Copyright 1998 IEEE. Published in the Proceedings of VL'98, Sept. 1-4, 1998 at Halifax, Nova Scotia, Canada. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions, IEEE Service Center/445 Hoes Lane/PO Box 1331, Piscataway, NJ 08855-1331 USA.

Implementing Level 4 Liveness in Declarative Visual Programming Languages

Margaret M. Burnett*, John W. Atwood, Jr., and Zachary T. Welch
Department of Computer Science, Oregon State University, Corvallis, OR 97331
{burnett, atwoodj, welchza}@cs.orst.edu

Abstract

An increasingly common characteristic in visual programming languages (VPLs) is level 4 liveness—the constant monitoring of the system state with continuous redisplay as events arrive and computations progress. However, level 4 liveness can be expensive.

In this paper, we present an implementation method that supports level 4 liveness in declarative VPLs, ensuring without "unreasonable" cost that all values on the screen are correctly updated as computations progress. The method is especially well-suited for the growing class of declarative VPLs that display continuously time-varying calculations and graphics, such as GUI specification VPLs, event-based or reactive VPLs, scientific visualization VPLs, or graphical simulation VPLs.

1. Introduction

Many declarative visual programming languages (VPLs) today employ immediate visual feedback to support programming. To categorize the immediacy of feedback provided, Tanimoto coined the term "liveness," which categorizes the immediacy of *semantic* feedback that is *automatically* provided during programming [17].

Tanimoto described four levels of liveness. At level 1 no semantic feedback about a program is provided to the user, and at level 2 the user can obtain semantic feedback about a portion of a program, but it is not provided automatically. Interpreters support level 2 liveness. At level 3, incremental semantic feedback is automatically provided whenever the user performs an incremental program edit, and all affected onscreen values are automatically redisplayed. This ensures the consistency of display state and system state if the only trigger for system state changes is user editing. The automatic recalculation feature of spreadsheets supports level 3 liveness. At level 4, the system responds to program edits as in level 3, and to other events as well, such as system clock ticks and mouse clicks over time, ensuring that all data on display accurately reflects the current state of the system as computations continue to evolve.

Level 4 liveness supports continuous visualization and steering of evolving computations, which is particularly important for declarative VPLs supporting interactive GUI specification, scientific visualization, and graphical simulations. In the VPL arena, these problem domains have been addressed almost entirely by declarative approaches, especially constraint-based, dataflow, and rule-based. To support these problem domains in our declarative spreadsheet VPL, Forms/3 [3, 4], we have been working on ways to efficiently support liveness at level 4.

In this paper, we discuss a number of implementation methods for supporting liveness at level 4 in declarative VPLs, and compare their efficiency. We also present a new method we have devised, termed "lazy marking with culprit tracking." We show that this method is more able than the others to maintain consistent response times, and that, for some triggers, it is in fact optimal in the number of variables[1] processed.

1.1 Implementation requirements

To support level 4 liveness, it is necessary to keep the display state up-to-date even when the program remains unedited, and it is mainly upon this situation that this paper focuses. (We defer until Section 7 the discussion of responding to program edits.)

Even when there are no edits, the display needs to change in response to four trigger types: (t1) interactions that change which portion of the program is visible to the user, such as de-iconifying the window with the graphical thermometer in Figures 1 or 2; (t2) computational events, such as a graphical clock that needs to be redrawn after each "tick"; (t3) user input events, such as clicking on the F-C button in Figures 1 or 2; and (t4) time travel events, in which the user manipulates a time control device to view the program's state at a different moment in time. Not all level 4 systems support t4 triggers, but some do,

* This work was supported in part by Harlequin, Inc., and by NSF under ASC-9523629 and a Young Investigator Award.

[1] The term "variable" is being used for consistency with traditional programming language literature. Generalizing to VPLs that do not have explicit variables, "variable" means any object that might need to be computed to produce and maintain the program's output; for example, a spreadsheet cell or an attribute of a character in a graphical rewrite rule.

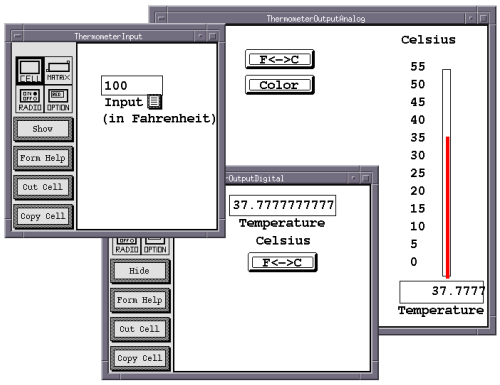

Figure 1: A user's view of a Thermometer program. The user can type in a temperature, and can control scale and color through the F-C and Color buttons.

