SIGAPP

Applied Computing Review

Dec. 2018, Vol. 18, No. 4

Frontmatter

Selected Research Articles

Applied Computing Review

Editor in Chief	Sung Y. Shin
Associate Editors	Hisham Haddad Jiman Hong John Kim Tei-Wei Kuo Maria Lencastre

SIGAPP FY'18 Quarterly Report

October 2018 – December 2018
Jiman Hong

Mission

To further the interests of the computing professionals engaged in the development of new computing applications and to transfer the capabilities of computing technology to new problem domains.

Officers

Notice to Contributing Authors

Next Issue

The planned release for the next issue of ACR is March 2019.

Document-based and Term-based Linear Methods for Pseudo-Relevance Feedback

Daniel Valcarce, Javier Parapar, Álvaro Barreiro
Information Retrieval Lab
Department of Computer Science
University of A Coruña, Spain
{daniel.valcarce, javierparapar, barreiro}@udc.es

ABSTRACT
Query expansion is a successful approach for improving Information Retrieval effectiveness. This work focuses on pseudo-relevance feedback (PRF) which provides an automatic method for expanding queries without explicit user feedback. These techniques perform an initial retrieval with the original query and select expansion terms from the top retrieved documents. We propose two linear methods for pseudo-relevance feedback, one document-based and another term-based, that models the PRF task as a matrix decomposition problem. These factorizations involve the computation of an inter-document or inter-term similarity matrix which is used for expanding the original query. These decompositions can be computed by solving a least squares regression problem with regularization and a non-negativity constraint. We evaluate our proposals on five collections against state-of-the-art baselines. We found that the term-based formulation provides high figures of MAP, nDCG and robustness index whereas the document-based formulation provides very cheap computation at the cost of a slight decrease in effectiveness.

CCS Concepts
•Information systems → Information retrieval; Information retrieval query processing; Query reformulation; *Retrieval models and ranking;*

Keywords
Information retrieval; linear methods; pseudo-relevance feedback; query expansion; linear least squares

1. INTRODUCTION
Two natural ways of approaching the enhancing of retrieval effectiveness are by improving the retrieval model or by modifying the query prompted by the user. In this paper, we focus on the latter: how to alter the original query to obtain a better rank. Query expansion techniques aim to add new terms to the query. This expanded query is expected to provide better retrieval results than the initial one. Relevance feedback is one of the most reliable types of query expansion methods, but it requires users to indicate which documents from those retrieved with the original query are relevant [29]. An alternative method for expanding the queries which does not need interaction from the user is pseudo-relevance feedback (PRF). This approach is based on the assumption that the top documents retrieved are relevant. From these pseudo-relevant documents (which form the so-called pseudo-relevant set), PRF techniques extract terms (with their corresponding weights) to expand the original query. This assumption is not too strong if the retrieval model provides decent results. In fact, research has shown that PRF is one the most effective techniques to improve the retrieval quality [28, 27, 8, 26, 4, 13, 6, 14, 15, 21, 16, 23, 41].

The language modeling framework is a fertile area of research for PRF techniques [15, 32, 16]. However, in this article, we propose a novel framework for the PRF task which is not based on language models, but in linear methods, which we call LiMe. In particular, we propose two modelings of the PRF task as matrix decomposition problems called DLiMe (Document-based Linear Methods) and TLiMe (Term-based Linear Methods). LiMe framework and the TLiMe model were first presented in our previous article [39]. In this work, we extend the LiMe framework by proposing DLiMe.

RFMF was the first formulation of PRF as a matrix decomposition problem [41] and computes a latent factor representation of documents/queries and terms using non-negative matrix factorization. In contrast, in this manuscript, we propose a different decomposition that stems from the computation of inter-document or inter-term similarities. Previous work on translation models has exploited this concept of inter-term similarities [2, 12]; however, to the best of our knowledge, no state-of-the-art PRF approach directly leverages inter-document or inter-term similarities. Our matrix formulations enable to compute these similarities that yield within the query and the pseudo-relevant set. We use the information of these relationships between documents or terms to expand the original query.

Since producing a good rank of expansion terms is critical for a successful PRF technique, the modeling of inter-term similarities seems to be a desirable property. Additionally, computing good weights for those expansion terms is a critical factor in the performance of a PRF technique. We also think that modeling the relationship between pseudo-relevant documents can be a faster way to produce expansion terms because the number of documents is much smaller than the number of terms in the pseudo-relevant set. In fact, our experiments show that the computation of inter-term similarities produces high-quality rankings of expansion terms and weights. In contrast, our proposal based on inter-document similarities is computationally very cheap at the expense of slightly worse expansion terms.

As [41] showed, an advantage of addressing PRF as a matrix decomposition problem is that it admits different types of features for representing the query and the pseudo-relevant set. Since these features are independent of the retrieval model, LiMe is a general framework for PRF that can be plugged on top of any retrieval engine. Although we can plug in retrieval-dependent features or a theoretical probabilistic weighting function into LiMe if desired, we leave those ideas for future work. In this and previous paper, we explore well-known and straightforward weighting functions which allow us to outperform state-of-the-art techniques.

LiMe modeling of the PRF task paves the way for developing multiple PRF algorithms since the proposed formulations of the matrix decompositions can be calculated in various ways. In this paper, we use a method based on regularized linear least squares regression. On the one hand, we employ a ℓ_2 regularization scheme to avoid overfitting. On the other hand, we use ℓ_1 regularization to enforce sparsity into the learned inter-document or inter-term similarities. This method provides an automatic feature selection which gives us a more compact solution with the corresponding efficiency gains. The combination of ℓ_1 and ℓ_2 regularization for linear least squares problems is also known as an elastic net regression in Statistics [44]. Additionally, we add non-negativity constraints to force the computed similarities to be positive to increase the interpretability of the models.

We thoroughly evaluate DLiMe and TLiMe on five TREC collections. The obtained results show that TLiMe outperforms state-of-the-art baselines regarding several common effectiveness metrics. Moreover, TLiMe achieved high values of robustness compared to the baselines. These findings highlight the applicability of TLiMe as a pseudo-relevance feedback technique. In contrast, DLiMe provides a computationally cheaper alternative with a slight decrease in effectiveness. It is important to note that LiMe framework can exploit different features allowing the exploration of further features schemes.

In summary, the contributions of this paper are DLiMe and TLiMe, two novel matrix decomposition formulations of the PRF task involving inter-document and inter-term similarities and an algorithm based on constrained elastic net regression for solving the proposed matrix decompositions and computing the expansion terms. The empirical evaluation of the effectiveness of the proposed methods against state-of-the-art baselines shows that DLiMe and TLiMe are competitive PRF techniques.

2. BACKGROUND

In this section, we first describe pseudo-relevance feedback (PRF). Then, we focus on state-of-the-art PRF techniques based on the language modeling framework [24] because they perform notably well in practice [13, 15, 16, 41]. Afterward, we introduce previous work on PRF using matrix factorization [41]. Finally, we introduce linear methods for regression problems since our proposal rests on these models.

2.1 Pseudo-Relevance Feedback (PRF)

Query expansion methods aim to add new terms to the original query. These techniques can improve the performance of retrieval models when answering the users' information needs. Using true relevance feedback from the user is highly effective, but also difficult to obtain. Hence, automatic query expansion techniques, which do not require feedback from the user, can be beneficial in practice [5]. Given the utility of these methods, it is not surprising that initial work on automatic query expansion dates from the sixties [18]. Manifold strategies for approaching this problem have been developed [5]; however, the foundations of PRF were established in the late seventies [8]. Pseudo-relevance feedback (also known as blind relevance feedback) is a highly effective strategy to improve the retrieval accuracy without user intervention [8, 26, 4, 42, 13, 6, 14, 15, 21, 23, 16, 41]. Instead of using explicit feedback information from the user, the top retrieved documents by the user's original query are assumed to be relevant. These documents constitute the pseudo-relevant set. PRF techniques produce an expanded version of the original query using the information from the pseudo-relevant set. PRF methods use the expanded query for a second retrieval, and the results of the second ranking are presented to the user.

A plethora of strategies for weighting the candidate expansion terms using the pseudo-relevant set information has been developed. The Rocchio framework [28] was one of the very early successful methods presented in the context of the vector space model. Rocchio algorithm modifies the query vector in a direction which is closer to the centroid of the relevant documents vectors and further from the centroid of non-relevant documents vectors. In [4], the authors used this framework with different term weighting functions including those based on pseudo-relevant feedback instead of relevance feedback such as the Binary Independence Model [27], the Robertson Selection Value [26], the Chi-square method [4] or the Kullback-Leibler distance method [4].

2.2 PRF based on Language Models

Among all the PRF techniques in the literature, those developed within the Statistical Language Model framework [24] are arguably the most prominent ones because of their sound theoretical foundation and their empirical effectiveness [15]. Within the language modeling framework, documents are ranked according to the KL divergence $D(\cdot\|\cdot)$ between the query and the document language models, θ_Q and θ_D, which is rank equivalent to the negative cross-entropy [12]:

$$Score(D, Q) = -D(\theta_Q \| \theta_D) \overset{\text{rank}}{=} \sum_{t \in V} p(t|\theta_Q) \log p(t|\theta_D) \quad (1)$$

where V is the vocabulary of the collection. To obtain better results, instead of using the original query model θ_Q, we use θ'_Q which is the result of the interpolation between θ_Q and the estimated feedback model θ_F [1, 15]:

$$p(t|\theta'_Q) = (1 - \alpha)\, p(t|\theta_Q) + \alpha\, p(t|\theta_F) \quad (2)$$

where $\alpha \in [0, 1]$ controls the relative importance of the feedback model with respect to the query model. Therefore, the task of a PRF technique under this framework is to provide an estimate of θ_F given the pseudo-relevant set F. Next, we remind two state-of-the-art PRF techniques based on the language modeling framework [15].

2.2.1 *Relevance-Based Language Models*

Relevance-based language models or, for short, Relevance Models (RM) are a state-of-the-art PRF technique that explicitly introduces the concept of relevance in language models [13]. Although RM were initially conceived for standard PRF [13], they have been used in different ways such as the generation of query variants [6], cluster-based retrieval [14] or collaborative filtering recommendation [22, 35, 36, 37].

Lavrenko and Croft [13] proposed two models for estimating the relevance: RM1 (which uses i.i.d. sampling) and RM2 (based on conditional sampling). We remind solely RM1 since it has shown to be more effective than RM2 [15]. RM1 estimates can be computed as follows when assuming uniform document prior probabilities:

$$p(t|\theta_F) \propto \sum_{D \in F} p(t|\theta_D) \prod_{q \in Q} p(q|\theta_D) \tag{3}$$

where $p(t|\theta_D)$ is the smoothed maximum likelihood estimate (MLE) of the term t under the language model of the document D with Dirichlet priors as the preferred smoothing technique [42, 13]. RM1 is typically called RM3 when it is interpolated with the original query (see Eq. 2) [1].

2.2.2 *MEDMM*

The Maximum-Entropy Divergence Minimization Model (also known as MEDMM) [16] is a PRF technique based on the Divergence Minimization Model (DMM) [42] which stems from the language modeling framework. It is similar to the Rocchio algorithm from the vector space model if we use the pseudo-relevant set to compute the relevant documents vectors and the collection model for the non-relevant documents vectors [28]. MEDMM aims to find a feedback model θ_F which minimizes the distance to the language models of the documents of the pseudo-relevant set and, at the same time, maximizes the distance to the collection model θ_C (the assumed non-relevant model). This model has a parameter λ to control the IDF effect and parameter β to control the entropy of the feedback language model:

$$\theta_F = \arg\min_{\theta} \sum_{D \in F} \alpha_D \, H(\theta, \theta_D) - \lambda \, H(\theta_F, \theta_C) - \beta \, H(\theta) \tag{4}$$

where $H(\cdot, \cdot)$ denotes the cross entropy and $H(\cdot)$ denotes the entropy.

MEDMM also gives a weight α_D for each document based on the posterior of the document language model:

$$\alpha_D = p(\theta_D|Q) = \frac{p(Q|\theta_D)}{\sum_{D' \in F} p(Q|\theta'_D)} = \frac{\prod_{t \in Q} p(t|\theta_D)}{\sum_{D' \in F} \prod_{t' \in Q} p(t'|\theta'_D)} \tag{5}$$

The analytic solution to MEDMM, obtained with Lagrange multipliers, is given by [16]:

$$p(t|\theta_F) \propto \exp\left(\frac{1}{\beta} \sum_{D \in F} \alpha_D \log p(t|\theta_D) - \frac{\lambda}{\beta} \log p(t|\theta_C)\right) \tag{6}$$

where $p(t|\theta_D)$ is the smoothed MLE of the term t under the language model θ_D using additive smoothing with parameter γ. On the other hand, $p(t|\theta_C)$ represents the MLE of the term t in the collection. The feedback model computed by MEDMM is also interpolated with the original query as in Eq. 2.

2.3 PRF based on Matrix Factorization

Other authors have focused on developing PRF models based on different ideas. In particular, RFMF was the first technique that applied matrix factorization to the PRF task [41]. This approach builds a document-term matrix X from the query and the pseudo-relevant set. They built this matrix using TF-IDF or weights derived from the language modeling framework. RFMF reconstructs, through non-negative matrix factorization (NMF), the document-term matrix and use the new weights as a scoring function to rank candidates terms for expansion. This approach is inspired by the Recommender Systems literature where matrix factorization techniques are commonplace [11]. RFMF finds the latent document and term factors with a particular parameter for the number of dimensions d of the latent factors.

Formally, NMF is a matrix factorization algorithm which decomposes the matrix $X \in \mathbb{R}_+^{m \times n}$ in two matrices $U \in \mathbb{R}_+^{m \times d}$ and $V \in \mathbb{R}_+^{d \times n}$ such that $X \approx UV$. U represents the latent factors of the query and the pseudo-relevant documents whereas V represents the latent factors of the terms.

2.4 Linear Methods

Linear methods are a simple but successful collection of techniques that have been used for regression and classification tasks. Given n features and m data points, $\vec{y} = (y_1, \ldots, y_m)^T$ is the column vector which contains the response and $\vec{x_1}, \ldots, \vec{x_n}$ are the m-dimensional vectors that contains each of the n features of the m observations. A linear method try to predict the response $\vec{y}$ using a linear combination of $\vec{x_1}, \ldots, \vec{x_n}$. The vectors of features can be arranged in the form of a matrix X of m rows and n columns. Linear regression aims to find the optimal values of the coefficients $\vec{w} = (w_1, \ldots, w_n)^T$ that minimize the error $\vec{\epsilon}$:

$$\vec{y} = X\vec{w} + \vec{\epsilon} = w_1\vec{x_1} + \cdots + w_n\vec{x_n} + \vec{\epsilon} \tag{7}$$

In particular, ordinary linear least squares models try to find the best approximate solution of this system of linear equations where the sum of squared differences between the data and the prediction made by the model serves as the measure of the goodness of the approximation:

$$\vec{w}^* = \arg\min_{\vec{w}} \|\vec{\epsilon}\|_2^2 = \arg\min_{\vec{w}} \|\vec{y} - X\vec{w}\|_2^2 \tag{8}$$

Linear least squares loss is strictly convex; thus, it has a unique minimum. Moreover, the simplicity of the model favours its explainability and interpretability. However, this model suffers from overfitting. For tackling this problem, it is common to add ℓ_2 or Tikhonov regularization (this model is also known as ridge regression in Statistics [9]). Imposing a penalty based on the squared ℓ_2-norm of the coefficients $\vec{w}$ produces a shrinking effect which is controlled by the non-negative parameter β_2:

$$\vec{w}^* = \underset{\vec{w}}{\arg\min} \|\vec{y} - X\vec{w}\|_2^2 + \beta_2 \|\vec{w}\|_2^2 \tag{9}$$

An alternative strategy to ridge regression is imposing a penalty based on the ℓ_1-norm of the coefficient vector. This approach is commonly known as lasso regression in Statistics [34]. This approach performs automatic feature selection as the value of the non-negative parameter β_1 grows:

$$\vec{w}^* = \underset{\vec{w}}{\arg\min} \|\vec{y} - X\vec{w}\|_2^2 + \beta_1 \|\vec{w}\|_1 \tag{10}$$

Since both, ridge and lasso regressions, have beneficial properties, Zou and Hastie [44] developed a technique combining both ℓ_1 and ℓ_2 regularization: the elastic net, which is a generalization of ridge and lasso regression. This approach can perform shrinkage and feature selection at the same time controlled by the non-negative parameters β_1 and β_2:

$$\vec{w}^* = \underset{\vec{w}}{\arg\min} \|\vec{y} - X\vec{w}\|_2^2 + \beta_1 \|\vec{w}\|_1 + \beta_2 \|\vec{w}\|_2^2 \tag{11}$$

3. LIME: LINEAR METHODS FOR PRF

LiMe is designed for ranking the candidate terms for producing an expanded query Q'. As it is usual in PRF, LiMe uses only information about the original query Q and the pseudo-relevant set F. The set F is composed of the top-k documents retrieved using the original query Q. We should note that LiMe treats the query as another document. Thus, for convenience, we define the extended feedback set F' as the pseudo-relevant set plus the original query ($F' = \{Q\} \cup F$) and we denote its cardinality by $m = |F'| = k + 1$. We consider as candidate terms the subset of words from the collection vocabulary V that appear in F'. We refer to this set by $V_{F'}$ and we denote its cardinality by $n = |V_{F'}|$.

3.1 LiMe Framework

We can define LiMe using matrix or vector formulation. To understand better the idea behind LiMe, we initially present our technique under a matrix formulation. Afterward, we introduce the vector representation which is much more convenient for its implementation.

Considering the query as another pseudo-relevant document, we define the matrix $X = (x_{ij}) \in \mathbb{R}^{m \times n}$. The first row represents the original query Q while the rest rows correspond to the k documents from F. Each column of X corresponds to a term from $V_{F'}$. Each element x_{ij} represents a feature between the document (or query) corresponding to the i-th position and the term t_j represented with the j-th column of X. Therefore, each row of X is a sparse feature vector representing the query or a pseudo-relevant document.

The objective of LiMe is to factorize this matrix X into the product of itself and another matrix. In the case of TLiMe, we build an inter-term matrix $W = (w_{ij}) \in \mathbb{R}_+^{n \times n}$ whereas in the case of DLiMe, we build an inter-document matrix $Z = (z_{ij}) \in \mathbb{R}_+^{m \times m}$.

3.1.1 TLiMe Formulation

The matrix W represents the inter-term similarity between pairs of words in $V_{F'}$. In particular, each entry w_{ij} symbolizes the similarity between terms t_i and t_j. To increase the interpretability of the model, we constrain the similarities to be non-negative. Moreover, to avoid the trivial solution (W equal to the identity matrix) we enforce that the main diagonal of W are all zeros. Formally, we define TLiMe as an algorithm that computes the following decomposition:

$$\begin{gathered} X \approx X\,W \\ s.t.\ \mathrm{diag}(W) = 0,\ W \geq 0 \end{gathered} \tag{12}$$

We formulate this matrix decomposition task as a constrained linear least squares optimization problem. We want to minimize the residual sum of squares of the factorization. Additionally, to avoid overfitting and to enforce a sparse solution we apply the elastic net penalty which combines ℓ_1 and ℓ_2 regularization. In this way, the objective function of LiMe is the following one:

$$\begin{aligned} W^* = \ & \underset{W}{\arg\min} && \frac{1}{2}\|X - X\,W\|_F^2 + \beta_1 \|W\|_{1,1} + \frac{\beta_2}{2}\|W\|_F^2 \\ & \text{s.t.} && \mathrm{diag}(W) = 0,\ W \geq 0 \end{aligned} \tag{13}$$

Note that the matrix $\ell_{1,1}$-norm (denoted by $\|\cdot\|_{1,1}$) is equivalent to the sum of the ℓ_1-norm of the columns. On the other hand, the squared Frobenius norm (denoted by $\|\cdot\|_F^2$) is calculated as the sum of the squares of each matrix element which is equivalent to the sum of the squared ℓ_2-norm of the columns. Using these equivalences between the matrix and vector norms, we can split this matrix formulation by columns rewriting the optimization problem in the following vector form:

$$\begin{aligned} \vec{w}_{\cdot j}^* = \ & \underset{\vec{w}_{\cdot j}}{\arg\min} && \frac{1}{2}\|\vec{x}_{\cdot j} - X\vec{w}_{\cdot j}\|_2^2 + \beta_1 \|\vec{w}_{\cdot j}\|_1 + \frac{\beta_2}{2}\|\vec{w}_{\cdot j}\|_2^2 \\ & \text{s.t.} && w_{jj} = 0,\ \vec{w}_{\cdot j} \geq 0 \end{aligned} \tag{14}$$

where the non-negativity constraint is applied to the elements of $\vec{w}_{\cdot j}$ vector which is the j-th column of the W matrix. Similarly, $\vec{x}_{\cdot j}$ represents the j-th column of the X matrix. For each term j in $V_{F'}$, we train an elastic net [44] with an equality constraint to zero in one coefficient and non-negativity constraints on the rest of the coefficients.

We merge the solutions of the regression problems depicted in Eq. 14 to build the inter-term similarity matrix W^*. We use the computed matrix decomposition to reconstruct the first row of X (which we will denote by $\hat{x}_{1\cdot}$) as follows:

$$\hat{x}_{1\cdot} = \vec{x}_{1\cdot} W^* \tag{15}$$

Note that, by construction, X is a sparse matrix (hence also the row vector $\vec{x}_{1\cdot}$) and W^* will be a sparse matrix due to the ℓ_1 regularization. Thus, the product between the row vector $\vec{x}_{1\cdot}$ and the matrix W^* is highly efficient. We use the pseudo-relevant documents for learning the inter-term similarities, but we reconstruct the first row of X because we want to expand only the query.

3.1.2 DLiMe Formulation

The document-based linear method for PRF (DLiMe) is based on the computation of the matrix $Z = (z_{ij}) \in \mathbb{R}_+^{m\times m}$. This matrix represents the inter-document similarity between pairs of elements from the extended pseudo-relevant set F' (i.e., the query and the pseudo-relevant documents). The matrix formulation of DLiMe is analogous to TLiMe:

$$\begin{aligned} X &\approx Z\, X \\ s.t.\, \operatorname{diag}(Z) &= 0,\; Z \geq 0 \end{aligned} \tag{16}$$

We also constrain Z to be non-negative to foster interpretability and enforce the diagonal to be zero to avoid the trivial solution. Since we are only interested in reconstructing the first row of X, we only need to compute the first row of Z. Therefore, DLiMe factorization can be reduced to a single constrained linear least squares optimization problem as follows:

$$\begin{aligned} \vec{z}_{1\cdot}^{*} \;=\; & \underset{\vec{z}_{1\cdot}}{\arg\min} \;\; \frac{1}{2}\left\|\vec{z}_{1\cdot} - \vec{z}_{1\cdot}X\right\|_2^2 + \beta_1 \left\|\vec{z}_{1\cdot}\right\|_1 + \frac{\beta_2}{2}\left\|\vec{z}_{1\cdot}\right\|_2^2 \\ & \text{s.t.} \quad z_{11} = 0,\; \vec{z}_{1i} \geq 0 \end{aligned} \tag{17}$$

Note that compared to TLiMe, where n least squares problem have to be solved, DLiMe is much more efficient because it only involves solving one least squares problem. To reconstruct the first row of X we simply need to perform the following vector-matrix multiplication:

$$\hat{x}_{1\cdot} = \vec{z}_{1\cdot}^{*} X \tag{18}$$

3.2 LiMe Feedback Model

LiMe feedback model is created from $\hat{x}_{1\cdot}$, which can be reconstructed using either DLiMe or TLiMe. We can normalize this vector to obtain a probability estimate. In this way, the probability of the j-th term given the feedback model is given by:

$$p(t_j|\theta_F) = \begin{cases} \dfrac{\hat{x}_{1j}}{\sum_{t_v \in V_{F'}} \hat{x}_{1v}} & \text{if } t_j \in V_{F'}, \\ 0 & \text{otherwise} \end{cases} \tag{19}$$

We only rank those terms that appear in the pseudo-relevant set or the query. Although some PRF techniques can rank all the terms in the collection, in practice, it is common to only rank those appearing in the pseudo-relevant set or the query [13, 41]. In fact, scoring terms that do not appear in F' would contradict the foundations of PRF since this approach is based on local information (i.e., the pseudo-relevant set and the query).

Although both LiMe and RFMF decomposes a similar matrix, they use different objective functions and optimization algorithms. Additionally, LiMe employs elastic net regularization. In contrast, RFMF is based on non-negative factorization which can deal with non-negative and sparse data while LiMe deals with this data by enforcing non-negativity constraints in the optimization problem. Additionally, LiMe discovers inter-document (DLiMe) or inter-term similarities (TLiMe) that yield within the pseudo-relevant set and the query while RFMF learns document and term latent factor representations.

Next, we discuss how we fill matrix $X = (x_{ij})$ with features relating query/documents i with terms j.

3.3 Feature Schemes

One advantage of LiMe is its flexibility: we can use any feature scheme to build matrix X. To foster sparsity in matrix X, we decided to fill with zeros all those entries that correspond to terms that do not appear in the current document. This approach will provide a quite sparse matrix which can be more efficiently decomposed than a dense one.

Let $s(w, D)$ be the function that assigns a score to the term w given the document D and let $f(w, D)$ be the frequency of occurrence of term w in document D, the matrix X is filled in the following manner:

$$x_{ij} = \begin{cases} s(w_j, Q) & \text{if } i = 1 \text{ and } f(w_j, Q) > 0, \\ s(w_j, D_{i-1}) & \text{if } i > 1 \text{ and } f(w_j, D_{i-1}) > 0, \\ 0 & \text{otherwise} \end{cases} \tag{20}$$

We explored several strategies based on well-known weighting functions used in Information Retrieval. We studied several term frequency measures: raw frequency counts, binarized counts and logarithmic versions. Additionally, we tried different TF-IDF formulations. We achieved the best results using the following TF-IDF weighting function proposed by Salton [31]:

$$s_{tf\text{-}idf}(w, D) = (1 + \log_2 f(w, D)) \times \log_2 \frac{|\mathcal{C}|}{df(w)} \tag{21}$$

where $|\mathcal{C}|$ is the number of documents in the collection and $df(w)$ represents the document frequency of term w (i.e., the number of documents in the collection where the term w occurs).

In any case, other alternatives may be possible. In fact, in previous work, we also reported the performance for the logarithmic TF heuristic [39]. For example, it may be worth exploring features related to the first retrieval such as the contribution of an individual term to the document score within a particular retrieval model; however, in that case, LiMe would not be independent of the retrieval technique. Also, we could derive probabilistic weighting functions (as RFMF does) at the expense of introducing a few new parameters to tune into the model. We leave for future work the investigation of additional features schemes. Nevertheless, the ability of LiMe for performing well with simple and well-known features such as TF-IDF is remarkable. Also, this weighting function is supported by decades of research in Information Retrieval.

3.4 Implementation Details

Equation 14 shows that the computation of matrix W^* can be divided in multiple linear regression problems, one for each vector $\vec{w}_{\cdot j}^{*}$ which represents a term in $V_{F'}$. Thus, each column of matrix W^* can be computed separately and, if needed, in parallel without any dependencies among them. In contrast, DLiMe only requires to solve one least squares problem (Eq. 17). To solve these regression problems, we used the highly efficient BCLS[1] (Bound-Constrained Least

[1]See `http://www.cs.ubc.ca/~mpf/bcls`

Squares) library, which implements a two-metric projected-descent method for solving bound-constrained least squares problems.

An additional optimization for TLiMe is to drop part of the matrix W^*. This matrix is used for computing expansion terms when multiplied by vector $\vec{x}_{1\cdot}$ (see Eq. 15). Therefore, we only need those rows that correspond to a term in the original query. If we only store those similarities, we save much space since the number of terms in a query prompted by a user is tiny compared to the number of rows.

4. EXPERIMENTS

In this section, we assess the performance of LiMe against state-of-the-art techniques. The experiments were performed using Terrier [17] on five TREC collections. We describe the evaluation methodology and explain the choice of baselines and the parameter setting. Finally, we present and analyze the results comparing the behavior of LiMe concerning the baselines.

4.1 Evaluation Methodology

We conducted the experiments on diverse TREC collections commonly used in PRF literature [15, 16, 41]: AP88-89, TREC-678, Robust04, WT10G and GOV2. The first one is a subset of the Associated Press collection from years 1988 and 1989. The second collection is based on TREC disks 4 and 5. The third dataset was used in the TREC Robust Track 2004 and consists of poorly performing topics. The fourth one, the WT10G collection, is a general web crawl used in the TREC Web track 2000-2001. Finally, we also ran our experiments on a large dataset, the GOV2 collection, which is a web crawl of `.gov` websites from 2004 (used in the TREC Terabyte track 2004-2006 and the Million query track 2007-2008). We applied training and test evaluation on all collections. We found the model hyperparameters that maximize MAP (mean average precision) using the training topics, and we used the test topics to evaluate the performance of the methods. Table 1 describes each collection and the training and test splits.

We produced a rank of 1000 documents per query. We evaluated MAP and nDCG (normalized discounted cumulative gain) using `trec_eval`[2] at a cut-off of 1000. Additionally, we measured the RI (robustness index or reliability of improvement [30]) against the non-expanded query. This metric, which ranges in the interval $[-1, 1]$, is computed as the number of topics improved by using PRF minus the number of topics hurt by the PRF technique divided by the number of topics. We employed one-tail permutation test with 10,000 randomizations and $p < 0.05$ to measure if the improvements regarding MAP and nDCG were statistically significant [33]. We cannot apply a paired statistic to RI because it is a global metric.

We used title queries from TREC topics. We preprocessed the collections with the standard Terrier stopwords removal and Porter stemmer since previous work recommended the use of stemming and stopwords removal [15].

[2]See `http://trec.nist.gov/trec_eval`

Table 1: Collections statistics.

Collection	#docs	Avg doc length	Topics	
			Training	Test
AP88-89	165k	284.7	51-100	101-150
TREC-678	528k	297.1	301-350	351-400
Robust04	528k	28.3	301-450	601-700
WT10G	1,692k	399.3	451-500	501-550
GOV2	25,205k	647.9	701-750	751-800

4.2 Baselines and Parameter Setting

We employed the state-of-the-art language modeling framework for performing the first and second stage retrievals [24]. In particular, we used the KL divergence model (see Eq. 1) which allow us to introduce a feedback model easily [12]. For smoothing the document language models, we used Dirichlet priors smoothing [43] with parameter $\mu = 1000$. To compare the effectiveness of our proposals, we employed the following state-of-the-art baselines:

LM First, we should always compare a PRF technique against the performance of a retrieval model without feedback information. We used language modeling retrieval with Dirichlet priors ($\mu = 1000$).

RFMF We included this PRF technique because it is based on the non-negative factorization of a document-term matrix obtained from the query and the pseudo-relevant set [41]. We set the number of dimensions of the factorization, d, to the size of the relevant set plus one as the authors recommended [41]. We used the TF-IDF weighting function.

