Garey et al. (1976)}]\), following similar approach adopted by Pereira and Palsberg \([\text{Pereira and Palsberg (2006)}]\). Full details appear in our earlier work on this subject \([\text{Ferreira et al. (2012)}]\). \(\square\)

Because \(\text{CMOS}\) is NP-complete, we solve it via heuristics. In order to create a program like that in Figure 12, we start with a set of functors in which every writer is of type \texttt{writer}. Then, by following the rules that we will introduce in Section 4.5, we replace some of these functors by \texttt{cwriters}. To illustrate this transformation, we assume that our starting configuration is that seen in Figure 14(a).

4.5. Finding a good scheduling of EGO components.

The dependence graph of a set of functors \(F\), is the digraph that we obtain by removing all the back-edges that reach multiplexers in \(F\). If \(f\) and \(g\) are functors, then we say that \(f < g\) if, and only if, \(f\) must precede \(g\) in the topological ordering of the dependence graph. In order to build an executable scheduling \(S\) of the components in \(F\), we must ensure that, for any \(f, g \in F\), such that \(f < g\), we have that \(f\) precedes \(g\) in \(S\). Given this restriction, we build a scheduling in two steps: first, we perform a dataflow analysis to detect data access conflicts. Second, we run a dependency analysis to schedule functors that write data after functors that read such data.
Dataflow Analysis. The objective of the dataflow analysis is to identify paths in the program dependence graph that might lead to functors where data is written or read. We run this analysis for each data-structure that the program manipulates. Because different data-structures are analyzed independently, we call this analysis a Partitioned Variable Problem, following Zadeck’s nomenclature Zadeck (1984). Thus, henceforth we assume that there is only one data-structure being propagated along the EGO Script program. The equations in Figure 13 define the transfer functions used in our dataflow analysis.

We let the constraint variable \([f]\) be the abstract state of the data in the edges that reach the functor \(f\). Our dataflow analysis considers two abstract states: \(r\) and \(w\). The state \(r\) indicates a path along which the data might be read, i.e., dumped by a reader into the output channel \(\Theta\). The state \(w\) indicates a path along which the data might be updated by a writer. Notice that copy-writers do not propagate abstract information, as they never overwrite data, always producing a new location to store it. Figure 14(a) shows the result of the dataflow analysis performed in our running example.

Data Access Conflicts. the dataflow analysis gives us the subsidies to determine which nodes might be the source of data access conflicts. A conflict
Figure 14: (a) Result of the dataflow analysis in our running example. (b) Final scheduling. Pseudo-dependences are represented by dashed arrows. The dependence graph has no back-edges.

might occur whenever a careless allocation of stores might compromise the correct execution of the EGO program. In structural terms, there exists a conflict with origin at functor $f$ whenever there are two paths of channels leaving $f$, and at least one of these paths ends in a writer, without going across a cwriter. We call this functor, at the end of the path, a dangerous writer. We say that a functor $f$ is the source of a data conflict if it has two successors, $f_0$ and $f_1$, and either: (i) $[f_0] = r$ and $[f_1] = w$, or (ii) $[f_0] = [f_1] = w$. Notice that we write data conflict to avoid confusion with the term data race used in parallel programming. Conflicts happen even if we have just one thread of execution, depending on how functors are scheduled to run. If a functor $f$ is the source of a data-conflict, then we might have to replace one or more dangerous writers by copy-writers. However, there are
First, we can avoid copy-writers in pseudo write-write conflicts. If two functors, \( f_0 \) and \( f_1 \), might participate in a write-write conflict that originates at a functor \( g \), then we have a pseudo conflict whenever \( f_0 \leq g < f_1 \). That is, \( f_1 \) must be scheduled necessarily after \( f_0 \), due to the topological ordering of the dependence graph. The rational behind these rules is straightforward: \( f_1 \) must, necessarily, write on the data that \( f_0 \) updates. As an example, in Figure 14(a), neither the functor \( f \), nor the functor \( g \), need to be transformed in copy-writers, as they are the sources of pseudo-conflicts.
The topological ordering between functors must take loops into consideration. In Figure 15(a), it seems like \( d > b \); however, this program contains a true write-write conflict. If we replace \( d \) by a copy-writer, then, as we see in Figure 15(b), we obtain a different output. If we unroll the loop once, as in Figure 15(c), then we see that node \( c \) is the source of a true write-write conflict. If node \( d \) were outside the loop, as in Figure 15(d), then we would have a pseudo-conflict. Even if we unroll the loop once, in this case, as Figure 15(e) shows, node \( d \) is always after any execution of node \( c \).

