Memory Scalability in Constraint-Based Multimedia Style Sheet Systems

Terry Cumaranatunge and Ethan V. Munson

Department of EECS
University of Wisconsin–Milwaukee
Milwaukee, WI 53201
{cumarana,munson}@cs.uwm.edu

Abstract. Multimedia style sheet systems uniformly use a constraint-based model of layout. Constraints provide a uniform mechanism for all aspects of style management and layout and are better-suited to non-textual media than flow models.
We have developed a prototype style sheet system, Proteus, and have used it with a variety of document types, including program source code. This work has exposed a critical performance problem in constraint-based style sheet runtime systems: memory usage. Existing constraint systems treat cached attribute values and constraints as first-class objects, each with its own storage. Program syntax trees are very large and the constraint data for a medium-sized source file can easily consume tens of megabytes of main memory. This scalability problem would be exposed by any document of any type containing thousands of objects.
We present here a new constraint-based runtime system that is substantially faster and dramatically more space-efficient than its predecessor, which had first-class constraint objects. The improved performance is the result of exploiting important common cases and a sophisticated constraint representation that allows considerable sharing of information between individual constraints.

1 Introduction

Multimedia style sheet systems [7, 14] uniformly use a constraint-based model of layout, because constraints are better suited to non-textual media than the flow models used in text-oriented style sheet systems [3, 6, 12]. Some multimedia style sheet systems [14] only use constraints for layout, but in our work on the Proteus style sheet system [2, 11] and the PSL style sheet language [9], we use constraints as the foundation of the entire style management process. This approach gives the PSL language uniform and relatively simple semantics and the runtime system of Proteus exhibits a similar uniformity in its underlying mechanisms.

We have been using Proteus to manage style information for program source code in the Ensemble software development environment [1], and in the process have exposed a serious limitation to the scalability of existing constraint-

management systems: memory usage. The central problem is that existing constraint systems [4, 13] treat the attributes of document elements and the constraints between them as first-class objects.

Program syntax trees are quite large. In Ensemble, the abstract syntax tree for a 21 line Java fragment contains 91 lexemes (the text fragments that must be laid out and displayed) and over 500 nodes. By extrapolation, a 1000-line program would have nearly 4300 lexemes and about 24,000 nodes.

Ensemble's text formatter requires Proteus to manage the values of 22 style attributes for each node. This means that, for a 1000-line program, Proteus would have to manage about 500,000 independent attribute values. If each value is represented by a first-class object, then for large programs (or any other large document), the memory requirements of Proteus's runtime system must be measured in megabytes. Furthermore, for many style sheets, there are more constraints than attribute values and if constraints are represented by first-class objects, even more megabytes will be consumed. Memory usage at this level places tremendous pressure on system resources, both by consuming available memory and by slowing performance due to virtual memory page faults.

Prior to the research described in this paper, Proteus used a runtime system originally developed for the Shilpe style sheet system [8]. This runtime system does indeed treat attribute values and constraints as first-class objects and it uses about 2.3 kilobytes of memory per document node.

This paper describes a new runtime system, called Protean, that reduces memory consumption by a factor of 8.8 and improves runtime performance by a factor of 2. Protean's improved performance results from the identification of certain common cases that can be handled without using first-class constraint or attribute objects and from a novel representation for constraint information in the less common cases.

Section 2 describes the original runtime system of Proteus. Section 3 presents the new runtime system and our initial performance results. Section 4 gives our conclusions.

2 The Shilpe Runtime System

Prior to this research, Proteus used a runtime system originally developed for Shilpe [8]. The Shilpe runtime system uses the incremental attribute evaluation algorithm developed by Hudson [4] for the Higgens user interface management system [5]. The Higgens UIMS was used in an interactive editor for graph-structured music documents. In contrast, Proteus assumes that its documents are tree-structured, but this difference is not important to the attribute evaluation algorithm, because both systems allow the dependencies between attribute values to form a directed-acyclic graph.