MEDMM We also employed the maximum-entropy divergence minimization model which is recognized as one of the most competitive PRF techniques [16]. We followed the recommendations of the authors, and we set the IDF parameter λ to 0.1, the entropy parameter β to 1.2 and the additive smoothing parameter γ to 0.1 [16].

RM3 Relevance-based language models are an effective PRF technique based on the language modeling framework. We use Dirichlet priors for smoothing the maximum likelihood estimate of the relevance models. We used RM3, the most effective estimate, which uses i.i.d. sampling method and interpolates the original query with the feedback model [13, 1]. We set the Dirichlet priors smoothing parameter μ' to 1000 as it is typically done [15, 16, 41].

For all the PRF models, we swept the number of top k documents retrieved in the first stage among {5, 10, 25, 50, 75, 100} and the number of expansion terms e among {5, 10, 25, 50, 75, 100}. We swept the interpolation parameter α from 0 to 1 in steps of 0.1. Regarding LiMe, we trained the β_1 and β_2 parameters. We tuned the values of β_1 among {0.01, 0.1, 1.0} and parameter β_2 among {10, 25, 50, 100, 150, 200, 250, 300, 350, 400, 450}. We selected those parameters that maximize the values of MAP in the training set.

4.3 Results and Discussion

The results of the experiments regarding MAP, nDCG, and RI are summarized in Table 2. Overall, all the PRF techniques outperform the language modeling baseline without query expansion. However, TLiMe is the only method that offered significant improvements over LM in MAP and nDCG on all collections. DLiMe showed competitive effectiveness concerning MEDMM and RM3.

To further analyze if PRF techniques are beneficial, we measured the robustness index. This value is positive for all the methods on every collection. This value means that, on average, more queries were improved rather than worsened due to the PRF techniques. Either DLiMe or TLiMe achieved the highest figures in RI on every dataset except for MEDMM on the WT10G collection. Additionally, RM3 achieve the same robustness index as TLiMe does on the Robust04 collection.

On all datasets, TLiMe achieved the highest results regarding MAP and nDCG. No baseline outperformed TLiMe on any dataset. TLiMe significantly surpassed RFMF on four out of five datasets regarding MAP and nDCG. Regarding RM3, TLiMe significantly outperformed RM3 on three collections (concerning MAP or nDCG). The strongest baseline, MEDMM, was only significantly surpassed by TLiMe on the AP88-89 collection. However, on all datasets, TLiMe showed higher values in nDCG and MAP than MEDMM. Although no baseline significantly improved TLiMe, MEDMM significantly surpassed RM3 and DLiMe regarding nDCG and MAP on the TREC-678 collection. Also, DLiMe, RM3, and MEDMM significantly improved RFMF in terms of MAP and nDCG on several datasets.

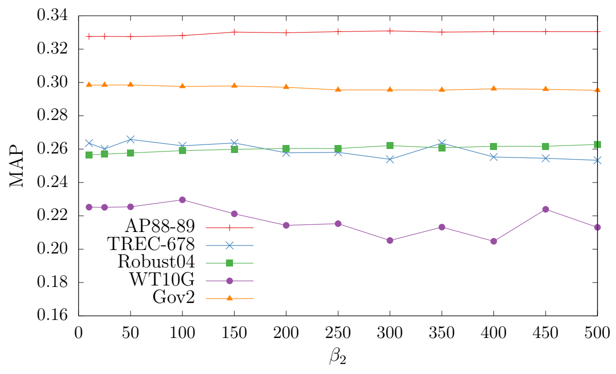
It is interesting to remark that the PRF techniques achieved the smallest improvements in the WT10G collection. This small improvement is probably due to the nature of the web which is a noisy media. Also, the values of RI on this dataset are the lowest.

Regarding the differences between DLiMe and TLiMe, the latter approach showed better figures of MAP and nDCG on all datasets. Nevertheless, the differences are significant only on the TREC-678 collections. In contrast, DLiMe provided higher RI than TLiMe on GOV2 and the same figure on AP88-89 collections

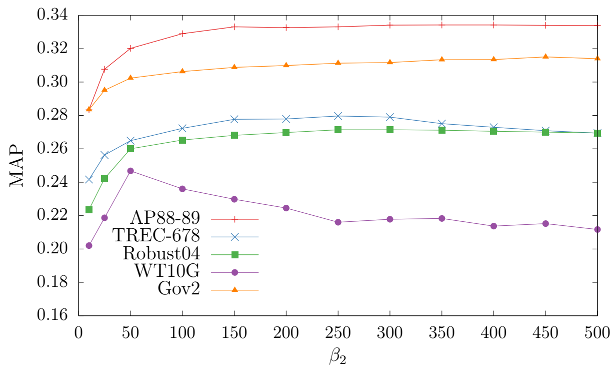
4.3.1 *Query Analysis*

To provide insights into the good results achieved by DLiMe and TLiMe, we manually studied the expanded queries produced by the tested PRF methods. Table 3 shows the top 10 expansion terms for the TREC topic 664 (“American Indian Museum”) on the Robust04 collection.

RM3 provided bad expansion terms by adding very common uninformative terms such as “will”, “1” or “new”. Those terms seem to be a problem of low IDF effect. In contrast, MEDMM yielded much better expansion terms. However, some of them are of dubious utility such as “live” or “part”. RFMF provided specific terms, but some of them are completely unrelated to the topic (e.g., “dolphin” or “rafaela”). Hence, the inferior performance of RFMF is likely to be due to the introduction of noisy terms. Regarding our methods, we can see than DLiMe provided good expansion terms. Still, this approach included the term “hey” which we think is uninformative. In this case, TLiMe yielded the best expansion terms. All of them are specific and related to the



a) DLiMe



b) TLiMe

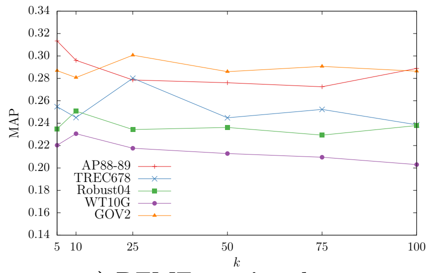
Figure 1: Sensitivity of DLiMe and TLiMe techniques to β_2 on each collection. The rest of the parameters were fixed to their optimal values.

topic.

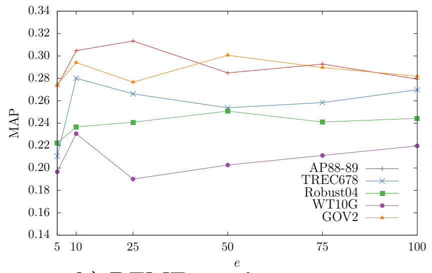
In the light of the results, we can claim that RM3 and MEDMM tend to foster those terms that appear in the majority of the pseudo-relevant set in contrast to matrix factorization approaches. LiMe was capable of selecting very specific and relevant terms such as “smithsonian” or “chumash”. RFMF was also able to include relevant terms such as “professor” but it also added non-related terms. Therefore, the main advantage of the matrix formulation is its ability to select discriminative words without being biased to popular and non-informative terms in the pseudo-relevant set. However, our approach based on inter-term or inter-doc similarities can select relevant terms while RFMF factorization approach based on document and term latent factors is incapable of filtering non-related terms.

4.3.2 *Sensitivity Analysis of Parameters*

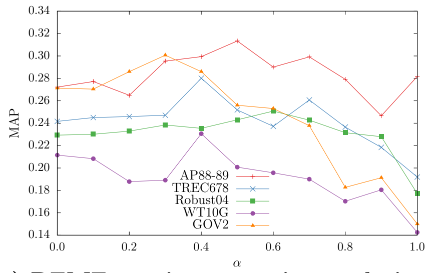
Regarding the parameters of LiMe, we observed that the differences in effectiveness between DLiMe and TLiMe when we changed the value of β_1 were minor. We can set β_1 to 0.01 reducing the number of parameters to tune and obtaining good results. Nevertheless, the inclusion of ℓ_1 regularization into LiMe models is still beneficial since it provides sparsity to the learned matrix W with the corresponding space sav-



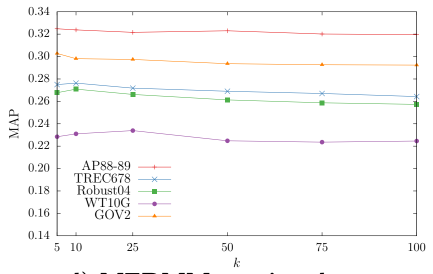
a) RFMF varying docs.



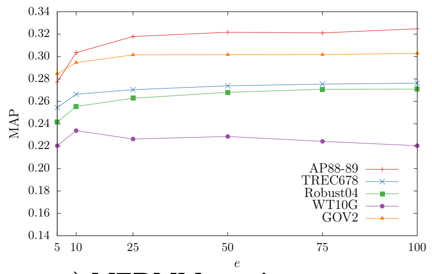
b) RFMF varying terms.



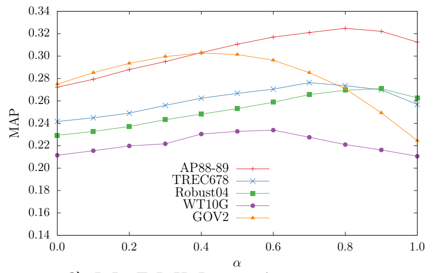
c) RFMF varying query interpolation.



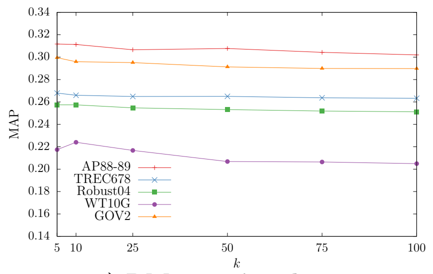
d) MEDMM varying docs.



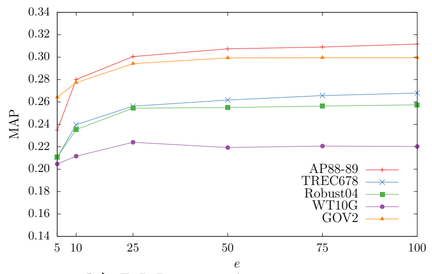
e) MEDMM varying terms.



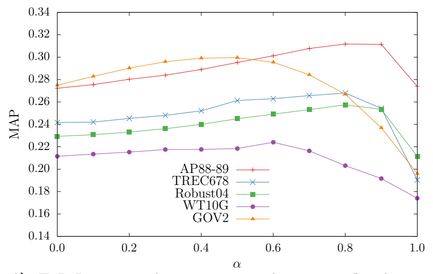
f) MEDMM varying query interpolation.



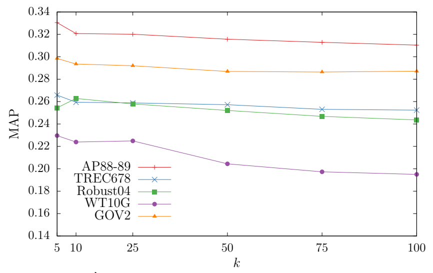
g) RM3 varying docs.



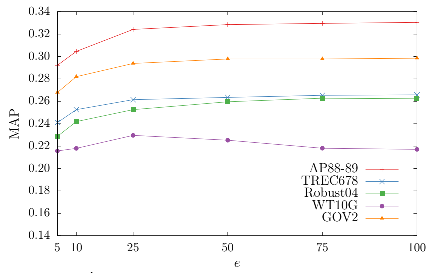
h) RM3 varying terms.



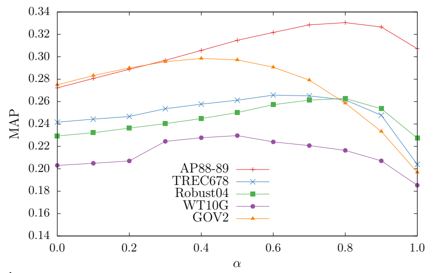
i) RM3 varying query interpolation.



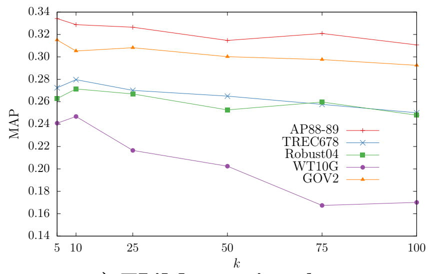
j) DLiMe varying docs.



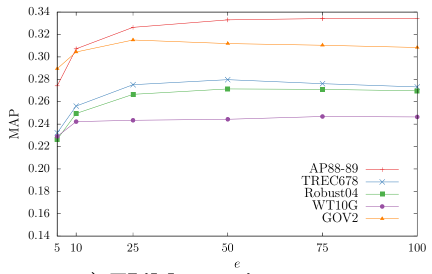
k) DLiMe varying terms.



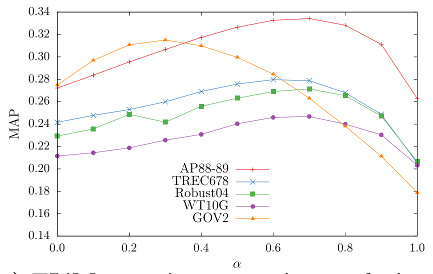
l) DLiMe varying query interpolation.



m) TLiMe varying docs.



n) TLiMe varying terms.



o) TLiMe varying query interpolation.

Figure 2: Sensitivity of RFMF, MEDMM, RM3, DLiMe and TLiMe to k (the number of feedback documents), e (the number of expansion terms) and α (the interpolation parameter of the original query with the expansion terms) on each collection. The rest of the parameters were fixed to their optimal values.

Table 2: Values of MAP, P@5, nDCG and RI for LM, RFMF, MEDMM, RM3, DLiMe and TLiMe techniques on each collection. Statistically significant improvements according to permutation test ($p<0.05$) w.r.t. to LM, RFMF, MEDMM, RM3, DLiMe and TLiMe are superscripted with a, b, c, d, e and f, respectively.

Collection	Metric	LM	RFMF	MEDMM	RM3	DLiMe	TLiMe
AP88-89	MAP	0.2349	0.2774^{a}	0.3010^{ab}	0.3002^{ab}	0.3112^{ab}	**0.3149**^{abcd}
	nDCG	0.5637	0.5749^{a}	0.5955^{ab}	0.6005^{ab}	0.6058^{ab}	**0.6085**^{ab}
	RI	–	0.42	0.42	0.50	**0.52**	**0.52**
TREC-678	MAP	0.1931	0.2072	0.2327^{abde}	0.2235^{ab}	0.2206^{ab}	**0.2357**^{abde}
	nDCG	0.4518	0.4746	0.5115^{abde}	0.4987^{ab}	0.4936^{ab}	**0.5198**^{abde}
	RI	–	0.23	0.26	0.40	0.44	0.46
Robust04	MAP	0.2914	0.3130^{a}	0.3447^{ab}	0.3488^{ab}	0.3435^{ab}	**0.3517**^{ab}
	nDCG	0.5830	0.5884	0.6227^{ab}	0.6251^{ab}	0.6247^{ab}	**0.6294**^{ab}
	RI	–	0.07	0.32	**0.37**	0.32	**0.37**
WT10G	MAP	0.2194	0.2389^{a}	0.2472^{a}	0.2470^{a}	0.2368^{a}	**0.2476**^{a}
	nDCG	0.5212	0.5262	0.5324	0.5352	0.5290	**0.5398**^{a}
	RI	–	0.30	**0.36**	0.20	0.26	0.30
GOV2	MAP	0.3310	0.3580^{a}	0.3790^{ab}	0.3755^{ab}	0.3731^{ab}	**0.3830**^{ab}
	nDCG	0.6325	0.6453	0.6653^{ab}	0.6618^{ab}	0.6588^{ab}	**0.6698**^{abd}
	RI	–	0.42	0.66	0.60	**0.72**	0.62

Table 3: Top 10 expansion terms for the TREC topic 664 ("American Indian Museum") when using the different PRF methods on the Robust04 collection.

a) RFMF

term	weight
indian	0.1725
museum	0.1685
american	0.1505
professor	0.0193
tribal	0.0160
ancient	0.0155
dolphin	0.0153
rafaela	0.0140
activist	0.0137
racist	0.0137

b) MEDMM

term	weight
indian	0.1511
museum	0.0802
american	0.0780
cultur	0.0210
year	0.0177
live	0.0153
nation	0.0148
artifact	0.0146
part	0.0139
tribal	0.0127

c) RM3

term	weight
indian	0.1285
american	0.0895
museum	0.0874
year	0.0219
will	0.0209
west	0.0182
1	0.0167
tribal	0.0158
time	0.0149
new	0.0147

d) DLiMe

term	weight
indian	0.1392
museum	0.1365
american	0.1257
smithsonian	0.0394
artifact	0.0307
hey	0.0272
tribal	0.0271
cultur	0.0250
chumash	0.0219
tribe	0.0213

e) TLiMe

term	weight
indian	0.1392
museum	0.1364
american	0.1256
tribe	0.0393
artifact	0.0306
cultur	0.0272
tribal	0.0271
nation	0.0249
chumash	0.0219
smithsonian	0.0212

ings. Regarding β_2, we plotted the values of MAP achieved by DLiMe and TLiMe with different amount of ℓ_2 regularization in Fig. 1. Except for the WT10G collection, the parameter β_2 is relatively stable among the values 150 and 400 for both DLiMe and TLiMe.

We also studied how DLiMe and TLiMe behave varying the size of the pseudo-relevant set k, the number of expansion terms e and the interpolation parameter α against the baselines RFMF, MEDMM, and RM3. Figure 2 summarizes the results of the sensitivity analysis regarding MAP. The general trend is that a high number of pseudo-relevant documents hurts the performance of the PRF techniques. The optimal number of feedback documents was never higher than 25. LiMe methods and RM3 are quite stable, and they behave optimally with 5-10 documents. In contrast, RFMF and MEDMM may require up to 25 documents in the pseudo-relevant set depending on the dataset.

The optimal number of expansion terms is quite variable. MEDMM and RM3 require more expansion terms than any other approach except on the WT10G dataset which is the noisiest one. LiMe methods are robust to noisy collections and work well with a high number of terms on WT10G. In contrast, RFMF is the technique that requires the smallest number of expansion terms in general. Finally, DLiMe and TLiMe are situated between the two extremes.

Regarding the interpolation parameter α, except for the GOV2 collection, we observed that the optimal values for DLiMe and TLiMe lie within a narrower interval than the optimal values for RFMF, MEDMM, and RM3. Nevertheless, we can see that α has a notable impact on any PRF technique and we should adequately tune it. Overall, the performance of RFMF is very unstable when we vary α (to a lesser extent, this is also true when varying the other parameters). We also found that when we do not interpolate the feedback model with the original query by setting $\alpha = 1$ (i.e., when we use the feedback model as the expanded query), RM3 showed the lowest performance. In general, we observed that DLiMe, TLiMe, and MEDMM generate better feedback models to use in isolation.

5. RELATED WORK

Pseudo-relevance feedback (PRF) is a fertile area of research in Information Retrieval [28, 27, 8, 26, 4, 13, 6, 14, 15, 32, 21, 23, 16, 41]. Among the PRF techniques, those based on the language modeling framework have showed great effectiveness [15]. Therefore, we used them as baselines and described them in Section 2. Additionally, we included RFMF as a baseline because it was the first work that modeled the PRF task as a matrix factorization problem [41].

PRF methods have been adapted to collaborative filtering recommendation with great success [22]. In particular, relevance-based language models [22, 36, 37, 38] and the Rocchio framework [35]. Conversely, RFMF is a case of a recommendation technique applied to PRF [41].

Following this analogy between PRF and collaborative filtering, we can find a state-of-the-art recommendation technique, SLIM [20], which is also based on linear methods. SLIM decomposes the full user-item feedback producing an item-item similarity matrix using ℓ_1 and ℓ_2 regularization. With this decomposition, they reconstruct the full user-item feedback matrix to generate recommendations. In contrast, we only need to predict the first row of X since we only have to expand the query. As SLIM does, LiMe fills with zeros all the missing values of the input matrix. In the beginning, in Recommender Systems, those unknown values were not set to zero. Instead, the objective function was optimized only for the known elements. However, later research found that this procedure produces worse rankings than dealing with the whole matrix considering all missing values as zeros [7].

Although RFMF and LiMe are PRF techniques based on matrix factorization, they compute different decompositions. The differences in performance are explained by the use of different objective functions and optimization algorithms. LiMe minimizes the elastic net loss and RFMF minimizes the KL-divergence of the NMF decomposition. This diversity in performance is also found in collaborative filtering where approaches such as SLIM outperforms several alternative matrix factorization techniques [20].

Linear methods have also been used in Information Retrieval. For example, [19] proposed a learning to rank approach based on linear models that directly maximize MAP. Moreover, linear methods have been applied to other tasks such a query difficulty prediction [3]. In the context of PRF, [25] used logistic regression (a linear classification method) to discriminate between relevant and non-relevant terms. However, to the best of our knowledge, multiple elastic net models have never been applied before to the PRF task.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we presented LiMe, a framework where the PRF task is modeled as a matrix decomposition problem which involves the computation of inter-term similarities. In previous work, we proposed TLiMe, a technique based on inter-term similarities. In this extended version, we also present DLiMe which is based on an inter-document matrix. TLiMe and DLiMe factorizations are solved as linear least squares problems with ℓ_1 and ℓ_2 regularization and non-negativity constraints. For that purpose, we use not only the information from the pseudo-relevant set but also the original query before expansion. The experimental evaluation showed that TLiMe outperforms state-of-the-art baselines on five TREC datasets whereas DLiMe shows competitive effectiveness with a reduced computational cost.

This work paves the way for further investigation on linear methods for pseudo-relevance feedback. The obtained results reveal the potential of LiMe as a general PRF method usable on top of any retrieval model. LiMe is a flexible framework that allows the introduction of different document-term features. The good results achieved by DLiMe and TLiME using only TF-IDF indicate that there may be room for improvements. Therefore, exploring alternative feature schemes seems to be a promising research direction.

We also envision to include a richer representation of text features into the model. For example, the use of features extracted from Wikipedia has proved to be beneficial in the PRF task [40]. Additionally, we plan to study how other similarity measures may be useful for PRF. In particular, we plan to study translation models because they usually rely on inter-term similarities [2, 12]. Previous work on translation models learned inter-term similarities from training data [2] or employed mutual information [10].

7. ACKNOWLEDGMENTS

This work was supported by project TIN2015-64282-R (MINECO/ERDF), project GPC ED431B 2016/035 (Xunta de Galicia) and accreditation 2016-2019 ED431G/01 (Xunta de Galicia/ERDF). The first author also acknowledges the support of grant FPU014/01724 (MECD).

8. REFERENCES

[1] N. Abdul-Jaleel, J. Allan, W. B. Croft, F. Diaz, L. Larkey, X. Li, M. D. Smucker, and C. Wade. UMass at TREC 2004: Novelty and HARD. In *TREC 2004*, pages 1–13, 2004.

[2] A. Berger and J. Lafferty. Information Retrieval as Statistical Translation. In *Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '99, pages 222–229, New York, NY, USA, 1999. ACM.

[3] D. Carmel and E. Yom-Tov. Estimating the Query Difficulty for Information Retrieval. *Synthesis Lectures on Information Concepts, Retrieval, and Services*, 2(1):1–89, 2010.

[4] C. Carpineto, R. de Mori, G. Romano, and B. Bigi. An Information-Theoretic Approach to Automatic Query Expansion. *ACM Transactions on Information Systems*, 19(1):1–27, 2001.

[5] C. Carpineto and G. Romano. A Survey of Automatic Query Expansion in Information Retrieval. *ACM Computing Surveys*, 44(1):1:1–1:50, 2012.

[6] K. Collins-Thompson and J. Callan. Estimation and use of uncertainty in pseudo-relevance feedback. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '07, page 303, New York, NY, USA, 2007. ACM.

[7] P. Cremonesi, Y. Koren, and R. Turrin. Performance of Recommender Algorithms on Top-N Recommendation Tasks. In *Proceedings of the 4th ACM Conference on Recommender Systems*, RecSys '10, pages 39–46, New York, NY, USA, 2010. ACM.

[8] W. B. Croft and D. J. Harper. Using Probabilistic Models of Document Retrieval Without Relevance Information. *Journal of Documentation*, 35(4):285–295, 1979.

[9] A. E. Hoerl and R. W. Kennard. Ridge Regression: Biased Estimation for Nonorthogonal Problems. *Technometrics*, 12(1):55–67, 1970.

[10] M. Karimzadehgan and C. Zhai. Estimation of Statistical Translation Models Based on Mutual Information for Ad Hoc Information Retrieval. In *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '10, page 323, New York, NY, USA, 2010. ACM.

[11] Y. Koren and R. Bell. Advances in Collaborative Filtering. In F. Ricci, L. Rokach, and B. Shapira, editors, *Recommender Systems Handbook*, pages 77–118. Springer US, 2nd edition, 2015.

[12] J. Lafferty and C. Zhai. Document Language Models, Query Models, and Risk Minimization for Information Retrieval. In *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '01, pages 111–119, New York, NY, USA, 2001. ACM.

[13] V. Lavrenko and W. B. Croft. Relevance-Based Language Models. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '01, pages 120–127, New York, NY, USA, 2001. ACM.

[14] K. S. Lee, W. B. Croft, and J. Allan. A Cluster-based Resampling Method for Pseudo-relevance Feedback. In *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '08, pages 235–242, New York, NY, USA, 2008. ACM.

[15] Y. Lv and C. Zhai. A Comparative Study of Methods for Estimating Query Language Models with Pseudo Feedback. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, CIKM '09, pages 1895–1898, New York, NY, USA, 2009. ACM.

[16] Y. Lv and C. Zhai. Revisiting the Divergence Minimization Feedback Model. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, CIKM '14, pages 1863–1866, New York, NY, USA, 2014. ACM.

[17] C. Macdonald, R. McCreadie, R. L. T. Santos, and I. Ounis. From Puppy to Maturity: Experiences in Developing Terrier. In *Proceedings of the SIGIR 2012 Workshop in Open Source Information Retrieval*, pages 60–63, 2012.

[18] M. E. Maron and J. L. Kuhns. On Relevance, Probabilistic Indexing and Information Retrieval. *Journal of the ACM*, 7(3):216–244, 1960.

[19] D. Metzler and W. B. Croft. Linear Feature-Based Models for Information Retrieval. *Information Retrieval*, 10(3):257–274, 2007.

[20] X. Ning and G. Karypis. SLIM: Sparse Linear Methods for Top-N Recommender Systems. In *Proceedings of the 2011 IEEE 11th International Conference on Data Mining*, ICDM '11, pages 497–506, Washington, DC, USA, 2011. IEEE Computer Society.

[21] J. Parapar and Á. Barreiro. Promoting Divergent Terms in the Estimation of Relevance Models. In *Proceedings of the 3rd Cnternational Conference on Advances in Information Retrieval Theory*, ICTIR '11, pages 77–88. Springer-Verlag, sep 2011.

[22] J. Parapar, A. Bellogín, P. Castells, and Á. Barreiro. Relevance-based language modelling for recommender systems. *Information Processing and Management*, 49(4):966–980, 2013.

[23] J. Parapar, M. A. Presedo-Quindimil, and Á. Barreiro. Score Distributions for Pseudo Relevance Feedback. *Information Sciences*, 273:171–181, 2014.

[24] J. M. Ponte and W. B. Croft. A Language Modeling Approach to Information Retrieval. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '98, pages 275–281, New York, NY, USA, 1998. ACM.

[25] K. Raman, R. Udupa, P. Bhattacharya, and A. Bhole. On Improving Pseudo-Relevance Feedback Using Pseudo-Irrelevant Documents. In *ECIR '10*, pages 573–576. Springer-Verlag, 2010.

[26] S. E. Robertson. On Term Selection for Query Expansion. *Journal of Documentation*, 46(4):359–364, 1990.

[27] S. E. Robertson, K. Spärck Jones, and K. S. Jones. Relevance Weighting of Search Terms. *Journal of the American Society for Information Science*, 27(3):129–146, may 1976.

[28] J. J. Rocchio. Relevance Feedback in Information Retrieval. In G. Salton, editor, *The SMART Retrieval System - Experiments in Automatic Document Processing*, pages 313–323. Prentice Hall, 1971.

[29] I. Ruthven and M. Lalmas. A Survey on the Use of Relevance Feedback for Information Access Systems. *The Knowledge Engineering Review*, 18(2):95–145, 2003.

[30] T. Sakai, T. Manabe, and M. Koyama. Flexible Pseudo-Relevance Feedback via Selective Sampling. *ACM Transactions on Asian Language Information Processing*, 4(2):111–135, 2005.

[31] G. Salton. *The SMART Retrieval System—Experiments in Automatic Document Processing*. Prentice-Hall, Inc., 1971.

[32] J. Seo and W. B. Croft. Geometric Representations for Multiple Documents. In *Proceedings of the 33rd*

International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '10, page 251, New York, NY, USA, 2010. ACM.

[33] M. D. Smucker, J. Allan, and B. Carterette. A Comparison of Statistical Significance Tests for Information Retrieval Evaluation. In *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*, CIKM '07, page 623, New York, NY, USA, 2007. ACM.

[34] R. Tibshirani. Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288, 1996.

[35] D. Valcarce, J. Parapar, and Á. Barreiro. Efficient Pseudo-Relevance Feedback Methods for Collaborative Filtering Recommendation. In *Proceedings of the 38th European Conference on Information Retrieval*, ECIR '16, pages 602–613, Berlin, Heidelberg, 2016. Springer International Publishing.

[36] D. Valcarce, J. Parapar, and Á. Barreiro. Item-based Relevance Modelling of Recommendations for Getting Rid of Long Tail Products. *Knowledge-Based Systems*, 103:41–51, 2016.

[37] D. Valcarce, J. Parapar, and Á. Barreiro. Language Models for Collaborative Filtering Neighbourhoods. In *Proceedings of the 38th European Conference on Information Retrieval*, ECIR '16, pages 614–625, Berlin, Heidelberg, 2016. Springer.

[38] D. Valcarce, J. Parapar, and Á. Barreiro. Axiomatic Analysis of Language Modelling of Recommender Systems. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 25(Suppl. 2):113–127, 2017.

[39] D. Valcarce, J. Parapar, and Á. Barreiro. LiMe: Linear Methods for Pseudo-Relevance Feedback. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, pages 678–687, New York, NY, USA, 2018. ACM.

[40] Y. Xu, G. J. Jones, and B. Wang. Query Dependent Pseudo-Relevance Feedback Based on Wikipedia. In *Proceedings of the 32Nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '09, page 59, New York, NY, USA, 2009. ACM.

[41] H. Zamani, J. Dadashkarimi, A. Shakery, and W. B. Croft. Pseudo-Relevance Feedback Based on Matrix Factorization. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, CIKM '16, pages 1483–1492, New York, NY, USA, 2016. ACM.

[42] C. Zhai and J. Lafferty. Model-based Feedback in the Language Modeling Approach to Information Retrieval. In *Proceedings of the 10th International Conference on Information and Knowledge Management*, CIKM '01, page 403, New York, NY, USA, 2001. ACM.

[43] C. Zhai and J. Lafferty. A Study of Smoothing Methods for Language Models Applied to Information Retrieval. *ACM Transactions on Information Systems*, 22(2):179–214, 2004.