We solve true write-write conflicts by transforming one of the writers in a copy-writer, giving preference to the nodes outside loops. In Figure 14(b), we have transformed \( i \) in this way. We have also created a dependence edge \( i \to j \), to ensure that \( i \) will not read an updated instance of its input. In general, if functor \( f \) is the source of \( n \) dangerous paths, and \( f_1, \ldots, f_n \) are the writers at the end of these paths, we must transform \( n - 1 \) of them in copy-writers, and must schedule the unchanged node to execute first. We ensure this scheduling by inserting new edges in the dependence graph. If it is not possible to force a scheduling without creating cycles in the dependence graph, then every writer is transformed.

Once we are done inserting copies to eliminate write-write conflicts, we proceed to handle read-write conflicts. We can eliminate a copy in a read-write conflict, whenever it is possible to schedule the writer to execute after the reader. For each functor that is a potential source of a data conflict, we try to force dependencies among the components involved in the problem, in hopes of finding a scheduling that makes the copy unnecessary. We say that the sequence of functors \( f_0, f_1, \ldots, f_n \) makes up a reading path if \( f_n \) reads data that originates at \( f_0 \), and no functor along this path writes on this data, i.e., \([f_i] = r\), \(1 \leq i \leq n\). Let's assume that our flow graph contains two edges, \( \overrightarrow{fr} \) and \( \overrightarrow{fw} \), such that \([f_r] = r\) and \([f_w] = w\). Let \( f_n \) be the last functor in any reading path that starts at \( f_r \), and let \( f_m \) be the first writer in any dangerous path that starts at \( f_w \). If we can create an edge \( \overrightarrow{fm} \) without closing a cycle in the dependence graph, then we can schedule \( f_m \) to run after \( f_n \). In this case, we eliminate a potential conflict, as \( f_m \) will only write the data after \( f_r \) reads it. As an example, in Figure 14(b) we have created pseudo-edges \( \overrightarrow{hi} \) and \( \overrightarrow{hj} \) to avoid a read-write conflict.

**Performance Improvement due to Copy Elimination.** In this section, we analyzed two benchmarks. The first is Dinamica’s “getting-started” example,
<table>
<thead>
<tr>
<th>MS</th>
<th>D</th>
<th>$T_u$</th>
<th>$T_o$</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>20,000</td>
<td>23</td>
<td>5</td>
<td>4.60</td>
</tr>
<tr>
<td>120</td>
<td>28,800</td>
<td>44</td>
<td>7</td>
<td>6.28</td>
</tr>
<tr>
<td>140</td>
<td>39,200</td>
<td>77</td>
<td>10</td>
<td>7.70</td>
</tr>
<tr>
<td>160</td>
<td>51,200</td>
<td>125</td>
<td>13</td>
<td>9.61</td>
</tr>
<tr>
<td>180</td>
<td>64,800</td>
<td>196</td>
<td>17</td>
<td>11.52</td>
</tr>
<tr>
<td>200</td>
<td>80,000</td>
<td>289</td>
<td>21</td>
<td>13.76</td>
</tr>
<tr>
<td>220</td>
<td>96,800</td>
<td>418</td>
<td>25</td>
<td>16.72</td>
</tr>
<tr>
<td>240</td>
<td>115,200</td>
<td>580</td>
<td>29</td>
<td>20.12</td>
</tr>
<tr>
<td>260</td>
<td>135,200</td>
<td>791</td>
<td>34</td>
<td>23.26</td>
</tr>
<tr>
<td>280</td>
<td>156,800</td>
<td>1,083</td>
<td>39</td>
<td>27.76</td>
</tr>
<tr>
<td>300</td>
<td>180,000</td>
<td>1,451</td>
<td>45</td>
<td>32.24</td>
</tr>
</tbody>
</table>

Figure 16: MS: size of map’s side, in number of cells. Each map is a square matrix. D: Number of dynamic copies without optimization. $T_u$: Execution time without optimization (sec). $T_o$: Execution time with optimization (sec). S: Speedup.

which derives statistics from a map, like a typical spreadsheet application does. Values from the map are first stored in a table. Then, the model iterates over this table, computing data such as number of non-null values, sum of these values, arithmetic mean, variance and standard deviation. Figure 16 shows the behavior of this model for different map sizes. Without our optimization, each table row is updated by a CWriter, which copies the entire table. The larger the map, the higher the number of dynamic copies, and the higher the execution time. This model had two C Writers, and both could be replaced by IWriters. For our largest example map, having 300x300 cells, we could improve execution time by more than 32x.

The second case study comes from a professional model used to detect hilltops in altimetric maps. This model receives two inputs: an elevation map and a vertical resolution value. The EGO script divides the elevation map vertically into slices of equal height. This height is defined by the vertical resolution. Before finding hilltops, this script performs other analyses to calculate average slopes, to compute the area of each region and to find the average elevation of each region. The model outputs relative height, plateau identifiers, hilltops, local minima and local maxima. This EGO script uses tables intensively; hence, data replication was a bottleneck preventing it from scaling to higher resolutions before the deployment of our optimization.