Hudson's algorithm uses a mix of eager and lazy techniques to achieve good performance. For each node, the most recently computed value of each attribute is cached. The algorithm eagerly propagates information about the invalidation of these cached values while updating them lazily. When evaluating all attributes

from scratch, its performance is linear in the size of the document. When updating attributes in response to an editing change, its worst-case performance is linear in the number of attribute values dependent on the change.

In Shilpe, attribute invalidation is performed by maintaining a data structure called a *dependency chain*, which keeps track of the attributes that must be invalidated due to an invalidation or changed caused by an editing operation. For example, the following rule constraints a node's `Width` attribute to be the same as its parent's `Width`.

```
Width = Parent.Width;
```

Given this rule, whenever the parent's `Width` attribute value is changed or invalidated, the `Width` attribute of the child should also be invalidated. A dependency chain is a data structure that keeps track of those attributes that must to be invalidated due to changes in other attribute values or in the structure of the document caused by editing operations.

2.1 Problems with Shilpe

The dependency chains consume a large amount of memory, even for small documents, for two reasons:

1. The system represents all attribute dependencies using the same structure. This ignores two important common cases: inheritance and bounding box attributes.
   - Inheritance of formatting attributes is very common in style sheets for tree-structured documents. Inheritance is one kind of constraint, so each inheritance relationship must be represented by a dependency chain.
   - Proteus's box layout system maintains complete bounding box information for all document nodes. A node's bounding box is computed from (i.e. constrained by) the bounding boxes of each of its children, which must be represented by a (often long) dependency chain.
2. The dependency chain representation in Shilpe is not particularly space-efficient. Even if the aforementioned special cases were handled more efficiently, memory consumption for dependency chains would remain a problem.

The Shilpe runtime system was instrumented and its memory usage analyzed for an application that requires the maintenance of 16 attribute values for each node. The results of the experiment are shown in Figure 1. These results indicate that for each additional node, memory usage increased by almost 2.3 Kb. For large documents (over 10,000 nodes) this can result in very high memory usage.

The space-efficiency problems seen with the Shilpe runtime system are not unique to it or to Hudson's algorithm. They would also be observed with any other constraint system that treats cached attribute values and constraints as first-class objects, such as the SkyBlue constraint solver [13].

Number of Nodes	Shilpe		Protean	
	Size of dependency data structures	Overall memory usage	Size of dependency data structures	Overall memory usage
10	17.75 Kb	33.76 Kb	80 bytes	18.37 Kb
11	19.94 Kb	36.06 Kb	88 bytes	18.63 Kb
12	21.95 Kb	37.90 Kb	96 bytes	18.90 Kb
13	24.51 Kb	40.20 Kb	104 bytes	19.15 Kb
500	1095.75 Kb	1160.76 Kb	3.91 Kb	140.87 Kb
1000	2195.75 Kb	2310.76 Kb	7.81 Kb	265.87 Kb
10000	21995.75 Kb	23010.76 Kb	78.12 Kb	2515.87 Kb

Fig. 1. Memory usage of Shilpe vs. Protean. Shilpe's memory usage grows at a rate 8.8 times higher than Protean's memory usage. (This experiment was performed on a document that used a style sheet containing 16 attributes.)

3 The Protean Runtime System

The Protean runtime system preserves the approach to incremental attribute evaluation that was used in Shilpe, but uses a more sophisticated algorithm to handle dependencies along with very compact runtime data structures to improve the memory usage and runtime performance.

Each node in the tree has two parts: a cache, and dependency information. The framework for this algorithm is a simple, yet very effective representation of the cache and dependency information.