[44] H. Zou and T. Hastie. Regularization and Variable Selection via the Elastic Net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2):301–320, 2005.

ABOUT THE AUTHORS:

Daniel Valcarce is a Ph.D. student at the University of A Coruña (Spain). He obtained his B.Sc. and M.Sc. degrees in Computer Science from the same university. His research interests cover recommender systems, information retrieval and machine learning. He is particularly interested in adapting information retrieval models to solve recommendation problems. He has been reviewer for multiple conferences such as ECIR, RecSys, CIKM and SIGIR. For more information, go to https://www.dc.fi.udc.es/~dvalcarce.

Javier Parapar is an Assistant Professor at the University of A Coruña (Spain). He was President of the Spanish Society for IR from 2014 to 2018. Javier Parapar holds a B.Sc.+M.Sc. in Computer Science and he got his Ph.D. in Computer Science (cum laude) in 2013, both from the University of A Coruña (Spain). His current research interests include but are not limited to information retrieval evaluation, recommender systems, text mining, and summarization. He regularly serves as reviewer and PC member of conferences such as ACM RecSys, The Web Conference (WWW), ACM SIGIR, ECIR, etc. He is a member of IP&M editorial board and a regular reviewer for journals such as ACM TOIS, IRJ, DKE and TKDE. For more information, go to https://www.dc.fi.udc.es/~parapar.

Álvaro Barreiro is a Professor in Computer Science at the University of A Coruña (Spain) and the group leader of the Information Retrieval Lab. He has supervised five doctoral theses and five research projects of the National R&D program, as well as other regional projects and R&D projects with companies. His research interests include information retrieval models, efficiency in information retrieval systems, text and data analysis and classification, evaluation and recommender systems. He has been acknowledged as ACM Senior Member and he is a member of ACM and ACM-SIGIR, BCS and BCS-IRSG, AEPIA and SERI societies. For more information, go to https://www.dc.fi.udc.es/~barreiro.

Real-time and Energy Efficiency in Linux: Theory and Practice

Claudio Scordino
Evidence Srl
Via Carducci 56
San Giuliano Terme, Pisa, Italy
claudio@evidence.eu.com

Luca Abeni
Scuola Superiore S. Anna
Via Moruzzi 1
Pisa, Italy
luca.abeni@santannapisa.it

Juri Lelli
Red Hat, Inc.
100 E. Davie Street
Raleigh, NC
juri.lelli@redhat.com

ABSTRACT

The recent changes made in the Linux kernel aimed at achieving better energy efficiency through a tighter integration between the CPU scheduler and the frequency-scaling subsystem. However, in the original implementation, the frequency scaling mechanism was used only when there were no real-time tasks in execution. This paper shows how the deadline scheduler and the cpufreq subsystem have been extended to relax this constraint and implement an energy-aware real-time scheduling algorithm. In particular, we describe the design issues encountered when implementing the GRUB-PA algorithm on a real operating system like Linux. A set of experimental results on a multi-core ARM platform validate the effectiveness of the proposed implementation, which has been recently merged into the official Linux kernel.

CCS Concepts

•Computer systems organization → Embedded software; Real-time operating systems;

Keywords

DVFS, Power-Aware Scheduling, Real-Time Scheduling, Linux

1. INTRODUCTION

During the recent years, the ICT industry has faced a growing pressure for increasing the processing capabilities of mobile devices (like smartphones or IoT nodes) and, at the same time, extending (or, at least, not reducing) their autonomy. The battery technology on this kind of devices, in fact, has evolved too slowly for being capable of satisfying the processing needs posed by modern applications. Even worse, on these devices there is often the additional requirement of avoiding the mechanical cooling systems typically used for dissipating the heat produced by powerful processing units (because they could easily break and, in any case, would increase the overall size and energy consumption).

Moreover, some of the applications running on this new generation of embedded devices need a sufficient amount of CPU time to provide an acceptable Quality of Service (QoS), or are characterized by some *temporal constraints* that have to be respected. Hence, the system has to find a trade-off between two opposite needs: *providing an acceptable QoS* (or, respecting the applications' temporal constraints) and *reducing the energy consumption*.

Such a trade-off can be achieved by using different kinds of approaches. For example, Dynamic Voltage and Frequency Scaling (DVFS) is a renowned technique for reducing the power-consumption of CPUs by dynamically adjusting their frequency and voltage. On a General-Purpose Operating System (GPOS), the load estimation for selecting the CPU's Operating Performance Point (OPP) is usually based upon some kind of moving average. The reason is that on GPOSs there is no standard way for measuring the QoS provided by the applications. Real-Time Operating Systems (RTOS), instead, can implement algorithms that select the lowest possible OPP that does not break the specific real-time guarantees.

Although several DVFS algorithms for real-time systems have been proposed in the literature (see Section 6 for more details), none of them has convinced the Linux kernel community about its effectiveness before this work. Notice that most of the previous work only considered single-core systems, while almost all of the recent hardware architectures are characterized by multi-core CPUs. This paper, instead, presents and evaluates the design and implementation of a new DVFS algorithm (based on a multi-core extension of GRUB-PA) that has been recently integrated into the official Linux kernel.

2. THE PROBLEM

In this paper, we consider the issue of reducing the energy consumption in a Linux-based OS running a mix of real-time and non real-time tasks. The Linux kernel is widely used in the new generation of embedded and mobile devices, and can support both (soft or hard) real-time applications as well as non real-time activities. The Linux approach to DVFS is traditionally oriented to non real-time workloads, allowing to save energy (with an amount of aggressiveness that can be configured by the user) when no real-time tasks are active in the system.

To better understand the problem addressed in this paper, consider a Linux-based mobile device running some non real-time workload (scheduled with the default POSIX `SCHED_OTHER` policy) together with a time sensitive thread.

Such thread needs to periodically execute (every $100ms$) an activity requiring $30ms$ of CPU time when the CPU runs at the maximum frequency. This kind of activity is generally implemented as a real-time thread scheduled with fixed priority (using the POSIX `SCHED_FIFO` or `SCHED_RR` policy) or with EDF (using the Linux `SCHED_DEADLINE` policy described in the next sections).

GPOSs like Linux typically use an estimation of the CPU load to reduce the CPU frequency. However, this approach risks to break the timing guarantees of the real-time activities. Using the example above, if the CPU frequency is lowered too much, the computation time of the real-time activity can become larger than $100ms$; as a consequence, the activity cannot be terminated before the next periodic activation. As an alternative, it is possible to set the CPU frequency to the maximum value, thus that the real-time activity's requirements are respected. However, this solution increases the energy consumption without any good reason (when the CPU runs at the maximum frequency, the real-time activity only requires 30% of the CPU time). This issue can be partly mitigated by setting the CPU frequency to the maximum value only when the real-time thread is active. However, there is still a non negligible amount of wasted energy.

The solution proposed in this paper is based on a *theoretically sound* estimation of the real-time workload (based on the GRUB and GRUB-PA algorithms) setting the CPU frequency to lower values with respect to the maximum frequency originally set by the Linux kernel; the implemented algorithms ensure that the frequency is lowered still respecting the requirements of the real-time activities running in the system.

3. THEORETICAL BACKGROUND

In order to understand how the proposed technique works, it is useful to introduce some basic terminology about real-time systems, and some background about theoretical scheduling and DVFS algorithms.

3.1 System model

A *real-time task* τ_i can be seen as a sequence of repetitive actions, named *jobs*. Each job $J_{i,j}$ (the j^{th} instance of task τ_i) becomes ready for execution ("arrives", causing a wake-up of the task) at time $r_{i,j}$ and needs an amount of CPU time $c_{i,j}$ for completing. After executing for the amount of time $c_{i,j}$, the job finishes at time $f_{i,j}$ (and the related task blocks, unless some other job has arrived in the meanwhile); notice that by definition $f_{i,j} \geq r_{i,j} + c_{i,j}$.

The temporal constraints associated with the real-time task can be modelled as *absolute deadlines* $d_{i,j}$, where a single deadline is respected if $f_{i,j} \leq d_{i,j}$.

Task τ_i can be periodic if $\forall j, r_{i,j+1} = r_{i,j} + T_i$ (where T_i is the *task's period*) or sporadic if $\forall j, r_{i,j+1} \geq r_{i,j} + T_i$ (in this case, T_i is called minimum inter-arrival time). The Worst-Case Execution Time (WCET) of the task is defined as $C_i = \max_j\{c_{i,j}\}$, while absolute deadlines are generally set as $d_{i,j} = r_{i,j} + D_i$, where D_i is the *relative deadline* of the task. If $D_i = T_i$, we talk about *implicit deadlines*.

3.2 The Constant Bandwidth Server

While the real-time theory (starting from [15]) has provided a huge amount of different algorithms for scheduling real-time tasks, in this paper we focus on the class of *reservation-based algorithms* [21], based on the idea of assigning the resource to task τ_i for an amount of time Q_i every period P_i. The specific algorithm considered in this paper is the Constant Bandwidth Server (CBS) [1], because it is the algorithm originally implemented in the Linux kernel (as described in the next section). The CBS, in turn, is based on the Earliest Deadline First (EDF) [15] algorithm: in particular, CBS allows to assign deadlines to tasks that are then used by EDF for CPU scheduling.

If we consider the CPU speed as invariant (i.e. all the CPUs / CPU cores in the system have the same constant speed), the behaviour of the algorithm is very simple: each real-time task is assigned a CPU reservation (Q_i, P_i), meaning that the task needs to execute at least for the *"runtime"* Q_i[1] every *period* of time P_i. The ratio Q_i/P_i denotes the fraction of CPU time reserved for the task.

If a task tries to execute for more than Q_i in a period P_i, then it gets *throttled* (i.e., not selected for execution) until the end of the current reservation period P_i. This effect is achieved by implementing three mechanisms:

- accounting: each task is assigned a *current runtime*, that is decreased when the task executes. In particular, if the task executes for a time δ, its current runtime is decreased by δ.
- enforcement: when the current runtime of a task arrives at 0, the task is throttled and cannot be scheduled until the current runtime is replenished.
- replenishment: at the end of a reservation period (when the time becomes equal to the scheduling deadline of the task), the current runtime of the task is replenished to the maximum value Q_i (and the scheduling deadline is postponed by P_i).

In this way, each task is constrained to not use more than its reserved CPU share — i.e., a maximum of Q_i every P_i units of time. This behavior (known as "hard reservation" [21]) has been designed to avoid the "deadline aging" phenomenon [4, 23], where a task consumes its future reservation due to other real-time tasks not ready to run. Moreover, the hard reservation behaviour avoids the starvation of lower priority tasks due to misbehaving real-time tasks (i.e., *isolation* property). On the other hand, however, it makes the scheduler not work conserving, since a real-time task might be throttled even when the rest of the system is idle.

3.3 DVFS

When adding DVFS, resource-reservation algorithms become more complex. For example, it has to be decided if the algorithm reserves a fixed amount of CPU time to the task (independently from the CPU speed), or if it allows the task to execute some kind of "fixed amount of work" in every period

[1]Notice that the Linux "runtime" is often called "budget" in the real-time literature.

P_i (in this case, the reserved amount of time Q_i has to be rescaled according to the current CPU speed / frequency).

The solution adopted in the Linux kernel is to let the runtime Q_i specify the amount of time reserved to the task *when the CPU runs at its maximum speed*, and to allow tasks to execute the same amount of work in a period P_i independently from the actual CPU speed. This means that the CPU scheduler has two different possibilities:

1. Always execute real-time tasks at the maximum CPU frequency. This was the approach originally taken by the Linux kernel [2].

2. Dynamically rescale the runtime Q_i according to the actual CPU speed, as the execution time increases when reducing the CPU frequency.

To avoid complex hardware-specific models, the execution time is generally assumed to scale linearly with the frequency: if C_i is the WCET of τ_i at the maximum frequency f_{max}, then the WCET at frequency $f' < f_{max}$ will be $C'_i = C_i \frac{f_{max}}{f'}$.

In case of EDF, the utilization $U = \sum_i \frac{C_i}{T_i}$ of a real-time taskset $\Gamma = \{\tau_i\}$ at frequency f' will be $U' = \sum_i \frac{C_i}{T_i} \frac{f_{max}}{f'} = U \frac{f_{max}}{f'}$. If the system has only one CPU, the tasks are guaranteed to respect all of their deadlines if $U' \leq 1$, and this leads to

$$U' \leq 1 \Rightarrow U \frac{f_{max}}{f'} \leq 1 \Rightarrow f' \geq U f_{max}$$

In case of other scheduling algorithms and/or multi-core systems, the analysis is more complex but can still be performed.

3.4 GRUB-PA

The previous approach is quite conservative, because it assumes that all the jobs execute for the maximum possible amount of time. The real-time literature has therefore proposed various algorithms to further reduce the CPU frequency, exploiting the fact that the actual execution times could be smaller than the WCETs. In the case of CBS, in particular, this approach can be optimized by introducing an estimation of the actual utilization. The active utilization U^{act} denotes the fraction of CPU time actually used by the tasks that are active on a CPU. This metrics has been originally used by the GRUB algorithm [10] to modify the accounting mechanism.

Considering the task τ_i served by a reservation (Q_i, P_i), the U^{act} is immediately increased by $U_i = Q_i/P_i$ when τ_i wakes up (i.e., it is added to the runqueue of the scheduler). When τ_i blocks, however, the active utilization cannot be immediately decreased, otherwise this could break the real-time guarantees (e.g., if τ_i unblocks immediately later and the bandwidth has been already assigned to another task). Therefore, when τ_i blocks, its utilization Q_i/P_i is removed from U^{act} only at the so-called "0-lag time". If τ_i wakes up

[2]However, this approach caused some issues on heterogeneous multicore architectures, such as ARM big.LITTLE.

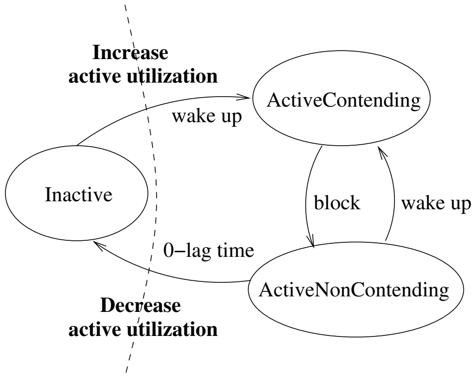


Figure 1: Diagram of the task states in the GRUB algorithm.

again before the 0-lag time, then nothing is done. Figure 1 shows the state transitions for a GRUB task.

The GRUB algorithm can be divided into two different parts: a set of rules for identifying the reclaimable bandwidth, and a set of rules for exploiting such bandwidth. The original GRUB algorithm reassigns the reclaimed bandwidth to let the active real-time tasks execute for a longer time. In the GRUB-PA algorithm [25], instead, the reclaimed CPU time is used to slow down the processor (i.e., to lower the CPU frequency). The real-time tasks execute for a longer time, but at a slower speed. The net effect is a reduction of the CPU energy consumption still respecting the real-time guarantees.

The specification of the two algorithms is very similar, as GRUB-PA only adds a couple of extensions:

- it considers the processor speed when performing the runtime accounting;
- it sets the processor speed equal to U^{act}.

The original paper [25] also devised a mechanism for taking into account the bandwidth reduction due to the time needed for switching frequency. Even if not stated, however, the proposed formula is valid only for systems with two CPU frequencies, since it does not take into account "ramp up" scenarios in which the frequency is increased multiple times. The proposed formula can be easily extended to multiple frequencies by following the original approach. The interested readers can refer to the original papers for an in-depth description of the algorithms.

4. IMPLEMENTATION IN LINUX

Some of the algorithms illustrated in the previous section were been implemented in the Linux kernel, and have been used as a base for the new DVFS mechanism.

4.1 Deadline Scheduling

Since release 2.6.23, Linux has a modular scheduling framework consisting of a main core and a set of *scheduling classes*, each encapsulating a specific scheduling policy. The binding between each policy and the related scheduler is done through a set of *hooks* (i.e., function pointers) provided by each scheduling class and called by the core scheduler.

The SCHED_DEADLINE [14] scheduling policy, available by default since version 3.14 of the Linux kernel, uses the CBS algorithm to assign deadlines to tasks, and then EDF to schedule the tasks based on such deadlines. The default algorithm implemented by SCHED_DEADLINE is based on global EDF, but tasks are inserted on per-CPU (or per-CPU core) ready queues, named *runqueues*.

Since version 4.13 of the kernel, the SCHED_DEADLINE scheduling class has been improved to optionally support CPU reclaiming. Obviously, the original GRUB algorithm could not be directly implemented in the Linux kernel, mainly because by reclaiming 100% of the CPU time it could cause a starvation of non SCHED_DEADLINE tasks and also because the algorithm did not support multiple CPUs / CPU cores. Hence, a slightly different algorithm called M-GRUB [2, 3] has been implemented by modifying the mechanism for accounting the execution time. M-GRUB improves the algorithm by adding support for multiple cores and by allowing to reclaim only a pre-defined (and user-configurable) fraction of the total CPU time (so that starvation of other tasks can be avoided). In order not to break real-time guarantees on multi-core systems, M-GRUB performs reclaiming based on the *inactive utilization*, tracked per core / runqueue. The inactive utilization of a runqueue can be computed as the difference between the total utilization of the SCHED_DEADLINE tasks associated to the runqueue (even if they are blocked) and the runqueue's active utilization U^{act}. Hence, to implement CPU reclaiming, the kernel tracks the active utilization U^{act} of each runqueue (which is stored in a per-runqueue variable named running_bw), and this value can be used to drive the DVFS mechanism, similarly to what GRUB-PA does.

The modified accounting rule can be selected per-task, by using the new SCHED_FLAG_RECLAIM flag introduced in the user-space API. If set, this flag allows a task to explicitly reclaim some further CPU time (if any) unused by other SCHED_DEADLINE tasks.

4.2 Frequency Scaling

The Linux kernel contains a subsystem, named "cpufreq", that scales the CPU frequency (and voltage) according to different user-selectable policies. It contains a set of "governors", each adopting a different power management strategy, and a set of "drivers", implementing the actuators for the frequency scaling decisions made by the selected governor. In other words, the cpufreq drivers implement the frequency scaling mechanisms (for different kinds of CPUs), while the various governors implement the different policies.

The *ondemand* and *conservative* governors aim at dynamically adjusting the CPU frequency based on some kind of estimation of the system load, but have been designed considering non real-time workloads. As a consequence, when used in real-time systems, they present two issues: their goal is not formally specified (to correctly schedule real-time tasks, instead, it is important to formalize the invariants that the DVFS mechanism has to respect) and they have poor performance due to a coarse integration with the CPU scheduler.

To be more effective, a DVFS algorithm needs to be more tightly integrated with the CPU scheduler, so that the applications' requirements can be tracked more easily. For example, all the DVFS algorithms for real-time tasks require some kind of interaction between the CPU scheduler and the frequency scaling mechanism, so that the CPU frequency can be set to the appropriate value at the right time and the temporal requirements are respected.

To address this issue, the new *schedutil* governor has been introduced (since version 4.7 of the kernel), implementing the interactions between the cpufreq subsystem and the CPU scheduler. Load estimation for non real-time tasks is achieved through the scheduler's Per-Entity Load Tracking (PELT) mechanism, which gives more importance to recent load contributions by using a geometric series. Unfortunately, the original schedutil governor preserved the real-time and SCHED_DEADLINE guarantees by running fixed priority and deadline tasks at the highest CPU frequency (while DVFS was performed only for SCHED_OTHER tasks based on the PELT estimation).

While setting the CPU at the maximum frequency is reasonable for fixed priority (i.e. SCHED_FIFO and SCHED_RR) tasks, because their CPU requirements are not known in advance, a smarter approach can be taken when scheduling SCHED_DEADLINE tasks, whose CPU demand is specified explicitly in terms of runtime and period. In particular, the utilization already tracked by the CPU reclaiming mechanism provides an estimation of the fraction of CPU time used by the SCHED_DEADLINE tasks currently running on a CPU core. In the same way as the GRUB algorithm has been modified to be usable in practice in the Linux kernel [2, 3], this paper shows how to implement a DVFS mechanism inspired by GRUB-PA.

4.3 Implementing GRUB-PA

Remember that if a task τ_i is scheduled by SCHED_DEADLINE with $Q_i \geq \max_j\{c_{i,j}\}$, $P_i \leq \min_j\{r_{i,j+1} - r_{i,j}\}$ and some admission test is respected[3], then the task is guaranteed to respect all its deadlines. Hence, the new DVFS mechanism must be defined not to break this guarantee.

Considering a single CPU core, if its frequency is set to the maximum value f^{max}, then the core will not execute SCHED_DEADLINE tasks for a fraction $(1 - U^{act})$ of the time. Slowing down the CPU frequency, the jobs' execution times $c_{i,j}$ will increase proportionally; hence all the runtimes Q_i have to be increased proportionally in order not to miss any deadline. As a consequence, the amount of time not used by SCHED_DEADLINE tasks (which can be seen as an "idle time"[4]) will decrease. If the CPU frequency f is higher than $f^{max} \cdot U^{act}$, then it is possible to guarantee that each job of each SCHED_DEADLINE task will be able to finish before its deadline (that is, it will be able of receiving an amount of

[3]The admission test for single-processor systems is $\sum_i \frac{Q_i}{T_i} \leq 1$. For multi-processor systems it is more complex — for example see Section 4.2 of [3].

[4]Notice that the term "idle" could be misleading here, as the CPU is not idle but it is simply not executing SCHED_DEADLINE tasks.

CPU time equal to $\frac{c_{i,j} \cdot f^{max}}{f}$ before the end of the reservation period). Hence, it is possible to save some energy by setting the CPU frequency to $f \geq f^{max} \cdot U^{act}$, as requested by GRUB-PA.

Again, remember that since lowering the CPU frequency slows down the performance, the tasks will need more time to do their work (in other words, job $J_{i,j}$ will execute for an amount of time $\frac{c_{i,j} \cdot f^{max}}{f}$ instead of $c_{i,j}$). Hence, the scheduler rescales the tasks' runtimes accordingly, by modifying the accounting rule. In particular, when a task executes for a time δ, its current runtime is not decreased by δ, but by $\delta \cdot f^{max}/f$ (where f is the current frequency). If $f = f^{max} \cdot U^{act}$, then the current runtime is decreased by $\delta \cdot U^{act}$, as done by GRUB (and by the new reclaiming mechanism set by `SCHED_FLAG_RECLAIM`).

When considering multiple CPUs / CPU cores, the DVFS mechanism can be based on the M-GRUB algorithm. As already mentioned, the kernel tracks two different per-runqueue utilizations: the active utilization U^{act} (as done by the original GRUB algorithm) and the total runqueue utilization (also taking into account non-active tasks). Both these utilizations can be used for frequency scaling, resulting in different DVFS behaviors: using the active utilization results in a more aggressive frequency scaling, potentially achieving more energy saving but resulting in many frequency switches; using the total runqueue utilization, instead, results in a more conservative approach, saving less energy but reducing the number of frequency switches. One of the first design choices when implementing the new DVFS mechanism has therefore concerned which of these two utilizations to use.

To better understand the implications of using the active utilization or the total runqueue utilization, consider a real-time task τ_i scheduled with a runtime Q_i and a period P_i resulting in an utilization $U_i = Q_i/P_i = 70\%$. If the jobs of this task finish after executing for less than the reserved runtime Q_i, setting the CPU frequency based on the total runqueue utilization (i.e., 70% of the maximum frequency) is a more conservative approach, that has the drawback of a poor energy efficiency. With such an approach, in fact, a real-time task contributes to the CPU frequency even when blocked. On the other hand, it has the benefit of a lower number of frequency switches (because the frequency is changed only at task creation and destruction). Considering that the WCET is often much higher than the average execution time, and aiming at reaching a better energy performance, we have rather preferred a more aggressive approach. We have followed the GRUB-PA algorithm more strictly by relying on the active utilization. This approach has the advantage of further reducing the CPU frequency whenever a blocked task enters the inactive state.

Note that, in line with the other DVFS mechanisms available in Linux, we have assumed the speed of the executed task to be proportional to the CPU frequency[5]. Moreover, it is important to point out that changing the frequency of the cores affects their processing speed, but not the latency of the memory accesses. More complex models can be elaborated to take into account both the processing and the memory access speeds.

[5]This assumption can be easily removed by integrating the scheduler and the schedutil governor with the Energy Aware Scheduling (EAS) subsystem that is currently being developed by the Linux community.

Summing up, the DVFS mechanism proposed in this paper, inspired by GRUB-PA, scales the CPU frequency based on the active utilization U^{act}. This result has been obtained by modifying the schedutil governor to use U^{act} as an estimation of the CPU load: when considering only `SCHED_DEADLINE` tasks, the resulting frequency scaling is identical to the one obtained by using GRUB-PA.

Since real CPUs do not allow to set their operating frequencies to arbitrary values (but permit to select only a limited number of discrete values), the cpufreq subsystem selects the minimum possible frequency that is higher than $f^{max} \cdot U^{act}$. Additionally, it automatically discards requests of setting a CPU frequency equal to the one already in use. The cpufreq subsystem also implements a timing mechanism for blocking all frequency switch requests until a certain amount of time has elapsed since the last switch. Without this mechanism, too many requests could keep the CPU busy, resulting in low performance and additional energy consumption. When dealing with real-time tasks, however, it is important to increase the CPU frequency immediately, otherwise some real-time task may miss its deadline. We have therefore modified cpufreq to increase the CPU frequency immediately, crossing such mechanism.

The frequency set by the schedutil governor has of course to take into account also the processing needs of the other scheduling classes. This has been easily achieved by extending the existing data structures to keep track of the load contributions of the various scheduling classes.

Another important difference with the original GRUB-PA algorithm is that it performed the accounting operations (i.e. decreasing the remaining runtime of the executing task) assuming that the CPU frequency had been set only based on the active utilization U^{act}. Since the CPU frequency can be different from $f^{max} \cdot U^{act}$ (due to the load contributions of the other scheduling classes), the accounting must be performed considering the actual frequency, and not the value of U^{act} (as originally done by GRUB and GRUB-PA). In this way, the current runtime of the task is decreased based on how much processing power (i.e., CPU frequency) it actually obtained.

Notice that, as mentioned, other scheduling classes are free to update their own load contributions (thus affecting the CPU frequency) without informing the deadline scheduler of such changes. This means that, when performing the runtime accounting, the real-time task could have been executed at a frequency different than the one currently used. We had three possible options to deal with these asynchronous frequency changes:

1. Add a notification mechanism to inform the deadline scheduler about every change of the CPU frequency made by the other scheduling classes. Despite the precise accounting, however, this approach would have introduced a large amount of unwanted overhead.

2. Prevent the other scheduling classes from changing

the CPU frequency when there is some real-time load. Even if viable, this approach would be seldom accepted by the kernel community (who traditionally is more concerned with energy efficiency rather than precise real-time execution).

3. Use the current value of the CPU frequency at the moment of the accounting, regardless of any frequency change that may have occurred since the last accounting operation. This — of course, slightly inaccurate — approach is the one that has been suggested by the kernel community.

Currently, a reliable information about the actual CPU frequency is available to the schedutil governor only on ARM platforms (through a patch recently merged into the mainline kernel). Therefore, to take advantage of the proposed DVFS mechanism on non-ARM platforms [6] without breaking the real-time guarantees, the user has to explicitly request CPU reclaiming by setting the `SCHED_FLAG_RECLAIM` flag for important deadline tasks.

Finally, we faced a few technical difficulties due to some details of the cpufreq subsystem. The schedutil governor relies on a worker kernel thread for driving frequency changes on platforms that do not have fast switching capabilities. This thread was originally scheduled with the `SCHED_FIFO` policy and a priority equal to (`MAX_USER_RT_ PRIO` / 2). However, it must have higher priority than all the `SCHED_DEADLINE` tasks, otherwise a CPU-hungry task would be able of delaying the frequency switches. As a temporary workaround, the priority of this task has been raised to the maximum possible value[7]. The kernel community expressed a bit of concern due to unwanted scheduling behaviors that may happen when mixing such a high priority kernel thread with priority inheritance or similar resource sharing protocols. However, these are considered corner cases, and this temporary solution has been accepted waiting for a more general approach.

As a final remark, it is important to observe how the modular `cpufreq` structure allowed us to implement our new DVFS policy (based on GRUB-PA) by simply modifying the schedutil governor, without having to cope with the hardware details, with the CPU interface, or with the implementation of the drivers. These modifications of the schedutil governor have been merged in the official Linux kernel since version 4.16.

5. EXPERIMENTAL EVALUATION

To validate the proposed scheduler we have performed a set of extensive tests using different ARM-based embedded boards. The presented results extend the preliminary evaluation introduced in [24]. Where not explicitly specified, the results have been measured using a "CubieTech cubietruck" board based on a dual-core Allwinner A20 ARM Cortex-A7 SoC. The CPU cores can run at seven different frequencies: 144 MHz, 312 MHz, 528 MHz, 720 MHz, 864 MHz, 912 MHz, and 960 MHz.

The real-time load has been generated through the rt-app framework[8], implementing one or more real-time tasks composed by periodic jobs with a fixed execution time, run through the ARM's LISA toolkit[9]. To achieve consistent results, all the provided values were averaged over 10 consecutive runs.

The energy consumption has been measured through the Baylibre's ACME Cape board[10] integrated with LISA. We highlight the fact that the measured values are related to the energy consumed by the whole embedded board, not just the SoC.

5.1 Hard real-time schedulability

In the first set of experiments, we verified that the new DVFS mechanism does not introduce any unexpected deadline miss in tasksets that are not supposed to miss deadlines. Note that, since `SCHED_DEADLINE` uses a global EDF algorithm to schedule tasks over multiple cores, the admission test implemented by the kernel ($\sum_i Q_i/P_i \leq x \cdot M$, where M is the number of CPU cores and x is a user-definable value between 0 and 1) does not always guarantee the hard respect of all deadlines, but only that the maximum tardiness $f_{i,j} - r_{i,j}$ has an upper bound. Hence, we decided to start with the simplest possible taskset that is guaranteed not to miss any deadline, composed by one single task.

A periodic real-time task $\tau = (C, T)$ scheduled with a reservation (Q, P) is guaranteed not to miss any deadline if $C \leq Q$; since the Linux kernel performs the CPU time accounting at every system tick, this condition must be changed in $C \leq Q - T^{tick}$, where T^{tick} is the length of a system tick. Hence, in these first experiments, we used a periodic task with period $100ms$ and execution time C, scheduled by `SCHED_DEADLINE` with a runtime Q such that $C = 0.9Q$. Multiple experiments have been performed, with the runtime Q ranging from $10ms$ to $100ms$ (hence, C ranging from $9ms$ to $90ms$). The experiments with the new DVFS mechanism have been performed using a 4.17.10 kernel (which includes the new frequency scaling policy described in this paper), while the experiments with the "old schedutil" have been performed using a 4.15.18 kernel (which did not include the new policy yet).

The results in terms of both energy consumption and deadline misses have been measured for the previous schedutil governor, for the performance governor (which keeps the CPU always at the maximum frequency) and for the new DVFS algorithm. Since the previous schedutil governor always missed all the deadlines when the runtime was set equal to the reservation period, such a configuration has been removed from the results presented in this section, and replaced with the N.A. ("Not Applicable") acronym.