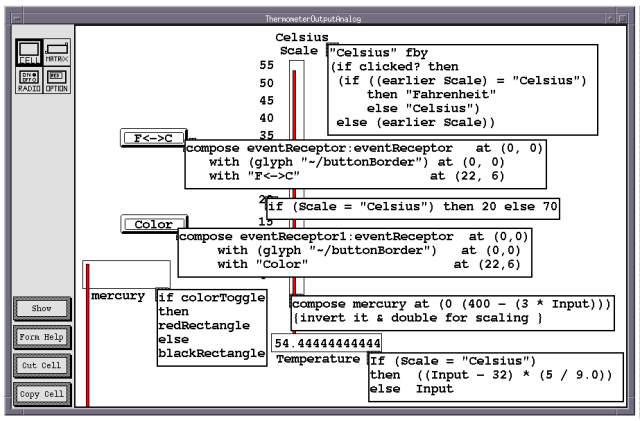

Figure 2: A programmer's view of the graphical thermometer window of the Thermometer program, with some of the formulas shown next to the corresponding cells. (Cells have been rearranged for better visibility of their formulas. The mercury cell on the left is not normally visible to the user).

so we include it for completeness.

Three of these four trigger types are user interactions, and this raises the issue of response time. For quick response time in an interactive VPL with continuous display, costs must be distributed on a "pay only for what you use" basis, where "what you use" is determined by the semantics of the user's interactions. Semantically insignificant tasks such as re-exposing an unchanged value should incur insignificant costs, but it is acceptable for more significant tasks, such as clicking on a button to start a series of calculations, to incur more significant costs. Although this point may seem obvious, prior implementation methods have not been able to meet it.

We state these expectations in the following responsiveness requirement:

Responding to a trigger requiring display update should require processing only of changed variables that affect the onscreen variables.

1.2 A simple model of time

Since level 4 liveness maintains the visual display continuously over time, it is necessary to include time explicitly in our discussion of implementation methods. The model of time we will use includes a global time axis and a way of aligning variables along the axis that allows explicit description of dependency relationships among simple and time-based variables found in level-4 VPLs, such as progressive values of a system clock, successive animation frames, and streams of input events.

Let "logical time" be defined by (T, t_1, t_{max}, *next, onOrAfter*), where time axis T is a sequence of "time" elements from any domain in which binary operators $<$, $=$, and $>$ are well-defined (e.g., N^+); where $\forall t_i,t_j \in T$, $t_i<t_j$ iff $i<j$; t_1 is the minimum and t_{max} the maximum element of T; *next* is a function that, given a time t_i, returns t_{i+1}; and *onOrAfter* is a function whose specifics may vary from one VPL to another but that, given a subsequence S of T, must return a time no smaller than any $t_i \in S$.

Variables are aligned with logical time as follows. Each variable x's definition defines instead of a single value for x, a sequence of (v, defTime) "vt-tuples," where v is a value and defTime$\in$ T is the time at which v becomes defined, subject to these constraints: (1) no two vt-tuples for the same variable have the same defTimes; (2) $\forall$(v', defTime') such that v' is needed to compute v, defTime$\geq$ defTime'[1]; and (3) defTime=t_1 if x's definition is a constant. If one of x's vt-tuples (v, t_i) is "marked" out-of-date (via one of the methods described in this paper) as of time t_j, or if (v', t_j) is also defined for x where $j>i$, then v and (v, t_i) are said to be "out-of-date" as of time t_j.

Given this model, we refine the term "time travel" to mean using a navigational device in the programming environment to select for display vt-tuple values at times before or after those currently displayed.

1.3 Input events

A way to view input events in a declarative VPL is through an abstraction device that maps event queues to variables and events to values[2], adding to our model the constraint that for any vt-tuples ($event_i$, $defTime_i$) and ($event_j$, $defTime_j$), $defTime_i<defTime_j$ iff $event_i$ happened at a "real" (clock-on-the-wall) time before $event_j$. Thus, for each event-sensitive abstraction, such as a menu or event-sensitive area of the screen, we will consider the queue of input events to be just another variable. This allows the

[1] Although this constraint precludes cyclic relationships among vt-tuples, "spiral" relationships are allowed (such as a vt-tuple of x depending on another, earlier vt-tuple of x).

[2] This stream-like view is better-suited than a monadic [16] view would be to the model used in this paper.

model to cover input events in the same way as other computations, minimizing the need for special rules and definitions. (In fact, Forms/3's treatment of events explicitly implements this model [3].)

2. Prior solutions

In this section we describe three traditional implementation methods for keeping evolving data up-to-date. Later in the paper, we will present a comparative analysis of their costs.

We will use producer/consumer terminology for clarity about dependency relationships. For example, in Figure 2, Scale is a "producer" of data for Temperature, and Temperature is a "consumer" of Scale's data.