Figure 17 shows the speedup that we obtain via our copy elimination
<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>$T_u$</th>
<th>$T_o$</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>1,956</td>
<td>20</td>
<td>20</td>
<td>1.00</td>
</tr>
<tr>
<td>15</td>
<td>2,676</td>
<td>28</td>
<td>26</td>
<td>1.07</td>
</tr>
<tr>
<td>13</td>
<td>3,270</td>
<td>30</td>
<td>29</td>
<td>1.03</td>
</tr>
<tr>
<td>11</td>
<td>4,677</td>
<td>32</td>
<td>32</td>
<td>1.03</td>
</tr>
<tr>
<td>10</td>
<td>6,126</td>
<td>36</td>
<td>36</td>
<td>1.00</td>
</tr>
<tr>
<td>9</td>
<td>9,129</td>
<td>39</td>
<td>36</td>
<td>1.08</td>
</tr>
<tr>
<td>8</td>
<td>15,150</td>
<td>49</td>
<td>39</td>
<td>1.25</td>
</tr>
<tr>
<td>7</td>
<td>29,982</td>
<td>87</td>
<td>50</td>
<td>1.74</td>
</tr>
<tr>
<td>5</td>
<td>137,745</td>
<td>995</td>
<td>76</td>
<td>13.09</td>
</tr>
<tr>
<td>4</td>
<td>279,495</td>
<td>4,817</td>
<td>116</td>
<td>41.52</td>
</tr>
<tr>
<td>3</td>
<td>518,526</td>
<td>18,706</td>
<td>197</td>
<td>94.95</td>
</tr>
</tbody>
</table>

Figure 17: V: Vertical resolution (m). D: Number of dynamic copies without optimization. $T_u$: Execution time without optimization (sec). $T_o$: Execution time with optimization (sec). S: Speedup.

algorithm. We have run this model for several different vertical resolution values. The smaller this value, more slices the map will have. This model has three CWriters, but given that they happen inside loops, the dynamic number of copies is much greater. Figure 17 shows the number of dynamic copies in the unoptimized program. Our optimization has been able to eliminate all the copy-writers. The end result is a speedup of almost 100x, as we observe in the fourth column of Figure 17.

5. Related Work

Other Geomodeling Tools. There exist many different tools that support the development of land use models. Some of them enjoy commercial success; others are popular in the academia. We are aware of four frameworks having a moderate to large user base that might compete with Dinamica in the geomodeling niche: ArcGis\(^5\), Idrisi\(^6\), and PCRaster\(^7\). A direct comparison between all these tools is not possible, because they use different algorithms to perform simulations. Furthermore, there is not a common benchmark suite that they all can handle. Nevertheless, there exist a few limited studies

\(^5\)http://www.esri.com/software/arcgis
\(^6\)http://clarklabs.org/
\(^7\)http://pcraster.geo.uu.nl/
<table>
<thead>
<tr>
<th>Modeling method</th>
<th>15 m (9602)</th>
<th>5 m (2,8802)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dinamica, 2 iterations</td>
<td>3 sec</td>
<td>31 sec</td>
</tr>
<tr>
<td>Dinamica, 3 iterations</td>
<td>5 sec</td>
<td>46 sec</td>
</tr>
<tr>
<td>Dinamica, optimum</td>
<td>38 sec</td>
<td>6 min 6 sec</td>
</tr>
<tr>
<td>Idrisi MacroModeler</td>
<td>10 min</td>
<td>–</td>
</tr>
<tr>
<td>ArcGIS ModelBuilder</td>
<td>22 min</td>
<td>–</td>
</tr>
</tbody>
</table>

Figure 18: A comparison between three different tools: Dinamica v1.8, Idrisi 32 v.2 and ArcGis 9.3.

