Migrating Business Logic to an Incremental Computing DSL: A Case Study

Daco C. Harkes
Delft University of Technology
The Netherlands
d.c.harkes@tudelft.nl

Elmer van Chastelet
Delft University of Technology
The Netherlands
e.vanchastelet@tudelft.nl

Eelco Visser
Delft University of Technology
The Netherlands
e.visser@tudelft.nl

Abstract

To provide empirical evidence to what extent migration of business logic to an incremental computing language (ICL) is useful, we report on a case study on a learning management system. Our contribution is to analyze a real-life project, how migrating business logic to an ICL affects information system validatability, performance, and development effort.

We find that the migrated code has better validatability; it is straightforward to establish that a program 'does the right thing'. Moreover, the performance is better than the previous hand-written incremental computing solution. The effort spent on modeling business logic is reduced, but integrating that logic in the application and tuning performance takes considerable effort. Thus, the ICL separates the concerns of business logic and performance, but does not reduce effort.

CCS Concepts • **Information systems → Web applications**; *Enterprise information systems*; • **Software and its engineering → Domain specific languages**;

Keywords Incremental Computing, Information Systems, Domain-Specific Languages

ACM Reference Format:
Daco C. Harkes, Elmer van Chastelet, and Eelco Visser. 2018. Migrating Business Logic to an Incremental Computing DSL: A Case Study. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (SLE '18), November 5–6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3276604.3276617

1 Introduction

Information systems are systems for the collection, organization, storage, and communication of information. Information systems aim to support operations, management, and decision-making. In order to do this, the data in information systems is filtered and processed to create new data: derived data. Often these information systems contain large amounts of data and receive frequent updates to this data. The derived data should be updated as base data is updated, and that should happen fast. However, realizing a high performance implementation typically requires invasive changes to the basic business logic in the form of cache and cache invalidation code. Unfortunately, this obfuscates the original intent of the business logic in an abundance of caching patterns. These programming patterns are an obstacle to understanding of programs by human readers [14], and thus reduce validatability. In other words, it is less straightforward to establish that a program 'does the right thing'.

Incremental computing languages (ICLs) aim to address this tension between performance and validatability by automatically incrementalizing non-incremental specifications. Since none of the existing ICLs could express business logic of our information systems concisely, we created IceDust, an ICL with support for recursive aggregation and composition of multiple incremental computing strategies.

Contribution To provide empirical evidence to what extent migration of business logic to an ICL is useful, we report on a case study on a learning management system: WebLab. Our contribution is to analyze a real-life project, how migrating business logic to an ICL affects validatability and performance, and how much effort migration takes.

Audience We target language engineering researchers (interested in empirical data justifying their work or looking for new problems to solve) and information system developers (seeking to understand how ICLs can help them in practice).

Structure We organize this paper according to the structure proposed for case studies by Runeson et al. [36] and Yin [47], similar to a recent case study by Voelter et al. [44]. We start with the background on information systems, language engineering, and incremental computing languages in Section 2. Section 3 introduces our research questions and collected data. Section 4 describes the relevant context of the case study as suggested by Dyba et al. [12]. Section 5 provides an overview of the IceDust-based implementation of WebLab. Section 6 answers the research questions. Section 7 discusses validity. Section 8 contrasts our work to related work, and we conclude in Section 9.

SLE '18, November 5–6, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6029-6/18/11…$15.00
https://doi.org/10.1145/3276604.3276617

2 Background

In this section we cover the background information for this case study: information system engineering, language engineering, and incremental computing.

2.1 Web-based Information System Engineering

Nowadays, many applications — including information systems — are web applications, as these are easily accessible. This is illustrated by the fact that the most widely used programming, scripting, and markup languages by Stack Overflow users in 2018 were JavaScript, HTML, and CSS [1].

Many organizations have unique requirements for their information systems. Organizations differ on the exact structure of their data, who gets access to what data, what derived data is computed, and how many users concurrently use the system. Consequently, many organizations use custom-built information systems. Most of these organizations do not require a large-scale infrastructure for their web-based information system, usually a single-shard web-server suffices. While new storage technologies are on the rise, the predominant technology used for storage of data for information systems is relational databases. The code interacting with databases is usually object-oriented, as illustrated by the Correlated Technologies in the same developers survey [1].

Developing information systems poses challenges. One of these challenges is bridging the gap between domain concepts and the encoding of these concepts in a programming language [14, 25]. The validatability of a program is a measure of the size of this gap. Better validatability, a smaller gap between intent and encoding, makes it straightforward to establish that a program 'does the right thing'. Another challenge is performance [16, 48]. Realizing a high performance implementation typically requires invasive changes to a basic expression of intent, reducing validatability. The last challenge is reducing the effort spend on engineering information systems, as budget overruns and delays are a constant problem for information system engineering [46].

2.2 Incremental Computing Languages and IceDust

Incremental computing is a software feature which, when a piece of data changes, attempts to save time by reusing previous results to compute new results [2, 8, 10]. This can be orders of magnitude faster than computing new results from scratch. A programming language is an incremental computing language if all programs written in it use incremental computing. ICLs can be general purpose, for example self-adjusting computation [2] and Adapton [19]; or domain-specific, such as type system languages [38], array computation languages [32, 49], and SQL materialized views [18, 28].

IceDust [20, 22] is a domain-specific ICL for the domain of information systems. It targets small to medium-sized information systems of small organizations that run on a single shard server and are programmed with a relational database and an object-oriented language. In IceDust, a data model and derived values can be specified. These derived values can be calculated by a variety of incremental calculation strategies [20]. Moreover, these calculation strategies can also be composed [22]. IceDust uses static dependency tracking so that for each page request only the relevant data needs to be loaded in memory (dynamic approaches require all data in memory or persistence of the dynamic dependency graph).

2.3 Language Engineering with Spoofax

Language engineering refers to building, extending and composing general-purpose and domain-specific languages [43]. Language workbenches [13, 15] are tools for efficiently implementing languages and their integrated development environments (IDEs). Spoofax is a language workbench for developing textual (domain-specific) programming languages [26]. Spoofax provides meta-languages for high-level declarative language definition. It provides an interactive environment for developing languages using these meta-languages. Moreover, it produces parsers, type checkers, compilers, interpreters, and other tools from these language definitions.

3 Case Study Setup

Our goal is to find out the degree to which ICLs are useful for developing information systems. We adopt the case study method to investigate the use of IceDust in a mission critical project, as we believe that the true risks and benefits of DSLs can be observed only in such projects. Focusing on a single case allows us to provide significant details about that case.

To structure the case study, we introduce three specific research questions in Section 3.1. They are aligned with the general challenges for information systems discussed in Section 2.1. The data collected to evaluate these research questions is introduced in Section 3.2.

3.1 Research Questions

Encoding of domain concepts in programming language constructs makes it hard to validate that a program behaves as intended. To this end the domain-specific language and modeling communities aim to eliminate the gap between domain concepts and language constructs. IceDust is such an attempt, thus the first research question is as follows:

> *RQ-Validatability: Are the language features provided by IceDust beneficial for establishing that an information system 'does the right thing'?*

Performance for information systems is important as the amount of information and the amount of users tends to grow over time, and the filtering and processing to create new data can depend on a lot of data. We capture this in question two:

> *RQ-Performance: Do IceDust-based information systems perform well with real-world data and workloads?*

Independent of how useful an approach is in terms of the first two research questions, it must not require significant additional effort. Hence, our last research question is:

RQ-Effort: How much effort is required for developing information systems with IceDust?

3.2 Data Collected

Below we list the data collected to answer the research questions. As this is a real project, with real users, some data is not available (see discussion in Section 7.5).

RQ-Validatability We look at the source code of the derived value calculations in the vanilla system (without IceDust), and the system with IceDust. We qualitatively asses their impact on the amount of encoding. Moreover, we quantitively asses their impact on the amount of encoding by looking at lines of code, where we assume fewer lines of code means less accidental complexity. (WebDSL and IceDust feature similar syntax and both organize code in entities.) The business logic in IceDust, and in vanilla, we can make available as artifact.

RQ-Performance We measure the original and migrated system performance under a variety of simulated workloads with real-world data sets, and analyze the achieved performance. In addition we measure when and how performance degrades under increasing workloads. The raw performance numbers we can make available as artifact. The private user data we benchmarked on, we cannot make available.