2.1 EE: Eager evaluation

In eager evaluation (EE), a new vt-tuple is computed as soon as there is enough producer data to do so. Thus, t2 and t3 triggers (computational and user input events) require new vt-tuples for *all* variables affected, regardless of whether they affect the screen. (If no prior results are saved, this is also the case for the other two trigger types: manipulations to the display or to the time travel control.)

EE has been widely used since the early days of VPLs. For example, it has been used by several commercial spreadsheet systems, by the research spreadsheet VPL NoPump [18], and by early dataflow VPLs for GUI specification such as Fabrik [12]. EE fails the responsiveness requirement because vt-tuples for variables that do not affect the screen may be computed. For example, if the user presses the F-C button in the Thermometer example, EE recomputes the graphical thermometer even if it is off-screen.

2.2 LazyNM: Lazy evaluation, no memoization

With lazy evaluation, a vt-tuple is computed at most once during computation of a single expression, and is not computed at all unless it is needed for output. To achieve the "at most once" characteristic, each subexpression's result is saved temporarily until the computation is completed. However, traditional implementations of lazy evaluation do not retain results *after* the computation is complete and the output painted on the screen. We refer to such a traditional implementation as LazyNM.

We know of no VPLs that use LazyNM: it is not well-suited to them. In LazyNM, every user manipulation—even one with no semantic content, such as moving a window aside—would generate recomputations to redraw even values that had not changed, since none of these values would have been saved. For example, re-exposing the graphical thermometer in the thermometer program would require recomputation of vt-tuples for every variable in the program. Thus this approach does not meet the responsiveness requirement.

2.3 LazyEM: Lazy evaluation, eager marking

To avoid needless recomputation, a lazy evaluation engine can "memoize" [13] (cache) computed vt-tuples for longer periods of time than in LazyNM. If this is done, the system must track when producers' computations cause consumers' values to become out-of-date, so that it can update the screen. One approach to this is LazyEM, which computes lazily, but eagerly "marks" consumers' affected vt-tuples out-of-date as of t_i whenever a producer's (v, t_i) vt-tuple is computed.

When a trigger (of any of the four trigger types) causes such a new computation, LazyEM visits every consumer of the new computation at least once to mark a vt-tuple out-of-date. There may be many interrelationships among the consumers, and analysis suggests that it is infeasible in a live environment to apply standard optimizations in a way that eliminates all duplicated visits [5]. Hence, we count LazyEM's worst-case marking cost as quadratic. (In contrast, EE and LazyNM have no marking cost.)

At this point, marking is complete but the screen is still out-of-date. Demands must follow to replace out-of-date values on the screen. Clearly LazyEM violates the responsiveness requirement, since it processes (during marking) variables that do not affect the screen.

LazyEM is widely used. Garnet [14] and Amulet [15], two versions of a well-known visual/textual GUI builder based on one-way constraints, are an example of visual systems that use LazyEM. Clock [9], a visual functional language for groupware development, is another example. Some textual declarative languages also use methods similar to LazyEM to maintain continuous display of GUI objects (c.f. Fudgets [6], Concurrent Clean's I/O system [1]).

Hudson has formally presented and analyzed the LazyEM method, and has shown that LazyEM is optimal in the variables it selects for computation [10-11]. However, our responsiveness requirement considers not only the cost of computation, but also adds in the cost of marking, and LazyEM is not optimal in that total.

3. LazyLM: Lazy evaluation, lazy marking

LazyLM is the basis of the approach we have devised for satisfying the responsiveness requirement. Instead of marking a value's consumers' vt-tuples out-of-date, LazyLM predicts a value's own expiration time as soon as the value is computed. This eliminates the need for a separate marking step.

LazyLM has not been described in the literature before,

but it has been used by Forms/3 since 1991. LazyLM builds upon earlier research related to temporal aspects of lazy declarative languages. Typical of this work were the optimizations done in the context of the lazy dataflow language Lucid [2]. Most of these approaches include memoization, some using LazyEM and some using prediction of expiration times. More recent uses of predicted expiration times include Plane Lucid [7] and Fran [8]. None of these systems have addressed the issue of responsiveness in the presence of liveness.