Figure 2 shows the average energy consumption measured by the energy meter for different values of the reservation. The registered percentage of deadline misses is summarized

[6]Note, however, that nowadays ARM-based platforms represent the vast majority of modern mobile devices.

[7]Technically, it has been transformed into a special `SCHED_DEADLINE` task without the traditional reservation values (i.e., runtime and period) and executed for all the needed time.

[8]`https://github.com/scheduler-tools/rt-app`

[9]`https://github.com/ARM-software/lisa`

[10]`http://baylibre.com/acme/`

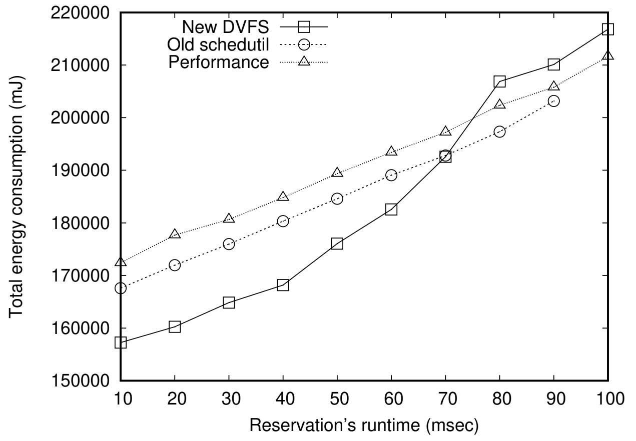


Figure 2: Energy consumption of one `SCHED_DEADLINE` task with $Q = C/0.9$ and $P = T = 100$ msec.

Table 1: Fraction of missed deadlines for one `SCHED_DEADLINE` task with $Q = C/0.9$ and $P = T = 100$ msec.

Resv. runtime	New DVFS	Performance	Schedutil
10%	0.0%	0.0%	11.5%
20%	0.0%	0.5%	47.5%
30%	0.0%	0.0%	4.7%
40%	0.0%	0.0%	43.4%
50%	0.0%	0.0%	11.3%
60%	0.0%	0.0%	3.1%
70%	0.1%	0.0%	28.6%
80%	0.0%	0.0%	13.3%
90%	0.0%	0.0%	2.8%
100%	0.0%	0.0%	N. A.

in Table 1. Such figures show that both the performance governor and the new algorithm basically miss 0 deadlines, as expected. However, the old schedutil policy presents a considerable number of unexpected missed deadlines. The bad real-time performance of the previous schedutil governor (visible in Table 1) looked strange at a first glance (by setting the CPU frequency to the maximum value when the `SCHED_DEADLINE` task is executing, the schedutil governor should be able to respect all of the deadlines). Further investigations revealed that this behavior is due to the fact that the original schedutil governor was expecting the CPU frequency to be switched immediately. Hence, when the `SCHED_DEADLINE` task executed for an amount of time δ, its current runtime was decreased by δ. Our new implementation, instead, always considers the current CPU frequency (and not the requested CPU frequency) for CPU time accounting, hence compensating for non-negligible frequency switch times.

By looking at the energy consumption, it is possible to notice that the new algorithm allows to considerably reduce the energy consumption when the utilization Q/P is lower than 70%, while maintaining a good real-time performance for all the values of the utilization. In particular, our implementation provides real-time performance similar to the performance governor without a huge loss in terms of energy efficiency.

To better understand the energy performance of the various governors, we also recorded the CPU frequency used during the experiments. The performance governor obviously drove the CPU at maximum frequency for 100% of the time, while the CPU frequencies used by the new DVFS mechanism and by the original schedutil governor (expressed as percentages of time in which the CPU executed at a given frequency) are reported in Figures 3 (new DVFS mechanism) and 4 (previous schedutil governor). As it is possible to notice, the new mechanism always tries to drive the CPU at the "correct" frequency (using the lowest possible frequency — 144 MHz — when the CPU is idle), while the old schedutil

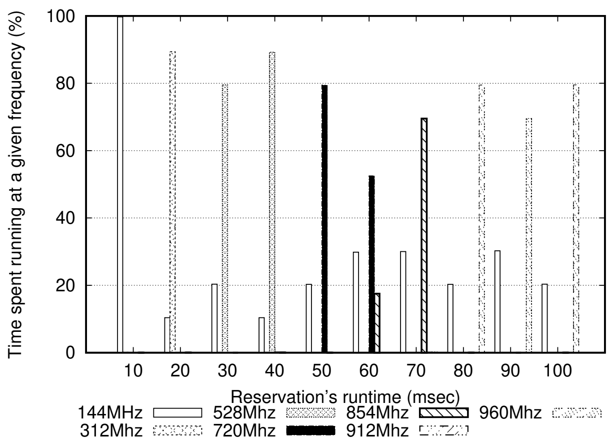


Figure 3: Percentages of time spent running the CPU at various frequencies when running a `SCHED_DEADLINE` task with $Q = C/0.9$ and $P = T = 100$ msec with the new DVFS mechanism.

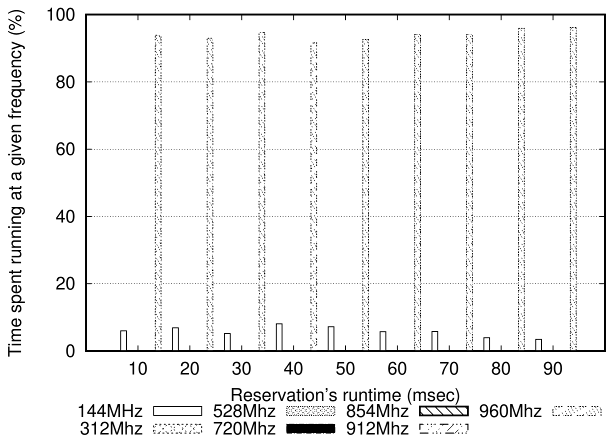


Figure 4: Percentages of time spent running the CPU at various frequencies when running a `SCHED_DEADLINE` task with $Q = C/0.9$ and $P = T = 100$ msec with the old schedutil governor.

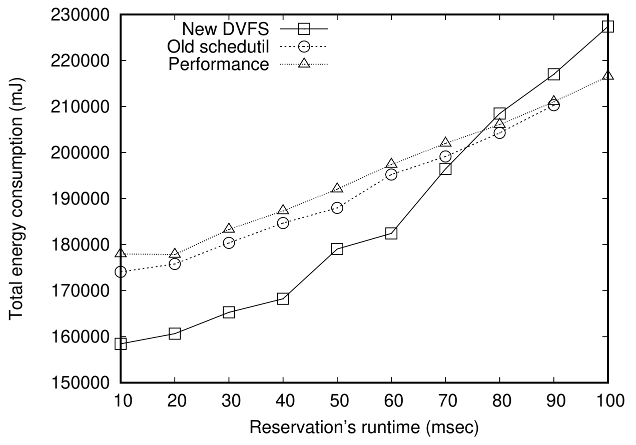


Figure 5: Energy consumption of one `SCHED_DEADLINE` task with $Q = C$ and $P = T$ =100 msec.

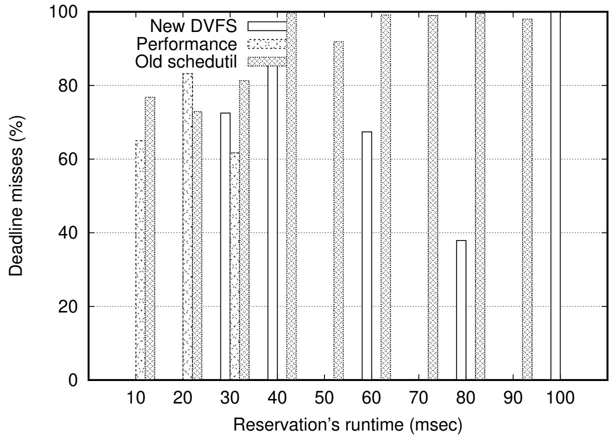


Figure 6: Fraction of missed deadlines for one `SCHED_DEADLINE` task with $Q = C$, $P = T$ =100 msec.

governor does not make use of intermediate frequencies but only uses the maximum and the minimum values.

More investigations revealed that the increased energy consumption of the new algorithm for utilizations larger than 70% is due to the kernel thread used by schedutil. When the utilization increases, frequency switches are triggered more often and the kernel thread ends up consuming a considerable amount of CPU time and increasing the energy consumption.

A large number of experiments with one single periodic task having execution time C smaller than the reserved runtime Q have been performed, all confirming the results presented above.

5.2 Badly dimensioned reservations

A second set of experiments further stressed the deadline scheduler by setting the execution time C of the jobs exactly equal to runtime Q of the the reservation serving the task. The experimental results, shown in Figure 5 (energy

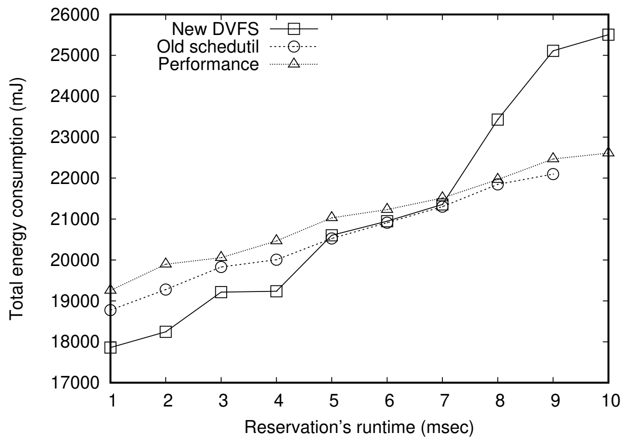


Figure 7: Energy consumption of one `SCHED_DEADLINE` task with $Q = C/0.9$ and $P = T = 10$ msec.

Table 2: Fraction of missed deadlines for one `SCHED_DEADLINE` task with $Q = C/0.9$ and $P = T = 10$ msec.

Resv. runtime	New DVFS	Performance	Schedutil
10%	0.0%	0.0%	3.6%
20%	0.0%	0.0%	3.5%
30%	0.0%	0.0%	91.3%
40%	0.9%	0.0%	0.0%
50%	0.0%	0.0%	95.7%
60%	19.2%	0.0%	49.3%
70%	39.5%	0.0%	96.3%
80%	0.0%	0.0%	49.6%
90%	49.8%	0.0%	99.0%
100%	99.8%	0.0%	N. A.

consumption) confirm the behavior already illustrated. On the other hand, the percentage of missed deadlines, shown in Figure 6, is high for all the DVFS algorithms (including the performance governor, which does not perform frequency scaling). This is expected, because with Q exactly equal to C, every small variation in the execution times can result in missed deadlines (from the theoretical point of view, it is a meta-stable situation). In other words, this figure suggests to slightly over-allocate the reservation with respect to the actual task needs, even when using the default governors.

5.3 Changing the reservation periods

A third set of experiments aimed at investigating the behavior of the scheduler when reducing the timing granularity. We have thus reduced the task period (and the `SCHED_DEADLINE` reservation period) to $10ms$, with the reservation's runtime Q ranging from $1ms$ to $10ms$ and the jobs execution time C (equal to $0.9Q$) ranging from $0.9ms$ to $9ms$. The results in Figure 7 (energy consumption) again show a gain in terms of energy efficiency for utilizations smaller than $U = 70\%$. Looking at the distribution of the misses with respect to the reservation's utilization (reported in Table 2 and

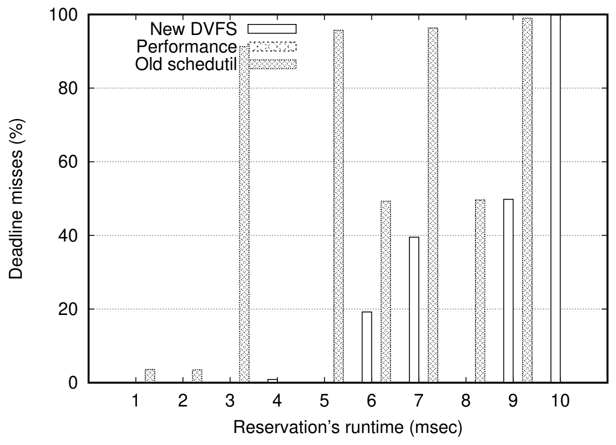


Figure 8: Fraction of missed deadlines for one `SCHED_DEADLINE` task with $Q = C/0.9$ and $P = T =$ 10 msec.

plotted in Figure 8), we can see that the performance governor never misses deadlines (as expected), while the new algorithm has a worse number of misses than the current schedutil governor for values of the reservation bandwidth higher than 60%; the default schedutil governor, instead, generally presents a non-negligible percentage of misses. Again, the average percentage of misses is lower using the new algorithm than the governor previously available in the mainline kernel.

5.4 Coping with frequency switch times

The difference between the theoretical algorithm behavior (it should cause 0 deadline misses) and the actual behavior (noticeable deadline miss percentage for utilization > 60%) is due to the fact that the physical CPU needs some time to switch the frequency (on the board used for the experiments, it is about $1.3ms$). This can be accounted for by increasing the `SCHED_DEADLINE` runtime used for scheduling the task: if the job execution time is C, the runtime has to be set to $Q \geq C + \epsilon$, where ϵ is the frequency switch time. Of course, this pessimistic assignment of the scheduling parameters will admit less `SCHED_DEADLINE` tasks in the system (for example, if $\epsilon = 3ms$ a task with job execution time $C = 8ms$ and period $P = 10ms$ would be rejected); on the other hand, it allows to respect more deadlines (making the performance of the new algorithm comparable with the ones of the performance governor for what concerns deadline misses).

The effectiveness of this approach has been confirmed by running some additional experiments, scheduling a periodic task (having period P) with `SCHED_DEADLINE`, reservation period P and runtime Q, and varying the task's execution time. For example, Table 3 shows the percentage of missed deadlines when $P = 10ms$, $Q = 7ms$, and the execution time ranges from $5ms$ to $7ms$. As it is possible to notice, the task starts to miss deadlines for $C > 5.75ms$, consistently with the fact that $\epsilon = 1.3ms$: for $C + \epsilon < Q$, no deadline is missed.

Table 3: Percentage of missed deadlines for one `SCHED_DEADLINE` task with $Q =$ 7 msec and $P = T =$ 10 msec using the new DVFS algorithm.

Task's execution time (msec)	Missed deadlines (%)
5.00	0.0
5.25	0.0
5.50	0.0
5.75	33.0
6.00	49.3
6.25	33.5
6.50	49.7
6.75	15.2
7.00	57.2

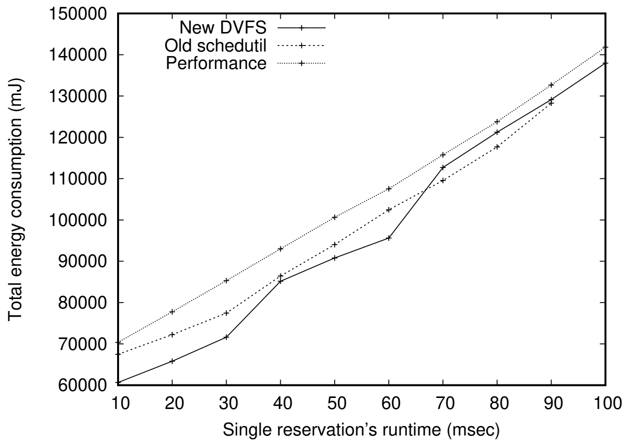


Figure 9: Energy consumption for four tasks with $Q = C/0.9$ and $P = T =$ 100 msec.

5.5 Using multiple cores

The next sets of experiments aimed at investigating the performance when increasing the number of real-time tasks and using more CPU cores. For these experiments, we used a Freescale Sabre board based on a quad-core Cortex-A9 ARM SoC are reported. The CPU cores can run at three different frequencies: 396 MHz, 792 MHz and 996 MHz.

In the fourth test, four `SCHED_DEADLINE` tasks (equal to the number of the available cores) have been executed. The four tasks, encapsulated in different reservations, had a period of $100ms$ and execution time equal to 90% of their reservation's runtime, similarly to the first test. The experimental results, shown in Figure 9 and Table 4, confirm the behavior already illustrated. Additionally, we experienced a deadlock of the target using the schedutil governor and a utilization equal to 100%.

5.6 Soft real-time schedulability

Finally, the fifth set of experiments aimed at investigating the behavior of the scheduler with generic tasksets (composed by an even higher amount of real-time tasks). We have therefore generated sets of eight reservations with het-

Table 4: Fraction of missed deadlines for four tasks with $Q = C/0.9$ and $P = T = 100$ msec.

Resv. runtime	New DVFS	Performance	Schedutil
10%	0.1%	0.1%	40.3%
20%	0.2%	0.1%	40.1%
30%	0.9%	0.1%	40.2%
40%	0.2%	0.1%	24.0%
50%	0.3%	0.1%	17.6%
60%	0.6%	0.2%	15.7%
70%	0.4%	0.3%	8.9%
80%	0.5%	0.6%	10.8%
90%	0.5%	0.5%	7.6%
100%	0.7%	0.5%	N.A.

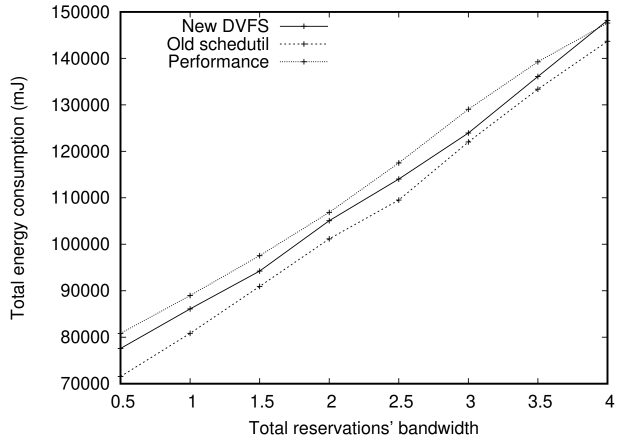


Figure 10: Energy consumption for eight randomly generated tasks with $Q = C/0.9$ and $P = T$.

erogeneous values of runtime and period, using the Randfixedsum algorithm [9]. Each reservation has been used to serve a real-time task with a runtime equal to 90% of its reservation's runtime. The measured values averaged over 10 consecutive runs with different sets of reservations are shown in Figure 10 and Figure 11. Several patterns can be observed in these figures:

- In terms of energy efficiency, the proposed scheduler does not perform as well as the current schedutil governor (even if still better than the performance governor).
- The real-time performance, however, was significantly improved, as the default governor had a high percentage of deadline misses even for low real-time loads.
- For very high values of the reservations' bandwidth, all the schedulers tend to show almost the same amount of deadline misses.

Also in this case, the proposed implementation presented the best trade-off between energy consumption and real-time performance.

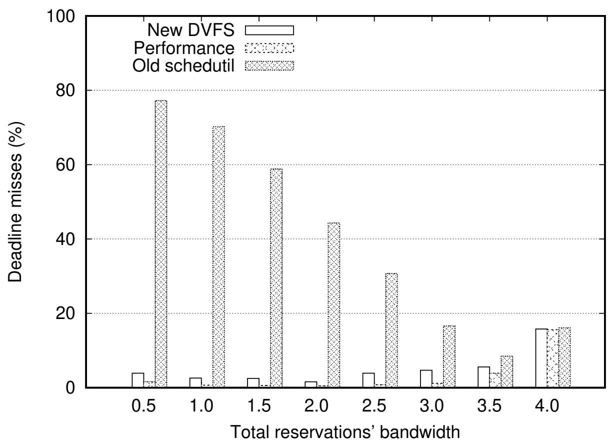


Figure 11: Fraction of missed deadlines for eight randomly generated tasks with $Q = C/0.9$ and $P = T$.

6. RELATED WORK

Several energy-aware scheduling algorithms have been proposed in the literature during the years [26, 7]. Notice that, although this paper focuses on embedded systems and real-time scheduling, energy efficiency is important not only in embedded systems, but also in other fields of computer science (for example, in high performance computing, to reduce the environmental impact of clusters [8]).

While some previous works used an estimation of the workload, based on a moving average, to drive the frequency scaling mechanism [11], most of the previous work in the real-time literature uses real-time schedulability analysis to reduce the energy consumption without breaking some kind of (hard or soft) real-time guarantees. Pillai and Shin [17] proposed the RTDVS family of algorithms, using a static approach (set $f' = Uf_{max}$) similar to the one described in Section 3.3, and then allowing to dynamically exploit the slack time to further decrease the CPU frequency. The proposed dynamic scaling only works for periodic tasks. Aydin et al. [6] used a similar static scaling algorithm, and proposed the dynamic DRA algorithms for reclaiming the spare time on Earliest Deadline First (EDF). Again, these dynamic algorithms are based on the periodic task model (using this model, it is possible to know the future arrival pattern of the task, reclaiming spare time without the risk to break temporal constraints). A power-aware algorithm for EDF scheduling based on dynamic voltage scaling has been proposed by Zhu and Mueller [27] as well. In this work, a feedback mechanism is used to cope with dynamic workloads. While most of the previously cited works focus on EDF or on dynamic priorities, dynamic voltage scaling has been applied also to schedulers based on fixed priorities, for example by by Saewong and Rajkumar [22]. An alternative approach can be to characterize the frequency requirements of each application (modeling applications as set of real-time tasks), so that compositional analysis can be applied [12].

All these works are based on the *hard real-time* task model, and therefore do not fit the scenario of kernels like Linux, that have to serve a unique mix of hard real-time tasks, soft real-time tasks, and non real-time tasks.

Some algorithms have been proposed for soft real-time tasks too, for example using DVFS techniques to deliver probabilistic real-time guarantees [28]. Lorch and Smith also addressed variable voltage scheduling of tasks with soft deadlines in [16]. Pouwelse et al. [19, 18] presented a study of power consumption and power-aware scheduling applied to multimedia streaming. Kumar et al. [13] proposed a prediction mechanism for fixed-priority scheduling of soft periodic tasks. However, these techniques are based on heuristics and cannot provide guarantees to hard real-time tasks. Qadi et al. [20] presented the DVSST algorithm that reclaims the unused bandwidth of sporadic hard real-time tasks.

Finally, the GRUB-PA algorithm [25] takes a more general approach, that allows supporting periodic, sporadic and aperiodic tasks. The algorithm has shown better energy-efficiency than most of the mentioned algorithms [25]. Moreover, it can be used to schedule both hard and soft real-time tasks, and can be integrated in systems containing non real-time tasks too.

While most of the previous works focused on scheduling algorithms and DVFS policies, there has also been some research on implementation issues, for example considering the impact of the used frequency scaling mechanism [29] (instead of focusing on the policy only).

Excluding some notable exceptions [5], most of the previous work only considered single-core / single-CPU systems.

7. CONCLUSIONS

In this paper, we have described the GRUB-PA energy-aware real-time scheduling algorithm recently integrated in the official Linux kernel, illustrating the design issues and the main choices that we have faced when implementing a theoretical scheduling algorithm in a real operating system.

The experimental results measured on a real multi-core ARM platform have shown the limits in terms of real-time performance of the schedutil governor currently available in the Linux kernel. They also confirmed that the proposed approach allows real-time performance similar to the performance governor but with a lower energy consumption.

The next activities will focus on integrating with the Energy-Aware Scheduling (EAS) project recently proposed by ARM Ltd. and Linaro.

8. ACKNOWLEDGMENTS

This work has been partially funded by the European Commission through the HERCULES (H2020 GA-688860) and the RETINA (EUROSTARS E10171) projects.

9. REFERENCES

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[2] L. Abeni, J. Lelli, C. Scordino, and L. Palopoli. Greedy CPU reclaiming for SCHED_DEADLINE. In *Proceedings of the 9th Real-Time Linux Workshop*, Dusseldorf, Germany, 2014.

[3] L. Abeni, G. Lipari, A. Parri, and Y. Sun. Multicore cpu reclaiming: Parallel or sequential? In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pages 1877–1884, New York, NY, USA, 2016. ACM.

[4] L. Abeni, L. Palopoli, C. Scordino, and G. Lipari. Resource reservations for general purpose applications. *IEEE Transactions on Industrial Informatics*, 5(1):12–21, 2009.

[5] N. Almoosa, W. Song, Y. Wardi, and S. Yalamanchili. A power capping controller for multicore processors. In *2012 American Control Conference (ACC)*, pages 4709–4714, June 2012.

[6] H. Aydin, R. Melhem, D. Mossé, and P. Mejía-Alvarez. Power-aware scheduling for periodic real-time tasks. *IEEE Transactions on Computers*, 53(5):584–600, May 2004.

[7] M. Bambagini, M. Marinoni, H. Aydin, and G. Buttazzo. Energy-aware scheduling for real-time systems: A survey. *ACM Transactions on Embededded Compututing Systems*, 15(1):7:1–7:34, Jan. 2016.

[8] A. CocaÃśa-FernÃąndez, L. SÃąnchez, and J. Ranilla. A software tool to efficiently manage the energy consumption of hpc clusters. In *2015 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, pages 1–8, Aug 2015.

[9] P. Emberson, R. Stafford, and R. I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 6–11, 2010.

[10] G.Lipari and S. Baruah. Greedy reclaimation of unused bandwidth in constant bandwidth servers. In *IEEE Proceedings of the 12th Euromicro Conference on Real-Time Systems*, Stokholm, Sweden, June 2000.

[11] L. Hu, W. Hu, R. Li, C. Li, and Z. Zhang. A time slices based novel dvs algorithm for embedded systems. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 1500–1505, Aug 2015.

[12] J. H. Kim, D. Gangadharan, O. Sokolosky, A. Legay, and I. Lee. Extensible energy planning framework for preemptive tasks. In *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 32–41, May 2017.

[13] P. Kumar and M. Srivastava. Predictive strategies for low-power rtos scheduling. In *Proceedings of the IEEE International Conference On Computer Design: VLSI In Computers & Processors (ICCD '00)*, Austin, Texas, USA, Sept. 2000.

[14] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli. Deadline scheduling in the linux kernel. *Software: Practice and Experience*, 46(6):821–839, 2016.

[15] C. L. Liu and J. Layland. Scheduling alghorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1), 1973.

[16] J. R. Lorch and A. J. Smith. Improving dynamic voltage scaling algorithms with pace. In *In Proceedings of the ACM SIGMETRICS 2001 Conference*, Cambridge, MA, June 2001.

[17] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceeding of the 18th ACM Symposium on Operating Systems Principles*, 2001.

[18] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. In *7th ACM Int. Conf. on Mobile Computing and Networking (Mobicom)*, 2001.

[19] J. Pouwelse, K. Langendoen, and H. Sips. Energy priority scheduling for variable voltage processors. In *Int. Symposium on Low Power Electronics and Design (ISLPED)*, 2001.

[20] A. Qadi, S. Goddard, and S. Farritor. A dynamic voltage scaling algorithm for sporadic tasks. In *Proceedings of the 24th Real-Time Systems Symposium*, pages 52 – 62, Cancun, Mexico, 2003.

[21] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998.

[22] S. Saewong and R. Rajkumar. Practical voltage-scaling for fixed-priority rt-systems. In *Proceedings of the ninth IEEE Real-Time and Embedded Technology and Applications Symposiuam (RTAS)*, May 2003.

[23] C. Scordino. *Dynamic Voltage Scaling for Energy-Constrained Real-Time Systems.* PhD thesis, University of Pisa, Dec. 2007.

[24] C. Scordino, L. Abeni, and J. Lelli. Energy-aware real-time scheduling in the linux kernel. In *33rd ACM/SIGAPP Symposium On Applied Computing (SAC 2018)*, Pau, France, Apr. 2018.

[25] C. Scordino and G. Lipari. A resource reservation algorithm for power-aware scheduling of periodic and aperiodic real-time tasks. *IEEE Transactions on Computers*, 55(12):1509–1522, 2006.

[26] C. S. Stangaciu, M. V. Micea, and V. I. Cretu. Energy efficiency in real-time systems: A brief overview. In *2013 IEEE 8th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, pages 275–280, May 2013.

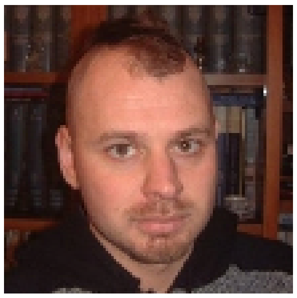
[27] Y.Zhu and F.Mueller. Feedback edf scheduling exploiting dynamic voltage scaling. In *10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*, Toronto, Canada, May 2004.

[28] Z. Zhang, X. Chen, D.-j. Qian, and C. Hu. Dynamic voltage scaling for real-time systems with system workload analysis. *IEICE transactions on electronics*, 93(3):399–406, 2010.

[29] Y. Zhu and F. Mueller. Exploiting synchronous and asynchronous dvs for feedback edf scheduling on an embedded platform. *ACM Trans. Embed. Comput. Syst.*, 7(1):3:1–3:26, Dec. 2007.

ABOUT THE AUTHORS:

Claudio Scordino graduated in Computer Engineering at the University of Pisa in 2003. In 2007 he received the PhD in Computer Science from the same university. His PhD thesis, done in collaboration with Scuola Superiore Sant'Anna, was about power-aware real-time scheduling. In 2005 he was a visiting student at the University of Pittsburgh, doing research about energy-efficiency of real-time servers. His main research interests include operating systems, real-time scheduling, energy efficiency, hypervisors. He has collaborated to the Linux kernel development since 2008. He currently works as Project Manager and team leader at Evidence Srl, involved in several international R&D projects funded by the European commission.



Luca Abeni is an Associate Professor at the ReTiS Lab of Scuola Superiore S. Anna, Pisa. He graduated in Computer Engineering at the University of Pisa in 1998, and has been a PhD student at Scuola Superiore S. Anna from 1999 to 2002, carrying out research on real-time operating systems and scheduling algorithms. In 2000, he was a visiting student at Carnegie Mellon University, working on resource reservation algorithms for real-time kernels. In 2001, he was a visiting student at Oregon Graduate Institute (Portland), working on real-time performance of the Linux kernel. From 2003 to 2006, Luca worked in Broadsat S.R.L., developing IPTV applications. Then, he moved to University of Trento as a full-time researcher, where he became later Associate Professor. Since 2017, he is back at Scuola Superiore S. Anna as Associate Professor.

Juri Lelli received a Bachelor's degree in Computer Engineering at the University of Pisa (Italy) in 2006, and a Master's degree in Computer Engineering at the University of Pisa (Italy) in 2010 with a thesis titled "Design and development of real-time scheduling mechanisms for multiprocessor systems". He then earned a PhD degree at the Scuola Superiore Sant'Anna of Pisa, Italy (ReTiS Lab). His PhD thesis focused on reducing the gap between classical real-time theory and practical implementation of real-time scheduling algorithms on General Purpose Operating Systems, with a special focus on Linux. He is one of the original authors of the SCHED_DEADLINE scheduling policy in Linux, and he is actively helping maintaining it. At the moment, he works at Red Hat, Inc., where he continues contributing to the Linux scheduler development.

