Optimizing Content Management System Pipelines

Separation and Merging of Concerns

Markus Noga[1] and Florian Krüper[2]

[1] Universität Karlsruhe
Program Structures Group
Adenauerring 20a
76133 Karlsruhe, Germany
markus@noga.de

[2] Sparkasse.de
Rotherstr. 9
10245 Berlin, Germany
florian@krueper.net

Abstract. Content management systems support the dissemination and maintenance of documents. In software engineering terms, they separate the concerns of content, application logic and visual styling. Current systems largely maintain this separation of concerns after document deployment. Their runtime processing pipeline is a composition of generators, or document transformations. We exploit commutativity to enable new static evaluations of the composite during document deployment. Unlike traditional caching, we arrive at closed-form composites even for styled, database-driven documents. This eliminates the runtime penalties of a separation of concerns while preserving their software engineering benefits.

1 Introduction

A *content management system*, short *CMS*, supports the process of publishing documents in software. Formerly used exclusively by large organizations, e.g., newspapers, they have become the mainstay of web publishing for medium-size businesses. This paper discusses optimizations that enable economic deployment of a CMS by even smaller businesses and individuals.

Content management is a complex affair. Users have distinct preferences when editing documents. In many cases, publishing these documents also requires significant amounts of application logic. In a web context, e.g., navigation must be generated. Finally, entire professions build a living around visualizing documents. Thus, the notion of a *separation of concerns* [1] is as central to content management as it is to software engineering. A CMS primarily separates the concerns of *content*, *logic* and *presentation*.

Separating concerns allows their development to be temporally, spatially and personally separated as well. E.g., visuals take a significant fraction of initial

D. Batory, C. Consel, and W. Taha (Eds.): GPCE 2002, LNCS 2487, pp. 252–267, 2002.
© Springer-Verlag Berlin Heidelberg 2002

development time for customer-specific CMS solutions. In terms of remuneration, web designers are cheaper than application logic programmers. Thus, separation of concerns in design and development yields immediate economic benefits.

Unfortunately, webspace providers do not charge for traffic and space alone. They also apply the *utility pricing model* to the runtime environment, i.e., the more complex the runtime, the higher the charges. Therefore, separation of concerns in a deployed system imposes economic penalties. Removing these software engineering artifacts from the deployed system is a competitive advantage.

We aim to reap this advantage. This paper describes an experimental CMS based on composed transformations of documents. On a restricted domain of transformations, we show the composition to be commutative and exploit this property to optimize the deployed system with static evaluation. This eliminates the penalties for separation of concerns during system operation while preserving their design and development benefits. Additionally, our approach increases online throughput by an order of magnitude. This enables professional content management on inexpensive webspace accounts.

The remainder of the paper is organized as follows: We examine the state of the art in section 2. In section 3, we introduce an abstract processing model and discuss optimizations on it. Our prototype system implementing these optimizations is described in section 4. section 5 summarizes our results and outlines directions for future work.

2 State of the Art

To clarify a CMS' runtime environment, we first look at technical and economical aspects of renting webspace and examine LAMP, the most popular and economic variant of webspace in more detail. Then, we examine some professional CMS of the open-source and commercial variety. We look at traditional build managers used in programming and examine their suitability for CMS. Finally, we summarize our findings.

2.1 Webspace

The options for renting webspace are bewildering. Offerings range from limited amounts of plain HTML pages, typically offered for free, to personal dedicated servers. Managed service providers like Loudcloud offer a wide array of associated services, from simple email support over 24/7 hotlines up to full site management.

For the purposes of this paper, we disregard managed service providers and employ a simple model of increasing technical complexity. We distinguish webspace, virtual servers and dedicated servers. *Webspace* is storage space on a server. A simple application server or parts thereof may be available, but webspace users have no control over configuration. *Virtual servers* are virtual sandboxes within a server. They support more complex environments like Enterprise JavaBeans [2]. Additionally, virtual servers offer some control over server configurations, but fall short of supporting the installation of arbitrary software. This requires a server entirely under user control, that is, a *dedicated server*.

Rental fees increase in line with technical complexity. Depending on the configuration, webspace is available at low or no cost. Virtual servers impose moderate costs, and dedicated servers even higher ones. These costs are duplicated with the user: configuring virtual or dedicated servers requires technical expertise that must be bought. These types of servers may also require additional maintenance personnel.