3.1 How LazyLM works

Predicting expiration times can be viewed as the problem of correctly computing defTimes for out-of-date marks in the sequence of variable x's vt-tuples, and viewing it in this way facilitates comparison with the other approaches. With LazyLM, initially a variable contains a single mark with defTime t_1. Whenever vt-tuple (v, t_i) is computed for x, a vt-tuple (mark, t_j) with $j>i$ is created for x, unless x already has a vt-tuple at time t_j. No other marks are created. Thus, t_j is (v, t_i)'s expiration time. (If a value at t_j is later computed, it replaces the mark.) Expiration time t_j is conservatively predicted by these rules:

- Constants (same as sample values): For a variable whose definition is a constant (such as a bitmap), the first vt-tuple never expires; thus, $t_j = t_{max}$.
- Input and system/sensor events: A vt-tuple in an event variable expires when the next event arrives; if that time is unknown (which is usually the case), the conservative estimate that $t_j = next(t_i)$ is used.
- Results of primitive operations: Each language primitive defines the expiration time of its result in relation to its arguments.
- Results of user-defined operations: The result of each user-defined function invocation expires at the minimum expiration time of all its arguments' expiration times.

Note than an overly conservative expiration time does not harm correctness, but eventually generates demands for vt-tuples that haven't really expired (which then adjust the expiration times). Thus, the above rules are most efficient when the expiration times are exact, because then duplicated computation of a vt-tuple (solely to ascertain that its predicted expiration time was too conservative) is avoided. Expiration times are exact when a vt-tuple's producers' expiration times are known, which can occur with t1 and t4 triggers. For example, if x has been off-screen (and undemanded) and then is brought on-screen, if its producers have by now discovered exact expiration times for their vt-tuples, x's new vt-tuple's expiration time will be exact. The LazyLM algorithms are given in Table 1.

Although it may appear from this discussion that all computed vt-tuples are saved forever, this is not necessarily the case. In a declarative VPL, for any variable except an event queue, the variable's definition provides enough information to recompute vt-tuples whenever desired. Thus, in LazyLM as in the other approaches, how many and which vt-tuples to save can be decided by any arbitrary memoization (caching) approach.

3.2 Correctness and marking cost

Since this method always results in a mark being placed as soon as a vt-tuple could possibly expire, no value that is out-of-date as of time t can be erroneously used to render the screen's display of time t.

LazyLM is done during the normal course of computing a value for variable x, so no traversals of x's consumers are required. Because LazyLM accesses times of only direct producers (arguments explicitly included in x's definition), whose values are already being accessed for computation purposes, marking adds only O(1) to the cost. The incremental propagation of the expiration times avoids the need to explicitly consider expiration times of indirect producers (the producers' producers). LazyLM requires the same space order as LazyEM, namely O(1) per variable.

4. LazyLM$_{CT}$: LazyLM, culprit tracking

An advantage of LazyLM is that it avoids separate dependency graph traversals for marking. However, in rendering the screen, it polls all onscreen variables to ensure that up-to-date values are displayed. In the Thermometer

Render-screen(t)

For each onscreen window:
  For each variable x on the window:
  Display (Demand (x,t)).

Demand(x, t)

If an unexpired value of variable x exists for time t,
  Return the value, its defTime, and its expiration time.
Else if x is an event queue or a primitive,
  Do something language-specific (see Section 3.1).
Else if x's definition is a constant,
  Return the constant, defTime t_1, expiration time t_{max}.
Else
  For each argument needed to compute x:
  Demand the argument at time t.
  value = result of applying the operations to the arguments.
  defTime = onOrAfter (arguments' defTimes).
  expirationTime = min (arguments' expiration times).
  Return value, defTime, and expirationTime.

Table 1: LazyLM. Render-screen and the underlined portions of Demand will be altered in Section 4 to create LazyLM$_{CT}$. Display(Demand(x,t)) is a shortcut for Display (the first result returned from (Demand(x,t)). The renderer is triggered by the four trigger types given in Section 1, and issues demands as needed.

example, as a result of the Color button being pressed, variables dependent only on the F-C button will be polled even though they cannot have changed. Because it processes these unchanging variables (by polling them), LazyLM fails the responsiveness requirement. LazyLM$_{CT}$ is a modified version of LazyLM that solves this problem.

4.1 A categorization of variables

Recall that, when the system receives a user input event, the event is considered to be the newest value of a variable representing an event queue. We call this user event the "culprit event," because it is the reason the system needs new computations to become up-to-date again.

We categorize the variables of a program as follows: (1) Event-sensitive variables are dependent on a user input event. (2) Temporal variables are computed solely due to the passage of time, such as animated objects or variables whose definitions refer to the system clock. (For temporal variables, the culprit event is the passage of a certain amount of time). (3) Static variables are those that are not in either of the first two categories.

We further categorize the event-sensitive category into separate subcategories for each event queue. For example, in the Thermometer program all variables dependent on buttons are in the event-sensitive category, which has two subcategories: (1) variables dependent on the F-C button, and (2) variables dependent on the Color button. A variable can belong to multiple (sub)categories; for example a variable might depend on two different event queues. The culprit event's event queue and all that event queue's consumers are termed "Active" until all processing generated by the culprit event has been completed.