3.1 Cache

A central premise behind Hudson's attribute evaluation algorithm is that caching of attribute values leads to critical runtime performance gains. We performed an experiment and discovered that Protean, on average, takes 27.64 μsec to determine the value of a node's attribute when values were cached, while it takes 279.4 μsec when values were not cached — a performance difference by a factor of 10. The runtime performance when attribute values are not cached leads to poor and unacceptable response times that makes the system almost unusable for large documents, especially in interactive applications.

Each node has an *attribute cache*, which is a set of value/status-flag pairs stored as an array of two-word *values*, and a packed array of 4-bit *status flags*.

A typical medium in the Ensemble system has about 20 attributes. Since 4 bits [1] are required to represent each status flag, the entire cache status flag representation for each node would take 80 bits (or 12 bytes on a modern workstation,

[1] In PSL, an attribute value can be computed in four different methods: a node-specific rule, a default rule, an implicit rule, and a global default value. Each of the rules has a unique odd and even values to represent the valid and invalid states. An additional value to represent complete invalidity makes a total of 9 bits, which are represented as 4 bits.

rounded to the next word). For each node, the storage for cached attribute values would be 160 bytes.

3.2 Invalidating the Cache

When an editing operation is performed, attribute values can change. Such changes trigger a chain of cache status flag invalidations to invalidate both the attribute values directly affected by it and the ones indirectly affected due to dependencies caused by rules specified in the style sheet.

3.3 Dependencies

Each node contains a data structure, the *dependency list*, which stores information about the cached values (in this node or another) that are dependent on each of the node's attributes. Protean implements a dependency list using a very compact bit representation. The runtime system performs the following administrative tasks at startup time: (1) style sheet rules are stored in a shared data structure called the *component array*, (2) a mechanism, called an *inverse path*, is established to reach dependent attributes, (3) the component array is ordered, and (4) a dependency list is created in each node.

The Component Array Each style sheet rule is internally represented as a list of its *atomic components*. For the style sheet rule in Figure 2, the atomic components are: a rule header for attribute `VertPos:Y`; a tree navigation function (`LeftSib`); and an access operation (on attribute `Y`).

```
VertPos:Y = LeftSib.Y;
```

Fig. 2. A rule defined for the "RAYS" node in the style sheet shown in Figure 7. The rule asserts that the node's `VertPos:Y` attribute is constraint by its left sibling's `Y` attribute.

Fig. 3. Inverse path for the rule shown in Figure 2

The list of atomic components for a rule is linked together by storing a forward pointer in each component. Our implementation arranges the components as infix expressions. When the rule needs to be evaluated, the list of components is followed in the forward direction. The details of this implementation, however, are not important to the algorithm presented in this paper.

Inverse Paths When an attribute is invalidated due to an editing operation, attributes that are dependent on it must be invalidated. Protean uses a mechanism called an *inverse path* to reach the affected attributes. The key to determining the inverse path is an observation in the PSL style sheet language: Dependencies between attributes can arise only from attribute-access operations and tree navigation functions. Such components of a rule are called *dependency-causing* components.

An inverse path for a rule is created by storing backward pointers in the dependency-causing components of a rule. For the rule that we have been discussing (in Figure 2), the inverse path is shown in Figure 3.

The process of invalidating affected attributes follows the inverse path (backward pointers) in a rule's component list and inverts the meaning of each tree navigational component, in order to locate the nodes that are dependent by the rule. Given the rule shown in Figure 2, if "RAYS" node's left sibling's `Y` attribute were invalidated, the `LeftSib` tree navigation function would be inverted as `RightSib` to reach the "RAYS" node.

The `Parent` tree navigation function does not have a unique inverse when there are multiple children. However, the algorithm maintains dependency information in the parent and the child to resolve this problem. The details of marking (establishing) dependencies will be described in section 4.4.

Ordering the Component Array The rules specified in the style sheet are decomposed into its atomic components and stored in a *component array*, which is logically divided into three contiguous blocks: the *attribute-access block*, the *navigation block*, and the *miscellaneous block*. The attribute-access block contains all components that access attributes; the navigation block contains all components that are tree navigation functions; and the miscellaneous block contains all components that are neither an attribute-access operation nor a tree navigation function. Such components include constants, rule headers, and arithmetic operators.