As more and more smaller companies and individuals come online, the growth of the internet occurs mainly within low-cost webspace. This market segment restricts server providers to free software environments as the most economical choice of implementation. We explore that environment in more detail below.

2.2 LAMP

What technologies are used in cheap webspace accounts? Usually, free open-source ones, the foremost example being *LAMP* [3]. The acronym refers to its constituent parts: the Linux operating system, the Apache web server, the MySQL relational database and one of the scripting languages PHP, Perl or Python, usually PHP.

Linux, Apache and MySQL are fairly well-known. The newest addition to LAMP is *PHP* [4], a pre-processor for XML and HTML that seamlessly integrates scripting into the XML and HTML files. Because of its simplicity, PHP is open to less-sophisticated developers for whom Perl or Python present formidable challenges. In a LAMP environment, PHP is the application server. Together with the other programs, it enables the provision of dynamic content and database-backed services.

Monthly web surveys can convey a measure of LAMP's popularity [5]. Currently, roughly $\frac{2}{3}$ of all servers run Apache. PHP is the most popular Apache module, running in about $\frac{1}{2}$ of all Apache servers. These numbers make LAMP the most widely deployed web environment. Most large ISPs provide LAMP environments in the very lowest price segment, although with restricted configurability. In Germany, the two largest providers Strato and Puretec do so [6, 7].

2.3 Open-Source CMS

Open-source CMS abound. On Sourceforge, there are some 800 CMS projects, most of them in combination with PHP or LAMP. We discuss only the technology leader, Apache Cocoon, which runs in a more sophisticated Java environment.

Cocoon focuses on plain content management [8]. The Cocoon architecture follows the architectural pattern *Pipeline* [9]. Here, we present the standard case of the configurable pipeline. Each step is a document transformation and encapsulates a concern in the domain of content management: document content, addition and evaluation of application logic, and finally visual formatting or presentation (see fig. 1).

Technically, Cocoon relies on XML and server-side Java. Content is encoded in XML. Application logic libraries are XSLT scripts that replace XML logic

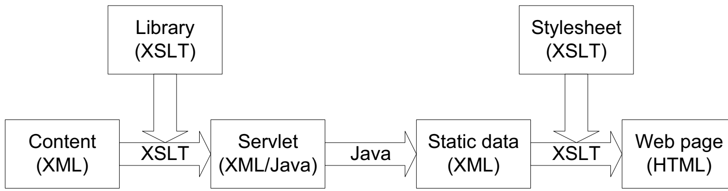

Fig. 1. The Cocoon processing pipeline

tags with server-side Java code embedded in XML. Styling for presentation is also expressed as an XSL transformation. In effect, this is an infrastructure for XML-based domain-specific languages extensible through XSLT rule definitions.

Cocoon requires server-side Java and access to the server configuration. Thus, a virtual server is the minimal runtime environment for Cocoon. In most cases, dedicated servers are required.

2.4 Commercial CMS

Prominent examples of commercial CMS are BroadVision, Gauss, Imperia and Vignette. We discuss Vignette as a classical system and BroadVision as a modern XML-based one.

Vignette Content Suite V6 is a large system split into six modules [10]. Apart from content management and sophisticated caching strategies, the system also caters to personalization, integration and content analysis. Vignette achieves a separation of concerns between content and logic. A separation between logic and presentation is not evident. Content is stored in relational databases and files. Logic takes the form of proprietary modules, TCL scripts or JSPs. Presentation is performed in those scripts. Vignette requires a dedicated server to run.

BroadVision also offers a wide array of tools, which cover personalization, e-commerce and portals in addition to plain CMS [11]. The system achieves a separation of concerns between content, logic and presentation. BroadVision employs XML and relational databases to store content. Logic and scripting interfaces are accessible from Java, JavaScript, COM and others. Like Cocoon, BroadVision uses XSLT for presentation and styling. Like Vignette, BroadVision requires a dedicated server to run.

In general, commercial CMS deal with many issues besides plain content management. They tend towards high complexity and include many proprietary components, although XML technology is being adapted steadily. Unfortunately, heavyweight commercial CMS usually require dedicated server environments.