The system can use this categorization of variables to meet the responsiveness requirement. When a culprit event arrives, the system must render the screen to display current values. However, the system need only render (demand and display) variables in the temporal category and a single subcategory—the active event queue's subcategory—and can avoid the processing required even to consider any other variables for rendering. This is how LazyLM$_{CT}$ meets the responsiveness requirement.

4.2 How LazyLM$_{CT}$ works

There are two parts to the method: tracking the culprit event during demands, and the screen rendering portion of the system where the benefits of tracking are realized in a reduced number of variables to consider.

Tracking the culprit event. The system categorizes the variables into the categories described above by tracking which variables are consumers of which event queues, and which variables are consumers of temporal variables as follows. When the user defines a new event-sensitive variable, the system creates an event-sensitive subcategory for its event queue. Each variable is in the same event-sensitive subcategory as its producers. (The built-in temporal variables such as the system clock are predefined to be in the temporal category.) Being lazy, the system discovers during computation which variables belong in which (sub)categories. Category information is thus propagated through the system incrementally.

The algorithm for this tracking part of LazyLM$_{CT}$ augments the demand algorithm of LazyLM. Two new data structures are needed: for each variable the set of the categories (denoted Category in the algorithms) to which it belongs, and for each window a table (denoted CTtable in the algorithms) that stores the categories affecting each variable in an inverted form so that the rendering routine can efficiently identify the active event-sensitive variables. To maintain these data structures, LazyLM$_{CT}$ adds the italicized operations shown in Table 2 to the underlined portion of LazyLM's Demand algorithm from Table 1.

Rendering the screen. A culprit event triggers the renderer, which then uses the category information as shown in Table 3 to select which variables must be demanded to keep the screen up-to-date. The algorithm in Table 3 is what calls the algorithm in Table 2.

4.3 Correctness and marking cost

We have shown that LazyLM's mark placement in log-

For each argument needed to compute x:
  Demand the argument at time t.
  x's Category = x's Category ∩ argument's Category.
  For each i ∈ x's Category:
  Add x to window's CTtable in subcategory i.

Table 2: LazyLM$_{CT}$'s additions to the underlined portion of LazyLM's Demand algorithm.

Render-screen(t)

For each window W on the screen:
  Case t1: bringing an off-screen window W1 on-screen:
  Select all variables on W1.
  Case t2: a temporal computational event:
  Select from W's CTtable all temporal variables.
  Case t3: an input event:
  Select from W's CTtable all temporal variables and all variables in the active event-sensitive subcategory.
  Case t4: a time travel event:
  Select from W's CTtable all temporal and event-sensitive variables.
For each selected variable:
  Display(Demand (variable,t)).

Table 3: LazyLM$_{CT}$'s rendering algorithm for each trigger.

ical time is correct in that it prevents out-of-date values from being rendered on the screen. LazyLM$_{CT}$ does not change the placement of marks; its only modification is that it does not consider for redrawing onscreen variables that could not possibly have changed. Thus, the screen remains up-to-date. Culprit tracking adds O(|EQs$_x$|), the cost to notify variable x of the event queues affecting it, to each non-trivial demand for x, as is clear from Table 2.

5. Global time complexity

The marking costs given thusfar were local costs, i.e., from the perspective of a single variable. Now consider cost for an entire program written in a level 4 VPL, upon receiving a trigger. T1 and t4 triggers are straightforward: assuming memoization, EE "wins" because its computational cost has already been paid prior to this trigger, LazyLM$_{CT}$ behaves similarly to LazyLM, and the three other methods cost the same as for t2 and t3 triggers. Thus, we focus on t2 and t3. For example, suppose all onscreen variables' values are up-to-date, and then the user interacts with the application, such as by pushing the thermometer example's F-C button. How much work is required to get from this state back to a state in which the screen is again up-to-date?

5.1 Time complexities of the five methods

We consider worst-case cost because that is what is needed to ascertain ability to satisfy the responsiveness requirement. We will assume memoization of all values for ease of comparison (except in LazyNM, which does not memoize) and data structures supporting O(1) access to any vt-tuple. Thus the worst-case cost of a method is the sum of the time spent rendering (r), marking (m), and computing (c). We will refer to the asymptotic growth rates of r, m, and c as R, M, and C.

Let $\{x_1, x_2, ..., x_n\}$ be the variables in which vt-tuples are actually computed by a particular method. Then $c = \sum dc_i \ \forall i \in \{1..n\}$, where dc_i is the direct cost of computing a vt-tuple for x_i *given* that the needed producers' vt-tuples (arguments) have previously been computed (+ O(|EQs$_{x_i}$|), for LazyLM$_{CT}$). To facilitate comparisons of global worst-case costs, let $dc_E = \max \{dc_1, ...dc_n\}$ under eager evaluation, $dc_L = \max \{dc_1, ...dc_n\}$ under lazy evaluation, and |EQs$_x$|= max {|EQs$_{x1}$|, ...|EQs$_{xn}$|}. Then C for EE is the number of variables computed * dc_E, and C for the other four methods is the number of variables computed * dc_L (+ O(|EQs$_x$|), for LazyLM$_{CT}$).