RQ-Effort We measure and discuss the effort required for migrating WebLab to IceDust, distinguishing expressing business logic, embedding it in the rest of the application, performance engineering, and benchmarking. Moreover, we measure and discuss the effort spent on the IceDust compiler triggered by the WebLab case study.

4 Case Study Context

In order to better contextualize our case study, we describe the context as proposed by Dyba et al [12].

4.1 WebLab

WebLab is a learning management system in which students can submit assignments that get graded semi-automatically. Students can submit answers to programming, essay, and multiple choice questions. Individual programming submissions are graded automatically based on (hidden) unit tests. Multiple choice questions also are graded automatically, and all types of submissions can be graded on checklists by teaching assistants. Moreover, WebLab provides many features for calculating the final grades of students: weighted averaging, pass-n-out-of-m assignments, deadlines with late penalties, personal deadline overrides, personal grade overrides, bonus assignments, optional assignments, minimum grade to pass, and calculation traces to explain the final grades. Finally, these grades are used in all kinds of statistics about the assignments and courses in WebLab.

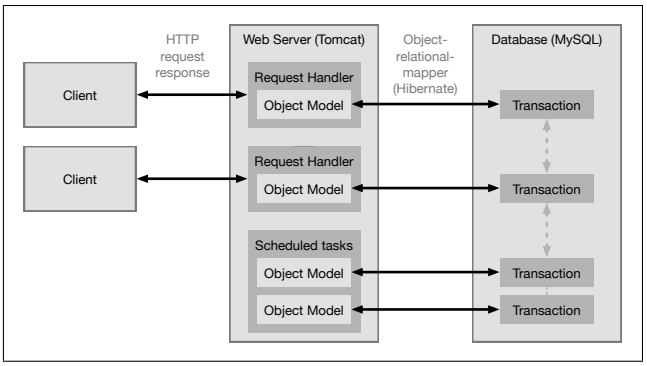

Figure 1. WebLab uses a standard stateless architecture for web servers. HTTP requests are serviced in isolation by a request handler. A request handler loads (saves) the data from (to) the database by means of an object-relational mapper. Request handlers interact concurrently with the database through transactions. The scheduled task executor handles periodic or asynchronous tasks.

The primary success criterion for WebLab is whether the system is reliable enough to use for course labs and exams. This reliability has two aspects. First, its availability should be high. During exams, or labs with deadlines, WebLab should not succumb to the peak loads, as this would invalidate exams or labs. This is reflected in *RQ-Performance*. Second, the computed final grades of students should respect all features which interact with grade calculation, otherwise the final grades are not reliable, and teachers will have to resort to spreadsheets again. WebLab has so many features interacting with grade calculation that this proved to be a non-trivial task the past few years while WebLab was evolving. This is reflected in *RQ-Validatability*.

4.2 Software Architecture

WebLab is a web-based information system that runs on a single shard server. WebLab uses a standard web architecture (Figure 1): a relational database for data persistence and transactions, a HTTP request handler which handles each request in isolation, and an object-relational mapper for loading and storing data every request. Its technology stack is the Tomcat web server, the Hibernate object-relational mapper [34], and MySQL. WebLab is built in the domain-specific language WebDSL [17], which provides a typed integration between a Java-like object-oriented language, a SQL-like language (HQL), a custom UI templating language, AJAX interactions, and access control rules. While WebDSL is a domain-specific language, the code for grade and statistics calculation (which we migrate to IceDust) is written in the Java-like language of WebDSL. So for the purpose of this case study, we can regard this code as non-domain-specific.

4.3 Server Setup

WebLab runs on a large, but relatively old web server. It has 4 12-core 1.7 GHz AMD Opteron processors, 96 GB memory, and 4 500 GB conventional hard disks in RAID 10 configuration. In order to scale under a large parallel workloads, the Tomcat server is configured to use up to 5 GB of memory. The disk bottleneck is mitigated by giving MySQL an InnoDB buffer pool size of 30 GB.

Our development machines, which we use both for development and for benchmarking before continuing to the acceptance stage, are mid-2014 MacBook Pro's. These feature 2.8 GHz Intel Core-i7 processors, 16 GB memory, and a fast PCI-e solid state drive. As memory is limited on these machines, both Tomcat and MySQL only get 2 GB of ram.

4.4 Development Timeline

Migration to IceDust started in September 2015, but stalled in October 2015 after 15 person days (PD) due to lack of expressiveness of IceDust. The migration restarted in September 2017. As of July 2018, WebLab-IceDust is in acceptance stage. Since the restart, 80 PDs have been spent. Full-time work was not feasible due to other pressing projects.

4.5 Tools

IceDust is available as a standalone Eclipse plugin. However, we used the language workbench Spoofax (also integrated in Eclipse) as IDE, as WebLab-IceDust used the nightly version of IceDust during development. WebDSL is also available as an Eclipse plugin.

4.6 Organization and Team

The developers working on WebLab are a team of two scientific programmers within a university. The projects built by these scientific programmers are funded by internal customers (within the university) or external scientific organizations. As such, decisions made about these projects are based on limited resources and potential sources of funding.

The team working on the WebLab migration was this team of scientific programmers plus the IceDust developer. These scientific programmers were not familiar with IceDust before, but were already familiar with Spoofax. As the scientific programmers are housed at the same floor as the IceDust developer, a lot of informal knowledge transfer happened 'at the coffee machine' [11].

5 The WebLab IceDust Implementation

This section provides an overview of WebLab-IceDust and illustrates its use of IceDust's features.

5.1 Overall Structure and Migration

The WebLab code is organized in components as detailed in Figure 2. In WebLab-IceDust we migrated the data model partially from WebDSL to IceDust. Moreover, we migrated

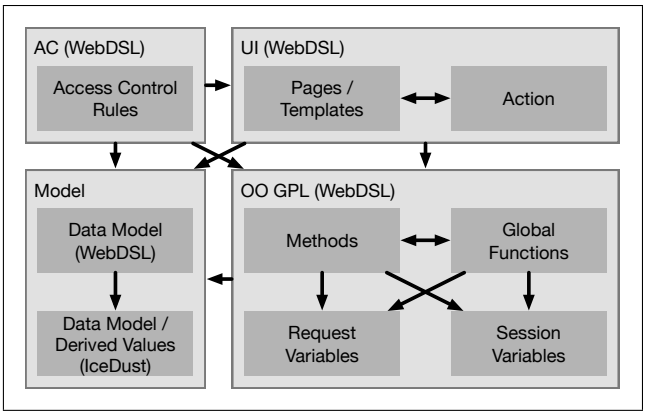

Figure 2. The WebLab code is organized in the following components. The base component is the data model which is partially defined in IceDust, and partially in WebDSL. The data model can be manipulated by the Java-like GPL base language of WebDSL. WebDSLs request variables are global per HTTP request, and its session variables are global per browser session. The user-interface is defined in two parts: actions which manipulate data by means of the GPL, and pages and templates which render information from the data model or GPL and define a navigation structure. Finally, the access control component provides or prevents access to pages and actions based on calls to the model or GPL.

the calculation of derived values from object-oriented GPL code (methods and global functions) to IceDust derived value expressions. Finally, we refactored the rest of the code to use these derived values rather than the GPL methods.

5.2 Size of the System

The WebLab implementation consists of code written in multiple languages. Table 1 shows the size of the code; it is ca. 40,000 lines of code (LOC) when not counting external libraries. WebLab is a medium sized application with 61 different interactive pages using 549 UI-templates and displaying and modifying 98 different object-types through 4013 methods and functions.

Table 2 shows the number of instances of language concepts in IceDust. The business logic specified in IceDust is 542 LOC, where the hand-written incremental calculation is over 800 LOC.[1] The generated WebDSL code from this IceDust code is over 20,000 LOC. This demonstrates that IceDust significantly cuts down on boilerplate for fine-grained incremental calculation strategies. In fact, it would be infeasible to write this by hand, let alone keep it correct while the model is evolving. Note that the hand-written solution provided only coarse-grained incrementality (checking a whole course for changes after a single change), while IceDust recomputes only derived values which are influenced by changes.

[1] Line count obtained by manually listing the methods and fields which contribute to calculation of grades and statistics.