2.5 Build Managers

Build managers automate the process of building a target from sources. Both targets and sources are software artifacts. make [12], Odin [13] and ClearCase [14] are some well-known tools. They organize builds according to *derivation trees*,

whose nodes are software artifacts and whose edges are individual derivators like compilers or linkers. The three tools make increasingly complex efforts to cache intermediate results: make uses regular files in the user directory, Odin maintains a private cache where source names and tools configurations are encoded into names and ClearCase relies on custom file systems with view capabilities.

However, all three consider the order of individual derivation steps as fixed. Consider fig. 1, where visualization occurs after interpretation. If interpretation hinges on the runtime environment, e.g. because of database accesses, caching must stop before interpretation. Build managers cannot eliminate the final transformations because they do not support commutativity. In the CMS domain, that prohibits deployment of to lightweight platforms.

2.6 Summary

All professional CMS achieve a separation of concerns in design and development to varying degrees. Unfortunately, they maintain that separation in deployment. Their complex runtime environments require virtual or even dedicated servers to run. This makes professional CMS unsuitable for the majority of current and future web page operators. For them, it is necessary to support LAMP environments while preserving software engineering benefits. Thus, the separated concerns must be merged in deployment. Conventional build managers cannot achieve this because they regard the order of individual derivation steps as fixed.

3 Model

In this section, we present an abstract model of a content management system pipeline. We introduce a set of restrictions on pipeline steps that demonstrably ensures commutative composition. Then, we discuss how the resulting preprocessing steps can be expressed in closed form. In the final subsection, we discuss applications to document deployment.

3.1 Processing Model

We model the processing pipeline P of a CMS as a composition of operators $P = O_1 \circ O_2 \circ \ldots \circ O_n$. The individual operators $O_i\colon X \to X$ are maps on the set of XML documents X.

The classical processing pipeline consists of XSL transformations and language interpretation. Let $X_\sigma = XSLT(\cdot, \sigma)$ be the XSLT transformation operator with script σ and $I_e = interpret(\cdot, e)$ be the programming language interpretation with runtime environment e, which may include file systems and databases. We define interpretation to replace special interpretation elements with well-formed, computed XML fragments.

Thus, the classical processing pipeline is can be expressed as $P = X_s \circ I_e \circ X_l$, where s is the styling transformation and l is the logic library transformation.

Many optimization strategies for database query optimizations rely on the algebraic properties of query operators (see [15]). Here, we focus on associativity and commutativity.

Operator composition is associative. Now, let us assume the composition in $X_s \circ I_e$ to be commutative as well. If that were the case, we could transform the pipeline equation to $P = P' = I_e \circ X_s \circ X_l$. As $X_s \circ X_l$ is independent of the runtime environment e, we could precompute $d' = X_s \circ X_l(d)$ when deploying a document d. The online computation would be reduced to $P''(d) = I_e(d')$.

Unfortunately, $X_s \circ I_e$ does not commute in general. While the identity stylesheet does commute with any logic, there are numerous counterexamples. Consider, e.g., application logic generating a list of random numbers and a stylesheet sorting them. A list of numbers not yet generated cannot be sorted. For commutative composition to hold, we must thus restrict ourselves to a subset of all stylesheets and applications. There is a family of such restrictions: the more restrictions are made on stylesheets, the less restrictions on logic are necessary, and vice versa. Here, we are interested in restrictions on s that leave many degrees of freedom to l.

3.2 Restrictions

We proceed by partitioning the set of admissible XML elements by name into three sets, the structural $\mathcal{S}$, logical $\mathcal{L}$ and generated $\mathcal{G}$ ones. Additionally, we know the set of interpreter elements $\mathcal{I}$ and the set of HTML elements $\mathcal{H}$. W.l.o.g., we assume all five sets to be mutually disjunct.

In this framework, X_l transforms XML documents over $\mathcal{S} \cup \mathcal{L}$ to documents over $\mathcal{S} \cup \mathcal{G} \cup \mathcal{I}$. I_e transforms the results to documents over $\mathcal{S} \cup \mathcal{G}$. Finally, X_s transforms these documents to documents over $\mathcal{H}$. With this framework in place, we are ready to impose restrictions.

1. Interpreter scripts may not generate or reference XML elements $\mathcal{S} \cup \mathcal{G}$, save direct copies of well-formed XML elements in the stylesheet with optional generation of attribute values.
2. Attribute value computations are free of side-effects.
3. Styling transformations are restricted to statically:
   - copying by default
   - changing element or attribute names
   - adding arbitrary XML elements immediately before or after opening or closing tags, provided the result is well-formed. It may contain copies of attribute values.