Figure 6 shows the component array resulting from parsing the style sheet shown in Figure 7. The component array is a single data structure shared by all nodes.

Creating Dependency Lists The dependencies between attributes are represented using a dependency list in each node. The dependency list is implemented as a packed array of bits, a compact data structure compared to heavier-weight data structures used in other constraint-based style sheet runtime systems.

The ordering of the component array places the dependency-causing components in a contiguous section of the array. This contiguity allows us to create a dependency list for each node, where each bit in the dependency list has a corresponding dependency-causing component in the component array.

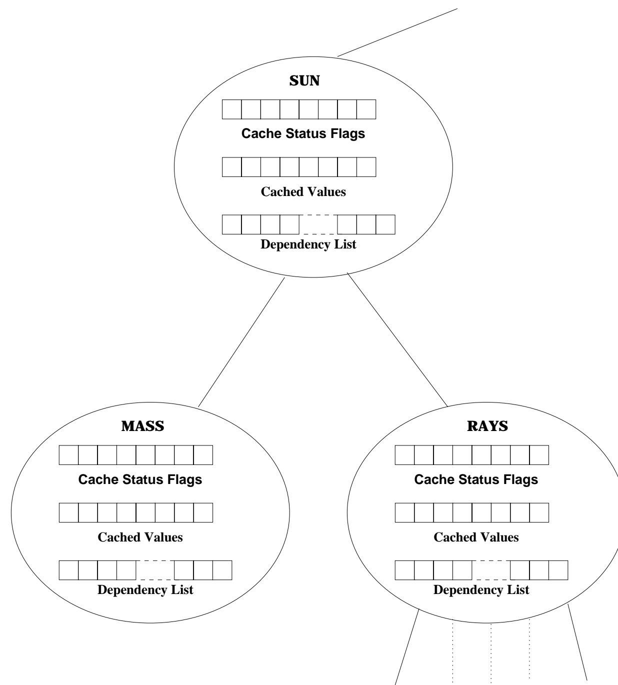

Fig. 4. A sample tree depicting the cache and dependency list for each node.

3.4 Marking Dependencies

The relationship between two attributes is represented by setting the appropriate bits in the dependency list of each of the relevant nodes. Given the rule shown in Figure 2, its dependency-causing components are listed in the component array at indices 17 (`LeftSib`) and 8 (access on attribute `Y`), as shown in Figure 6. If you apply this rule to the tree shown Figure 4, the dependency can be represented by setting bits 8 and 17 in "MASS" node's dependency list. The dependency representation of the "MASS" node is shown in Figure 5.

Since the rule shown in Figure 2 constraints the "RAYS" node's `VertPos:Y` attribute to "MASS" node's `Y` attribute, if the "MASS" node's `Y` attribute were invalidated, the "RAYS" node's `VertPos:Y` attribute must be invalidated. This is done by searching the attribute-access block in the component array to determine if there are components that access the `Y` attribute. In our example, the component at index 6 in the component array accesses the attribute `Y`. So the "MASS" node's bit 8 of the dependency list is checked to see if it is set. Since the bit is set, starting from the component at index 8, the inverse path (shown in Figure 3) is followed to locate the affected attribute. When the affected attribute is located, it is invalidated and the process continues recursively to invalidate attributes that are affected by the recent invalidation.

Recall from earlier that when following an inverse path, the `Parent` navigation function can give rise to ambiguity when there are multiple children. This problem is resolved by identifying the `Parent` tree navigation function's component index in the component array, and setting the bit both in the child's and parent's dependency lists. When a `Parent` function is encountered while following an inverse path, the bit for that component is matched with the children to identify the correct child (or children if the rule affects multiple children).