On the Detection of Applications in Co-Resident Virtual Machines via a Memory Deduplication Side-Channel

Jens Lindemann
Security and Privacy Group
Department of Computer Science
University of Hamburg, Germany
lindemann@informatik.uni-hamburg.de

Mathias Fischer
Security and Privacy Group
Department of Computer Science
University of Hamburg, Germany
mfischer@informatik.uni-hamburg.de

ABSTRACT

Nowadays, hosting services of multiple customers on the same hardware via virtualiation techniques is very common. Memory deduplication allows to save physical memory by merging identical memory pages of multiple Virtual Machines (VMs) running on the same host. However, this mechanism can leak information on memory pages to other. In this paper, we propose a timing-based side-channel to identify software versions running in co-resident VMs. The attack tests whether pages that are unique to a specific software version are present in co-resident VMs. We evaluate the attack in a setting without background load and in a more realistic setting with significant background load on the host memory. Our results indicate that, with few repetitions of our attack, we can precisely identify software versions within reasonable time frames and nearly independent of the background load. Finally, we discuss potential countermeasures against the presented side-channel attack.

CCS Concepts

•**Security and privacy** → **Side-channel analysis and countermeasures; Virtualization and security;** *Vulnerability scanners;* •**Computer systems organization** → **Cloud computing;**

Keywords

security, side-channel attack, virtualization, cloud computing

1. INTRODUCTION

Our society relies more and more on the availability of Internet services. These services are increasingly provided by virtualised servers operated by cloud providers in their server infrastructures.

Attacks on cloud providers take place all the time. However, only few of them cause real damage. Mostly, such successful attacks are prepared by extensive reconnaissance in which attackers actively scan their targets. Obtaining information about the software configuration of other VMs, other users on the same VM, or the host operating system allows them to specifically exploit security vulnerabilities known to exist in the identified software versions. Conducting vulnerability scans in a provider's infrastructure from virtual machines is forbidden by the acceptable use policies of many cloud service providers [13]. Furthermore, when such scans are conducted via the network, they can be easily detected, e.g. by an Intrusion Detection System (IDS). However, so-called side-channel attacks, which do not use standard communication paths, can reveal information on a target system by evading standard detection mechanisms at the same time.

Such side-channel attacks are a danger especially in virtualised environments (e. g. [9, 11]), in which multiple virtual machines share the same hardware. One type of side-channel attacks in virtualised environments is based on the memory deduplication mechanism, which identifies and merges identical memory pages. This can save large amounts of physical memory [4, 27]. However, this mechanism can adversely affect the confidentiality of data in virtual machines. Suzaki et al. [24] have shown that it is possible to detect a software running within another VM on the same host by writing a copy of the binary into memory. This copy will then be deduplicated by the hypervisor. When this deduplicated copy of the binary is overwritten, this will take longer than overwriting random and non-deduplicated data. Thus, this gives an attacker the information that another copy of the binary is present on the host. A vulnerable software version identified in another VM does not lead to a direct attack path, as the IP address will normally be unknown and would have to be obtained using another method. If the IP address is known to the attacker, however, knowing the software version being executed enables the attacker to launch an attack specifically targeted at vulnerabilities in this version. Also, a vulnerable hypervisor version or a vulnerable software version being executed by another user on the same VM will be directly attackable.

The main contribution of this paper is a novel side-channel attack based on memory deduplication that has already been published by us as a conference paper [14]. The attack allows a curious attacker controlling a VM to gain information about the software configuration of (a) other co-located VMs, (b) other users of the same VM or (c) the host operating system. Our attack is based on identifying memory pages of a software version that are unique across all other versions of that software. Once such signature pages have been identified, their existence in co-resident VMs can be easily tested by loading just these pages into the memory of the attacker VM. Thus and contrary to related work, our attack does not presume to load a software binary completely.

Our evaluation results indicate that we can distinguish software versions to the precision of the distribution patch level. Our attack is faster than other attacks that test whether individual pages have been deduplicated as timing differences. It requires fewer measurements to detect versions that have a large number of unique pages. Furthermore, memory activity has an impact on the number of measurements required: without load, fewer measurements are required to achieve a certain level of accuracy. For example, to achieve an accuracy of more than 99.9 % in the detection of the software version with background activity on the host, at least three measurements with a minimal signature size of five pages are necessary. Such an attack would take at least 32 minutes.

As previous work did not analyse the impact of differences in software versions and the underlying operating system, we present an analysis of the effectiveness of such an attack across different software and operating system versions. We found that software binaries of the same upstream release share almost no common pages across different Linux distributions. Furthermore, we have analysed the potential for memory savings by deduplicating memory pages containing executable code across different OS and software versions.

Compared to our former work [14], we have improved our attack code by eliminating some of the noise observed in the measurements. As a result, we require fewer measurements to achieve the same level of accuracy in detecting application versions. We also evaluate the attack under more realistic conditions, i. e. with background load on the host. Furthermore, we more extensively discuss countermeasures against the identified side-channel attack.

The remainder of this paper is structured as follows: In Section 2 we discuss background information and related work. Section 3 describes our side-channel attack. In Section 4, we evaluate effectiveness and efficiency of the attack. Section 5 presents potential countermeasures and Section 6 concludes the paper.

2. BACKGROUND AND RELATED WORK

In this section we first explain the concept of memory deduplication as well as its implementation in popular hypervisors. Then, the attacker model is presented, before we discuss how executable files are loaded on Linux. Finally, we will discuss related work.

2.1 Memory Deduplication

Memory deduplication is a technique for saving physical memory on a computer. It is often deployed on hosts for virtual machines as a cost-saving measure. The memory of a computer is organised by the operating system as a set of memory pages $\mathcal{M}$. Typically, the size of a memory page $p_i \in \mathcal{M}$ is 4 096 bytes. Every memory page resident in physical memory will consume these 4 096 bytes. Memory deduplication takes advantage of the fact that there are often sets D_i of multiple identical pages $p_i = p_j \in \mathcal{M}$. To save memory, all but one page $p_m \in D_i$ will be removed from the physical memory and all memory mappings $\forall p_i \in \mathcal{M} : p_i = p_m$ updated to point to p_m instead. Subsequently, when a page $p_i \in D_i$ is to be changed, the deduplication mechanism copies it to a different memory region so that it can be modified without affecting the other copies of the page.

Note that only pages actually resident in memory can be deduplicated, whereas pages that are swapped out cannot be deduplicated. This implies that it may not be possible to detect some pages of a file by means of a deduplication side-channel attack, despite the file being loaded into the virtual memory of the host.

Figure 1 shows an example of memory deduplication. Let us assume two VMs running on a host. Each VM is assigned four pges of virtual memory. Without deduplication (Figure 1a) every page in the virtual memory of the two VMs are mapped to exactly one distinct page in physical memory. When deduplication is activated (Figure 1b), it will scan the memory and find that there are two pages of identical content. These pages are then merged into one physical memory page, resulting in the two VMs sharing a physical memory page.

In the following, we will describe the memory deduplication mechanisms of the popular KVM, Xen and VMWare ESXi hypervisors.

KVM.

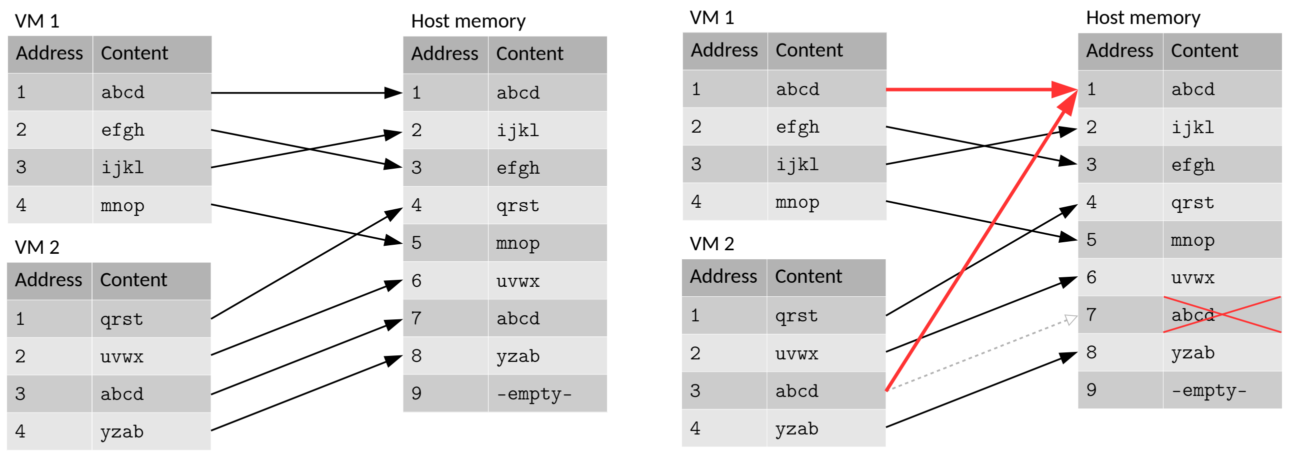
The KVM hypervisor is built into the Linux kernel and uses the "Kernel Samepage Merging" (KSM) technique [1] for memory deduplication. The guest OSs do not have to be modified for KSM. KSM runs in the background ans scans the memory pages of virtual machines for identical pages, which are then deduplicated. It will scan in specified intervals. When an interval has passed, a specific number of memory pages is scanned and, if appropriate, deduplicated.

Instead of using fixed values for interval and number of pages to scan per interval, the ksmtuned daemon can be used [22]. It will tune the KSM configuration according to the current memory usage on the host: The higher the memory usage is, the more pages will be scanned per interval. When memory usage decreases, the number of pages scanned per interval will also be decreased. Ksmtuned also supports turning off KSM when memory deduplication does not exceed a specified threshold.

Xen.

The Xen hypervisor provides a mechanism for sharing memory pages between VMs [5]. However, no mechanism for automatically identifying such pages is provided as part of the hypervisor, so that this feature is of little use in practice.

However, mechanisms that enable live deduplication in Xen have been developed by researchers. One such mechanism is the Difference Engine proposed by Gupta et al. [8]. Similar to KSM, it periodically scans the memory for shareable pages. Besides deduplicating identical pages, it also supports sharing similar pages. This is achieved by storing the difference between the deduplicated pages and patching the appropriate page when it is being accessed. Note that, unlike deduplicating identical pages, this also causes



(a) no deduplication **(b) deduplication active**

Figure 1: Memory deduplication

an overhead when pages are merely being read. In addition, it compresses unshareable pages that are not being used frequently. Compared with techniques that work on identical pages only, Difference Engine achieves higher memory savings. While the researchers originally published their code online, the project now seems dead and the repository is no longer available.

Another mechanism is Satori [18]. Unlike most other deduplication mechanisms, it requires modifications to the guest OS. Instead of periodically scanning the full VM memory for shareable pages, it checks whether a page can be deduplicated when it is being loaded. This means that even pages that only remain in memory for a short period of time can be deduplicated. However, pages that are changed to become identical to another page after they have initially been loaded into memory will not be detected. Satori allows a guest VM to specify pages that may not be shared. This implies disabling deduplication for specific memory areas and will eliminate both memory savings as well as memory timing side-channels in respect to these pages. We were unable to find a publicly available implementation of Satori.

VMWare ESXi.

VMWare ESXi uses its own deduplication mechanism, which has been described by Waldspurger [27]. Similar to the KSM mechanism used by KVM, the memory of guest VMs is regularly scanned for duplicate pages to deduplicate these. Modifications to the guest OS or the disk images used by the VM are not necessary.

2.2 Attacker Model

Our assumptions about the attacker's capabilities are as follows: A memory deduplication side-channel attack takes place on a host h that hosts a set of virtual machines M. We denote the set of all versions of an application i as A_i. Individual versions are denoted as $a_i^v \in A_i$, where v is used as a version identifier. Each virtual machine $m_k \in M$ runs a set of application versions, which are returned by $apps(m_k)$. The attacker controls at least one virtual machine $m_a \in M$. The attacker can only observe the network traffic of m_a, not that of h or any other VM $m \in M \setminus \mathrm{m_a}$. The attacker's intention is to determine a specific version $a_i^v \in A_i$ running outside their scope of control.

There are three possible attack scenarios:

- **Inter-VM.** The attacker is trying to determine a_i^v of an application A_i running on another virtual machine $m_v \in (M \setminus m_a)$
- **Intra-VM.** The attacker does not have root access to $\mathrm{m_a}$ and is trying to determine a_i^v of an application A_i being executed on m_a by another user.
- **VM-to-host.** The attacker is trying to determine a_i^v of an application A_i running on the host operating system of H.

For sake of clarity, we will concentrate on describing inter-VM attacks in the following. The mechanisms of intra-VM and VM-to-host attacks are identical. Intra-VM attacks will work on all hosts where inter-VM attacks are possible. Whether a VM-to-host attack can be performed on a host depends on whether the deduplication mechanism deduplicates pages of the host OS in addition to those of the VMs.

While the operator of a VM host trying to exploit a security vulnerability in software of a guest VM is the worst-case attacker, we do not consider Host-to-VM attacks. As the host has full access to the memory of all VMs on the host, its operator has a much easier attack path than a memory deduplication side-channel attack. Furthermore, they would also know how to communicate with an affected VM without having to find out the IP address using a separate side-channel.

2.3 Loading of Executables in Linux

As described in Sect. 2.1, the content of two memory pages must be identical for them to be deduplicated. However, the position of the page in memory is irrelevant. Thus, we need to know the content of an executing program's memory pages, but not their position in memory.

Linux uses the Executable and Linkable Format (ELF) for executable files. The current version of the standard is 1.2 [25]. ELF is also used by many other modern Unix operating systems, such as FreeBSD and Solaris.

The data in an ELF file is organised in sections and segments. A file contains one or more segments. A segment contains one or more sections. For executing a program, only segments are relevant. An executable contains a program header table describing the segments contained in the file. For each segment, it contains information such as the type of the segment, its position and size in the file, the virtual memory address that it shall be loaded to and alignment requirements.

In the following, we will describe how ELF executables are placed into memory pages by the Linux kernel. The loading mechanism for ELF files can be found in the source file fs/binfmt_elf.c. To load an ELF object file, the function load_elf_binary is called. The function iterates over all entries in the program header table. It checks whether the corresponding segment is a loadable (PT_LOAD) segment. If it is, it calls the elf_map function.

The elf_map function then maps the specified segment into memory. It maps full memory pages, even if the virtual memory address specified in the program header table points to a position within a page. If this is the case, the bytes directly preceding the segment are loaded until the memory page is filled. A similar approach is taken if the segment does not end on a page boundary: The bytes directly succeeding the segment are loaded until the page is filled.

2.4 Related Work

Data deduplication is similar to memory deduplication, but aims to save disk space by deduplicating copies of identical data in storage. It can be very effective (savings of 70 to 80 percent) when applied to images of similarly configured VM images [12, 16]. However, its effectiveness is reduced for heterogeneous software configurations on the VMs [12]. Timing side-channels also exist in data deduplication. They can reveal whether a file (or even a part of it) is already present on a storage service through timing differences caused by copy-on-write [9] or non-uploading of file contents [19]. Researchers have proposed Message-Locked Encryption as a countermeasure [2, 21].

Gruss et al. [6] demonstrate that it is possible to perform a *memory deduplication side-channel attack* from within a browser using JavaScript. Bosman et al. [3] apply this approach to the Microsoft Edge browser on Windows 8.1 and 10, which use memory deduplication by default. Their attack does not require a virtualised environment, but targets end-user computers. The authors show that it is feasible to read arbitrary data from the target computer's memory.

Irazoqui et al. [11] describe an approach to detect the version of a cryptography library executed on a co-resident VM. They make use of a Flush+Reload attack on functions characteristic to a library. This leads to a difference in reload time if the function has been called in another VM after the attacker has flushed it from the cache. For the attack to work, the page containing the attacked function needs to be deduplicated between the attacker VM and the victim VM. While their attack has a similar aim as ours, it uses a different technique that requires manual analysis of the attacked libraries to find a suitable function. Automatically generating signatures for this type of attack would be hard as the targeted function needs to be loaded into the cache by the victim, which would typically be triggered by its execution. Thus, signatures would need to take into account how likely a function is to be executed. If an automatic signature generation mechanism targeted a function that is unique for an application, but rarely executed (e. g. handling of an uncommon error), this would be of little use for detecting an application. Furthermore, as their attack targets a single function in the library, it will be unable to distinguish versions in which the analysed function is identical. This implies that different functions may have to manually be found to distinguish different pairs of versions.

Gulmezoglu et al. [7] describe a cache-based side-channel attack to detect applications in co-resident virtual machines. They use machine learning to train a classifier on the cache usage patterns of applications. While their attack has the advantage of not requiring memory deduplication to be active, it is unclear whether it can be used to exactly identify the executed version of an application.

Xiao et al. [28] show that memory deduplication can be used to establish a covert channel for communication between two (collaborating) co-resident virtual machines. Furthermore, they show that memory deduplication can be used to monitor the integrity of a VM's kernel from the outside.

Suzaki et al. [24] first described a side-channel attack exploiting timing differences caused by the KSM deduplication mechanism used in KVM. They demonstrate that it is possible to detect applications running in a co-resident VM. However, they only analyse a single version of each tested application. The authors do not analyse whether it is possible to tell different versions of an application apart. They used the full binary as a signature, ignoring whether pages may also be present in other versions of the applications or even other parts of the system.

Owens and Wang [20] describe an approach to detect the operating system running inside another virtual machine hosted on the same VMWare ESXi host through a memory deduplication side-channel attack. They generate their signatures by setting up the targeted OS versions, capturing memory images of the running system and then filtering out the memory pages unique to that OS version. However, their approach was only tested on four different major releases of Windows and two of Ubuntu Linux. The impact of the frequently published patches for these operating systems on the accuracy of their detection mechanism was not evaluated.

In *summary*, most other side-channel attacks on memory deduplication concentrate on either revealing data in the memory of another VM or on establishing a covert communications channel between two VMs. While some approaches are concerned with detecting the presence of applications, they do not thoroughly study detecting specific versions. The work of Owens and Wang [20], who aim to detect versions of operating systems, is the closest to ours.

3. MEMORY SIDE-CHANNEL ATTACK

Our memory deduplication side-channel attack is based on timing measurements and can reveal whether pages characteristic for a software version have been deduplicated. In the following, we will describe the general approach an attacker would take to identify software versions running in other VMs. We will also describe how to find characteristic memory pages that can serve as a signature for a specific software version.

3.1 Attack Procedure

Memory deduplication opens up a timing side-channel that can reveal to an attacker that a memory page holding a certain content is present on the host, e. g. within another virtual machine. A deduplicated page needs to be copied before it can be modified. Thus, there is an additional delay in modifying such a page compared to modifying a non-deduplicated page. This delay can be used to detect the presence of applications [24] or other data [3] in other VMs on a host. Note, however, that it will not allow an attacker to find out in which particular other VM the application is running.

We define $pages(a_i^v)$ to return all pages of a_i^v excluding duplicate pages within the binary and pages containing only zero or one bits. Each virtual machine $m_j \in M$ is running a set of applications R_j. An attacker is interested in whether an application version is present in another VM, i. e. $a_i^v \in apps(M \setminus m_a)$. We define $pages(m_j)$ as the set of all memory pages of a VM m_j, i. e.

$$pages(m_j) \supseteq \bigcup_{a_i^v \in R_j} pages(a_i^v) \quad (1)$$

The attacker first needs to establish a deduplication and a non-deduplication baseline. To obtain the non-deduplication baseline, the attacker fills a number of memory pages equal to the number of pages they wish to test with random data, so that

$$pages(m_a) \cap \{\cup_{m \in M \setminus (m_a)} pages(m)\} = \emptyset \quad (2)$$

It can be assumed that randomly-generated pages do not get deduplicated as it is extremely unlikely that an identical copy is present on the host or in another VM. The attacker then measures the time it takes to overwrite these pages as a baseline for non-deduplicated pages.

The assumption is that we can identify a particular application version based on a subset of pages of that application version a_i^v that are unique across all different versions of it. We refer to this subset of pages as a signature $sig(a_i^v)$ (cf. Sect. 3.2 for details on signature derivation). The attacker writes the signature of an application they believe to be present in another VM to the memory of their VM m_a. If another VM m_v is executing a_i^v, this implies $\{\mathcal{M}_a \cap \mathcal{M}_v\} \supseteq sig(a_i^v)$, which means that these pages can be deduplicated. The attacker then needs to wait for deduplication to take place. Afterwards, the attacker modifies the pages that serve as signature and measures the time needed for overwriting exactly these pages.

This measurement can be compared to the baselines. A threshold for classifying measurements into deduplicated and non-deduplicated needs to be determined. If the measurement is significantly higher than the non-deduplicated baseline and close to the deduplicated baseline, the attacker can infer that the pages were most likely deduplicated, so that another copy of them as part of application version a_i^v is present in another VM. However, if the measurement is very close to the non-deduplicated baseline, the pages have not been duplicated and have been modified directly. This could mean that another copy of the pages was indeed not present on the host, but there is a small probability that a copy of the pages *is* present on the host, but has not been scanned by the deduplication mechanism yet, e. g. due to the deduplication mechanism being configured to only activate itself when the memory of the host is almost full. An easy and naive classification rule would be to use the mean of the two baselines as a threshold, which works well enough if multiple pages are being measured at once (cf. Sect. 4.7).

To use this side-channel to detect the presence of application version a_i^v on a host, an attacker would act as follows:

1. Establish *baselines* by writing $length(sig(a_i^v))$ pages containing random information to the memory of m_a and measuring the time it takes to overwrite this random information (non-deduplicated baseline). Furthermore, write two copies of randomly generated pages into the memory of m_a and overwrite one of the copies (deduplicated baseline). The baselines should be based on multiple measurements.
2. Determine the *classification threshold* based on the baselines obtained in the previous step.
3. *Write* $sig(a_i^v)$ into the memory m_a.
4. *Wait for deduplication* to happen. The correct waiting time depends on the configuration of the host's deduplication mechanism.
5. *Overwrite* the signature, while *measuring the time* this operation takes to complete.
6. *Repeat* steps 2 to 4 until a sufficient number of measurements has been taken.
7. Calculate the mean of the measurements taken and *compare* it to the classification threshold.

If the attacker is not interested in particular pages, but in identifying pages that are unique to an application version (aka signature), the full set should be written at once. The timing differences observed between overwriting deduplicated and non-deduplicated pages will be more pronounced if multiple pages are being checked at the same time. Thus, an attacker can identify a program running on another VM with fewer measurements. This implies that the signatures for a software version should consist of as many pages unique to this version as possible.

As it is necessary to repeat the measurements several times, and each measurement comes with a delay, the attack takes a relatively long time. However, if the signatures to be checked are disjunct, i. e. they do not contain any pages that are also present in other signatures, multiple signatures can be checked in parallel. To avoid measurements influencing each other, the overwriting operations should not overlap. It is not a problem to perform an overwriting measurement on one of the signatures while other signatures are in the waiting phase, though.

The configuration of the deduplication mechanism will have an impact on the effectiveness of any memory deduplication side-channel attacks: If the interval between scans is long, potential attackers would be slowed down at the cost of decreased memory savings. If, on the other hand, the activation threshold is set relatively high, attacks will not be possible at all as long as the host's memory load remains beneath the threshold.

3.2 Signatures for the Timing Side-Channel

To be able to reliably detect a software version, we need to build signatures for each version. A signature should contain only pages unique to the respective version, as including pages that can also be found in other versions may lead to false classifications. This also holds true for pages of completely different applications. Due to the size of a page, it is however very unlikely that an identical page can also be found in a different application.

To derive a signature for a version of an application binary, we start with all its pages and then remove the following types of pages:

1. **Internal duplicates** because a duplicate page within the signature itself would be sufficient to trigger deduplication of these pages.
2. **Pages containing only zeroes or ones**, as another copy of these is very likely to be present on the host even if the surveyed application is not being executed in another VM at all.
3. **Pages present in other versions**, as these are unsuitable for distinguishing the version.

Any of these pages can be deduplicated without the probed version being present in another VM. Therefore, they must be removed to avoid false positives. In summary, the signature for an application version a_i^v is generated as follows:

$$sig(a_i^v) = pages(a_i^v) - \bigcup_{a \in \{A_i \setminus a_i^v\}} a \qquad (3)$$

Our approach for deriving signatures is similar to the one for detecting operating systems by Owens and Wang [20]. In their approach, they capture memory images of different OSs while executing them. Then, they derive signatures from these that contain pages unique among their OS dataset. Similar to their work, we aim to find memory pages that are unique to an application instead of an OS version to use them as signature. However, we consider the pages of the application binary only and can ignore all pages containing application data pertaining to the runtime state. Thus, any pages that do not contain executable code, such as data pages, are ignored. These pages may differ between two instances of an identical binary, e. g. due to a different runtime state, or may be identical for two different versions of a binary, e. g. due to a similar runtime state saved in a memory structure that has not been changed between versions. Thus, they are not well-suited for detecting the application version being executed.

Focusing on only the pages of the application binary renders our technique much more efficient compared to the work of Owens and Wang. These binary pages are the only pages that can safely be assumed to reside in memory on all systems executing it. Moreover, binaries need not be executed to generate signatures in the form of unique memory pages. Thus, in the following, we can use these signatures to specifically test via our side-channel attack described in Section 3.1 if there is an application running that matches the signature, i. e. contains the signature pages.

In the next section, we summarise our findings in evaluating the proposed side-channel attack.

4. EVALUATION

In this section, we first present the tools and datasets that we have created for our experiments. Also, we describe experiments that indicate that a timing side-channel exists that can reveal whether pages of an application are present within another VM. We then present experiments that indicate that versions of the same application are different enough from each other to detect them with this method. Furthermore, our experiments indicate that releases of the same upstream version from different distributions can easily be distinguished, too. We also analyse the impact of changing the page size on the deduplication of pages containing executable code. Finally, we present an analysis on the complexity of our attack before discussing the limitations of our approach.

4.1 Signature Generation and Measurements

To derive signatures and enable the experiments described in the reminder of this paper, tools have been developed that allow the automatic comparison of a large number of versions of a binary[1].

To analyse a software, we first need to obtain its different versions. Then, the main binary has to be extracted from the downloaded packages. To this end, shell scripts have been developed that can extract the binaries from RPM and deb packages. The scripts can easily be adapted to each application and distribution and will then process a large range of versions, as the location and name of a binary within the package rarely changes. The scripts will place the extracted binaries into a directory structure that can be processed by our analysis tool.

The main analysis tool is written in Java and can process ELF binaries, but can easily be extended to other executable formats as well. It supports two modes of analysis: First, all versions of a binary can be **compared** with each other to determine the number of matching pages between each pair of versions. Results will be output as a csv file. Second, the software can output the **signature** for each version, which will contain all pages unique to that version (cf. Sect. 3.2). Statistics about the signatures will also be created and saved in a csv file. These include the signature sizes, the number of internal duplicates, the number of pages that can also be found in other versions, and the number of pages containing

[1]The code of our tools is available at https://github.com/jl3/memdedup-app-detection.

only zeroes or ones. The page size used by the tool can be configured freely.

Furthermore, two C programs have been developed to perform timing measurements. The first one loads signature pages into memory and measures the time it takes to overwrite them after a specified amount of time has passed. Measurements will be output to the console and logged into a file. This tool has been improved in comparison with the version used in our former work [14]. The old version was susceptible to noise from disk I/O, which has been removed by pre-caching the data that is to be loaded into memory when overwriting. The second one loads signature pages into memory aligned to page boundaries to enable experiments that do not use a running executable.

4.2 Datasets

We created three datasets for our experiments: The first one contains all Apache web server releases for Debian on the *x86_64* platform and the second one all SSH daemon (*sshd*) releases. The third dataset contains releases of *sshd 7.9p1* for different distributions. Our datasets consist of the following application versions:

- The **Apache-Debian-x86_64** dataset consists of all 160 Debian releases of *Apache* available for the x86-64 platform and includes versions from 2.2.11-5 to 2.4.37-1.
- The **sshd-Debian-x86_64** dataset consists of all 211 Debian releases of *sshd* available for the x86-64 platform and includes versions from 4.2p1-7 to 7.9p1-4.
- The **sshd-crossdist** dataset consists of 10 package versions of *sshd 7.9p1* from Arch Linux, Debian, Fedora, Mageia and Ubuntu. Multiple revisions are included for Debian (3) and Fedora (4).

Our datasets contain only the main executable of each application (*httpd* for Apache, *sshd* for the SSH daemon). For the surveyed applications, these are typically the only executables that will be running as a daemon at all times. While both applications include additional executables (e. g. *ssh-keygen* for generating SSH keypairs), these would normally not be running long enough for the described attack to be possible. In case of packages containing multiple executables to be run constantly (e. g. as a daemon), all these executables should be included when generating signatures.

The packages for the Debian-based datasets were obtained from the snapshot archive[2], which provides historic package versions. A similar repository of old package versions is available for Fedora[3], which retains old versions of binary packages and keeps them publicly available.

Unfortunately, most distributions, among them openSUSE, OpenMandriva, Ubuntu and Arch, provide only very recent versions of the binary packages. This makes it hard to create a dataset that can be applied to other distributions as well. For the cross-distribution dataset, due to the lack of older versions of binary packages for many distributions, we had to use the recent upstream version 7.9p1 of sshd, which was available for download for a variety of distributions at the time of dataset creation.

[2]http://snapshot.debian.org

[3]https://koji.fedoraproject.org/koji/

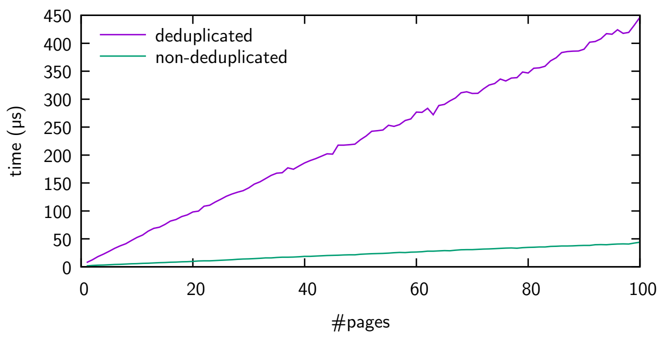


Figure 2: Average write times (in microseconds) to n deduplicated/non-deduplicated pages

4.3 Feasibility of the Side-Channel

In the following, we will show that a timing side-channel in memory deduplication exists that can be used to reveal the presence of memory pages in another VM or on the host.

For the experiments described in this section, two virtual machines m_a and m_v are used. The host is an Intel Core i7-4790 with 16 GiB RAM running KVM and KSM on Fedora 26. First, a number of pages is loaded into the memory of m_v. Then, the same pages are loaded into m_a. After that, we wait for deduplication to take place and overwrite the pages in the memory of m_a, measuring the time this takes.

Figure 2 shows the write times to sets of non-deduplicated and deduplicated pages depending on the number of pages in the respective application. All results in the figure are averaged over 1 000 measurements each. In the non-deduplication case, the pages on m_a and m_v are of identical size, but have different contents, so that no deduplication can take place. In the deduplication case, the pages on m_a and m_v are identical, so they can be deduplicated.