Do these restrictions guarantee commutative composition? To the XSLT processor, interpreter scripts are simply text interspersed with elements. Therefore, the XSLT script s can run on $X_l(d)$. As it copies by default, the interpreter markup $\mathcal{I}$ remains untouched, but structural and generated elements are transformed. The result of $X_s \circ X_l(d)$ is an XML document over $\mathcal{H} \cup \mathcal{I}$.

XML elements are generally not valid in interpreter scripts, so they must be contained in interpreter string literals. We enumerate the string literals l_i containing XML elements sequentially.

When changing names or adding XML content as per 3, this transforms the l_i to l'_i. The l'_i are valid interpreter literals because they can contain well-formed XML elements to begin with (per 1). The XSLT processor may reorder attributes arbitrarily within an element, but per 2, this cannot be observed on the outside. As 3 prohibits reordering or deletion of element contents, the number and order of the l'_i and the interspersed code remains invariant.

Thus, $X_s \circ X_l(d)$ can be interpreted with I_e. Their composition, we recall, was P'. Moreover, when simultaneously replacing the l_i, l'_i with `"li"` in this P and P', their static and dynamic semantics are the same, as there are no control dependencies on the contents of the l_i per 1. By construction of the l'_i, the resulting output documents are equal.

Therefore, the above restrictions are sufficient for commutative composition.

3.3 Closed Form

Commutative composition allows to preprocess the application library and styling transformations at document deployment, reducing on-line processing to a single interpreter run. We now ask if the initial composed transformations may be expressed in closed form. As they are both XSL transformations, we are interested in combining them into a single one.

We exploit the property that XSL transformation scripts are themselves XML documents. If $\mathcal{X}$ is the set of XSLT tags, the script l of the application library transformation X_l is an XML document over $\mathcal{X} \cup \mathcal{S} \cup \mathcal{G} \cup \mathcal{I}$. As X_s transforms elements in $\mathcal{S} \cup \mathcal{G}$ only, $X_s(l)$ is an XML document over $\mathcal{X} \cup \mathcal{I} \cup \mathcal{H}$. Because X_s does not reorder, $X_s(l)$ is a valid XSLT script. It replaces elements in $\mathcal{L}$ with interpreter elements in $\mathcal{I}$ containing script text and styled elements from $\mathcal{H}$.

A transformation with script $X_s(l)$ still leaves elements in $\mathcal{S} \cup \mathcal{G}$ invariant. However, as $\mathcal{S}$, $\mathcal{G}$ and $\mathcal{L}$ are mutually disjoint, the transformation rules in s and $X_s(l)$ may be combined without conflicts. We name the resulting script l'.

Analogous to the preceding section, we impose the following restriction on l:

4. The library transformation may not generate XML elements in $\mathcal{S} \cup \mathcal{G}$, save by copying input or stylesheet elements with optional generation of attribute values. It may not reference result tree fragments.

As XSL transformations are functional except for output, 2 of the preceding section always holds for the transformation script, too. Together with 3, the rationale of the preceding proof applies analogously. Thus, $X_{l'} = X_s \circ X_l$ holds.

l' is a static evaluation of s on all possible outputs of l. In terms of [16], we can also interpret l as a program family and s as a generator specification to obtain the individual family member l'.

3.4 Applications

In content management, deployment makes data on a production system available on a server. Simplifying things to text, there are three kinds of data: content, logic and presentation styling. When deploying them, commutative composition and closed form can be profitably employed to reduce runtime system requirements and enhance processing throughput.

When deploying new logic or presentation styling, we compute the closed form l' of the composite transformation on the client. We then redeploy the affected content.

When deploying content documents d, we apply the closed form of the composite transformation and upload $X_{l'}(d)$ to the server. Even for database-driven documents, this reduces online processing to a single step, which traditional CMS and build managers cannot. If I_e is independent of e, we may even upload $I_e \circ X_{l'}(d)$. This is equivalent to traditional caching of generated documents.

Under this scheme, content management systems may target inexpensive LAMP environments. Simultaneously, the software engineering benefits of a separation of concerns are fully retained.