3.5 Memory Usage of Protean

The memory usage of the Protean runtime system and its comparison to the Shilpe runtime system is shown in Figure 1. The results showed that, on average, Shilpe requires about 2.3 Kbytes per document node while Protean requires only .26 Kbytes. This improvement by a factor of 8.8 approaches a decimal order-of-magnitude.

3.6 Runtime Performance of Protean

The runtime performance of Protean was compared to that of Shilpe. The results showed that, on average, Shilpe takes 58.23 μsec to evaluate an attribute while Protean takes only 27.64 μsec — a binary order-of-magnitude improvement. (This experiment was performed on an HP 715/80 with 64MB RAM and a clock speed of 80MHZ running HP-UX 9.05.)

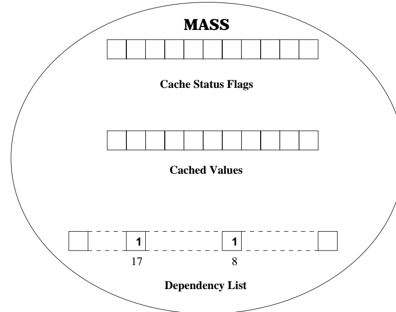

Fig. 5. Bits 8 and 17 are set to represent the dependencies caused by the rule in Figure 2.

Index	Attribute/ Function/ Constant	Node Name	Rule for Attribute
⋮			
6	X	RAYS	HorizPos: X
⋮			
8	Y	RAYS	VertPos: Y
⋮			
14	Parent	MASS	Width
15	Parent	MASS	Height
16	LeftSib	RAYS	HorizPos:X
17	LeftSib	RAYS	HorizPos:Y
⋮			
30	Width	MASS	(rule header)
31	*	MASS	Width
32	0.666667	MASS	Width
33	Height	MASS	(rule header)
34	*	MASS	Height
35	0.666667	MASS	Height
36	HorizPos:X	RAYS	(rule header)
37	VertPos:Y	RAYS	(rule header)

Fig. 6. A portion of the component array resulting from parsing the style sheet shown in Figure 7. Indices 0-9 represent the attribute-access block, indices 10-21 represent the navigation block, and indices 22-37 represent the miscellaneous block.

```
MASS {
     Width = Parent . Width * 0.666667;
     Height = Parent . Height * 0.666667;
     BgColor = "yellow";
     Fill = BG_SOLID;
}
RAYS {
     /* This will automatically space out all
        rays radially around the sun's mass */
     HorizPos:X = LeftSib . X;
     VertPos:Y = LeftSib. Y;
     Linewidth = 5.0;
}
```

Fig. 7. A portion of a style sheet.

4 Discussion and Future Work

The Protean runtime system provides dramatic improvements in both space and runtime efficiency over the Shilpe runtime system's implementation of Hudson's algorithm. Protean uses a novel system of bit arrays to represent dependencies between the attribute values of document elements, instead of the multi-byte first-class objects used by Shilpe.

Even with these improvements in space efficiency, the memory required to maintain style sheet information for a 1000-line Java program would still be about six megabytes, which might be tolerable given the recent rapid decline in memory prices but is still excessive. As a result, we expect to continue research into ways to reduce memory consumption. Some ideas that we will explore are

- Improving the component array data structure so that components that appear in multiple rules can be shared;
- Using a sparse representation for cached attribute values; and
- Representing layout information (especially bounding boxes) in relative coordinates, rather than absolute coordinates.

The Protean runtime system is quite general, but it makes certain assumptions and has certain limitations.