Using this, Table 4 summarizes the worst-case complexity of r, m, and c for the five methods. The table yields enough information to prove that LazyLM's worst-case cost is less than or equal to LazyEM's, because:

$(M+C)_{LazyLM} \leq (M+C)_{LazyEM}$ and

$R_{LazyLM} = R_{LazyEM}$.

Thus $(R+M+C)_{LazyLM} \leq (R+M+C)_{LazyEM}$. (1)

A glance at the exponents also shows that:

$(M+C)_{LazyLM} \leq (M+C)_{LazyNM}$, leading to:

$(R+M+C)_{LazyLM} \leq (R+M+C)_{LazyNM}$. (2)

Turning our attention to EE, note that $dc_L \leq dc_E$, since the overhead of saving and retrieving values is the same for both approaches under the memoization assumption, but lazy evaluation may omit some arguments from a computation. This fact and the set sizes show that:

$(M+C)_{LazyLM} \leq (M+C)_{EE}$. (3)

We also point out that:

$R_{LazyLM_{CT}} = R_{EE}$. (4)

(1)-(4) are what we can usefully prove. However, we draw from these results to speculate that if LazyLM$_{CT}$'s tracking overhead of O(|EQs$_x$|) is "small," LazyLM$_{CT}$ is likely to be more efficient in practice than LazyLM (and by (1)-(4), than *all* the other methods), because:

$(M+C)_{LazyLM_{CT}} \stackrel{?}{\approx} (M+C)_{LazyLM}$ and

$R_{LazyLM_{CT}} \leq R_{LazyLM}$.

We return to this point with observed effects of LazyLM$_{CT}$'s overhead in Section 6.

5.2 Variables processed: a lower bound

The number of variables that each method must process is a lower bound for that method. Whether a method can meet the responsiveness requirement can be found by considering these lower bounds, because to meet the

EE	R	|(Onscreen ∩ (Temporals ∪ AV))|
	M	zero
	C	$\|\text{Temporals} \cup \text{AV}\|^2 * dc_E$
LazyNM	R	|Onscreen|
	M	zero
	C	$\|\text{Producers}\|^2 * dc_L$
LazyEM	R	|Onscreen|
	M	$\|\text{Temporals} \cup \text{AV}\|^2$
	C	|(Producers ∩ (Temporals ∪ AV))| * dc_L
LazyLM	R	|Onscreen|
	M	|(Producers ∩ (Temporals ∪ AV))|
	C	|(Producers ∩ (Temporals ∪ AV))| * dc_L
LazyLM$_{CT}$	R	|(Onscreen ∩ (Temporals ∪ AV))|
	M	|(Producers ∩ (Temporals ∪ AV))|
	C	|(Producers ∩ (Temporals ∪ AV))| * (dc_L + |EQs$_x$|)

Table 4: Asymptotic growth rates of the methods for t2 and t3. (See [5] for the calculation of these rates.) Names are as in Section 4; also AV is an abbreviation for Active Event-Sensitive Variables, and here Producers means all the producers of the onscreen variables (inclusive).

requirement, a method must render, mark, and/or compute only variables that both (1) are on the screen or are producers of onscreen variables, and (2) have changed.

Figures 3 and 4 draw from Table 4 to show these lower bounds. The gray areas show the variables processed during rendering, and the bold areas show variables that require computation. More precisely, for methods that use marking, the bold area indicates variables visited for marking; the actual computation is deferred unless/until the values are actually needed, at which point the variables whose out-of-date vt-tuples supply variables being rendered will be computed. For EE and LazyNM, which do not mark, the computations in the bold area happen immediately.

To meet the responsiveness requirement, a method must not render, poll, or compute variables to the left of the dividing line that separates the static and changeable