4 Prototype

Our experimental system is a modification of the original Cocoon design. To support LAMP, we replaced server-side Java with PHP scripting. Content is still encoded in XML. Application logic libraries are XSLT scripts that replace XML logic tags with PHP scripts embedded in XML. Styling is again expressed as an XSL transformation.

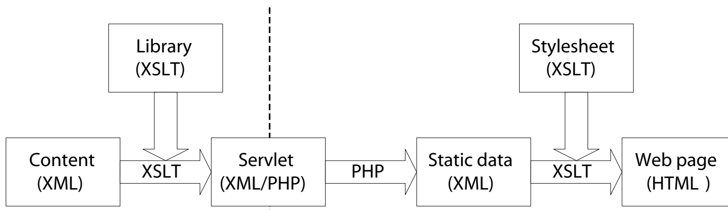

Fig. 2. The experimental processing pipeline. The dashed line represents the caching boundary for dynamic documents.

Fig. 2 shows the resulting design. The dashed line separates cacheable and non-cacheable parts of the system for dynamic documents. We refer to the full pipeline as the *standard* one and the pipeline to the right of the dashed line as the *caching* one.

Because the results in the previous section were obtained using an abstract notion of a CMS pipeline, they apply to our system as well as to Cocoon. Fig. 3 shows the *optimized* version of our system.

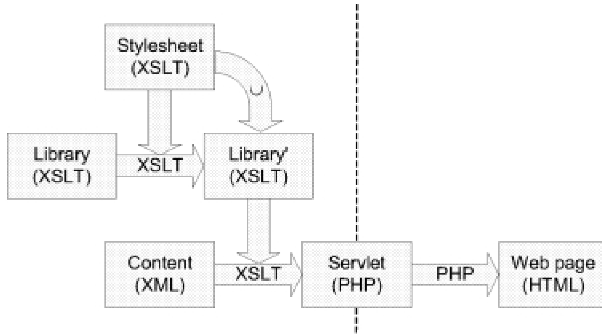

Fig. 3. The optimized processing pipeline. To the left the production system, to the right the deployed system.

4.1 Benchmarks

We prepared five benchmarks to evaluate the performance of the optimized pipeline as compared to the caching and standard ones. They are designed to closely mirror a variety of usage scenarios.

The *hello* benchmark is a simple hello world document. It prints a short message and applies basic styling to it. This benchmark is designed to establish a lower bound on the individual pipelines' execution times.

The *simpledb* benchmark is a database-driven document. It connects to a MySQL database of college classes, retrieves the list of classes in the current semester and applies plain styling to it (see the following subsection). *simpledb* represents first-generation database-driven web pages.

The *db* benchmark is a more sophisticated cousin of *simpledb*. In addition to database operations, it also performs fairly complex visual styling, including cascading stylesheets and table layouts. This benchmark should be fairly representative for contemporary database-driven web pages.

The *bib* benchmark visualizes a list of bibliography entries given as an XML file. There is comparatively little logic, but styling complexity is comparable to or slightly higher than for *db*. This benchmark represents contemporary dynamic web pages that do rely on databases.

Finally, the *fract* benchmark computes part of the Mandelbrot set. It visualizes the results as a colored HTML table. This benchmarks represents applications with complex logic but small to negligible libraries and styling.

4.2 A Benchmark in Depth

To demonstrate the optimized pipeline, we reproduce the *simpledb* in full and discuss its execution. Standard namespaces, default copying rules and error handling code have been removed to maintain readability.

Consider the following document d built from the structural elements `page` and `title` along with the logic element `classes`, which represents a database query to retrieve college classes in a given semester.

```
<page>
  <title>Current Classes</title>
  <classes year="2002" semester="S"/>
</page>
```

The following library transformation l replaces the logic tag in d with server-side PHP code that actually performs the query. Notice how attributes of `classes` are mixed into the code.

```
<xsl:stylesheet version="1.0">
  <xsl:template match="classes">
    <script language="php">
      print "<classlist>";
      $db=mysql_connect("localhost","root","");
      mysql_select_db("phptest",$db);
      $query=mysql_query("select name from classes
        where year=<xsl:value-of select="@year"/>
        and semester='<xsl:value-of select="@semester"/>'",$db);
      while($row=mysql_fetch_array($query))
        printf('<class name="%s"/>',$row['name']);
      print "</classlist>";
    </script>
  </xsl:template>
</xsl:stylesheet>
```