- Hudson's attribute evaluation algorithm was originally designed for use with graph-structured music documents [5]. The Proteus style sheet system assumes that its documents are tree-structured, but the Protean runtime system is not restricted to tree-structured documents.
- In general, Protean requires that dependency-causing components of style rules can be inverted unambiguously. The one violation of this assumption in Proteus (the `Parent` function) is handled as a special case.
- Protean is a "one-way" constraint system. The fact that an attribute value X is dependent on an attribute value Y does not mean that Y is dependent on

X. In contrast, the SkyBlue constraint solver [13] is an example of a "multi-way" constraint system, supporting dependencies in both directions. Multi-way constraint systems are interesting and very useful for certain graphic-object transformational applications in which all attributes are subject to change from external forces (such as users), but the one-way constraint model is simpler and appears to be well-suited to the multimedia style sheet task.
- SkyBlue also provides a general solution to the problem of cyclic constraints. Protean is capable of detecting cycles and will not crash or infinitely loop because of them, but does not guarantee that useful attribute values will be computed when cycles are present in the constraint rules.

Our future research will continue to explore style sheet systems and their applications to multimedia documents and programming environments. We seek to improve the power and ease of use of style sheet languages and to develop direct manipulation tools for style sheet development. We are continuing the development of the model of media used in configuring Proteus [10] and plan to extend it to better address issues of user interaction. Finally, the best document model for integrating program source code and multimedia documentation has not been identified and we hope to address this issue.

References

1. Susan L. Graham. Language and document support in software development environments. In *Proceedings of the Darpa '92 Software Technology Conference*, Los Angeles, April 1992.
2. Susan L. Graham, Michael A. Harrison, and Ethan V. Munson. The Proteus presentation system. In *Proceedings of the ACM SIGSOFT Fifth Symposium on Software Development Environments*, pages 130–138, Tyson's Corner, VA, December 1992. ACM Press.
3. Web Design Group. Cascading style sheets. Home Page at http://www.htmlhelp.com/reference/css.
4. Scott E. Hudson. Incremental attribute evaluation: A flexible algorithm for lazy updates. *ACM Transactions on Programming Languages and Systems*, 13(3):315–341, 1991.
5. Scott E. Hudson and Roger King. Semantic feedback in the Higgens UIMS. *IEEE Transactions on Software Engineering*, 14(8):1188–1206, August 1988.
6. ISO/IEC. *Information technology — Text and office systems — Document Style Semantics and Specification Language (DSSSL)*, August 1994. Draft International Standard ISO/IEC DIS 10179.2.
7. N. Layaida and L. Sabry-Ismail. *Maintaining Temporal Consistency of Multimedia Documents Using Constraint Networks*, pages 124–135. Multimedia Computing and Networking. SPIE, January 1996.
8. Alok Mittal. SHILPĒ: A presentation system for Ensemble. Master's thesis, University of California, Berkeley, California, December 1995.
9. Ethan V. Munson. A new presentation language for structured documents. *Electronic Publishing: Origination, Dissemination, and Design*, 8:125–138, September 1995. Originally presented at EP96, the Sixth International Conference on Electronic Publishing, Document Manipulation, and Typography, Palo Alto, CA, September 1996.

10. Ethan V. Munson. Toward an operational theory of media. In *Proceedings of the Third International Workshop on Principles of Document Processing*. Springer-Verlag, Palo Alto, CA, September 1996. To be published as part of the Lecture Notes in Computer Science series.
11. Ethan Vincent Munson. *Proteus: An Adaptable Presentation System for a Software Development and Multimedia Document Environment*. PhD dissertation, University of California, Berkeley, December 1994. Also available as UC Berkeley Computer Science Technical Report UCB/CSD-94-833.
12. Vincent Quint. The languages of Grif. Available by anonymous ftp from ftp.imag.fr in directory `/pub/OPERA/doc`, December 1993. Translated by Ethan V. Munson.
13. Michael Sannella. The SkyBlue constraint solver. Technical Report 92-07-02, University of Washington Department of Computer Science, 1992. Available at http://www.cs.washington.edu/research/constraints.
14. Louis Weitzman and Kent Wittenburg. Grammar-based articulation for multimedia document design. *Multimedia Systems*, 4(3):99–111, 1996.