Write times to deduplicated pages are higher than to non-deduplicated pages. For both types of pages, write time increases linearly with the number of pages overwritten. The gap in write times between non-deduplicated and fully deduplicated sets of pages increases when writing to a larger number of pages. This implies that when we measure the time to overwrite a larger number of pages at once, it will be easier to determine whether these pages have been deduplicated previously.

Figure 3a shows a histogram of 1 000 write times each for a single deduplicated or non-deduplicated page without background load on the system. As expected, the write times to non-deduplicated pages are typically lower than those to deduplicated pages. However, when overwriting a single page, some of the slower measurements for non-deduplicated pages fall into the same range as some of the faster measurements for deduplicated pages. This implies that performing a single measurement only will not be sufficient to reliably distinguish the two cases and thus determine whether another copy of the page is present on the host. Figure 3c shows the write times with background load, generated by running six instances of memtester 4.3.0 on the host while performing the measurements. While write times to non-deduplicated pages are still lower on average than those to deduplicated pages, measurements for both are spread out over a significantly larger range of times and overlap more. This makes the two cases harder to tell apart.

Figure 3b shows a histogram of 1 000 write times each for 100 deduplicated or non-deduplicated pages without background load on the system. As with the single pages, overwriting 100 deduplicated pages takes longer than overwriting 100 non-deduplicated pages. However, the measurements do not overlap. This implies that we could reliably distinguish whether a 100-page signature is present on the host based on a single measurement only in our test setup. Figure 3d shows shows the write times for 100 pages with background load. As for single pages, these are spread out over a wider range of times. However, they do not overlap and could still be distinguished reliably based on a single measurement.

4.4 Cross-version Similarities

We now present our analysis on cross-version similarities in the *Apache-Debian-x86_64* and *sshd-Debian-x86_64* datasets.

For that, we directly compare each version to every other version available. This direct comparison shows how many pages are identical among two specific versions. The more pages are identical, the harder it will be for an attacker to distinguish these versions from each other using our side-channel attack. However, a larger number of identical pages also implies that deduplication can save more memory.

Furthermore, we determine the number of pages that can be used as signature for each application version in our datasets. We also analyse how many pages have not been used for deriving signatures (cf. Section 3.2).

Figure 4a shows the number of matching pages between all versions for the *Apache-Debian-x86_64* dataset. Figure 5a shows the number of matching pages between all versions in the *sshd-Debian-x86_64* dataset. Each row and column corresponds to one version, namely from old versions on the top left to new versions on the bottom right. The colour at the intersection of the row corresponding to version v_r and the column corresponding to version v_c represents the number of pages in the binary of v_r that are also present in the binary of v_c. The bright diagonal line running from the top left to the bottom right represents the comparison of a version with itself ($v_r = v_c$) and shows the size of the respective version's binary in pages. It can be seen that newer versions of the binaries are larger than older ones.

The results indicate that some versions form clusters, whose binaries are relatively similar to each other. For the *Apache* dataset, these clusters correspond to multiple Debian revisions of an upstream version. An example of a range of versions whose binaries are very similar to each other is 2.2.16-1 to 2.2.16-6. However, the backport revisions corresponding to this upstream version are completely different and share only a single page with the non-backport versions. Further examples of similar binary versions are 2.2.22-6 to 2.2.22-13+deb7u2 and 2.4.10-1 to 2.4.10-9. While all versions in these clusters share more than 20 pages with each other, both clusters contain inner clusters of versions even more similar to each other. For example, while the versions in the former cluster all share at least 28 pages with each other, versions 2.2.22-7 to 2.2.22-10 share almost all their pages with each other. Figure 4b gives a detailed view of this cluster and shows how many of the pages in version 2.2.22-9 can also be found in each neighbouring version.

The clusters in the *sshd* dataset are similar in size and are also restricted to versions that have been released close to each other. Interestingly, while most of these clusters also contain only different Debian revisions of the same upstream software version, the *sshd* dataset – unlike the *Apache* dataset – contains a few clusters stretching across different upstream versions. While most of these span versions corresponding to directly adjacent upstream releases (e. g. 5.4p1 and 5.5p1), there are also clusters of slightly more distant versions. One notable example of this is the cluster comprising versions 7.2p2-6 to 7.2p2-8 as well as 7.4p1-1 to 7.4p1-5, which are more similar to each other in terms of memory pages than to the versions between. A detailed view of this cluster is shown in Figure 5b.

The results indicate that almost no similarities exist between binaries of packages across different upstream versions. The results also indicate that memory savings by means of deduplicating binaries of *Apache* on Debian x86-64 can only be achieved if multiple instances of the same upstream version and ideally the same or a very close Debian revision are being executed on the host. For *sshd*, limited sharing potential exists between releases of some close upstream versions.

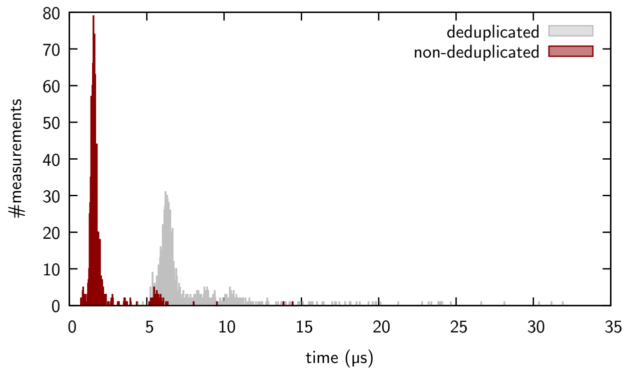
Figure 6a shows how many memory pages can be used in a signature for a binary of the *Apache-Debian-x86_64* dataset. The values are calculated on the assumption that signatures shall be used to identify not only the upstream version, but also the exact Debian patch level of the binary. The figure also shows how many pages of the binary are contained more than once within the binary or contain only zeroes or ones. It is also shown how many of the remaining pages are also contained in other versions of the binary. The remaining pages can be used as a signature. Figure 6b shows the results for the sshd-Debian-x86_64 dataset.

The results show that the size of the signature is large for many of the versions surveyed as they contain many unique pages. These versions can be precisely identified using our attack.

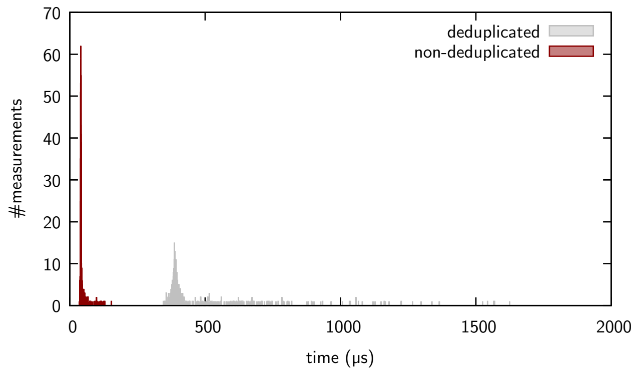
However, the size of the signature is small for many other versions. Due to the timing difference observed in a memory deduplication attack being far less pronounced for shorter signatures, it will be hard to identify these versions to the precision of a specific Debian revision. Signature size can be increased by grouping some of the affected versions with neighbouring versions and by creating a signature that describes the group. This reduces the precision of the version identification, but will make the memory deduplication attack easier to perform.

4.5 Inter-distribution Similarities

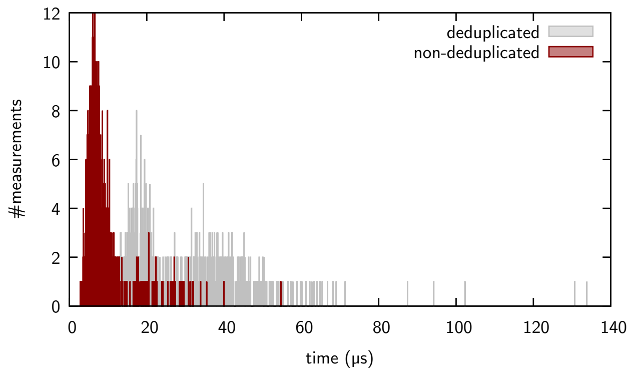
We now analyse whether signatures derived from the binaries of one Linux distribution can also be used to detect the version of the same software on another distribution. To



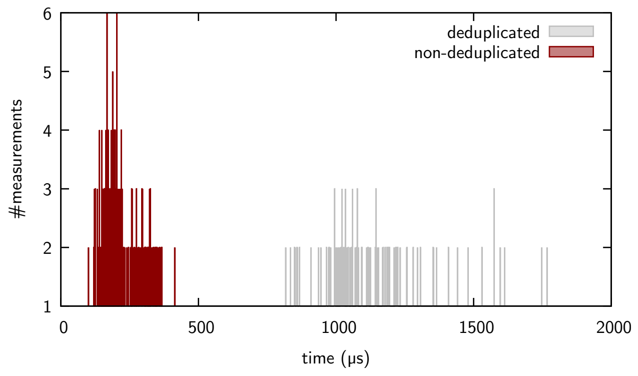
(a) single page, no background load



(b) 100 pages, no background load



(c) single page, with background load



(d) 100 pages, with background load

Figure 3: Histogram of write times (in microseconds) to deduplicated and non-deduplicated pages

this end, we compared binaries of the same software version from packages of several distributions in the same way as described in Sect. 4.4. Our experiments are based on the *sshd-crossdist* dataset.

Figure 7 shows the number of duplicate pages between the different binaries. It can be seen that the binaries distributed by Debian are very similar to each other. Furthermore, the Fedora releases are relatively similar to each other. We found release 7.9p1-1 for Fedora 29 to be more similar to 7.9p1-1 for Fedora 30 than to the 7.9p1-2 releases for both Fedora 29 and 30. The Debian and Ubuntu releases share five to seven pages with each other. All other cross-distribution pairs of releases exhibit no similarities.

4.6 Influence of Page Size

In the following, we will present an analysis on the influence of changing the page size on the effectiveness of our attack and the memory saving potential of deduplicating executable code. To analyse whether decreasing the page size from the standard of 4 096 bytes increases the proportion of binaries that can be deduplicated, we analyse the number of matching pages across versions of the *Apache-Debian-x86-64* dataset for non-standard page sizes in the same way as described in Sect. 4.4.

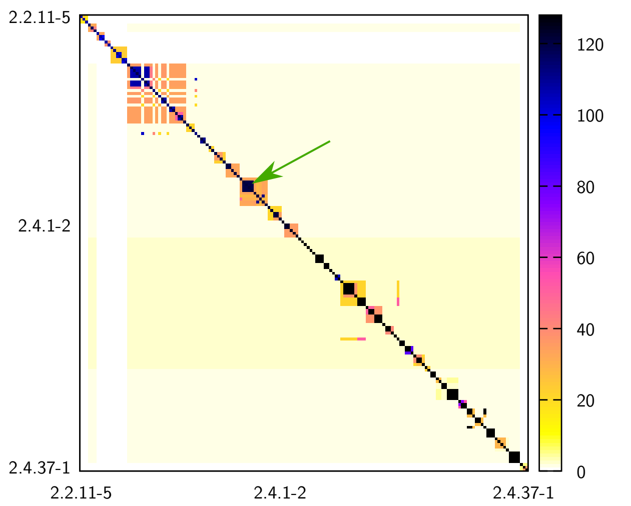
The results of the experiment are shown in Figure 8. We divide all pairs of versions into two categories: high-sharing pairs ($\geq 5\%$ of pages shareable) and low-sharing pairs ($< 5\%$ of pages shareable). The results indicate that reducing the page size increases the percentage of shareable pages for pairs of versions that were already similar at standard page size. However, sharing opportunities remain almost unchanged for lower page sizes.

4.7 Attack Complexity

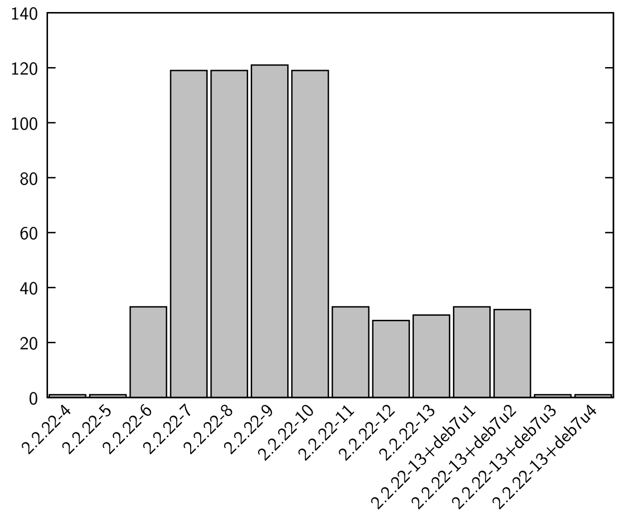
We will now analyse how long it takes to perform our attack. The duration for a successful run of our side-channel attack depends on the configuration of the deduplication mechanism and on the desired accuracy of the results.

How long it takes to perform a single measurement is defined by the time an attacker has to wait for deduplication to take place, which depends on how long the deduplication mechanism requires to scan the complete memory. In its standard configuration on Fedora 26 and RHEL 7.4, the ksmtuned daemon, which automatically configures KSM (cf. Sect. 2.1) according to memory usage, scans at least 1/65536 of the physical memory, i. e. it will take at most 655.36 seconds until all of a machine's memory has been scanned. For the remainder of this section, we will assume this as the time an attacker has to wait for deduplication to take place.

In the following, we want to establish which accuracy can be achieved depending on the signature size and the number of measurements performed (i. e. the time needed for the attack). As in Sect. 4.3, our test setup includes two VMs m_a and m_v. We first created a training dataset that



(a) Number of matching pages for all pairs of versions. The colour at the intersection of the line of version v_r with the column of version v_c indicates how many pages of v_r can also be found in v_c.



(b) Detail of selected cluster from (a) – Number of pages in Apache 2.2.22-9 (marked by the arrow in (a)) that can also be found in neighbouring versions.

Figure 4: Cross-version similarities – Apache-Debian-x86_64 dataset

was used to determine the classification threshold. As in a real attack (cf. step 1 in the attack procedure in Sect. 3), we loaded n pages into the memory of m_a. As the concrete content of the pages is irrelevant for this experiment, pages were generated randomly. Then, we loaded the same n pages into memory again as well as n pages filled with different data. After waiting for the deduplication to occur, we measure the time it takes to overwrite each set of pages. This process is performed 1 000 times, so that we have 1 000 training measurements for both the deduplicated and the non-deduplicated case. Note that this number of training measurements is not unrealistic in an actual attack, as measurements can be taken in parallel if different sets of data are used. The approach for creating our test dataset is identical except that the pages that are to be deduplicated are loaded into the memory of m_v (and later into m_a's memory only once).

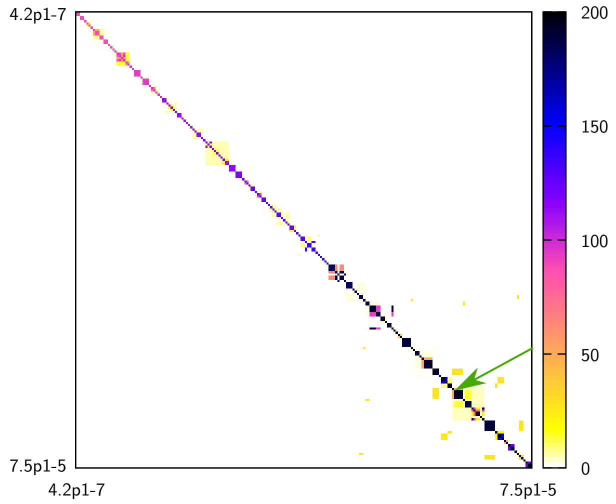
We perform our experiments in two scenarios: In the first scenario, only m_a and m_v are active on the host system. The VMs run only the OS and our analysis tools. The host was only running the base system and the two VMs. No further VMs were active. This ensures that there is relatively little load on the memory of the host that is not attributed to the measurements themselves.

In the second scenario, we simulate background memory activity on the host. As in the first scenario, the attack (m_a) and victim (m_v) VMs were active and no further VMs were active. However, in addition to the base system and the two VMs, the host was concurrently executing six instances of memtester 4.3.0. Each instance was configured to use 1 GiB of memory and run in an infinite loop. This ensures that there were constantly read and write accesses being made to the physical memory of the host.

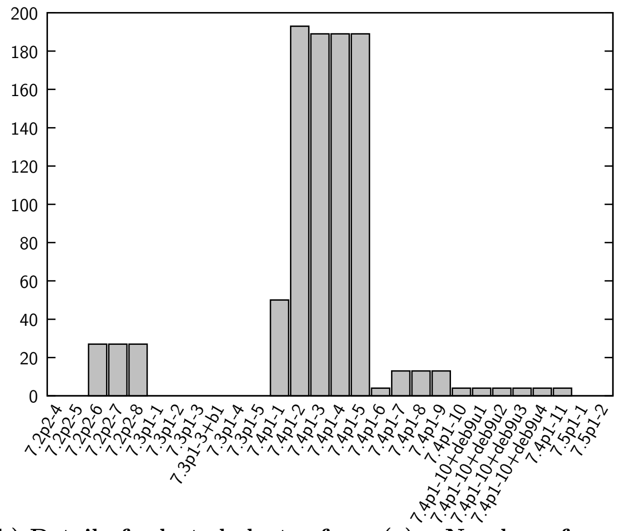
To probe a signature, multiple measurements should be performed to increase the accuracy of the results. The time this takes for one signature depends on the desired accuracy of the results. Figure 9a shows the impact of the number of measurements performed and the size of the signature on the accuracy of our version detection mechanism when there is no background load on the system. We calculated the mean of the training measurements for each test case to act as a baseline for classification (cf. Sect. 3.1). For different values of m, we then took 10 000 000 random samples of m measurements each from all our test cases and checked whether the mean of the sample was classified correctly. The accuracy value shown is aggregated over both the deduplicated and the non-deduplicated test case. As this classification rule is relatively simple, the accuracy values can be considered a lower bound of what is possible.

It can be seen that measuring multiple pages at once increases the accuracy. Thus, measurements should be performed based on signatures that contain all pages unique among the different versions of that application. Also, accuracy increases with the number of measurements performed. However, even a single deduplicated page in a set of non-deduplicated page will increase the write time and can lead to false classifications. Therefore, signatures should be as large as possible, but not contain any pages that are also present in other versions. In the best case, every version has completely different pages, so that all of them can be used as a signature.

Without background load on the host, relatively few measurements are required to achieve a high accuracy even for small signatures: For signatures of a single page, six measurements were needed for an accuracy of $\geq$ 99.9%. For larger signatures, fewer measurements were needed to achieve a similar level of accuracy, e. g. three measurements for sig-



(a) Number of matching pages for all pairs of versions. The colour at the intersection of the line of version v_r with the column of version v_c indicates how many pages of v_r can also be found in v_c.



(b) Detail of selected cluster from (a) – Number of pages in sshd 7.4p1-2 (marked by the arrow in (a)) that can also be found in neighbouring versions

Figure 5: Cross-version similarities – sshd-Debian-x86_64 dataset

natures of two pages and two measurements for signatures of five pages or more.

Under load, the number of measurements required to achieve a certain level of accuracy increases, as shown in Figure 9b. In our experiments, nine measurements were required to achieve an accuracy of $\geq 99.9\%$ for single-page signatures. For two-page signatures, we were able to achieve this level of accuracy with six measurements, while three measurements were sufficient for signatures of five pages.

For example, if six measurements are desired, it takes about 66 minutes to probe the signature pages. Increasing the number of measurements increases the time linearly. To probe all signatures of the *sshd-Debian-x86_64 dataset* consecutively takes about eight days if six measurements are performed per signature. However, as our signature pages are disjunct in between different application versions, they can actually be probed in parallel if enough memory is available in the attack VM v_a. This reduces the time to about 66 minutes, the same time it takes to probe a single signature. For that, all signatures are loaded into the memory of v_a at once. The attacker must then wait for deduplication to occur. Afterwards, the timing measurements can be performed consecutively. Each of them takes a fraction of a second. This process can then be repeated multiple times to achieve the desired number of measurements per signature.

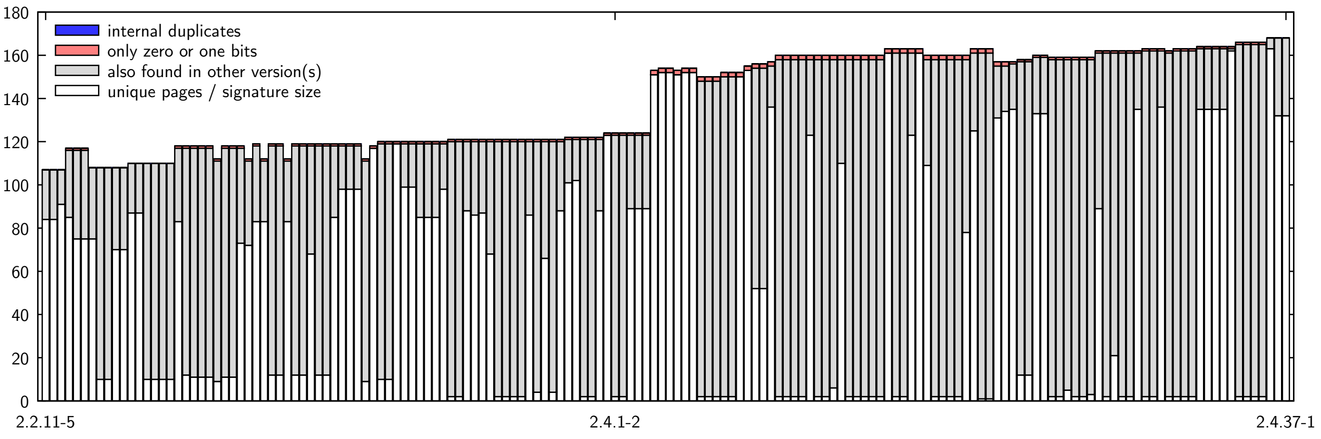
To further reduce the number of measurements required, similar versions can be grouped [15]. This results in larger signatures and eliminates all small signatures for our datasets. While this means that an attacker can no longer identify the exact version and distribution patch level of an application, we found that for our datasets, almost all groups contain only different distribution patch level releases belonging to the same upstream version released by the original developers of the software. The only exception from this was a group in the sshd-Debian-x86_64 dataset, which contained three versions from two neighbouring upstream versions. Thus, the versions in a group are likely to contain similar security vulnerabilities, which means that for most attackers, the version can still be identified precisely enough if groups are formed.

4.8 Limitations

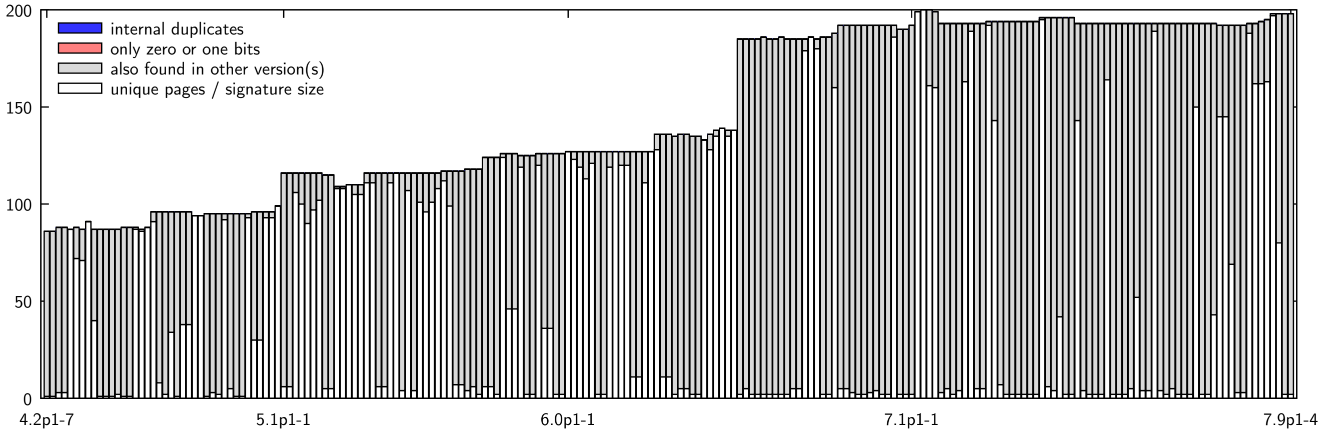
In the following, we discuss some limitations to the presented side-channel attack.

Attacker does not know IP of co-resident VMs.

The attack presented in this paper allows an attacker to find out whether a specific version of an application is running in another VM that is co-resident on the host. However, it does not allow an attacker to find out in which specific VM the application is being executed. It also does not provide any information on how to contact the VM that runs the identified application, which is necessary to exploit a potential security vulnerability. If an attacker is interested in attacking a specific online service, they may try to obtain a VM that is co-resident with a VM hosting it. Varadarajan et al. [26] have shown that this is realistic in public cloud environments. Depending on a cloud service provider's infrastructure, IP addresses may also be correlated with the placement and type of VMs [23], which would allow an attacker to increase its chances of obtaining a co-resident VM by choosing the deployment zone and instance type accordingly. This may also help an attacker that does not have a specific target in mind in narrowing down potential IP addresses of vulnerable co-resident VMs. For that, the attacker can randomly spawn attack VMs to find vulnerable VMs.



(a) Apache-Debian-x86_64 dataset



(b) sshd-Debian-x86_64 dataset

Figure 6: Number of unique pages that can be used as a signature (white) and reasons for the unsuitability of other pages

Memory deduplication must be active.

Our attack assumes that deduplication is activated on the host. Nowadays, many of the larger public cloud service providers such as Google [10] have turned off memory deduplication in fear of side-channel attacks. However, the technique can offer large memory savings [4, 27], which makes it attractive to server operators. This is especially true in environments where users of VMs are believed to be at least somewhat trustworthy, e. g. in private clouds.

Application versions might be indistinguishable.

Another assumption is that application versions are sufficiently different from each other. For the datasets we surveyed, this is the case. When generating signatures for individual application versions in our dataset, all signatures contain at least one page. This implies that all versions can be differentiated. Theoretically, however, it is possible for the signature generation to fail. This could be caused by two identical binaries in two different versions of a package, e. g. if only a default configuration file was changed between the package versions. It can also be caused by a version containing only pages that are also present in multiple different other versions, e. g. $v_1 = \{a, b, c\}, v_2 = \{a, d, e\}, v_3 = \{b, c, f\}$ will lead to the signature generation for version v_1 failing. Such situations can be resolved by creating group signatures [15] for the affected versions.

Our experiments were conducted on Linux only.

Experiments were only conducted for the Linux OS, which is dominant in cloud environments. However, we believe that our results are applicable to other operating systems as well. Most Unix-based OSs use the ELF file format for their executables and will employ a very similar loading mechanism to Linux. Adapting the attack to another OS requires taking into account how executables are structured and loaded on that system. For example, Windows uses the Portable Executable (PE) format [17] for its executables. For PE files, sections are loaded instead of segments. Sections are described in a similar header table as in ELF files, which can be used as a base for analysing a PE executable.

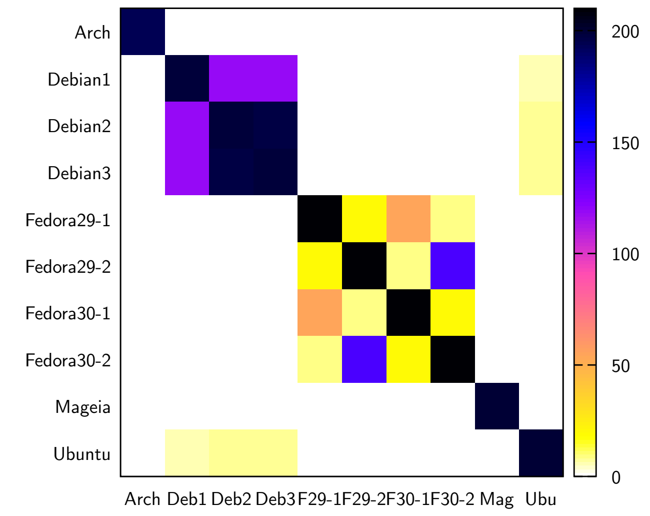


Figure 7: Cross-distribution similarities. Number of matching pages for all pairs of versions in the sshd-crossdist dataset. The colour at the intersection of the line of version v_r with the column of v_c indicates how many pages of v_r can also be found in v_c.

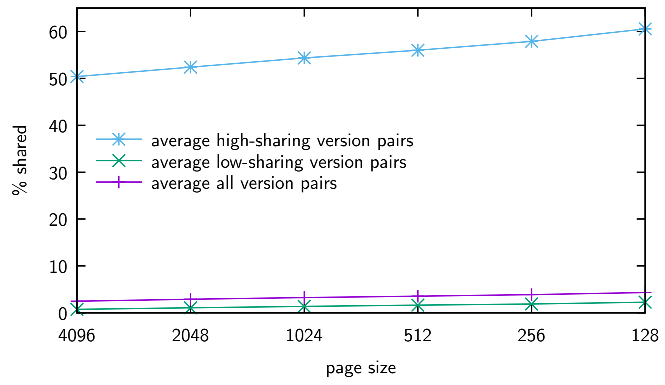


Figure 8: Influence of changing the page size on the proportion of shared pages. High-sharing version pairs are pairs of versions with $\geq 5\%$ shared pages at standard page size.

5. COUNTERMEASURES

In this section we discuss potential countermeasures against the presented attack. On the one hand, some of these aim at removing the side-channel altogether, but will also remove the memory savings offered by deduplication. On the other hand, some countermeasures aim at reducing the effectiveness of attacks without fully eliminating the side-channel.

Deactivating Memory Deduplication.

The easiest way of avoiding side-channel attacks by memory deduplication is to turn this feature off. However, this comes at the cost of eliminating all memory savings by deduplicating memory pages. Alternatively, this feature gets disabled only for pages belonging to executable binaries. According to our results, this will only require significantly more physical memory on systems hosting a large number of VMs that all run very similar software. However, modifications to the hypervisor and guest OS would be necessary to make them aware whether a page actually belongs to a binary.

Slow down writes to non-deduplicated pages.

Another approach that the operator of the host can take would be to slow down writes to non-deduplicated memory pages. If write operations are slowed down to the level of deduplicated pages, the side-channel is eliminated. However, this requires significant modifications to be made to the host operating system as write operations to non-deduplicated pages will normally not pass through the deduplication mechanism. This should not affect the performance of read-heavy workload, but it is unclear how large the adverse effect on performance for more write-heavy workloads would be.

Obfuscate Memory.