As l amounts to simple text substitution, we do not reproduce the transformation result $X_l(d)$ here. The following XSL transformation s performs simple presentation styling on $I_e \circ X_l(d)$:

```
<xsl:stylesheet version="1.0">
  <xsl:output method="html"/>

  <xsl:template match="page">
    <html><head><title>
      <xsl:value-of select="title"/>
    </title></head><body>
      <xsl:apply-templates/>
    </body></html>
  </xsl:template>

  <xsl:template match="title">
    <h1><xsl:value-of select="."/></h1>
  </xsl:template>

  <xsl:template match="classlist">
    <ul><xsl:apply-templates/></ul>
  </xsl:template>
```

```
  <xsl:template match="class">
    <li><xsl:value-of select="@name"/></li>
  </xsl:template>
</xsl:stylesheet>
```

The restrictions in section 3.2 hold by design for l and s, so commutative composition can be applied. Computing the closed form according to section 3.3 yields this styled version l' of the logic library:

```
<xsl:stylesheet version="1.0">
  <xsl:output method="html"/>

  <xsl:template match="classes">
    <script language="php">
      print "<ul>";
      $db=mysql_connect("localhost","root","");
      mysql_select_db("phptest",$db);
      $query=mysql_query("select name from classes
        where year=<xsl:value-of select="@year"/>
        and semester='<xsl:value-of select="@semester"/>'",$db);
      while($row=mysql_fetch_array($query))
        printf('<li>%s</li>',$row['name']);
      print "</ul>";
    </script>
  </xsl:template>

  <xsl:template match="page">
    <html><head><title>
      <xsl:value-of select="title"/>
    </title></head><body>
      <xsl:apply-templates/>
    </body></html>
  </xsl:template>

  <xsl:template match="title">
    <h1><xsl:value-of select="."/></h1>
  </xsl:template>

  <xsl:template match="classlist">
    <ul><xsl:apply-templates/></ul>
  </xsl:template>

  <xsl:template match="class">
    <li><xsl:value-of select="@name"/></li>
  </xsl:template>
</xsl:stylesheet>
```

Applying l' to d yields the styled document $X_{l'}(d)$:

```
<html>
  <head><title>Current Classes</title></head>
  <body>
    <h1>Current Classes</h1>
    <script language="php">
      print "<ul>";
      $db=mysql_connect("localhost","root","");
      mysql_select_db("phptest",$db);
      $query=mysql_query("select name from classes
        where year=2002 and semester='S'",$db);
      while($row=mysql_fetch_array($query))
        printf('<li>%s</li>',$row['name']);
      print "</ul>";
    </script>
  </body>
</html>
```

The above l' is deployed on the server. At runtime, only PHP interpretation remains to be done. If the database contains a fictitious list inspired by classes offered at our institute, interpretation results in the following static document $I_e \circ X_{l'}(d)$ for client consumption:

```
<html>
  <head><title>Current Classes</title></head>
  <body>
    <h1>Current Classes</h1>
    <ul>
      <li>Compiler Construction</li>
      <li>IA64 DivX Lab</li>
      <li>Software Components</li>
    </ul>
  </body>
</html>
```

4.3 Measurements

The optimized pipeline eliminates all but one stage of the transformation process. The caching pipeline eliminates the initial pipeline stage. Therefore, measurements in a client-side environment would skew the results in favour of the optimized and caching variants due to startup costs associated with the tools implementing individual pipeline stages, i.e., XSLT processors and PHP interpreters. Only a server-side environment, where the associated tools persist in main memory, allows a fair comparison.

This prevents classic build managers like make from participating in measurements. Instead, we implemented a custom framework for performance measurements in PHP. Using `eval`, output capturing and experimental XSLT extensions, startup costs can be avoided. To cope with system clock jitter, the framework loops each benchmark for a minimum of two seconds execution time.

Measurements were taken on an 1.2 GHz Mobile Pentium III-M with 512 MB of memory under Windows XP Professional. We executed our benchmarks with PHP 4.1.2 using version 0.9 of the Sablotron XSLT processor [17] and the NT executable of MySQL 3.23.49. Fig. 4 shows the results. The horizontal axis shows the individual benchmarks. Processing times in percentages of standard pipeline processing time are plotted along the vertical axis. The standard pipeline has been decomposed into its constituent phases, represented by bar graphs. Due to instruction cache effects, totals may differ from 100%. The caching and optimized pipelines are plotted as lines.