5.3 Use of IceDust's Features

Table 2 shows that WebLab makes use of many IceDust features, indicating their relevance for business logic in information systems. The rest of this subsection introduces these IceDust features and their use in WebLab in more detail.

Derived Values IceDust structures business logic into derived values. Derived values are calculated from base values or other derived values by means of derived value expressions. Figure 3 shows an example of derived value use in WebLab. These derived value expressions ensure that the definition of a derived value always is in one place, like a formula in a spreadsheet cell. This is good for traceability: the ability to verify why implemented business logic made certain decisions. When specifications are scattered, traceability tends to suffer [45]. Moreover, these derived value expressions lend themselves well to incrementalization. WebLab-IceDust uses 133 derived value expressions, 132 for attributes and 1 for a bidirectional relation.

Incremental Computing Derived values can be computed incrementally by IceDust. Figure 4 shows an example of an incrementally computed derived value in WebLab. Computed values are read from cache, and when underlying values change only the cached values depending on these changes are recomputed. With these incremental derived values, information systems can provide fast reads. However, in the WebLab specification we only use two incremental derived values, as we mostly use eventual computing.

Eventual Computing Although incremental computing improves read performance, it makes writes to base values slower, as the writes to base values include recalculating all changed derived values. Eventual computing speeds up writes to base values by sacrificing consistency between base values and derived values. The updates to derived values are postponed, temporarily allowing reads to return outdated derived values (Figure 5). Many derived values in the WebLab specification are eventually calculated. With eventual calculation, WebLab can reliably service many concurrent users who interact with the same data.

On-demand Computing On-demand computing in IceDust means no caching at all. The on-demand calculation strategy is used when performance gains of caching do not outweigh the space cost. When an on-demand expression refers to an eventually calculated value, the on-demand value is also potentially outdated when read. We indicate this by calling this calculation strategy `on-demand eventual`. Figure 6 shows examples of on-demand computed derived values in WebLab.

Computing Strategy Composition In IceDust, the mentioned calculation strategies can be composed within a single specification (Figure 6). To safeguard against erroneous compositions, IceDust employs a static check (Figure 8). In WebLab-IceDust all derived value compositions are checked. These

Table 1. Number of files and lines of code in various languages in the WebLab-IceDust implementation.

Language	Files	LOC	Language	Files	LOC
IceDust	1	542	SQL (migration)	1	153
WebDSL	109	35696	CSS (mostly libs)	18	8513
Java	10	1688	JS (mostly libs)	1321	29055

Table 2. Number of instances of language concepts in the WebLab IceDust model.

Concept	Count	Concept	Count
Attribute	*172*	*Entity*	*22*
abstract	3	base	14
user	27	sub type	8
derived	132	*Relation*	*23*
incremental	*2*	user	22
base	2	derived	1
eventual	*32*	*Function*	*17*
base	19		
overridden	13		
on-dem. eventual	*11*		
base	11		
inline	*87*		

```
progress         : Float = if(pass) 1.0 else 0.0
progressPercent  : Float = round1(progress*100.0)
progressWeighted: Float = progress * weight
```

Figure 3. Business logic is expressed in IceDust through derived value expressions. These expressions in this example calculate the progress of students on assignments.

```
deadline     : Datetime?
deadlineComp : Datetime? =
  deadline <+ parent.deadlineComp (incremental)
```

Figure 4. Derived value expressions can be calculated incrementally in IceDust. In this example snippet assignment deadlines are inherited from ancestors if not provided. When a `deadline` is set, IceDust automatically updates `deadlineComp` for that assignment on all its descendants. (The `<+` operator takes the left value if it is present, otherwise the right value.)

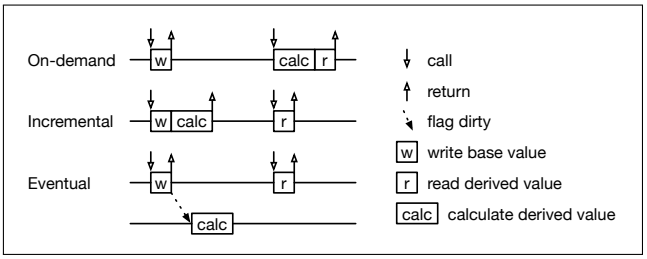

Figure 5. Thread activation diagrams for different calculation strategies in IceDust.

```
numAttempted         : Int = countTrue(subsInEval.attemptedComp)      (eventual)
numCompleted         : Int = countTrue(subsInEval.completedComp)      (eventual)
numPassed            : Int = countTrue(subsInEval.pass)               (eventual)
completedPercentage : Int = numCompleted * 100 /. numAttempted <+ 0 (on-demand eventual)
passPercentage       : Int = numPassed    * 100 /. numAttempted <+ 0 (on-demand eventual)
```

Figure 6. Derived values can be calculated eventually and on-demand in IceDust. On-demand values are not cached, they are recalculated on every read. In this example snippet the raw statistics of assignments are cached and eventually updated, while the percentages for presentation purposes are not cached.

```
gradeWeighted: Float = if(weightCustom > 0.0) totalGrade / weightCustom <+ 0.0 else totalGrade (inline)
gradeRounded : Float = max(gradeWeighted - (sub.penalty <+ 0.0) ++ 1.0).round1()               (inline)

gradeOnTime  : Float = if(sub.onTime <+ false) gradeRounded else 0.0                           (inline)

maxNotPassed : Float = max(0.0 ++ assignment.minimumToPass - 0.5).round1()                     (inline)
passSub      : Boolean = sub.filter(:AssignmentCollectionSubmission).passSub <+ true           (inline)
maxNotPass   : Float = if(passSub) gradeOnTime else min(gradeOnTime ++ maxNotPassed)           (inline)

grade        : Float = min(maxNotPass ++ scheme.maxGrade)                                    (eventual)
```

Figure 7. Derived values in IceDust can be inlined on use site, this controls the granularity of incremental and eventual calculation. Here a submission grade is calculated based on various parameters, but only the final grade is cached.

checks alert the developer when overlooking the impact of changing the calculation strategy of a derived value.

Inline Inline derived values enable breaking up a big expression into a set of smaller expressions, much like a let expression in functional programming languages (Figure 7). Toggling between inline and a calculation strategy controls the granularity of incremental computing. The majority of derived values in the WebLab specification are inline, favoring a somewhat larger granularity and smaller cache size.

Functions Functions enable reuse and abstraction in IceDust. Figure 9 shows examples of functions in the WebLab specification. The WebLab specification has 17 functions, 5 of these are reusable abstractions (such as the first function in Figure 9) while the rest is used to group together the markdown reporting scattered in the system.

Inheritance and Overriding Inheritance and overriding enables the modeling of variation in IceDust. Figure 10 shows an example of this. The WebLab specification uses 13 derived value attribute overrides.

Native Multiplicities All derived value expressions make use of native multiplicities: the cardinality of values is part of the type system, and operators and functions are automatically lifted [20]. The multiplicity type system prevents null-pointer errors, and the automatic lifting prevents boilerplate code dealing with collections and optional values.

5.4 IceDust Feature Requests

Apart from using the existing IceDust features, the WebLab implementation required features not previously supported in IceDust. For WebLab, two IceDust extensions have been developed; below we introduce these extensions and the specific rationales for developing them.

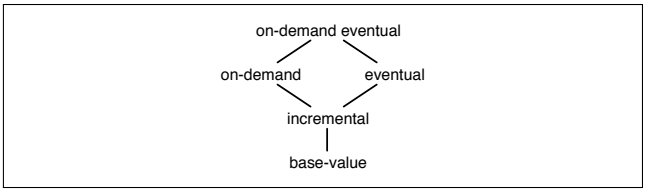

Figure 8. Calculation strategies guarantee their properties iff the derivation expressions refer to derived values with the same strategy or stronger strategies (lower in the lattice).

```
hasF(d : Float*) : Boolean = count(d) > 0

gradeTracePassFail(s:BasicSubmission) : String =
  "Pass or Fail assignment, result: " +
  if(s.pass) "**PASS**" else "**FAIL**"
```

Figure 9. Functions in IceDust enable abstraction. Incremental and eventual computing inlines functions on use-sites.

```
entity CollectionSubmission extends Submission {
  progress : Float =
    if(pass) 1.0
    else if(!isPassN)
      sum(subsForGrade.progressW) / totalWeight
    else
      sum(subsForGradeN.progressW)/ totalWeightN
    <+ 0.0                                        }
```

Figure 10. Inheritance with overriding enables modeling variation in IceDust. Progress in collection submissions takes the progress of the children (`subsForGrade`) into account. It overrides `progress` of `Submission` defined in Figure 3.