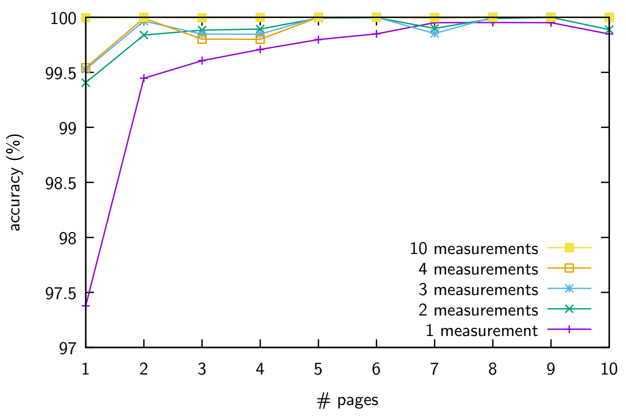
If a user who merely rents a VM on a host whose configuration they cannot control wants to prevent memory deduplication side-channel attacks on their VM, a possible solution would be to obfuscate the VM's memory. This could be achieved by deploying an Address Space Layout Randomization technique that – unlike the standard Linux implementation – does not only shuffle pages in memory, but randomises the memory contents on the sub-page level. This would ensure that all bits of the start address of a segment of an ELF segment are random. Therefore, the alignment of the segment's contents to page boundaries would be randomised, resulting in 4 096 possible alignments. As two pages will only be deduplicated if they match entirely, a different alignment prevents deduplication. An attacker could thus not simply use signatures as described in this paper. The attack would need to take all possible alignments of an application's pages into account, i. e. attackers would need to probe 4 096 times as many signatures. While these signatures can still be checked in parallel, this requires a lot more memory. If not enough memory is available, some signatures will have to be checked sequentially, increasing the time for the attack.

Modify Binaries.

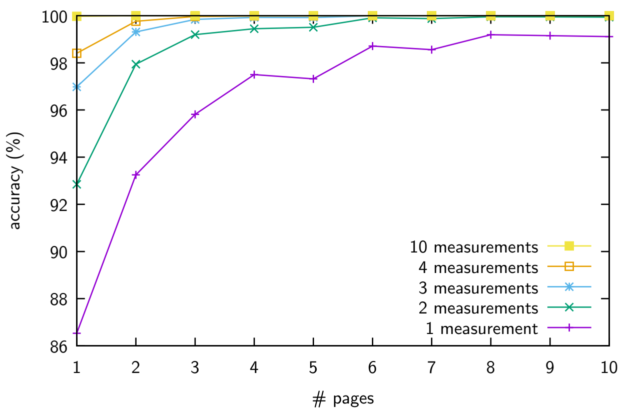
Another solution would be to slightly modify all executed binaries. This can be achieved without recompiling by inserting randomly-placed NOP opcodes into an application's binary. Alternatively, it should also be sufficient to compile the programs manually with some less commonly used compile options considering that the binaries released by different distributions are based on the same upstream version are highly different in their memory pages.

Encryption.

The user of a VM can also encrypt its memory. However, all of these techniques will make it very hard or impossible for the hypervisor to deduplicate memory pages, thus preventing memory savings.



(a) **no background load on the host**



(b) **with background load on the host**

Figure 9: Influence of signature size and number of measurements on accuracy of the version detection

Decoy Signatures.

Instead of preventing the side-channel attack outright, it would also be possible to deceive attackers by placing pages of binaries that are not actually running on any VM or the host into memory, e. g. pages that are contained in our signatures. This can be done by either the operator of the host or anyone controlling a VM on the host. While an attacker would still be able to detect the presence of software versions that are being executed, this would come with a certain number of false positives. An effective defence by such an approach would require much more memory to load a signatures for many versions of many applications. It may, however, be suitable to prevent that an attacker gets to know the exact version of a specific sensitive application from memory deduplication attacks. It is not a replacement for regularly updating the system, though: In case of a lack of updates of both the application and the decoy signatures, an attacker would still be able to establish an upper bound on the application version.

6. CONCLUSION

We have introduced a novel side-channel attack that is based on memory deduplication and that can detect software versions on co-resident VMs. We can even identify versions to the precision of a specific distribution patch level of an upstream release. This provides valuable knowledge to an attacker, who can perform attacks targeting specific vulnerabilities in the software versions that were detected by the side-channel attack. No significant similarities were found between binaries from different distributions that were based on the same upstream release. This means that releases of the same upstream software version from different distributions can be easily distinguished. It also implies that the potential for memory savings by deduplicating executable code is limited for computers hosting VMs with homogeneous software configurations. Changes to the page size increase deduplication potential only for pairs of versions that already share a significant number of pages at standard page size, i. e. only for some pairs of releases of the same or neighbouring upstream versions by the same OS and distributions.

Our results indicate that we can detect the presence of a signature of five pages or more in another VM or on the host with a reasonable amount of three measurements with an accuracy of $\geq 99.9\%$ even if there is significant load on the memory of the host. However, an actual attack takes time and for three measurements it will take about 32 minutes.

The side-channel can be prevented by disabling memory deduplication across multiple VMs – either completely or by modifying the deduplication mechanism to only not deduplicate executable code.

Possible future work includes studying a broader range of applications and extending the study to other operating systems, such as Windows. Furthermore, more advanced mitigation strategies should be developed to enable memory deduplication to take place without leaking information to other VMs.

7. REFERENCES

[1] A. Arcangeli, I. Eidus, and C. Wright. Increasing memory density by using KSM. In *Linux Symposium, Montreal, Canada*, pages 19–28, 2009.

[2] M. Bellare, S. Keelveedhi, and T. Ristenpart. Message-locked encryption and secure deduplication. In *EUROCRYPT*, pages 296–312, 2013.

[3] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *IEEE Symposium on Security and Privacy*, pages 987–1004, 2016.

[4] C. Chang, J. Wu, and P. Liu. An empirical study on memory sharing of virtual machines for server consolidation. In *IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA 2011*, pages 244–249, 2011.

[5] K. Fraser, S. H, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the xen virtual machine monitor. In *Operating System and Architectural Support for the On-demand IT Infrastructure (OASIS)*, 2004.

[6] D. Gruss, D. Bidner, and S. Mangard. Practical memory deduplication attacks in sandboxed javascript. In *ESORICS, Part I*, pages 108–122, 2015.

[7] B. Gulmezoglu, T. Eisenbarth, and B. Sunar. Cache-based application detection in the cloud using machine learning. In *ACM Asia Conference on Computer and Communications Security, AsiaCCS 2017*, pages 288–300, 2017.

[8] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: harnessing memory redundancy in virtual machines. *Commun. ACM*, 53(10):85–93, 2010.

[9] D. Harnik, B. Pinkas, and A. Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security & Privacy*, 8(6):40–47, 2010.

[10] A. Honig and N. Porter. 7 ways we harden our KVM hypervisor at Google Cloud: security in plaintext. https://cloud.google.com/blog/products/gcp/7-ways-we-harden-our-kvm-hypervisor-at-google-cloud-security-in-plaintext?m=1, January 2017. Accessed: November 28, 2018.

[11] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. Know thy neighbor: Crypto library detection in cloud. *PoPETs*, 2015(1):25–40, 2015.

[12] K. Jin and E. L. Miller. The effectiveness of deduplication on virtual machine disk images. In *The Israeli Experimental Systems Conference (SYSTOR)*, page 7, 2009.

[13] J. Lindemann. Towards abuse detection and prevention in IaaS cloud computing. In *International Conference on Availability, Reliability and Security (ARES)*, pages 211–217, 2015.

[14] J. Lindemann and M. Fischer. A Memory-Deduplication Side-Channel Attack to Detect Applications in Co-Resident Virtual Machines. In *ACM Symposium on Applied Computing (SAC) 2018*, pages 183–192, 2018.

[15] J. Lindemann and M. Fischer. Efficient identification of applications in co-resident vms via a memory side-channel. In *ICT Systems Security and Privacy Protection - IFIP TC 11 International Conference (SEC) 2018*, pages 245–259, 2018.

[16] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. *ACM Trans. Storage (TOS)*, 7(4):14:1–14:20, 2012.

[17] Microsoft Windows Dev Center. PE Format. https://docs.microsoft.com/en-gb/windows/desktop/Debug/pe-format, November 2018. Accessed: November 30, 2018.

[18] G. Milos, D. G. Murray, S. Hand, and M. A. Fetterman. Satori: Enlightened page sharing. In *USENIX Annual Technical Conference*, 2009.

[19] M. Mulazzani, S. Schrittwieser, M. Leithner, M. Huber, and E. R. Weippl. Dark clouds on the horizon: Using cloud storage as attack vector and online slack space. In *USENIX Security*, 2011.

[20] R. Owens and W. Wang. Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. In *IEEE International Performance Computing and Communications Conference (IPCCC)*, pages 1–8, 2011.

[21] P. Puzio, R. Molva, M. Önen, and S. Loureiro. ClouDedup: Secure deduplication with encrypted data for cloud storage. In *Cloud Computing Technology and Science (CloudCom)*, pages 363–370, 2013.

[22] Red Hat. Kernel Same-Page Merging (KSM). In: Red Hat Enterprise Linux 7 Virtualization and Optimization Guide.

[23] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security (CCS)*, pages 199–212, 2009.

[24] K. Suzaki, K. Iijima, T. Yagi, and C. Artho. Memory deduplication as a threat to the guest OS. In *European Workshop on System Security (EUROSEC)*, 2011.

[25] Tool Interface Standards (TIS) Committee. Executable and Linking Format (ELF) Specification Version 1.2. 1995.

[26] V. Varadarajan, Y. Zhang, T. Ristenpart, and M. M. Swift. A placement vulnerability study in multi-tenant public clouds. In *USENIX Security, 2015*, pages 913–928, 2015.

[27] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Symposium on Operating System Design and Implementation (OSDI)*, 2002.

[28] J. Xiao, Z. Xu, H. Huang, and H. Wang. Security implications of memory deduplication in a virtualized environment. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, 2013.

ABOUT THE AUTHORS:

	Jens Lindemann is a research associate and PhD student in the Security in Distributed Systems research group at the University of Hamburg. Previously, he studied Information Systems and IT-Management and -Consulting at the University of Hamburg, during which he spent one semester abroad at the School of Computer Science at the University of St Andrews. His research interests include IT and network security, virtualization and privacy.
	Mathias is an assistant professor at the University Hamburg since September 2016. Before that, he was an assistant professor at the University Münster (2015-16), a Postdoc at the International Computer Science Institute (ICSI) / UC Berkeley (2014-15), and Postdoc at the Center for Advanced Security Research Darmstadt (CASED) / TU Darmstadt from (2012-14). His research interests encompass IT and network security, resilient distributed systems, network monitoring, and botnets. Mathias received a PhD in 2012 and a diploma in computer science in 2008, both from TU Ilmenau.

OOlong: A Concurrent Object Calculus for Extensibility and Reuse

Elias Castegren
KTH Royal Institute of Technology
Kista, Sweden
eliasca@kth.se

Tobias Wrigstad
Uppsala University
Uppsala, Sweden
tobias.wrigstad@it.uu.se

ABSTRACT

We present OOlong, an object calculus with interface inheritance, structured concurrency and locks. The goal of the calculus is extensibility and reuse. The semantics are therefore available in a version for LaTeX typesetting (written in Ott), a mechanised version for doing rigorous proofs in Coq, and a prototype interpreter (written in OCaml) for typechecking an running OOlong programs.

CCS Concepts

•**Theory of computation** → **Operational semantics;** *Concurrency;* Interactive proof systems; •**Software and its engineering** → *Object oriented languages; Concurrent programming structures; Interpreters;*

Keywords

Object Calculi; Semantics; Mechanisation; Concurrency

1. INTRODUCTION

When reasoning about object-oriented programming, object calculi are a useful tool for abstracting away many of the complicated details of a full-blown programming language. They provide a context for prototyping in which proving soundness or other interesting properties of a language is doable with reasonable effort.

The level of detail depends on which concepts are under study. One of the most used calculi is Featherweight Java, which models inheritance but completely abstracts away mutable state [14]. The lack of state makes it unsuitable for reasoning about any language feature which entails object mutation, and many later extensions of the calculus re-adds state as a first step. Other proposals have also arisen as contenders for having "just the right level of detail" [3, 18, 26].

This paper introduces OOlong, a small, imperative object calculus for the multi-core age. Rather than modelling a specific language, OOlong aims to model object-oriented programming in general, with the goal of being extensible and reusable. To keep subtyping simple, OOlong uses interfaces and omits class inheritance and method overriding. This avoids tying the language to a specific model of class inheritance (*e.g.,* Java's), while still maintaining an object-oriented style of programming. Concurrency is modeled in a finish/async style, and synchronisation is handled via locks.

The semantics are provided both on paper and in a mechanised version written in Coq. The paper version of OOlong is defined in Ott [25], and all typing rules in this paper are generated from this definition. To make it easy for other researchers to build on OOlong, we are making the sources of both versions of the semantics publicly available. We also provide a prototype interpreter written in OCaml.

With the goal of extensibility and re-usability, we make the following contributions:

- We define the formal semantics of OOlong, motivate the choice of features, and prove type soundness (Sections 2–5).
- We provide a mechanised version of the full semantics and soundness proof, written in Coq (Section 6).
- We provide Ott sources for easily extending the paper version of the semantics and generating typing rules in LaTeX (Section 7).
- We give three examples of how OOlong can be extended; support for assertions, more fine-grained locking based on regions, and type-level tracking of null references (Section 8).
- We present the implementation of a simple prototype interpreter to allow automatic type checking and evaluation of OOlong programs. It provides a starting point for additional prototyping of extensions (Section 9).

This paper is an extended version of previous work [7]. Apart from minor additions and improvements, the section on the mechanised semantics has been expanded (Section 6). The extension to track null references through types is new (Section 8.3), and the prototype OOlong interpreter has not been described before (Section 9).

2. RELATED WORK

The main source of inspiration for OOlong is Welterweight Java by Östlund and Wrigstad [18], a concurrent core calculus for Java with ease of reuse as an explicit goal. Welterweight

	FJ	ClJ	ConJ	MJ	LJ	WJ	**OOlong**
State		×	×	×	×	×	×
Statements				×	×	×	
Expressions	×	×	×	×			×
Class Inheritance	×	×	×	×	×	×	
Interfaces		×					×
Concurrency			×			×	×
Stack				×		×	
Mechanised	×*				×		×
LaTeX sources					×	×	×

Figure 1: A comparison between Featherweight Java, `ClassicJava`, `ConcurrentJava`, Middleweight Java, Lightweight Java, Welterweight Java and OOlong.

Java is also defined in Ott, which facilitates simple extension and LaTeX typesetting, but only exists as a calculus on paper. There is no online resource for accessing the Ott sources, and no published proofs except for the sketches in the original treatise. OOlong provides Ott sources and is also fully mechanised in Coq, increasing reliability. Having a proof that can be extended along with the semantics also improves re-usability. Both the Ott sources and the mechanised semantics are publicly available online [5]. OOlong is more lightweight than Welterweight Java by omitting mutable variables and using a single flat stack frame instead of modelling the call stack. Also, OOlong is expression-based whereas Welterweight Java is statement-based, making the OOlong syntax more flexible. We believe that all these things make OOlong easier to reason and prove things about, and more suitable for extension than Welterweight Java.

Object calculi are used regularly as a means of exploring and proving properties about language semantics. These calculi are often tailored for some special purpose, *e.g.,* the calculus of dependent object types [1], which aims to act as a core calculus for Scala, or OrcO [19], which adds objects to the concurrent-by-default language Orc. While these calculi serve their purposes well, their tailoring also make them fit less well as a basis for extension when reasoning about languages which do not build upon the same features. OOlong aims to act as a calculus for common object-oriented languages in order to facilitate reasoning about extensions for such languages.

2.1 Java-based Calculi

There are many object calculi which aim to act as a core calculus for Java. While OOlong does not aim to model Java, it does not actively avoid being similar to Java. A Java programmer should feel comfortable looking at OOlong code, but a researcher using OOlong does not need to use Java as the model. Figure 1 surveys the main differences between different Java core calculi and OOlong. In contrast to many of the Java-based calculi, OOlong ignores inheritance between classes and instead uses only interfaces. While inheritance is an important concept in Java, we believe that subtyping is a much more important concept for object-oriented programming in general. Interfaces provide a simple way to achieve subtyping without having to include concepts like overriding. With interfaces in place, extending the calculus to model other inheritance techniques like mixins [12] or traits [24] becomes easier.

The smallest proposed candidate for a core Java calculus is probably Featherweight Java [14], which omits all forms of assignment and object state, focusing on a functional core of Java. While this is enough for reasoning about Java's type system, the lack of mutable state precludes reasoning about object-oriented programming in a realistic way. Extensions of this calculus often re-add state as a first step (*e.g.,* [2, 17, 23]). The original formulation of Featherweight Java was not mechanised, but a later variation omitting casts and introducing assignment was mechanised in Coq (∼2300 lines) [17]. When developing mixins, Flatt *et al.* define `ClassicJava` [12], an imperative core Java calculus with classes and interfaces. It has been extended several times (*e.g.,* [9, 27]). Flanagan and Freund later added concurrency and locks to `ClassicJava` in `ConcurrentJava` [11], but omitted interfaces. To the best of our knowledge, neither `ClassicJava` nor `ConcurrentJava` have been mechanised.

Bierman *et al.* define Middleweight Java [3], another imperative core calculus which also models object identity, null pointers, constructors and Java's block structure and call stack. Middleweight Java is a true subset of Java, meaning that all valid Middleweight Java programs are also valid Java programs. The high level of detail however makes it unattractive for extensions which are not highly Java-specific. To the best of our knowledge, Middleweight Java was never mechanised. Strniša proposes Lightweight Java as a simplification of Middleweight Java [26], omitting block scoping, type casts, constructors, expressions, and modelling of the call stack, while still being a proper subset of Java. Like Welterweight Java it is purely based on statements, and does not include interfaces. Like OOlong, Lightweight Java is defined in Ott, but additionally uses Ott to generate a mechanised formalism in Isabelle/HOL. A later extension of Lightweight Java was also mechanised in Coq (∼800 lines generated from Ott, and another ∼5800 lines of proofs) [10].

Last, some language models go beyond the surface language and execution. One such model is Jinja by Klein and Nipkow [16], which models (parts of) the entire Java architecture, including the virtual machine and compilation from Java to byte code. To handle the complexity of such a system, Jinja is fully mechanised in Isabelle/HOL. The focus of Jinja is different than that of calculi like OOlong, and is therefore not practical for exploring language extensions which do not alter the underlying runtime.

2.2 Background

OOlong started out as a target language acting as dynamic semantics for a type system for concurrency control [6]. The proof schema for this system involved translating the source language into OOlong, establishing a mapping between the types of the two languages, and reasoning about the behaviour of a running OOlong program. In this context, OOlong was extended with several features, including assertions, readers–writer locks, regions, destructive reads and mechanisms for tracking which variables belong to which stack frames (Section 8 outlines the addition of assertions and regions). By having a machine checked proof of soundness for OOlong that we could trust, the proof of progress and preservation of the source language followed from showing that translation preserves well-formedness of programs.

$$
\begin{array}{llll}
P & ::= & \mathit{Ids}\ \mathit{Cds}\ e & \textit{(Programs)} \\
\mathit{Id} & ::= & \textbf{interface}\ I\ \{\mathit{Msigs}\} & \textit{(Interfaces)} \\
 & | & \textbf{interface}\ I\ \textbf{extends}\ I_1, I_2 & \\
\mathit{Cd} & ::= & \textbf{class}\ C\ \textbf{implements}\ I\ \{\mathit{Fds}\ \mathit{Mds}\} & \textit{(Classes)} \\
\mathit{Msig} & ::= & m(x : t_1) : t_2 & \textit{(Signatures)} \\
\mathit{Fd} & ::= & f : t & \textit{(Fields)} \\
\mathit{Md} & ::= & \textbf{def}\ \mathit{Msig}\ \{e\} & \textit{(Methods)} \\
e & ::= & v \mid x \mid x.f \mid x.f = e & \textit{(Expressions)} \\
 & | & x.m(e) \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid \textbf{new}\ C \mid (t)\ e & \\
 & | & \textbf{finish}\{\textbf{async}\{e_1\}\ \textbf{async}\{e_2\}\};\ e_3 & \\
 & | & \textbf{lock}(x)\ \textbf{in}\ e \mid \boxed{\textbf{locked}_\iota\{e\}} & \\
v & ::= & \textbf{null} \mid \boxed{\iota} & \textit{(Values)} \\
t & ::= & C \mid I \mid \textbf{Unit} & \textit{(Types)} \\
\Gamma & ::= & \epsilon \mid \Gamma, x : t \mid \boxed{\Gamma, \iota : C} & \textit{(Typing environment)}
\end{array}
$$

Figure 2: The syntax of OOlong.

3. STATIC SEMANTICS OF OOlong

In this section, we describe the static semantics of OOlong. The semantics are also available as Coq sources, together with a full soundness proof. The main differences between the paper version and the mechanised semantics are outlined in Section 6.

Figure 2 shows the syntax of OOlong. Ids, Cds, Fds, Mds and Msigs are sequences of zero or more of their singular counterparts. Terms in grey boxes are not part of the surface syntax but only appear during evaluation. The meta-syntactic variables are x, y and **this** for variable names, f for field names, C for class names, I for interface names, and m for method names. For simplicity we assume that all names are unique. OOlong defines objects through classes, which implement some interface. Interfaces are in turn defined either as a collection of method signatures, or as an "inheriting" interface which joins two other interfaces. There is no inheritance between classes, and no overriding of methods. A program is a collection of interfaces and classes together with a starting expression e. An example of a full OOlong program (extended to handle integers) can be found in Figure 10.

Most expressions are standard: values (**null** or abstract object locations ι), variables, field accesses, field assignments, method calls, object instantiation and type casts. For simplicity, targets of field and method lookups must be variables, and method calls have exactly one argument (multiple arguments can be simulated through object indirection, and an empty argument list by passing **null**). We also use **let**-bindings rather than sequences and variables. Sequencing can be achieved through the standard trick of translating e_1; e_2 into $\textbf{let}\ \_ = e_1\ \textbf{in}\ e_2$ (due to eager evaluation of e_1). Parallel threads are spawned with the expression $\textbf{finish}\{\textbf{async}\{e_1\}\ \textbf{async}\{e_2\}\};\ e_3$, which runs e_1 and e_2 in parallel, waits for their completion, and then continues with e_3.

The expression $\textbf{lock}(x)\ \textbf{in}\ e$ locks the object pointed to by x for the duration of e. While an expression locking the object at location ι is executed in the dynamic semantics, it appears as $\textbf{locked}_\iota\{e\}$. This way, locks are automatically released at the end of the expression e. It also allows tracking which field accesses are protected by locks and not.

$\boxed{\vdash P : t \quad \vdash \mathit{Id} \quad \vdash \mathit{Cd} \quad \vdash \mathit{Fd} \quad \vdash \mathit{Md}}$ *(Well-formed program)*

WF-PROGRAM
$$\frac{\forall\, \mathit{Id} \in \mathit{Ids}.\ \vdash \mathit{Id} \qquad \forall\, \mathit{Cd} \in \mathit{Cds}.\ \vdash \mathit{Cd} \qquad \epsilon \vdash e : t}{\vdash \mathit{Ids}\ \mathit{Cds}\ e : t}$$

WF-INTERFACE
$$\frac{\forall\, m(x : t) : t' \in \mathit{Msigs}.\ \vdash t \wedge \vdash t'}{\vdash \textbf{interface}\ I\ \{\ \mathit{Msigs}\ \}}$$

WF-INTERFACE-EXTENDS
$$\frac{\vdash I_1 \qquad \vdash I_2}{\vdash \textbf{interface}\ I\ \textbf{extends}\ I_1, I_2}$$

WF-CLASS
$$\frac{\begin{array}{c}\forall\, m(x : t) : t' \in \textbf{msigs}\,(I).\textbf{def}\, m(x : t) : t'\ \{\ e\ \} \in \mathit{Mds} \\ \forall\, \mathit{Fd} \in \mathit{Fds}.\ \vdash \mathit{Fd} \qquad \forall\, \mathit{Md} \in \mathit{Mds}.\textbf{this} : C \vdash \mathit{Md}\end{array}}{\vdash \textbf{class}\ C\ \textbf{implements}\ I\ \{\ \mathit{Fds}\ \mathit{Mds}\ \}}$$

WF-FIELD
$$\frac{\vdash t}{\vdash f : t}$$

WF-METHOD
$$\frac{\textbf{this} : C, x : t \vdash e : t'}{\textbf{this} : C \vdash \textbf{def}\, m(x : t) : t'\ \{\ e\ \}}$$

Figure 3: Well-formedness of classes and interfaces.

Types are class or interface names, or **Unit** (used as the type of assignments). The typing environment Γ maps variables to types and abstract locations to classes.

3.1 Well-Formed Program

Figure 3 shows the definition of a well-formed program, which consists of well-formed interfaces and well-formed classes, plus a well-typed starting expression. A non-empty interface is well-formed if its method signatures only mention well-formed types (WF-INTERFACE), and an inheriting interface is well-formed if the interfaces it extends are well-formed (WF-INTERFACE-EXTENDS). A class is well-formed if it implements all the methods in its interface (the helper function **msigs** is defined in the appendix, *cf.,* Section A.3). Further, all fields and methods must be well-formed (WF-CLASS). A field is well-formed if its type is well-formed (WF-FIELD). A method is well-formed if its body has the type specified as the method's return type under an environment containing the single parameter and the type of the current **this** (WF-METHOD).

3.2 Types and Subtyping

Figure 4 shows the rules relating to typing, subtyping, and the typing environment Γ. Each class or interface in the program corresponds to a well-formed type (T-WF-*). Subtyping is transitive and reflexive, and is nominally defined by the interface hierarchy of the current program (T-SUB-*). A well-formed environment Γ has variables of well-formed types and locations of valid class types (WF-ENV). Finally, the frame rule splits an environment Γ_1 into two sub-environments Γ_2 and Γ_3 whose variable domains are disjoint (but which may share locations ι). The meta-syntactic variable γ abstracts over variables x and locations ι (to reduce clutter), and the helper function **vardom** extracts the set of variables from an environment (*cf.,* Section A.3). The frame rule is used

$\boxed{\vdash t}$ *(Well-formed types)*

$$\frac{\textbf{class}\ C\ \textbf{implements}\ I\ \{\ \_\ \} \in P}{\vdash C}\ \textsc{T-WF-Class} \qquad \frac{\textbf{interface}\ I\ \{\ \_\ \} \in P}{\vdash I}\ \textsc{T-WF-Interface}$$

$$\frac{\textbf{interface}\ I\ \textbf{extends}\ I_1, I_2 \in P}{\vdash I}\ \textsc{T-WF-Interface-Extends} \qquad \frac{}{\vdash \textbf{Unit}}\ \textsc{T-WF-Unit}$$

$\boxed{t_1 <: t_2}$ *(Subtyping)*

$$\frac{\textbf{class}\ C\ \textbf{implements}\ I\ \{\ \_\ \} \in P}{C <: I}\ \textsc{T-Sub-Class}$$

$$\frac{\textbf{interface}\ I\ \textbf{extends}\ I_1, I_2 \in P}{I <: I_1}\ \textsc{T-Sub-Interface-Left}$$

$$\frac{\textbf{interface}\ I\ \textbf{extends}\ I_1, I_2 \in P}{I <: I_2}\ \textsc{T-Sub-Interface-Right} \qquad \frac{t_1 <: t_2 \qquad t_2 <: t_3}{t_1 <: t_3}\ \textsc{T-Sub-Trans}$$

$$\frac{\vdash t}{t <: t}\ \textsc{T-Sub-Eq}$$

$\boxed{\vdash \Gamma}$ *(Well-formed environment)*

$$\frac{\forall\, x : t \in \Gamma.\ \vdash t \qquad \forall\, \iota : C \in \Gamma.\ \vdash C}{\vdash \Gamma}\ \textsc{WF-Env}$$

$\boxed{\Gamma_1 = \Gamma_2 + \Gamma_3}$ *(Frame Rule)*

$$\frac{\begin{array}{c}\forall\, \gamma : t \in \Gamma_2 . \Gamma_1(\gamma) = t \\ \forall\, \gamma : t \in \Gamma_3 . \Gamma_1(\gamma) = t \\ (\textbf{vardom}\,(\Gamma_2) \cap \textbf{vardom}\,(\Gamma_3)) \ = \ \emptyset\end{array}}{\Gamma_1 = \Gamma_2 + \Gamma_3}\ \textsc{WF-Frame}$$

Figure 4: Typing, subtyping, and the typing environment.

$\boxed{\Gamma \vdash e : t}$ *(Typing Expressions)*

$$\frac{\vdash \Gamma \qquad \Gamma(x) = t}{\Gamma \vdash x : t}\ \textsc{WF-Var} \qquad \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma, x : t_1 \vdash e_2 : t}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : t}\ \textsc{WF-Let}$$

$$\frac{\begin{array}{c}\Gamma \vdash x : t_1 \qquad \Gamma \vdash e : t_2 \\ \textbf{msigs}\,(t_1)(m) = y : t_2 \to t\end{array}}{\Gamma \vdash x.m(e) : t}\ \textsc{WF-Call} \qquad \frac{\Gamma \vdash e : t' \qquad t' <: t}{\Gamma \vdash (t)e : t}\ \textsc{WF-Cast}$$

$$\frac{\begin{array}{c}\Gamma \vdash x : C \\ \textbf{fields}\,(C)(f) = t\end{array}}{\Gamma \vdash x.f : t}\ \textsc{WF-Select} \qquad \frac{\begin{array}{c}\Gamma \vdash x : C \qquad \Gamma \vdash e : t \\ \textbf{fields}\,(C)(f) = t\end{array}}{\Gamma \vdash x.f = e : \textbf{Unit}}\ \textsc{WF-Update}$$

$$\frac{\vdash \Gamma \qquad \vdash C}{\Gamma \vdash \textbf{new}\ C : C}\ \textsc{WF-New} \qquad \frac{\vdash \Gamma \qquad \Gamma(\iota) = C \qquad C <: t}{\Gamma \vdash \iota : t}\ \textsc{WF-Loc}$$

$$\frac{\vdash \Gamma \qquad \vdash t}{\Gamma \vdash \textbf{null} : t}\ \textsc{WF-Null}$$

$$\frac{\Gamma = \Gamma_1 + \Gamma_2 \qquad \Gamma_1 \vdash e_1 : t_1 \qquad \Gamma_2 \vdash e_2 : t_2 \qquad \Gamma \vdash e : t}{\Gamma \vdash \textbf{finish}\ \{\ \textbf{async}\ \{\ e_1\ \}\ \textbf{async}\ \{\ e_2\ \}\ \}\ ; e : t}\ \textsc{WF-FJ}$$

$$\frac{\Gamma \vdash x : t_2 \qquad \Gamma \vdash e : t}{\Gamma \vdash \textbf{lock}(x)\ \textbf{in}\ e : t}\ \textsc{WF-Lock} \qquad \frac{\Gamma \vdash e : t \qquad \Gamma(\iota) = t_2}{\Gamma \vdash \textbf{locked}_{\iota}\{e\} : t}\ \textsc{WF-Locked}$$

Figure 5: Typing of expressions

when spawning new threads to prevent them from sharing variables[1].

3.3 Expression Typing

Figure 5 shows the typing rules for expressions, most of which are straightforward. Variables are looked up in the environment (WF-VAR) and introduced using **let** bindings (WF-LET). Method calls require the argument to exactly match the parameter type of the method signature (WF-CALL). We require explicit casts, and only support upcasts (WF-CAST). Fields are looked up with the helper function **fields** (WF-SELECT). Fields may only be looked up in class types (as interfaces do not define fields). Field updates have the **Unit** type (WF-UPDATE). Any class in the program can be instantiated (WF-NEW). Locations can be given any super type of their class type given in the environment (WF-LOC). The constant **null** can be given any well-formed type, including **Unit** (WF-NULL). Forking new threads requires that the accessed variables are disjoint, which is enforced by the frame rule $\Gamma = \Gamma_1 + \Gamma_2$ (WF-FJ). Locks can be taken on any well-formed target (WF-LOCK*).