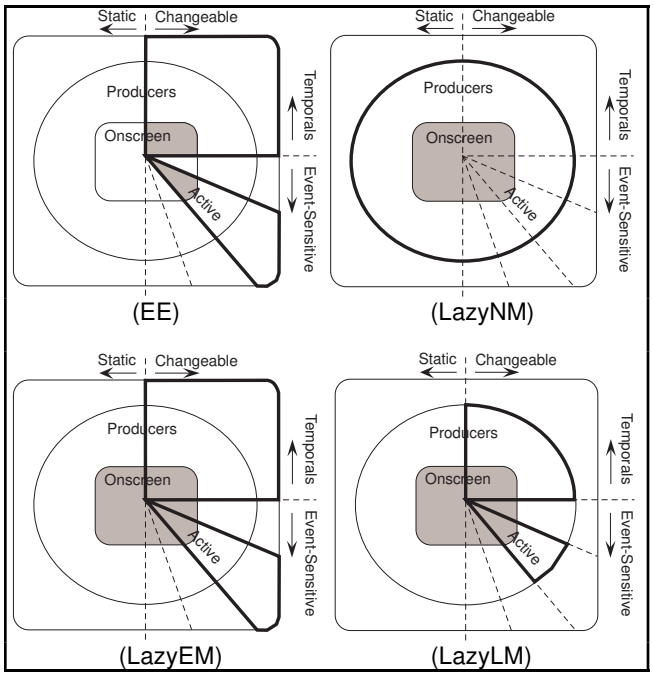

Figure 3: Comparison of variables processed by the four prior methods, categorized as in Table 4. Shaded areas are variables selected for rendering the screen in response to the trigger (r). Bold areas are variables that require marking and/or new computation (m+c).

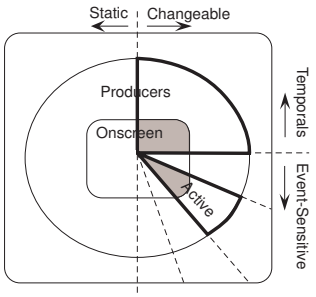

Figure 4: LazyLM$_{CT}$

variables; this is why LazyLM, LazyEM, and LazyNM cannot meet the requirement. Meeting the requirement also implies that a method must not compute or mark variables outside the Producers circle; this is why EE and LazyEM cannot meet the requirement.

However, EE's set of variables processed for rendering is optimally small, since it includes only changed, onscreen variables, and LazyLM's set of variables marked and/or computed is optimally small, since it includes only changed producers that affect the screen. LazyLM$_{CT}$ has both of these optimal characteristics, which makes it not only able to meet the responsiveness requirement, but also optimal in the number of variables processed. In contrast, it is worth noting that for supporting level 4 liveness, LazyEM combines the *dis*advantages of EE and LazyLM.

6. Empirical performance results

In Section 5.1, we raised the issue of whether LazyLM$_{CT}$'s overhead was small enough to allow it to outperform the others in practice. To investigate this, we ran a series of comparisons involving t2 and t3 triggers between LazyLM$_{CT}$ and LazyLM, which Section 5.1 suggests is LazyLM$_{CT}$'s closest competitor. All timings are for Forms/3 on a Sun UltraSparc with one user, under Liquid Common Lisp 5.0.3 with Garnet [14]. Although Forms/3 uses Garnet, all updates of the application-related objects such as the thermometer's buttons and mercury, are handled by LazyLM$_{CT}$ rather than by Garnet, and it is the timings for these application-related objects that we report.

Table 5 shows two system-independent measures—"top-level" demands called directly by the renderer, which makes concrete LazyLM$_{CT}$'s strategy for saving work, and total of all marks placed, which for these two methods is the same as total computations—and a system-dependent measure (runtimes). The table shows the thermometer ex-

Scenario	Top-level demands		Total marks		Total r+m+c in seconds	
	Lazy LM	Lazy LM$_{CT}$	Lazy LM	Lazy LM$_{CT}$	Lazy LM	Lazy LM$_{CT}$
Therm: Digital only	120	60	220	220	2.77	1.14
Therm: Graphical only	460	380	660	660	7.41	4.21
Therm: All	2980	440	1200	840	19.02	5.55
Therm: 1000 EQs	120	60	220	220	2.83	1.20
Animated sort	2992	264	1943	1855	41.51	26.79
Fibonacci	80	40	81	81	0.55	0.53
Pong: Play	44	22	543	543	3.26	3.22
Pong: All	3784	210	1012	576	14.27	3.94

Table 5: (Rows 1-3): The Thermometer program under three screen configurations. Times are CPU time triggered by 10 button clicks, averages of five runs of each scenario. (Row 4): Same as row 1 but with 1000 extra unneeded event queues. (Rows 5-up): Timings for other programs.

ample, an output intensive program (selection sort accompanied by a graphical animation view), a computation-intensive program (naive recursive Fibonacci invoked by a sequence of calls requesting different Fibonacci numbers), and two configurations of a highly interactive program (Pong). The "All" configurations had all the windows and variables displayed, which gave LazyLM$_{CT}$ the most opportunities for improvement. The Thermometer configuration with extra event queues demonstrates that LazyLM$_{CT}$'s overhead does not increase for event queues that are irrelevant to the onscreen variables. Although we have been able to construct naive examples in which LazyLM runs faster than LazyLM$_{CT}$, they are not included here because the only way to construct them is to omit sensible use of event abstraction, such as defining separate event queues for each pixel instead of a single event queue to monitor an object (group of related pixels).