Multi-Threaded Eventual Calculation WebLab has high peak workloads concentrated on a small subset of all data: online exams with hundreds of students. Moreover, the derived values depend on many values and influence many other derived values. For example, the `deadline` in Figure 4 flows to all descendant assignments, and influences the grade calculation for all student submissions to these assignments. This can create a performance problem when students are submitting answers an exam while concurrently a teacher tries to change the deadline of that exam. Eventual calculation minimizes the number of transaction conflicts, such that these interactions can all succeed concurrently.

However, if availability is not important (for example recalculating all derived values in a course, after a migration), incremental computing is much faster than eventual computing. To mitigate this issue we made eventual calculation multi-threaded, and the number of threads configurable at runtime. This enables us to allocate more resources during migration such that it takes hours instead of days.

Manual override During development of WebLab-IceDust, a performance caveat was discovered. With some derived bidirectional relations, IceDust fails to capture the dependencies precisely with its path-analysis. This leads to a slowdown for the single derived relation in Table 2. The relational calculus captures the dependencies more precisely. This prompted a feature request for IceDust to ignore incremental updates for a specific derived value. Instead, we use the underlying relational database to manually incrementalize this relation.

6 IceDust Evaluation

Many IceDust features are used to achieve a performant implementation with a validatable specification. However, achieving this also required work on the IceDust compiler. In this section we investigate these observations in more detail by evaluating the research questions introduced earlier.

6.1 RQ-Validatability

Migrating the business logic from WebDSL to IceDust reduced the line count by 32% (from 800+ to 542). As this is a real-world system, not a toy example designed to showcase the DSL, we believe this result to be significant. It indicates that WebLab-IceDust contains less accidental complexity. We assess IceDust's effect on validatability qualitatively below.

Improved Traceability using Derived Value Attributes Derived value attributes have been used extensively, as illustrated by Table 2 and all code figures in Section 5. The derived value attributes make sure that derived values have one unique definition: the derivation expression. This helps traceability, developers never have to doubt whether a derived value in the system comes from a specific piece of code. This in turn simplifies reasoning about the code.

Figuring out where a derived value came from was complicated in WebLab-vanilla. When debugging, more time was spent making sure that the derived value did not originate from another piece of code, rather than trying to understand why a specific piece of code could have produced a specific value. Expressing the business logic in IceDust shifted the debugging conversation from tracing implementation details to domain discussions about the business logic.

Improved Readability with Native Multiplicities The WebLab-vanilla implementation in WebDSL suffered from the billion dollar mistake: null-pointers [23]. The code-base is littered with non-`null` checks. Modern languages often adopt the `Option` Monad, with accompanying boiler-plate code containing `maps` and `flatMaps` or `do` notation. Native multiplicities solve the billion dollar mistake without introducing boiler-plate code. The business logic written in IceDust only mentions multiplicities when needed. This improves the readability of WebLab's business logic significantly.

Simpler Performance Engineering with Calculation Strategies In WebLab-vanilla it was very hard to validate that caching of derived values was correct. In fact, during migration we discovered inconsistencies in the data set from the live system. A grading parameter had been changed in a course, but the cached final student grades were never updated. With IceDust, cache invalidation is correct by construction. Moreover, multiple calculation strategies can only be soundly composed in IceDust. This means developers can stop worrying about the correctness of incremental computing and end users get correct out-of-date indicators for eventual calculation.

Another benefit of IceDust is that it is easy to see which calculation strategies are used. This makes it easier to understand and discuss performance trade-offs.

Separation of Concerns IceDust's design forces a separation of concerns between business logic and performance. Both can be edited separately in IceDust. Since we adopted IceDust, we noticed that business modeling and performance engineering have become two separate activities. The business logic specification is stable during performance engineering.

Remaining Intrinsic Complexity While IceDust reduces the accidental complexity of WebLab's business logic considerably, the business logic can still be complicated to understand. For example, Figure 7 takes some effort to understand, as many variables contribute to the grade of a student (the full specification is even longer). Note that in WebLab-vanilla it was not even possible to see that the business logic is inherently complicated. In future work we might explore how to better organize the remaining intrinsic complexity. We summarize as follows regarding RQ-Validatability:

> *Derived value expressions, as a single source of computation, give developers confidence that they understand what the business logic specification means.*

During performance engineering developers can reason about what the system is going to do based on calculation strategies, without worrying about inconsistencies.

6.2 RQ-Performance

To assess whether WebLab-IceDust performs well with real-world data sets and real-world workloads, we asked the main WebLab developer to describe all scenarios that could be performance bottlenecks. We identified three categories of interactions. Lightweight actions that hundreds of actors do concurrently. For example, students submitting new answers. Mediumweight actions have a larger effect and are performed by a single actor, while concurrently lightweight actions are performed. For example, a teacher postponing the deadline of the exam by 10 minutes, during the exam. And finally, heavyweight actions with a huge effect, performed by a single actor. For example, an administrator recalculating all derived values in a course after a migration.

For all these examples we used real-world data from the live database. (Which we cannot make available for privacy reasons.) Unfortunately, WebLab does not save all HTTP requests, so we could not replay real-world workloads. Fortunately, WebLab saves the history of programming submissions, so we could estimate the workload based on that.

We report on our final configuration of calculation strategies: mainly eventual computing. We experimented with other configurations, but none provided adequate availability under concurrent workloads. We vary the number of eventual computing threads to assess WebLab-IceDust's scalability. We report both the performance on a MacBook (our development machine) and the web server. Our baseline performance is the WebLab-vanilla implementation.

We identified three lightweight actions: random submission reads (browsing), random submission creations (first-time browsing), and random submission edits (working). Many students perform these actions concurrently. For these actions we are interested in the maximum workload WebLab can handle. Moreover, we also want to know under what workload derived value calculation starts to lag behind. If the workload is below that threshold, a teacher can see live statistics of exam progression during the exam. During a representative exam which lasted 3 hours and 30 minutes, we had 31836 code edits by 278 students. This is on average 3 edits per second. The busiest second was 15 edits, and the 99th percentile is 8 edits per second.

We identified two mediumweight actions: change deadline and change checklist weight. The checklist is a grading tool for teaching assistants, and the checklist weight determines the ratio between other means of (automated) grading and the checklist. These two actions are performed by teachers, possibly while students are submitting answers. For these actions we are interested in how long it takes for all derived values depending on the change to be computed. Changing a deadline changes all deadlines lower in the assignment tree, but only influences grades if submissions are late. On the other hand, checklist weights are sure to influence the grades, but only of the assignment and its ancestors.

We identified only one heavyweight action: recalculate a course. This action is performed by administrators after a migration. For this action we are also interested in how long it takes to compute all derived values.

Results Table 3 shows the results of our benchmarks. IceDust enables live statistics during exams (create, edit, and read submissions), which was not possible with vanilla due to availability issues. Moreover, it can provide live statistics for well above 3 edits per second. IceDust speeds up the medium actions (change deadline and checklist weight) significantly, as IceDust's fine-grained incrementality does not have to visit the whole course. Also, with enough worker-threads, IceDust improves the recalculation speed of whole courses.

However, WebLab-IceDust can handle less peak load. With more IceDust worker-threads running, less processing power is available for request-handler threads. Moreover, object creation (in create submission) is more costly with IceDust. All relations in IceDust are bidirectional, opposed to many unidirectional relations in vanilla. Unfortunately, WebDSL and the ORM unnecessarily load objects in memory for keeping relations bidirectionally consistent, even when the other side of the relation is never used. We tried fixing this, but after 4 person days we concluded that it was not worth the effort. Thus, IceDust provides up-to-date statistics at the cost of slower object creation in this case study.