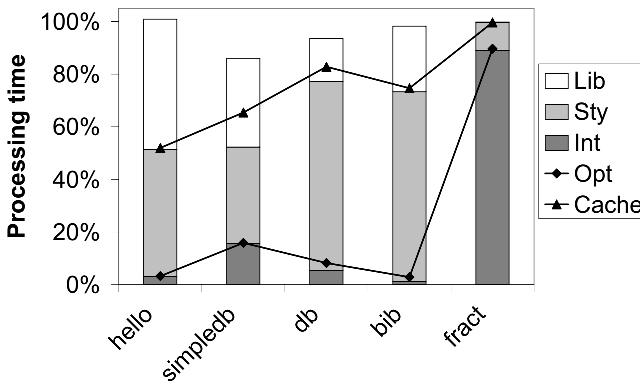

Fig. 4. Performance comparison: The standard, caching and optimized pipelines.

Looking at the curves, we see why caching is a mainstay of traditional CMS. It removes the overhead for library transformation, outperforming standard processing by a factor of $1.0-1.9$x on our benchmarks. Clearly, caching is a profitable strategy.

The optimized pipeline outperforms both the standard and caching variants on our benchmarks. The speedups over caching range from $1.1-26.2$x, those over standard processing from $1.1-35.1$x. Visually, benefits appear to be largest for the benchmarks using complex visual styling. The fact that our optimization renders a separate styling stage obsolete would support this interpretation. However, numerical speedups for *hello* exceed those for the database examples. Those two benchmarks demonstrate that increases in computing time have a proportionally larger impact on the optimized system than on the other variants.

We postulate that execution time for the single remaining optimized pipeline step is dominated by logic execution time. Reporting is reduced to simple copying, so reporting time is almost irrelevant for typical web-based document sizes.

4.4 Tools and Quoting

In preparing our benchmark, we noticed that quoting is an issue with two of the tools used, PHP and Sablotron.

XML mandates that the `<` and `&` characters be escaped with `<` and `&` in regular text, respectively. Unfortunately, the PHP interpreter does not recognize these representations as operators. Thus, PHP requires non-well-formed XML for input. This is probably due to its origins with the HTML community which has long been accustomed to laxer syntactical constraints. In our benchmarks, we resolved this issue through automatic postprocessing. For the long run, we would like the PHP community to address this issue, as it is critical to XML and XHTML embedding of PHP.

In a similar vein, the Sablotron XSLT processor does not preserve the quoting style used in element attributes. Whether single or double quotes are used, transformation results always employ double quotes. This is not a Sablotron issue alone — the XSLT transformation model and the underlying XML information set model simply do not contain quoting information. Most applications are unaffected by this. However, due to the embedding of XML attributes in language string constants, any change in quotation characters may break language syntax. For our benchmarks, we resolved this issue through manual postprocessing.

More viable alternatives come in many shapes. Of cause, we would welcome the addition of quotation information to the XML information set and XSLT processing models. But this may take a long time. Maybe adding a new switch to XSLT processors that allows users to toggle the standard quotation character would already be sufficient. Alternatively, languages like PERL can use arbitrary characters as quotes via the `q` and `qq` prefixes, respectively. However, these quick suggestions address individual processors and languages only.

4.5 Impact of Restrictions

How restrictive are the restrictions from section 3.2 in practice?

Restriction 1 prohibits the interpreter to generate or reference XML elements in $S \cup G$, other than copying them. In practice, this means that element names cannot be constructed in a piecemeal way, e.g. from string variables. However, as DTD grammars are finite, the possible elements names can be enumerated and used in a switch construct. Thus, restriction 1 may lead to an inconvenient notation, but it does not limit the expressive power of interpreter scripts.

Restriction 2 mandates that attribute value computations must be free of side effects. It is necessary because XML transformations may not preserve attribute order. In practice, attribute values can be precomputed and stored in variables, so restriction 2 does not limit the power of interpreter scripts, either. However, the quoting issues discussed in the previous subsection still render the process of using variables somewhat cumbersome.

Restriction 3 restricts styling to stateful tree decoration. This is a true restriction that destroys the Turing-completeness of the styling phase. Some restriction on styling is necessary, because it can only commute with interpretation if it does

not rely on values computed by the interpreter. Any such restriction is bound to mingle logic and styling concerns to some degree.