7. An open question: level 3 liveness

Finally, consider the other problem any implementation of level 4 liveness must address: what is an effective method for supporting level 3 liveness, i.e., immediately updating the display whenever the user interactively edits the program?

This remains an open question. Other than Hudson's analysis of LazyEM, we are not aware of research analyzing implementation costs of level 3. Our analysis of level 4 does not extend directly to level 3, because the problem of marking and model of time as formulated to analyze level 4 are not exactly what is needed for analysis of level 3. However, from a cursory look at the issue, it appears questionable whether any of the five methods in this paper can meet the responsiveness requirement at level 3. This is because EE and LazyEM process variables that do not necessarily affect the screen, LazyNM processes variables that have not changed, and LazyLM and LazyLM$_{CT}$ draw on static information to predict in advance which variables will change, and it is not clear whether this static information can be used effectively to predict change activity generated by a program edit.

8. Conclusion

In this paper, we have presented LazyLM$_{CT}$, a new implementation method for level 4 liveness, and have compared it with four prior implementation methods. We have shown that:

- It is not possible for the prior four implementation methods to meet the responsiveness requirement.
- In terms of ability to meet the responsiveness requirement, the widely-used LazyEM exhibits the *worst* characteristics of EE and LazyLM.
- For t1 and t4 triggers, EE is the fastest, because it loads all computing onto t2 and t3 triggers. Second fastest on t1 and t4 are LazyLM and LazyLM$_{CT}$.
- For t2 and t3 triggers, LazyLM$_{CT}$ is the only one able to meet the responsiveness requirement, performs well in practice under a variety of examples, and is in fact optimal in the number of variables processed.

References

1. Achten, P., J. van Gronigen, M. Plasmeijer, High level specification of I/O in functional languages, *Glasgow Wkshp. Functional Prog.*, Springer Verlag, 1992.
2. Ashcroft, E., W. Wadge, *Lucid, the Dataflow Programming Language*, Academic Press, London, 1985.
3. Atwood, J., M. Burnett, R. Walpole, E. Wilcox, S. Yang, Steering programs via time travel, *IEEE Symp. Vis. Languages*, Boulder, CO, 4-11, Sept. 1996.
4. Burnett, M., H. Gottfried, Graphical definitions: Expanding spreadsheet languages through direct manipulation and gestures, *ACM Trans. Computer-Human Interaction*, 1-33, March 1998.
5. Burnett, M., J. Atwood, Z. Welch, Implementing level 4 liveness in declarative visual programming languages, TR 98-60-03, Oregon State University, June 1998.
6. Carlsson, M., T. Hallgren, Fudgets - a graphical user interface in a lazy functional language, in *Functional Prog. and Computer Arch. '93*, Copenhagen, Denmark, 1993.
7. Du, W., W. Wadge, A 3D spreadsheet based on intensional logic, *IEEE Software*, 7(3), 78-89, May 1990.
8. Elliott, C., P. Hudak, Functional reactive animation. *Int. Conf. Functional Prog.*, Amsterdam, 263-273, June 1997.
9. Graham, T., T. Urnes, R. Najabi, Efficient distributed implementation of semi-replicated synchronous groupware, *ACM UIST'96*, Seattle, WA, 1-10, Nov. 1996.
10. Hudson, S. Incremental attribute evaluation: a flexible algorithm for lazy update, *ACM Trans. Prog. Languages and Systems*, 315-341, July 1991.
11. Hudson, S. A system for efficient and flexible one-way constraint evaluation in C++, Georgia Institute of Technology GRU Center TR GIT-GVU 93-15, April 1993.
12. Ingalls, D., S. Wallace, Y. Chow, F. Ludolph, K. Doyle. Fabrik: a visual programming environment, *ACM OOPSLA '88*, San Diego, CA, 176-190, Sept. 1988.
13. Michie, D. 'Memo' functions and machine learning, *Nat.* 218(5136), 19-22, Apr. 6, 1968.
14. Myers, B. et al. Garnet: Comprehensive support for graphical, highly interactive user interfaces, *Computer*, 71-85, Nov. 1990.
15. Myers, B., R. Miller, R. McDaniel, A. Ferrency, Easily adding animations to interfaces using constraints, *ACM UIST'96,* Seattle, WA, 119-128, Nov. 1996.
16. Peyton-Jones, S., P. Wadler, Imperative functional programming, *ACM Princ. Prog. Languages*, Charleston, SC, 71-84, Jan. 1993.
17. Tanimoto, S. VIVA: A visual language for image processing, *J. Vis. Languages Computing*, 127-139, June 1990.
18. Wilde, N., C. Lewis, Spreadsheet-based interactive graphics: from prototype to tool, *ACM CHI '90*, 153-159, Apr. 1990.