In terms of scalability, more parallelizable workloads benefit more from more worker threads. Recalculate course scales better than the mediumweight actions (change deadline and checklist weight). And the mediumweight actions performed on larger courses benefit more from extra threads than the same actions performed on smaller courses. Surprisingly, 2 threads for recalculating courses on the laptop consistently takes less than half the time of 1 thread. We cannot explain this, but we think it might be due to the CPU speed-step algorithm. Also, scalability on the laptop is hampered by throttling (up to 4ghz for single thread down to 2.6ghz for 4+ threads). The server does not have throttling, so it scales better. However, the server has slower CPUs in general, so it needs more threads to achieve live statistics. As the server has many more CPUs, its throughput (req/sec) is higher; but as the CPUs are slower the latency is also slightly higher (sec/req). In terms of performance, we summarize:

The WebLab implementation in IceDust enables live statistics, which was infeasible manually.

WebLab-IceDust performs similar or better compared to WebLab-vanilla, except for object creation.

6.3 RQ-Effort

Migrating the WebLab business logic to IceDust took 80 PDs in total. Table 4 shows the different development tasks.

Table 3. Benchmark results. The first three benchmarks are maximum system throughput under concurrent student actions. We report the average requests per second over 30 second runs, higher is better. More IceDust threads decrease performance, as less processing power is available for requests. (Vanilla calculation cannot run concurrently with load, hence no measurements for one thread.) The next two benchmarks are the system throughput under which live statistics can be maintained by IceDust. Also these we run for 30 seconds. More IceDust threads increase performance, as derived values are calculated faster. The final three benchmarks are the medium- and heavyweight teacher and administrative actions. For these we measure time to completion in seconds, lower is better. More IceDust threads increase performance, as derived values are calculated faster. All benchmarks have been performed three times. We report the median. All measurements lie within ±10% of the median.

		Machine	MacBook								Server										
		Impl.	Vanilla	IceDust							Vanilla	IceDust									
		Threads	0	1	0	1	2	3	4	5	6	0	1	0	2	4	6	8	10	12	14	16
Action	**Unit**	**Course**																				
read	req/sec	Small	90.99	-	84.76	80.67	76.15	72.37	69.77	65.90	62.37	136.04	-	129.65	112.53	109.58	103.11	92.68	84.39	74.66	67.29	61.18
submission		Medium	89.68	-	90.85	84.65	80.86	77.49	73.83	71.00	68.01	152.95	-	145.04	125.37	119.33	112.02	104.82	93.88	88.26	80.14	71.66
		Large	95.96	-	94.56	89.42	84.60	80.70	77.04	74.01	70.64	132.45	-	119.62	141.99	130.21	115.44	105.22	97.49	96.07	95.98	96.19
edit	req/sec	Small	34.17	-	84.41	79.21	75.40	71.77	68.27	64.42	61.07	31.06	-	120.49	115.52	115.42	114.88	114.48	114.18	114.18	113.55	113.25
submission		Medium	63.65	-	75.53	64.24	61.87	59.40	57.03	54.36	52.24	111.91	-	96.13	90.92	87.08	82.02	69.74	64.73	62.30	63.18	58.96
		Large	60.28	-	75.94	71.57	68.37	65.11	62.28	59.46	57.24	104.17	-	107.84	103.64	96.30	89.56	80.08	75.68	71.30	66.63	62.63
create	req/sec	Small	75.72	-	55.72	50.04	44.71	40.25	37.25	34.47	32.28	100.72	-	67.70	64.39	61.95	60.58	57.31	54.49	51.91	48.94	46.85
submission		Medium	101.18	-	56.61	50.17	43.71	40.95	37.37	34.96	32.24	137.07	-	67.91	64.45	65.55	62.12	58.37	58.17	55.65	55.80	49.03
		Large	101.12	-	30.60	25.85	22.99	21.45	19.58	18.15	16.61	139.34	-	35.93	35.73	34.02	33.58	32.52	31.73	31.40	29.47	28.79
edit	req/sec	Small	-	-	-	22.52	34.46	40.07	42.32	44.16	45.16	-	-	-	117.45	114.38	113.85	113.31	112.85	112.41	111.61	113.68
submission		Medium	-	-	-	6.94	13.12	15.78	16.86	19.60	19.87	-	-	-	6.59	11.29	16.49	20.93	23.95	26.75	29.15	30.72
(live stats)		Large	-	-	-	7.94	15.01	17.88	19.62	20.95	22.26	-	-	-	6.12	12.47	16.62	19.69	20.86	21.68	25.3	24.66
create	req/sec	Small	-	-	-	9.74	16.25	18.95	20.34	21.27	21.48	-	-	-	9.02	15.73	20.56	22.86	24.92	26.85	27.83	29.30
submission		Medium	-	-	-	4.73	8.34	10.55	11.38	12.41	13.18	-	-	-	4.49	8.31	11.26	13.40	14.82	15.77	16.45	17.30
(live stats)		Large	-	-	-	3.95	6.86	8.20	8.68	8.99	9.20	-	-	-	3.60	6.26	8.29	9.54	10.54	11.81	12.35	12.68
change	sec/	Small	-	120	-	145	79	63	58	51	45	-	303	-	195	110	78	64	56	51	48	47
deadline	10 reqs	Medium	-	210	-	186	96	75	70	62	55	-	571	-	252	140	106	79	68	62	59	57
		Large	-	777	-	296	156	122	115	103	92	-	2324	-	408	232	172	143	128	120	117	102
change	sec/	Small	-	600	-	24	18	17	16	16	14	-	1220	-	21	15	14	12	12	11	11	12
checklist w.	100 reqs	Large	-	6430	-	106	64	54	50	46	43	-	19130	-	123	71	55	46	41	39	41	45
recalculate	sec/req	Small	-	21	-	125	57	44	34	40	30	-	49	-	147	76	53	42	36	32	30	28
course		Medium	-	530	-	1295	598	472	425	373	327	-	1718	-	1557	795	549	434	373	333	318	302
		Large	-	5508	-	10368	4786	3670	3244	2910	2653	-	18494	-	12624	6833	5078	4365	4049	3821	3797	3748

Table 4. WebLab migration to IceDust effort

Development Task	**Effort**	**% Total**
Modeling	9 PD	11%
Integration / Migration	24 PD	30%
IceDust Compiler	11 PD	14%
Benchmarking / Performance Engineering	36 PD	45%

We spent 11% of the total effort (9 PDs) on reverse engineering WebLab's business logic and modeling it in IceDust.

We spent 30% (24 PDs) on integrating that business logic into the rest of the application. This included writing migration code to port the data from vanilla's calculation to IceDust's calculation, new user-interface (UI) elements to indicate calculation progress, and retro-fitting unit tests. Also new functionality was added during the integration: student grades finalization (after a course is over and grades are final). Modeling finalization in IceDust was a matter of minutes, creating UI elements took more effort.

We spent 14% of the total effort (11 PDs) on the IceDust compiler to add new features. Both the IceDust developer and the WebLab developers made changes to the IceDust compiler. (Remember that the WebLab developers are familiar with Spoofax and WebDSL.) The added language features enabled us to keep the separation of concerns between business logic and performance, effort well spent.

We spent 45% of the total effort (36 PDs) on benchmarks and performance engineering. As WebLab is used for exams at our university, it is of paramount importance that we can trust its performance. Designing benchmarks, setting up a benchmark infrastructure, and performing the benchmarks took the most time. While it is technically not part of the migration, it was required to give the responsible developers the confidence in WebLab-IceDust. A benefit of the calculation strategies is that is easy to switch between them. Proper benchmarking requires time, but getting a correctly functioning variant implementation to benchmark was a matter of minutes. Concerning RQ-Effort we conclude:

The effort for additional business logic is significantly lower in the ICL, but the total effort is not reduced.

While IceDust does not lead to an overall effort reduction or increase, it does increase separation of concerns.

7 Discussion

The preceding sections show how IceDust affects validatability, performance, and effort of a real-life information system. We put our results in a broader perspective in this section.

7.1 Internal Validity

Internal validity concerns whether our results can be trusted.

Bias One factor that affects this question is the bias because of the involvement of the authors in this case study itself. The authors are the developers of IceDust and WebLab. To counter this bias, we focused on aspects that can be objectively measured (size, concept counts, performance, effort).