In practice, we noticed restriction 3 several times. E.g., we initially tried to convert recursion depths to HTML colors in the styling phase of the fractal example, only to realized that recursion depths are unavailable in that phase.

Finally, restriction 4 is a restriction of the library transformation equivalent to restriction 1 on the interpreter stage. Thus, the same rationale applies.

5 Conclusions

The separation of concerns inherent to CMS may be beneficially dissolved during deployment using commutative composition and closed forms. This requires a pair of restrictions to application logic and presentation styling transformations. We devised restrictions that are lax on logic and stricter on styling. We showed them to be sufficient for commutativity and the computation of a closed form, enabling us to apply static preprocessing in the deployment phase.

Our approach lowers software requirements on the deployment platform. By eliminating software technology artifacts in the deployed system, we defeated the webspace utility pricing mechanism. In theory, it is now possible to deploy a modern content management system with inexpensive or free providers.

While some of our benchmarks are synthetic, we chose them in a representative manner. On these benchmarks, commutative composition and closed forms increase processing performance by an order of magnitude on average.

We explain this acceleration with the elimination of all but one pipeline stage from online execution. Reporting apart, this reduces online processing times to logic execution time. Simultaneously, the benefits of separating logic from styling concerns are preserved. Historically, the web progresses to ever more visually intricate document representations. The importance of all optimizations that address or eliminate late pipeline stages should increase in line with that trend.

Of cause, there is plenty of room for future work. The presented set of restrictions are not minimal. Techniques like partial evaluation and abstract interpretation may be capable of tightening them further. Entirely different tradeoffs between logic and presentation restrictions are conceivable and remain to be explored. Practical experience in preparing the benchmarks showed the styling restrictions to be the most pressing ones, so relaxing them is a priority.

We have not addressed the question of validating conformance to a set of restrictions. In this paper and benchmarks therein, we manually asserted conformance by careful design. I.e., we looked hard at the sources, a process prone to errors. Automatic validation would remove that potential for errors. However, to grapple problems of decidability, any such validation of application logic must by nature be aggressive. Whether the resulting fake negatives are relevant in practice remains to be evaluated.

The question of validation leads to other questions about real-world usability. We did not address any of the issues raised by multiple, distributed users in this paper. Neither did we address asset management, interface concerns or any of

the more advanced modules found in commercial CMS. To truly bridge the gap between content management and the end user, we have to extend our prototype into a more complete environment.

References

1. Dijkstra, E.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs, NJ (1976)
2. SUN Microsystems: Enterprise JavaBeans 1.1 specification. http://java.sun.com/products/ejb/docs.html (1999)
3. O'Reilly: LAMP. http://www.onlamp.net/ (2002)
4. The Apache Foundation: PHP hypertext procesor. http://www.php.net/ (2002)
5. Security Space: Internet research reports. http://www.securityspace.com/s_survey_data/ (2002)
6. Strato Medien AG: Strato Medien AG. http://www.strato.de/ (2002)
7. Puretec: 1&1 WebHosting. http://www.puretec.de/ (2002)
8. The Apache Foundation: Apache cocoon. http://xml.apache.org/cocoon2/ (2001)
9. Shaw, M., Graham, D.: Software Architecture in Practice – Perspectives on an Emerging Discipline. (1996)
10. Vignette: Vignette content suite v6. http://www.vignette.com/ (2002)
11. BroadVision: Broadvision. http://www.broadvision.com/ (2002)
12. Feldman, S.I.: Make-a program for maintaining computer programs. Software - Practice and Experience **9** (1979) 255–65
13. Clemm, G.M.: The Odin system. Lecture Notes in Computer Science **1005** (1995) 241–262
14. Leblang, D.B.: The CM challenge: Conguration management that works. In Tichy, W.F., ed.: Configuration Management, Wiley (1994)
15. Silberschatz, A., Korth, H.F., Sudarshan, S.: 12. In: Database System Concepts. 4th edn. McGraw-Hill (2001)
16. Eisenecker, U.W., Czarnecki, K.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
17. Ginger Alliance: Sablotron XSLT, DOM and XPath processor. http://www.gingerall.com/charlie/ga/xml/p_sab.xml (2002)