comparing some of these tools. As an example, Pérez-Vega et al. [Pérez-Vega et al. (2012)] have compared Dinamica and Idrisi, not from a performance perspective, but from the point of view of the accuracy of each modeling algorithm. Dinamica is substantially faster, and more accurate in some situations. Idrisi’s algorithm, based on neural networks, is more accurate in others. We have re-implemented one of the models from the Idrisi 32 v.2 tutorial (land use map (worcwest), source map (newplant) and destination map (powerline)), in both Dinamica EGO v1.8 and ArcGis 9.3. Figure 18 shows the result. The goal of this model is to find an optimum pathway in a map, given the cost of traversing the terrain. We have experimented with different resolutions: 5 meters (2,880 × 2,880 cells) and 15 meters (960 × 960 cells). Dinamica uses an iterative algorithm, which we might stop after a few passes over the map, or we might iterate until reaching an optimal solution. We did not perceive visual difference between the path found within 3 iterations or Dinamica’s optimal solution. For the 15 meters models, Dinamica’s 3-passes solution was 120 times faster than Idrisi’s and 264 times faster than ArcGis’s. For the 5 meter version, Dinamica’s 3-passes model took 46 seconds, while neither Idrisi nor ArcGis were able to run the model. This difference is larger in multicore computers. For example, on a laptop DELL Alienware with 8 cores, Dinamica’s 2-passes solution takes less than 2 seconds to process the 15-meters model, almost doubling performance. In this same setup, the computing time of the other GIS platforms increased due to disk accesses. Mas et al. [Mas et al. (2014)] compared Dinamica EGO in relation to CLUE-S and GEOMOD.
Dataflow Programming Languages. Dinamica EGO implements a dataflow programming environment. Quoting Daniel Hils [Hils (1992)], “The central concept of the data flow model is that a program can be represented by a directed graph where nodes represent functions and where arcs represent the flow of data between functions.” The idea of representing computation as graphs has been studied at least as early as 1966 [Bohm and Jacopini (1966)]. For a comprehensive survey about data-flow programming languages we recommend the work of Johnston et al. [Johnston et al. (2004)]. We are not aware of any data flow language that supports geomodeling in particular. However, this paradigm has been employed in the most varied domains, ranging from music [Levitt (1986)] to image processing [Tanimoto (1990)], and has achieved commercial success [Johnson and Jennings (2001)].

Ensuring referential transparency in the presence of destructive updates. There exists a large body of work related to destructive updates of aggregate data structures in functional programming languages [Coutts et al. (2007); Hartel and Vree (1994); Korfiatis et al. (2011); Leshchinskiy (2009); Odersky (1991); Vries et al. (2008); Wadler (1990)]. They are all based on the fact that an update is always safe if the updated data structure is never accessed after being overwritten. In general, the optimizer receives a sequence of function applications and tries to recycle storage locations between actual and formal parameters. There are many ways to ensure referential transparency in face of destructive updates. Some of these methods are implemented automatically, by the compiler [Korfiatis et al. (2011); Leshchinskiy (2009)], whereas others require the direct intervention of the programmer [Wadler (1990)], or the use of specific libraries [Coutts et al. (2007)]. E.g., linear types admit destructive updates, but require every variable to be used exactly once [Vries et al. (2008)]. The main difference between the algorithm that we present in Section 4.3, and these previous works is the fact that we solve a simpler problem: our entire control flow graph is known before a program executes. This is often not the case in functional languages, due to the possibility of high order functions. Consequently, our optimizer is allowed to change the order in which each functional component is evaluated. There is another difference: some algorithms, as Hartel et al.’s [Hartel and Vree (1994)], might allow the same data structure to be updated by different writers, as long as they write at different places. We, like the majority of the other works, do not attempt to carry out this type of optimization.
6. Conclusion

This paper has presented DinamicaVM, the virtual machine that supports the execution of applications built on top of the Dinamica EGO geoscientific framework. DinamicaVM differs from other virtual machines, such as Oracle’s JVM or Microsoft’s .NET, in a number of ways. In particular, DinamicaVM is not a general purpose virtual machine: it is tailored to run applications that process very large images representing cartographic maps. Furthermore, as we saw in the paper, DinamicaVM has a high-level instruction set, which includes components borrowed from functional programming, such as \texttt{map} (e.g., \texttt{Apply}) and \texttt{Reduce}.

This work lets us draw a number of conclusions which are also valid to other data-flow - functional - programming environments. First, it is possible to design and implement a very expressive instruction set, and still ensure that these instructions can be interpreted efficiently. Key to this efficiency are the optimizations that we apply after typechecking the application, but before loading it up. Second, it is possible to make the most of the specific nature of applications that will run on the virtual machine with very positive results. In our case, the virtual machine is customized to handle well large images and tables, which are typical in geosciences. This last observation lets us believe that the techniques currently available in DinamicaVM can be of use in other systems that process large images.

DinamicaVM has still room for further improvements. One possible direction that we hope to explore in the future is the possibility to couple it with runtime value specialization, a technique used by Costa \textit{et al.} \cite{costa2014value, santos2013value} to speedup the execution of JavaScript programs. Value specialization will let us produce binary versions of Dinamica’s core components, e.g., \texttt{Apply}, \texttt{Window}, \texttt{While} and \texttt{Reduce} that are tailored to particular runtime data. We believe that this kind of just-in-time speculation can boost Dinamica’s performance, without compromising its programability.

\textit{Software}. Dinamica EGO is open source and is available on-line at \url{http://csr.ufmg.br/dinamica/}

References