Team Expertise To clarify the potential impact of the team on the case study outcomes, we describe the team's background and expertise. The scientific developers both have 5+ years experience in developing information systems on the WebDSL technology stack. The IceDust developer had little experience with the WebDSL technology stack, but 5+ years experience with developing web applications with object-oriented languages and relational databases. When the project started, the scientific developers understood the business logic written in IceDust, but had little understanding of IceDust's calculation strategies. During the migration they gained understanding of these strategies by inquiring the IceDust developer and through experimentation.

Benchmark Internal Validity The benchmark results depend on full stack of technologies: MySQL, Java, Hibernate, WebDSL, and IceDust. Moreover, the results also depend on the hardware used and the settings for MySQL and the JVM. We verified that we actually measured the impact IceDust by benchmarking vanilla, and that we did not measure noise by benchmarking multiple times with a low standard deviation.

In this paper, our benchmarks focus on external validity. As suggested by Vitek et al., we have benchmarks focusing on external and on internal validity [42]. Benchmarks focusing on internal validity are described in previous work [20].

7.2 Conclusion Validity

Our findings favor the using an ICL for separation of the business logic and performance concerns. Conclusion validity raises the question whether these findings can be explained.

Design of IceDust IceDust has been specifically designed to achieve the benefits reported in this case study. So the design rationale of IceDust forms the theoretical explanation of the case study outcomes. For an extensive description of this design rationale we refer to [20–22].

Cognitive Dimensions of Notations The specification of business logic in IceDust improves over the specification in an object-oriented language according to the cognitive dimensions of notations, a set of established language evaluation criteria [7]. Four dimensions are specifically improved.

IceDust greatly reduces *Error-Proneness* with regard to incremental and eventual computing. Developers can rely on the IceDust runtime to keep derived values consistent with their defining expressions. IceDust also removes *Hidden Dependencies* in derived values, as all derived values have a single definition (unlike fields in object-oriented language which can be assigned to in all methods). IceDust greatly reduces the *Viscosity* of calculation strategies, changing a strategy is changing a keyword. IceDust also reduces the *Verbosity of Language* by adopting native multiplicities.

Experience vs. Notation A rival explanation of the success for validatability we measured might be that it is easier to understand the code as we spent considerable time at the white-board trying to reverse engineer the original code. The WebLab team is skeptical, a lot of human working memory is required to fully grasp all details of WebLab's business logic (even though it is only 500 LOC in IceDust). Every time a developer has worked on another project, and comes back to WebLab, this business logic needs to be rediscovered. This rediscovering is much easier in the IceDust specification.

7.3 Construct Validity

When describing our case study setup (Section 3), we explained how the three aspects studied (validatability, performance, and effort) relate to our overall goal of assessing the usefulness of ICLs for developing information systems. From a construct validity point of view, there are additional aspects (constructs) that we could have studied. Unfortunately, our migration did not yield any data on these constructs. However, we do think that our study is still useful, as these constructs are largely orthogonal to aspects we did study.

Interactive Development Information system development requires experimenting to design and understand some of the business logic specifications. Validatable specifications and extensive testing can reduce, but not avoid this need.

Unfortunately, WebDSL impairs interactive development due to long compilation times (over 3 minutes for WebLab). While WebDSL features incremental compilation, it only applies to non-invasive WebDSL features (such as UI components). The IceDust to WebDSL compilation takes a couple of seconds, and the extra generated WebDSL code lengthens the WebDSL compilation by a minute. This extra minute can be explained by the huge amount of fine-grained incrementality code generated. We do not believe IceDust itself inherently impairs interactive development, but properly incrementalizing the WebDSL and IceDust compilers is separate project.

Maintainability In *Little Languages: Little Maintenance?* [40] van Deursen and Klint conclude that a DSL designed for a well-chosen domain and implemented with adequate tools may drastically reduce the costs for building new applications as well as for maintaining existing ones. While we have no experience with long-term maintainability, we did make observations which confirm their conclusion.

During the migration we added new functionality: grade finalization. Modeling finalization of grades, and finalization statistics of courses in IceDust was easy. The intended behavior could be expressed concisely in IceDust.

Another part of the effort in software maintenance is reunderstanding the existing code. As grade finalization interacts with grade calculation in general, it needed to be

'hooked in'. Due to the derived value expressions it was easy to see what part of the specification needed to be modified. IceDust's emphasis on validatability suggests that recomprehension of the system is simplified.

Business Logic Evolution The business logic of information system evolves over time. When decision policies change, different decisions can be made by the business logic based on the same data. The question is how to deal with this evolution. WebLab implements an ad hoc check and does not override previous decisions or derived values. The migration to IceDust did not address this question. However, a validatable specification, which only talks about business logic, might be a stepping stone for addressing this question.

7.4 External Validity

Here we discuss whether our results can be generalized.

Beyond WebLab IceDust is best suited for information systems with complex business logic and a considerable amount of concurrent interaction. As for these situations a validatable specification together with good performance is important. So far, WebLab is the only information system which we modeled with IceDust, integrated into the rest of the application, and benchmarked with user data. We did model other systems with IceDust, but did not integrate or benchmark them. The findings in this paper with regard to validatability apply to these other information systems as well.

Beyond the Team To be successful with IceDust, a team should have experience with building small to medium scale information systems with object-oriented languages and relational databases. IceDust provides separation of concerns between business logic specification and performance engineering, but the latter still requires expertise. If the IceDust calculation strategies provide enough performance, this experience should suffice. However, to modify or add strategies, the team in addition requires language engineering expertise. The IceDust compiler is not overly complicated, both the WebLab team and a master student were able to extend it independently. Note that they did use Spoofax before.

Beyond WebDSL IceDust probably generalizes beyond WebDSL, any object-oriented language with an object-relational mapper should do. IceDust provides an interface to the rest of the application with the getters and setters of fields and the constructors of objects. At this moment we have not implemented any other backends that persist their data. We have not explored targeting non object-oriented languages.

Beyond Spoofax The IceDust implementation in Spoofax is a close translation from its grammar, static semantics, and dynamic semantics [20–22]. IceDust does not feature exotic constructs in its semantics. Thus, with considerable effort, IceDust should be implementable in any language workbench or general purpose programming language.

7.5 Repeatability

This case study reports on the development of a real-world information system. WebLab was not specifically set up as a case study. This has advantages and drawbacks. The advantages include a realistic system, realistic performance constraints, realistic data sets, and an experienced team of developers. The drawback is the unavailability of the source code and user data. (The business logic in IceDust and vanilla, as well as the raw performance numbers, we can make available as artifact.) In McGraths's terms [31], this is a field study, it emphasizes realism over repeatability.

7.6 Research Implications

This paper provides an in-depth case study of the use of IceDust to implement the business logic of a learning management information system, focusing on validatability, performance, and effort. To corroborate and challenge our findings, additional studies are needed, both for IceDust-based systems as well as for other incremental computing approaches.

Furthermore, as this study detailed the need for new language features during application development, we propose future research on co-development of DSLs and applications.

8 Related Work

We are not aware of any other case studies of ICL use for information systems. Instead, we compare our work to incremental computing and DSL case studies. Also, we contrast IceDust to other ICLs which we might have used instead.

8.1 Case Studies in Incremental Computing

Chan et al. investigate the trade-off between query performance, incremental maintenance cost, and storage space for materializing views [9]. They find that the optimal solution is to materialize some views, not all. We did not report on how we select calculation strategies and what we materialize (`inline` is not materialized). However, we have observed a similar trade-off between query time and incremental maintenance cost while experimenting with various configurations.

Another case study in databases [24] investigates various performance optimizations for maximal event throughput. Changes grouped together in a larger commit increases performance. Our experience is similar. One big commit (with `incremental`) can recalculate a full course much faster than many small commits (with `eventual`). However, large commits introduce concurrency conflicts, hurting availability. They report the highest performance with the business logic on the clients instead of in the database (by means of triggers). In IceDust, we also run the business logic in the GPL.

Behrend and Schueller did a case study adopting a new materialized view update technique [6]. They observe that their technique improves performance only in specific conditions. Similarly, IceDust's incrementalization technique

improves over vanilla in specific conditions: small updates are processed much faster, while object creation is not.

These studies only report performance, not validatability or effort. So we cannot compare our validatability and effort findings with other incremental computing case studies.

8.2 Case Studies with DSLs

Adopting a state machine DSL in a case study [5] led to findings similar to ours. They estimate a 10-fold effort reduction in modeling new scenarios in the DSL. In contrast to us, they do not report any required effort for improving their DSL compiler. They report decreased complexity by a code size reduction of 2-3x. Similarly, we also have a 1.5x reduction.

Adopting the Risla DSL for financial applications [39] led to similar findings. The effort for new products was reduced 5-fold. They mention extending the DSL is not easy, but do not quantify this in effort. Like us, they say *"it has become much easier to validate the correctness of the software"*. Unfortunately, they do not provide evidence for this claim.

Adoption of the Pheasant DSL [4] was studied in a lab experiment setting. They report an effort reduction of roughly 1.5x. However, in this lab setting, all tasks were expressible with the DSL, and DSL compiler effort is excluded. They report fewer errors and higher confidence for inexperienced users, but no difference for experienced users. We find better validatability for experienced users. This might be explained by the much higher complexity of our multi-month migration, opposed to one hour lab experiment setting.

Ericsson adopted model-driven software engineering [37]. They conclude that coding is always necessary, including coding the code-generators. Our case study agrees, we added features to the IceDust compiler during migration as well. They report adoption needs to be broken up in phases. We did not have to. This could be explained by the fact that WebLab is a smaller system. One of their goals was to increase productivity, but they did not measure it.

Adoption of the mbeddr extensible language [44] also led to similar findings. mbeddr reduced complexity and improved readability while code size stayed the same. Similarly, IceDust increases validatability, but our code-size did reduce. 5% of their effort was spent on new language extensions, while we spent 14% on new IceDust features. This might be explained by being able to express everything in C when a DSL feature is lacking in mbeddr, while IceDust needs to cover everything. Like us, they report an effort reduction for adding additional functionality, but not for the total effort.

8.3 ICLs for Information Systems

Various ICLs target information systems or present one as running example. Here, we cover these ICLs. ICLs targeted at different domains, but which are similar to IceDust's mechanics are covered in previous work [20, 22].

Object-Set Queries (OSQ) [35] brings relational incremental updates to an object-oriented setting. This might be a viable approach for incrementalizing the bidirectional relation which we had to hand-optimize in this case study. However, OSQ only supports set comprehensions, not complex expressions for calculating primitive values. Moreover, OSQ works in-memory, it is not clear whether it would work with an object-relational mapper and concurrent interaction.

IncOQ [30] improves over OSQ by tracking dependencies statically and incorporating demand. However, the same limitations apply: only set comprehensions and only in-memory.

Complex Object Queries [33] is a predecessor of these. However, this early approach incurs a considerable performance penalty by translating everything into sets and tuples.

MOVIE [3] also incrementalizes OQL queries, and it does persist its data. However, it does not support recursion, which this case study requires.

OR-SQL has also been incrementalized [29]. However, it only supports additions and removals (as most relational approaches) which significantly impair its efficiency when calculating complex primitive value expressions.

idIVM [27] incorporates id-based-diffs in a relational database. idIVM persists its data, but it is used by submitting queries to a database rather than by an object-relational mapper. Moreover, idIVM does not support recursion.

LogiQL [16] also supports incremental relational updates [41]. In contrast to many relational approaches its implementation supports recursive aggregation, be it behind a compiler flag. However, also this database requires interaction through queries, rather than an object-relational mapper.

9 Conclusion

In this paper we present a case study that evaluates the use of the ICL IceDust for specification of the business logic of an information system. We conclude that the migrated code has better validatability, similar or better performance, and that the effort involved in modeling decreases, but total effort does not. The ICL creates a separation of concerns between business logic specification and performance engineering. Performance engineering still takes considerable effort, including pull requests to the ICL compiler.

In future work we would like to investigate the phenomenon of co-development of DSLs with their applications. What factors influence whether a DSL is co-developed with its application? And is this co-development similar to co-development of frameworks or libraries with applications?

Another direction for future work is composing incrementalization techniques. IceDust's path-based incrementalization works well for complex expressions, while the relational calculus works well for bidirectional relations. Can these be combined in a unified incremental computing approach?

Acknowledgements The work presented in this paper was partially funded by the NWO VICI *Language Designer's Workbench* project (639.023.206).

References

[1] 2018. Stack Overflow Developer Survey 2018. https://insights.stackoverflow.com/survey/2018. Accessed: 2018-03-21.

[2] Umut A Acar. 2005. *Self-adjusting computation.* Ph.D. Dissertation. Princeton University.

[3] M. Akhtar Ali, Alvaro A. A. Fernandes, and Norman W. Paton. 2003. MOVIE: An incremental maintenance system for materialized object views. *Data & Knowledge Engineering* 47, 2 (2003), 131–166. https://doi.org/10.1016/S0169-023X(03)00048-X

[4] Ankica Barisic, Vasco Amaral, Miguel Goulão, and Bruno Barroca. 2011. Quality in use of domain-specific languages: a case study. In *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools, PLATEAU 2011, Portland, OR, USA, October 24, 2011*, Craig Anslow, Shane Markstrum, and Emerson R. Murphy-Hill (Eds.). ACM, 65–72. https://doi.org/10.1145/2089155.2089170

[5] Don S. Batory, Clay Johnson, Bob MacDonald, and Dale von Heeder. 2002. Achieving extensibility through product-lines and domain-specific languages: a case study. *ACM Transactions on Software Engineering Methodology* 11, 2 (2002), 191–214. https://doi.org/10.1145/505145.505147

[6] Andreas Behrend and Gereon Schüller. 2014. A case study in optimizing continuous queries using the magic update technique. In *Conference on Scientific and Statistical Database Management, SSDBM '14, Aalborg, Denmark, June 30 - July 02, 2014*, Christian S. Jensen, Hua Lu, Torben Bach Pedersen, Christian Thomsen, and Kristian Torp (Eds.). ACM, 31. https://doi.org/10.1145/2618243.2618285

[7] Alan F. Blackwell, Carol Britton, Anna Louise Cox, Thomas R. G. Green, Corin A. Gurr, Gada F. Kadoda, Maria Kutar, Martin Loomes, Chrystopher L. Nehaniv, Marian Petre, Chris Roast, Chris Roe, Allan Wong, and R. Michael Young. 2001. Cognitive Dimensions of Notations: Design Tools for Cognitive Technology. In *Cognitive Technology: Instruments of Mind, 4th International Conference, CT 2001, Warwick, UK, August 6-9, 2001, Proceedings (Lecture Notes in Computer Science)*, Meurig Beynon, Chrystopher L. Nehaniv, and Kerstin Dautenhahn (Eds.), Vol. 2117. Springer, 325–341. https://doi.org/link/service/series/0558/bibs/2117/21170325.htm

[8] Magnus Carlsson. 2002. Monads for incremental computing. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional Programming (ICFP 2002)*. 26–35. https://doi.org/10.1145/581478.581482

[9] Goretti K. Y. Chan, Qing Li, and Ling Feng. 1999. Design and Selection of Materialized Views in a Data Warehousing Environment: A Case Study. In *DOLAP 99, ACM Second International Workshop on Data Warehousing and OLAP, November 6, 1999, Kansas City, Missouri, USA, Proceedings*. ACM, 42–47. https://doi.org/db/conf/dolap/ChanLF99.html

[10] Camil Demetrescu, Irene Finocchi, and Andrea Ribichini. 2011. Reactive imperative programming with dataflow constraints. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, Cristina Videira Lopes and Kathleen Fisher (Eds.). ACM, 407–426. https://doi.org/10.1145/2048066.2048100

[11] Scott E Donaldson and Stanley G Siegel. 2001. *Successful software development.* Prentice Hall Professional.

[12] Tore Dybå, Dag I. K. Sjøberg, and Daniela S. Cruzes. 2012. What works for whom, where, when, and why?: on the role of context in empirical software engineering. In *2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '12, Lund, Sweden - September 19 - 20, 2012*, Per Runeson, Martin Höst, Emilia Mendes, Anneliese Amschler Andrews, and Rachel Harrison (Eds.). ACM, 19–28. https://doi.org/10.1145/2372251.2372256

[13] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2013. The State of the Art in Language Workbenches - Conclusions from the Language Workbench Challenge. In *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings (Lecture Notes in Computer Science)*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.), Vol. 8225. Springer, 197–217. https://doi.org/10.1007/978-3-319-02654-1_11

[14] Matthias Felleisen. 1990. On the Expressive Power of Programming Languages. In *ESOP 90, 3rd European Symposium on Programming, Copenhagen, Denmark, May 15-18, 1990, Proceedings (Lecture Notes in Computer Science)*, Neil D. Jones (Ed.), Vol. 432. Springer, 134–151.

[15] Martin Fowler. 2005. Language Workbenches: The Killer-App for Domain Specific Languages? https://doi.org/articles/languageWorkbench.html

[16] Todd J. Green. 2015. LogiQL: A Declarative Language for Enterprise Applications. In *Proceedings of the 34th ACM Symposium on Principles of Database Systems, PODS 2015, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Tova Milo and Diego Calvanese (Eds.). ACM, 59–64. https://doi.org/10.1145/2745754.2745780

[17] Danny M. Groenewegen, Zef Hemel, Lennart C. L. Kats, and Eelco Visser. 2008. WebDSL: a domain-specific language for dynamic web applications. In *Companion to the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-13, 2007, Nashville, TN, USA*, Gail E. Harris (Ed.). ACM, 779–780. https://doi.org/10.1145/1449814.1449858

[18] Ashish Gupta and Inderpal Singh Mumick. 1995. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Eng. Bull.* 18, 2 (1995), 3–18. https://doi.org/db/journals/debu/GuptaM95.html

[19] Matthew A. Hammer, Yit Phang Khoo, Michael Hicks, and Jeffrey S. Foster. 2014. Adapton: composable, demand-driven incremental computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 18. https://doi.org/10.1145/2594291.2594324

[20] Daco Harkes, Danny M. Groenewegen, and Eelco Visser. 2016. IceDust: Incremental and Eventual Computation of Derived Values in Persistent Object Graphs. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), Vol. 56. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2016.11

[21] Daco Harkes and Eelco Visser. 2014. Unifying and Generalizing Relations in Role-Based Data Modeling and Navigation. In *Software Language Engineering - 7th International Conference, SLE 2014, Västeras, Sweden, September 15-16, 2014. Proceedings (Lecture Notes in Computer Science)*, Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju (Eds.), Vol. 8706. Springer, 241–260. https://doi.org/10.1007/978-3-319-11245-9_14

[22] Daco Harkes and Eelco Visser. 2017. IceDust 2: Derived Bidirectional Relations and Calculation Strategy Composition. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs)*, Peter Müller 0001 (Ed.), Vol. 74. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2017.14

[23] Tony Hoare. 2009. Null references: The billion dollar mistake. *Presentation at QCon London* 298 (2009).

[24] Andrzej Hoppe and Jarek Gryz. 2007. Stream Processing in a Relational Database: a Case Study. In *Eleventh International Database Engineering*

and Applications Symposium (IDEAS 2007), September 6-8, 2007, Banff, Alberta, Canada. IEEE Computer Society, 216–224. https://doi.org/10.1109/IDEAS.2007.41

[25] Daniel Jackson. 2006. *Software Abstractions - Logic, Language, and Analysis*. MIT Press. https://doi.org/catalog/item/default.asp?ttype=2&tid=10928

[26] Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, Reno/Tahoe, Nevada, 444–463. https://doi.org/10.1145/1869459.1869497

[27] Yannis Katsis, Kian Win Ong, Yannis Papakonstantinou, and Kevin Keliang Zhao. 2015. Utilizing IDs to Accelerate Incremental View Maintenance. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1985–2000. https://doi.org/10.1145/2723372.2750546

[28] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. 2014. DBToaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.* 23, 2 (2014), 253–278. https://doi.org/10.1007/s00778-013-0348-4

[29] Jixue Liu, Millist W. Vincent, and Mukesh K. Mohania. 2003. Maintaining Views in Object-Relational Databases. *Knowl. Inf. Syst.* 5, 1 (2003), 50–82. https://doi.org/10.1007/s10115-002-0067-z

[30] Yanhong A. Liu, Jon Brandvein, Scott D. Stoller, and Bo Lin. 2016. Demand-driven incremental object queries. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, James Cheney and Germán Vidal (Eds.). ACM, 228–241. https://doi.org/10.1145/2967973.2968610

[31] Joseph E McGrath. 1995. Methodology matters: Doing research in the behavioral and social sciences. In *Readings in Human–Computer Interaction*. Elsevier, 152–169.

[32] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org. https://doi.org/cidr2013/Papers/CIDR13_Paper111.pdf

[33] Hiroaki Nakamura. 2001. Incremental Computation of Complex Objects Queries. In *OOPSLA*. 156–165.

[34] Elizabeth J. O Neil. 2008. Object/relational mapping 2008: hibernate and the entity data model (edm). In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, Jason Tsong-Li Wang (Ed.). ACM, 1351–1356. https://doi.org/10.1145/1376616.1376773

[35] Tom Rothamel and Yanhong A. Liu. 2008. Generating incremental implementations of object-set queries. In *Generative Programming and Component Engineering, 7th International Conference, GPCE 2008, Nashville, TN, USA, October 19-23, 2008, Proceedings*, Yannis Smaragdakis and Jeremy G. Siek (Eds.). ACM, 55–66. https://doi.org/10.1145/1449913.1449923

[36] Per Runeson, Martin Höst, Austen Rainer, and Björn Regnell. 2012. *Case Study Research in Software Engineering - Guidelines and Examples*. Wiley. https://doi.org/WileyCDA/WileyTitle/productCd-1118104358.html

[37] Miroslaw Staron. 2006. Adopting Model Driven Software Development in Industry - A Case Study at Two Companies. In *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings (Lecture Notes in Computer Science)*, Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio (Eds.), Vol. 4199. Springer, 57–72. https://doi.org/10.1007/11880240_5

[38] Tamás Szabó, Sebastian Erdweg, and Markus Völter. 2016. IncA: a DSL for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 320–331. https://doi.org/10.1145/2970276.2970298

[39] Arie van Deursen. 1997. Domain-Specific Languages versus Object-Oriented Frameworks: A Financial Engineering Case Study. In *Proceedings Smalltalk and Java in Industry and Academia, STJA'97*. Ilmenau Technical University.

[40] Arie van Deursen and Paul Klint. 1998. Little languages: little maintenance? *Journal of Software Maintenance* 10, 2 (1998), 75–92. https://doi.org/10.1002/(SICI)1096-908X(199803/04)10:2<75::AID-SMR168>3.0.CO;2-5

[41] Todd L Veldhuizen. 2013. Incremental maintenance for leapfrog triejoin. *arXiv preprint arXiv:1303.5313* (2013).

[42] Jan Vitek and Tomas Kalibera. 2012. R3: Repeatability, reproducibility and rigor. *ACM SIGPLAN Notices* 47, 4a (2012), 30–36.

[43] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. 2013. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org. https://doi.org/INTERNALSTYLE-FILEERROR

[44] Markus Völter, Arie van Deursen, Bernd Kolb, and Stephan Eberle. 2015. Using C language extensions for developing embedded software: a case study. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 655–674. https://doi.org/10.1145/2814270.2814276

[45] Robert J. Walker and Kevin Viggers. 2004. Implementing protocols via declarative event patterns. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2004, Newport Beach, CA, USA, October 31 - November 6, 2004*, Richard N. Taylor and Matthew B. Dwyer (Eds.). ACM, 159–169. https://doi.org/10.1145/1029894.1029918

[46] Khim Teck Yeo. 2002. Critical failure factors in information system projects. *International journal of project management* 20, 3 (2002), 241–246.

[47] Robert K Yin. 2013. Validity and generalization in future case study evaluations. *Evaluation* 19, 3 (2013), 321–332.

[48] Kai Zeng, Sameer Agarwal, and Ion Stoica. 2016. iOLAP: Managing Uncertainty for Efficient Incremental OLAP. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1347–1361. https://doi.org/10.1145/2882903.2915240

[49] Weijie Zhao, Florin Rusu, Bin Dong, Kesheng Wu, and Peter Nugent. 2017. Incremental View Maintenance over Array Data. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang 0001, and Dan Suciu (Eds.). ACM, 139–154. https://doi.org/10.1145/3035918.3064041