Section 9 introduces a bidirectional version of the typing rules which are entirely syntax-directed (meaning they can be directly implemented by a type checker) and which handle implicit upcasts, *e.g.,* for arguments to method calls.

4. DYNAMIC SEMANTICS OF OOlong

In this section, we describe the dynamic semantics of OOlong. Figure 6 shows the structure of the run-time constructs of OOlong. A configuration $\langle H; V; T\rangle$ contains a heap H, a variable map V, and a collection of threads T. A heap H maps abstract locations to objects. Objects store their class, a map F from field names to values, and a lock status L which is either **locked** or **unlocked**. A stack map V maps variable names to values. As variables are never updated, OOlong could use a simple variable substitution scheme

[1]Since variables are immutable in OOlong, this kind of sharing would not be a problem in practice, but for extensions requiring mutable variables, we believe having this in place makes sense.

$$
\begin{array}{lcll}
\mathit{cfg} & ::= & \langle H; V; T \rangle & \textit{(Configuration)} \\
H & ::= & \epsilon \mid H, \iota \mapsto \mathit{obj} & \textit{(Heap)} \\
V & ::= & \epsilon \mid V, x \mapsto v & \textit{(Variable map)} \\
T & ::= & (\mathcal{L}, e) \mid T_1 \,||\, T_2 \rhd e \mid \textbf{EXN} & \textit{(Threads)} \\
\mathit{obj} & ::= & (C, F, L) & \textit{(Objects)} \\
F & ::= & \epsilon \mid F, f \mapsto v & \textit{(Field map)} \\
L & ::= & \textbf{locked} \mid \textbf{unlocked} & \textit{(Lock status)} \\
\textbf{EXN} & ::= & \textbf{NullPointerException} & \textit{(Exceptions)}
\end{array}
$$

Figure 6: Run-time constructs of OOlong.

instead of tracking the values of variables in a map. However, the current design gives us a simple way of reasoning about object references on the stack as well as on the heap, and makes it easier to later add support for assignable variables.

A thread collection T can have one of three forms: $T_1||T_2 \rhd e$ denotes two parallel asyncs T_1 and T_2 which must reduce fully before evaluation proceeds to e. $(\mathcal{L}, e)$ is a single thread evaluating expression e. $\mathcal{L}$ is a set of locations of all the objects whose locks are currently being held by the thread. The initial configuration is $\langle \epsilon; \epsilon; (\emptyset, e)\rangle$, where e is the initial expression of the program. A thread can also be in an exceptional state **EXN**, which is a well-formed but "crashed" state that cannot be recovered from. The current semantics only supports the **NullPointerException**.

4.1 Well-Formedness Rules

Figure 7 shows the definition of a well-formed OOlong configuration. A configuration is well-formed if its heap H and stack V are well-formed, its collection of threads T is well-typed, and the current lock situation in the system is well-formed (WF-CFG). Note that well-formedness of threads is split into two sets of rules regarding expression typing and locking respectively A heap H is well-formed under a Γ if all locations in Γ correspond to objects in H, all objects in the heap have an entry in Γ, and the fields of all objects are well-formed under Γ (WF-HEAP). The fields of an object of class C are well-formed if each name of the static fields of C maps to a value of the corresponding type (WF-FIELDS). A stack V is well-formed under a Γ if each variable in Γ maps to a value of the corresponding type in V, and each variable in V has an entry in Γ (WF-VARS). A well-formed thread collection requires all sub-threads and expressions to be well-formed (WF-T-*). An exceptional state can have any well-formed type (WF-T-EXN).

The current lock situation is well-formed for a thread if all locations in its set of held locks $\mathcal{L}$ correspond to objects whose lock status is **locked**. Two instances of $\textbf{locked}_\iota$ in e must refer to different locations ι (captured by *distinctLocks*(e), *cf.,* Section A.3), and for each $\textbf{locked}_\iota$ in e, ι must be in the set of held locks $\mathcal{L}$. The parallel case propagates these properties, and additionally requires that two parallel threads do not hold the same locks in their respective $\mathcal{L}$. Any locks held in the continuation e must be held by the first thread of the async. This represents the fact the first thread is the one that will continue execution after the threads join (WF-L-ASYNC). Exceptional states are always well-formed with respect to locking (WF-L-EXN).

$\boxed{\Gamma \vdash \langle H; V; T\rangle : t}$ *(Well-formed configuration)*

$$
\text{WF-CFG}\quad \frac{\Gamma \vdash H \qquad \Gamma \vdash V \qquad \Gamma \vdash T : t \qquad H \vdash_{\text{lock}} T}{\Gamma \vdash \langle H;\, V;\, T\rangle : t}
$$

$$
\text{WF-HEAP}\quad \frac{\forall \iota : C \in \Gamma.H(\iota) = (C, F, L) \land \Gamma; C \vdash F \qquad \forall \iota \in \textbf{dom}\,(H).\iota \in \textbf{dom}(\Gamma) \qquad \vdash \Gamma}{\Gamma \vdash H}
$$

$$
\text{WF-FIELDS}\quad \frac{\textbf{fields}\,(C) \;\equiv\; f_1 : t_1, \,..\,, f_n : t_n \qquad \Gamma \vdash v_1 : t_1, \,..\,, \Gamma \vdash v_n : t_n}{\Gamma; C \vdash f_1 \mapsto v_1, \,..\,, f_n \mapsto v_n}
$$

$$
\text{WF-VARS}\quad \frac{\forall x : t \in \Gamma.\, V(x) = v \land \Gamma \vdash v : t \qquad \forall x \in \textbf{dom}\,(V).x \in \textbf{dom}(\Gamma) \qquad \vdash \Gamma}{\Gamma \vdash V}
$$

$$
\text{WF-T-ASYNC}\quad \frac{\Gamma \vdash T_1 : t_1 \qquad \Gamma \vdash T_2 : t_2 \qquad \Gamma \vdash e : t}{\Gamma \vdash T_1 \,||\, T_2 \rhd e : t}
\qquad
\text{WF-T-THREAD}\quad \frac{\Gamma \vdash e : t}{\Gamma \vdash (\mathcal{L}, e) : t}
$$

$$
\text{WF-T-EXN}\quad \frac{\vdash t \qquad \vdash \Gamma}{\Gamma \vdash \textbf{EXN} : t}
\qquad
\text{WF-L-THREAD}\quad \frac{\forall \iota \in \mathcal{L}.H(\iota) = (C, F, \textbf{locked}) \qquad \mathit{distinctLocks}(e) \qquad \forall \iota \in \textbf{locks}\,(e).\iota \in \mathcal{L}}{H \vdash_{\text{lock}} (\mathcal{L}, e)}
$$

$$
\text{WF-L-ASYNC}\quad \frac{\textbf{heldLocks}\,(T_1) \cap \textbf{heldLocks}\,(T_2) \;=\; \emptyset \qquad \forall \iota \in \textbf{locks}\,(e).\iota \in \textbf{heldLocks}\,(T_1) \qquad \mathit{distinctLocks}(e) \qquad H \vdash_{\text{lock}} T_1 \qquad H \vdash_{\text{lock}} T_2}{H \vdash_{\text{lock}} T_1 \,||\, T_2 \rhd e}
$$

$$
\text{WF-L-EXN}\quad \frac{}{H \vdash_{\text{lock}} \textbf{EXN}}
$$

Figure 7: Well-formedness rules.

4.2 Evaluation of Expressions

Figure 8 shows the single-threaded execution of an OOlong program. OOlong uses a small-step dynamic semantics, with the standard technique of evaluation contexts (the definition of E in the top of the figure) to decide the order of evaluation and reduce the number of rules (DYN-EVAL-CONTEXT). We use a single stack frame for the entire program and employ renaming to make sure that variables have unique names[2]. Evaluating a variable simply looks it up in the stack (DYN-EVAL-VAR). A **let**-expression introduces a fresh variable that it substitutes for the static name (DYN-EVAL-LET). Similarly, calling a method introduces two new fresh variables—one for **this** and one for the parameter of the method. The method is dynamically dispatched on the type

[2]This sacrifices reasoning about properties of the stack size in favour of simpler dynamic semantics.

$$E[\bullet] ::= x.f = \bullet \mid x.m(\bullet) \mid \textbf{let}\ x = \bullet\ \textbf{in}\ e \mid (t)\ \bullet \mid \textbf{locked}_\iota\{\bullet\}$$

$\boxed{cfg_1 \hookrightarrow cfg_2}$ *(Evaluation of expressions)*

DYN-EVAL-CONTEXT
$$\frac{\langle H; V; (\mathcal{L}, e)\rangle \hookrightarrow \langle H'; V'; (\mathcal{L}', e')\rangle}{\langle H; V; (\mathcal{L}, E[e])\rangle \hookrightarrow \langle H'; V'; (\mathcal{L}', E[e'])\rangle}$$

DYN-EVAL-VAR
$$\frac{V(x) = v}{\langle H; V; (\mathcal{L}, x)\rangle \hookrightarrow \langle H; V; (\mathcal{L}, v)\rangle}$$

DYN-EVAL-LET
$$\frac{x'\ \textbf{fresh} \qquad V' = V[x' \mapsto v] \qquad e' = e[x \mapsto x']}{\langle H; V; (\mathcal{L}, \textbf{let}\ x = v\ \textbf{in}\ e)\rangle \hookrightarrow \langle H; V'; (\mathcal{L}, e')\rangle}$$

DYN-EVAL-CALL
$$\frac{\begin{array}{c} V(x) = \iota \qquad H(\iota) = (C, F, L) \\ \textbf{methods}\,(C)(m) = y : t_2 \to t, e \\ \textbf{this}'\ \textbf{fresh} \qquad y'\ \textbf{fresh} \\ V' = V[\textbf{this}' \mapsto \iota][y' \mapsto v] \\ e' = e[\textbf{this} \mapsto \textbf{this}'][y \mapsto y'] \end{array}}{\langle H; V; (\mathcal{L}, x.m(v))\rangle \hookrightarrow \langle H; V'; (\mathcal{L}, e')\rangle}$$

DYN-EVAL-CAST
$$\frac{}{\langle H; V; (\mathcal{L}, (t)v)\rangle \hookrightarrow \langle H; V; (\mathcal{L}, v)\rangle}$$

DYN-EVAL-SELECT
$$\frac{\begin{array}{c} V(x) = \iota \qquad H(\iota) = (C, F, L) \\ \textbf{fields}\,(C)(f) = t \qquad F(f) = v \end{array}}{\langle H; V; (\mathcal{L}, x.f)\rangle \hookrightarrow \langle H; V; (\mathcal{L}, v)\rangle}$$

DYN-EVAL-UPDATE
$$\frac{\begin{array}{c} V(x) = \iota \qquad H(\iota) = (C, F, L) \\ \textbf{fields}\,(C)(f) = t' \qquad H' = H[\iota \mapsto (C, F[f \mapsto v], L)] \end{array}}{\langle H; V; (\mathcal{L}, x.f = v)\rangle \hookrightarrow \langle H'; V; (\mathcal{L}, \textbf{null})\rangle}$$

DYN-EVAL-NEW
$$\frac{\begin{array}{c} \textbf{fields}\,(C) \equiv f_1 : t_1, .., f_n : t_n \\ F \equiv f_1 \mapsto \textbf{null}, .., f_n \mapsto \textbf{null} \\ \iota\ \textbf{fresh} \qquad H' = H[\iota \mapsto (C, F, \textbf{unlocked})] \end{array}}{\langle H; V; (\mathcal{L}, \textbf{new}\ C)\rangle \hookrightarrow \langle H'; V; (\mathcal{L}, \iota)\rangle}$$

DYN-EVAL-LOCK
$$\frac{\begin{array}{c} V(x) = \iota \qquad H(\iota) = (C, F, \textbf{unlocked}) \qquad \iota \notin \mathcal{L} \\ H' = H[\iota \mapsto (C, F, \textbf{locked})] \qquad \mathcal{L}' = \mathcal{L} \cup \{\iota\} \end{array}}{\langle H; V; (\mathcal{L}, \textbf{lock}(x)\ \textbf{in}\ e)\rangle \hookrightarrow \langle H'; V; (\mathcal{L}', \textbf{locked}_\iota\{e\})\rangle}$$

DYN-EVAL-LOCK-REENTRANT
$$\frac{V(x) = \iota \qquad H(\iota) = (C, F, \textbf{locked}) \qquad \iota \in \mathcal{L}}{\langle H; V; (\mathcal{L}, \textbf{lock}(x)\ \textbf{in}\ e)\rangle \hookrightarrow \langle H; V; (\mathcal{L}, e)\rangle}$$

DYN-EVAL-LOCK-RELEASE
$$\frac{\begin{array}{c} H(\iota) = (C, F, \textbf{locked}) \qquad \mathcal{L}' = \mathcal{L} \backslash \{\iota\} \\ H' = H[\iota \mapsto (C, F, \textbf{unlocked})] \end{array}}{\langle H; V; (\mathcal{L}, \textbf{locked}_\iota\{v\})\rangle \hookrightarrow \langle H'; V; (\mathcal{L}', v)\rangle}$$

Figure 8: Dynamic semantics (1/2). Expressions.

$\boxed{cfg_1 \hookrightarrow cfg_2}$ *(Concurrency)*

DYN-EVAL-ASYNC-LEFT
$$\frac{\langle H; V; T_1\rangle \hookrightarrow \langle H'; V'; T_1'\rangle}{\langle H; V; T_1 \,||\, T_2 \rhd e\rangle \hookrightarrow \langle H'; V'; T_1' \,||\, T_2 \rhd e\rangle}$$

DYN-EVAL-ASYNC-RIGHT
$$\frac{\langle H; V; T_2\rangle \hookrightarrow \langle H'; V'; T_2'\rangle}{\langle H; V; T_1 \,||\, T_2 \rhd e\rangle \hookrightarrow \langle H'; V'; T_1 \,||\, T_2' \rhd e\rangle}$$

DYN-EVAL-SPAWN
$$\frac{e = \textbf{finish}\ \{\ \textbf{async}\ \{\ e_1\ \}\ \textbf{async}\ \{\ e_2\ \}\ \}\,;\,e_3}{\langle H; V; (\mathcal{L}, e)\rangle \hookrightarrow \langle H; V; (\mathcal{L}, e_1) \,||(\emptyset, e_2) \rhd e_3\rangle}$$

DYN-EVAL-SPAWN-CONTEXT
$$\frac{\langle H; V; (\mathcal{L}, e)\rangle \hookrightarrow \langle H; V; (\mathcal{L}, e_1) \,||(\emptyset, e_2) \rhd e_3\rangle}{\langle H; V; (\mathcal{L}, E[e])\rangle \hookrightarrow \langle H; V; (\mathcal{L}, e_1) \,||(\emptyset, e_2) \rhd E[e_3]\rangle}$$

DYN-EVAL-ASYNC-JOIN
$$\frac{}{\langle H; V; (\mathcal{L}, v) \,||(\mathcal{L}', v') \rhd e\rangle \hookrightarrow \langle H; V; (\mathcal{L}, e)\rangle}$$

Figure 9: Dynamic semantics (2/2). Concurrency.

of the target object (DYN-EVAL-CALL).

Casts will always succeed and are therefore no-ops dynamically (DYN-EVAL-CAST). Adding support for downcasts is possible with the introduction of a new exceptional state for failed casts. Fields are looked up in the field map of the target object (DYN-EVAL-SELECT). Similarly, field assignments are handled by updating the field map of the target object. Field updates, which are always typed as **Unit**, evaluate to **null** (DYN-EVAL-UPDATE). We have omitted constructors from this treatise (Section 8.3 discusses how they can be added). A new object has its fields initialised to **null** and is given a fresh abstract location on the heap (DYN-EVAL-NEW).

Taking a lock requires that the lock is currently available and adds the locked object to the lock set $\mathcal{L}$ of the current thread. It also updates the object to reflect its locked status (DYN-EVAL-LOCK). The locks in OOlong are reentrant, meaning that grabbing the same lock twice will always succeed (DYN-EVAL-LOCK-REENTRANT). Locking is structured, meaning that a thread can not grab a lock without also releasing it sooner or later (modulo getting stuck due to deadlocks). The **locked** wrapper around e records the successful taking of the lock and is used to release the lock once e has been fully reduced (DYN-EVAL-LOCK-RELEASE). Note that a thread that cannot take a lock gets stuck until the lock is released. We define these states formally to distinguish them from unsound stuck states (*cf.,* Section A.1)

Dereferencing **null**, *e.g.,* using a **null** valued argument when looking up a field or calling a method, results in a **NullPointerException**, which crashes the program. These rules are unsurprising and are therefore relegated to the appendix (*cf.,* Section A.2).

4.3 Concurrency

Figure 9 shows the semantics of concurrent execution in OOlong. Concurrency is modeled as non-deterministic choice of what thread to evaluate (DYN-EVAL-ASYNC-LEFT/RIGHT). Finish/async spawns one new thread for the second async and uses the current thread for the first. This means that the first async holds all the locks of the spawning thread, while the second async starts out with an empty lock set (DYN-EVAL-SPAWN). The evaluation context rule, needed because DYN-EVAL-CONTEXT does not handle spawning, forces the full reduction of the parallel expressions to the left of $\rhd$ before continuing with e_3, which is the expression placed in the hole of the evaluation context (DYN-EVAL-SPAWN-CONTEXT). When two asyncs have finished, the second thread is removed along with all its locks[3], and the first thread continues with the expression to the right of $\rhd$ (DYN-EVAL-ASYNC-JOIN).

5. TYPE SOUNDNESS OF OOlong

We prove type soundness as usual by proving progress and preservation. This section only states the theorems and sketches the proofs. We refer to the mechanised semantics for the full proofs (*cf.,* Section 6).

Since well-formed programs are allowed to deadlock, we must formulate the progress theorem so that this is handeled. The *Blocked* predicate on configurations is defined in the appendix (*cf.,* Section A.1).

PROGRESS. *A well-formed configuration is either done, has thrown an exception, has deadlocked, or can take one additional step:*

$$\begin{array}{l}\forall \Gamma,\ H,\ V,\ T,\ t\ .\ \Gamma \vdash \langle H; V; T\rangle : t \Rightarrow \\ \quad T = (\mathcal{L}, v) \lor T = \mathbf{EXN} \lor \mathit{Blocked}(\langle H; V; T\rangle) \lor \\ \quad \exists \mathit{cfg}', \langle H; V; T\rangle \hookrightarrow \mathit{cfg}'\end{array}$$

PROOF SKETCH. Proved by induction over the thread structure T. The single threaded case is proved by induction over the typing relation over the current expression.

To show preservation of well-formedness we first define a subsumption relation $\Gamma_1 \subseteq \Gamma_2$ between environments. Γ_2 subsumes Γ_1 if all mappings $\gamma : t$ in Γ_1 are also in Γ_2:

$\boxed{\Gamma_1 \subseteq \Gamma_2}$ *(Environment Subsumption)*

$$\text{WF-SUBSUMPTION}\quad \frac{\forall \gamma : t \in \Gamma . \Gamma'(\gamma) = t}{\Gamma \subseteq \Gamma'}$$

PRESERVATION. *If $\langle H; V; T\rangle$ types to t under some environment Γ, and $\langle H; V; T\rangle$ steps to some $\langle H'; V'; T'\rangle$, there exists an environment subsuming Γ which types $\langle H'; V'; T'\rangle$ to t.*

$$\begin{array}{l}\forall \Gamma,\ H,\ H',\ V,\ V',\ T,\ T',\ t. \\ \quad \Gamma \vdash \langle H; V; T\rangle : t \land \langle H; V; T\rangle \hookrightarrow \langle H'; V'; T'\rangle \Rightarrow \\ \quad\quad \exists \Gamma' . \Gamma' \vdash \langle H'; V'; T'\rangle : t \land \Gamma \subseteq \Gamma'\end{array}$$

PROOF SKETCH. Proved by induction over the thread structure T. The single threaded case is proved by induction over the typing relation over the current expression. There are also a number of lemmas regarding locking that needs proving (*e.g.,* that a thread can never steal a lock held by another thread). We refer to the mechanised proofs for details.

[3]In practice, since locking is structured these locks will already have been released.

6. MECHANISED SEMANTICS

We have fully mechanised the semantics of OOlong in Coq, including the proofs of soundness. The source code weighs in at ∼4100 lines of Coq, ∼900 of which are definitions and ∼3200 of which are properties and proofs. In the proof code, ∼300 lines are extra lemmas about lists and ∼200 lines are tactics specific to this formalism used for automating often re-occurring reasoning steps. The proofs also make use of the LibTactics library [20], as well as the `crush` tactic [8]. We use Coq bullets together with Aaron Bohannon's "Case" tactic to structure the proofs and make refactoring simpler; when a definition changes and a proof needs to be rewritten, it is immediately clear which cases fail and therefore need to be updated.

The mechanised semantics are the same as the semantics presented here, modulo uninteresting representation differences such as modelling the typing environment Γ as a function rather than a sequence. It explicitly deals with details such as how to generate fresh names and separating static and dynamic constructs (*e.g.,* when calling a method, the body of the method will not contain any dynamic expressions, such as $\mathbf{locked}_\iota\{e\}$). It also defines helper functions like field and method lookup.

The Coq sources are available in a public repository so that the semantics can be easily obtained and extended [5]. The source files compile under Coq 8.8.2, the latest version at the time of writing.

As a comparison between the Coq definitions and the paper versions, here are the mechanised formulations of progress and preservation:

```
Theorem progress :
  forall P t' Gamma cfg t,
    wfProgram P t' ->
    wfConfiguration P Gamma cfg t ->
      cfg_done cfg \/ cfg_exn cfg \/
      cfg_blocked cfg \/
      exists cfg', P / cfg ==> cfg'.

Theorem preservation :
  forall P t' Gamma cfg cfg' t,
    wfProgram P t' ->
    wfConfiguration P Gamma cfg t ->
    P / cfg ==> cfg' ->
    exists Gamma',
      wfConfiguration P Gamma' cfg' t /\
      wfSubsumption Gamma Gamma'.
```

Other than some notational differences (*e.g.,* using the name `cfg` instead of spelling out $\langle H; V; T\rangle$), the biggest noticeable difference is that the program `P` is threaded through all the definitions, among other things to be able to do field and method lookups. For example, the proposition `P / cfg ==> cfg'` should be read as "`cfg` steps to `cfg'` when executing in program `P`". There is also an explicit requirement that this program is well-formed (`wfProgram P t'`, where `t'` is the type of the starting expression).

As another example, here is the lemma that states that if two threads have disjoint lock sets, stepping one of them will not cause the lock sets to overlap:

```
Lemma stepCannotSteal :
  forall P H H' V V' n n' T1 T1' T2,
    wfLocking H T1 ->
    wfLocking H T2 ->
    disjointLocks T1 T2 ->
    P / (H, V, n, T1) ==> (H', V', n', T1') ->
    disjointLocks T1' T2.
```

The propositions `disjointLocks T1 T2` and `wfLocking H T` correspond to $\textbf{heldLocks}(T_1) \cap \textbf{heldLocks}(T_2) = \emptyset$ and $H \vdash_{\text{lock}} T$ (*cf.,* Figure 7). The extra element `n` in the configuration `(H, V, n, T)` is an integer used to generate fresh variable names.

Finally, we first show how the evaluation context E (*cf.,* Figure 8) is expressed in Coq. An evaluation context is a function taking a single expression to another expression:

```
Definition econtext := expr -> expr.
```

Each case of E is represented by a Coq function of type `econtext`, for example:

```
Definition ctx_call (x : _) (m : _) : econtext :=
  (fun e => ECall x m e).
```

To capture which functions are valid evaluation contexts, we define a proposition `is_econtext`, which is used by all definitions which reason about evaluation contexts (the snippet below shows the dynamic rule DYN-EVAL-CONTEXT):

```
Inductive is_econtext : econtext -> Prop :=
  | EC_Call :
      forall x m,
        is_econtext (ctx_call x m)
  | ...

Inductive step (P : program) :
  configuration -> configuration -> Prop :=
  | ...
  | EvalContext :
      forall H H' V V' n n' E e e' Ls Ls',
        is_econtext E ->
        P / (H, V, n, T_Thread Ls e) ==>
          (H', V', n', T_Thread Ls' e') ->
        P / (H, V, n, T_Thread Ls (E e)) ==>
          (H', V', n', T_Thread Ls' (E e'))
  | ...
```

When performing case analysis over which `step` rules are applicable, Coq sometimes generates absurd cases where the an invalid evaluation context is applied. To handle these cases automatically, we define tactics for unfolding (applying) all evaluation contexts in scope and finding impossible equalities in the assumptions (`context[e]` is the Coq notation for matching any term with `e` in it):

```
Ltac unfold_ctx :=
  match goal with
    | [H: context[ctx_call] |- _] =>
      unfold ctx_call in H
    | [_ : _ |- context[ctx_call]] =>
      unfold ctx_call
    | ...
  end.

Ltac malformed_context :=
  match goal with
    | [Hctx : is_econtext ?ctx |- _] =>
      inv Hctx; repeat(unfold_ctx);
        try(congruence)
    | _ =>
      fail 1 "could not prove malformed context"
  end.
```

The tactic `malformed_context` tries to find an instance of `is_econtext` in the current assumptions, inverts it (`inv H` is defined as `(inversion H; subst; clear H)`), unfolds all evaluation contexts in scope and then uses `congruence` to dismiss all subgoals with absurd equalities in the assumptions. In proofs where case-analysis of the step relation is needed, the tactic `inv Hstep; try malformed_context` is used to only keep the sane cases around.

7. TYPESETTING OOlong

The paper version of OOlong is written in Ott [25], which lets a user define the grammar and typing rules of their semantics using ASCII-syntax. The rules are checked against the grammar to make sure that the syntax is consistent. Ott can then generate LaTeX code for these rules, which when typeset appear as in this paper. The Ott sources are available in the same repo as the Coq sources [5]. As an example, here is the Ott version of the rule WF-LET:

```
G |- e1 : t1
G, x : t1 |- e2 : t
----------------------------- :: let
G |- let x = e1 in e2 : t
```

The LaTeX rendering of `G` as Γ and `|-` as $\vdash$ is defined elsewhere in the Ott file, and the rule is rendered as:

$$\text{WF-LET}\quad \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma, x : t_1 \vdash e_2 : t}{\Gamma \vdash \textbf{let}\, x = e_1 \,\textbf{in}\, e_2 : t}$$

It is also possible to have Ott generate LaTeX code for the grammar, but these tend to require more whitespace than one typically has to spare in an article. We therefore include LaTeX code for a more compact version of the grammar, as well as the definitions of progress and preservation [5]. Ott also supports generating Coq and Isabelle/HOL code from the same definitions that generate LaTeX code. We have not used this feature as we think it is useful to let the paper version of the semantics abstract away some of the details that a mechanised version requires.

8. EXTENSIONS TO THE SEMANTICS

This section demonstrates the extensibility of OOlong by adding assertions, region based locking, and type-based null reference tracking to the semantics. They are chosen as examples of adding new expressions, adding new runtime constructs, and extending the type system respectively. Here we only describe the additions necessary, but these features have also been added to the mechanised version of the semantics with little added complexity to the code. They are all available as examples on how to extend the mechanised semantics [5].

8.1 Supporting Assertions

Assertions are a common way to enforce pre- and postconditions and to fail fast if some condition is not met. We add support for assertions in OOlong by adding an expression **assert(x == y)**, which asserts that two variables are aliases (if we added richer support for primitives we could let the argument of the assertion be an arbitrary boolean expression). If an assertion fails, we throw an **AssertionException**. The typing rule for assertions states that the two variables are of the same type. The type of an assertion is **Unit**.

$$\textsc{wf-assert}\quad \frac{\Gamma(x) = t \qquad \Gamma(y) = t}{\Gamma \vdash \textbf{assert}\,(x == y) : \textbf{Unit}}$$

In the dynamic semantics, we have two outcomes of evaluating an assertion: if successful, the program continues; if not, the program should crash.

$$\textsc{dyn-eval-assert}\quad \frac{V(x) = V(y)}{\langle H; V; (\mathcal{L}, \textbf{assert}\,(x == y))\rangle \hookrightarrow \langle H; V; (\mathcal{L}, \textbf{null})\rangle}$$

$$\textsc{dyn-exn-assert}\quad \frac{V(x) \neq V(y)}{\langle H; V; (\mathcal{L}, \textbf{assert}\,(x == y))\rangle \hookrightarrow \langle H; V; \textbf{AssertionException}\rangle}$$

Note that the rules for exceptions already handle exception propagation, regardless of the kind of exception (*cf.,* Section A.2).

In the mechanised semantics, the automated tactics are powerful enough to automatically solve the additional cases for almost all lemmas. The additional cases in the main theorems are easily dispatched. This extension adds a mere ∼50 lines to the mechanisation.

8.2 Supporting Region-based Locking

Having a single lock per object prevents threads from concurrently updating disjoint parts of an object, even though this is benign from a data-race perspective. Many effect-systems divide the fields of an object into *regions* in order to reason about effect disjointness on a single object (*e.g.,* [4]). Similarly, we can add regions to OOlong, let each field belong to a region and let each region have a lock of its own. Syntactically, we add a region annotation to field declarations ("$f : t$ **in** r") and require that taking a lock specifies which region is being locked ("$\textbf{lock}(x, r)$ **in** e"). Here we omit declaring regions and simply consider all region names valid. This means that the rules for checking well-formedness of fields do not need updating (other than the syntax).

Dynamically, locks are now identified not only by the location ι of their owning object, but also by their region r. Objects need to be extended from having one lock to having multiple locks, each with its own lock status. We model this by replacing the lock status of an object with a region map RL from region names to lock statuses. As an example, the dynamic rule for grabbing a lock for a region is updated thusly:

$$\textsc{dyn-eval-lock-region}\quad \frac{\begin{array}{c} V(x) = \iota \qquad H(\iota) = (C, F, RL) \qquad RL(\mathbf{r}) = \textbf{unlocked} \\ (\iota, \mathbf{r}) \notin \mathcal{L} \qquad \mathcal{L}' = \mathcal{L} \cup \{(l, r)\} \\ H' = H[\iota \mapsto (C, F, RL[\mathbf{r} \mapsto \textbf{locked}])] \end{array}}{\langle H; V; (\mathcal{L}, \textbf{lock}(x, \mathbf{r})\,\textbf{in}\,e)\rangle \hookrightarrow \langle H'; V; (\mathcal{L}', \textbf{locked}_{(\iota,\mathbf{r})}\{e\})\rangle}$$

Similarly, the well-formedness rules for locking need to be updated to refer to region maps of objects instead of just objects. A region map must contain a mapping for each region used in the object:

$$\textsc{wf-regions}\quad \frac{\forall f : t\ \textbf{in}\ \mathbf{r} \in \textbf{fields}\,(C).\mathbf{r} \in \textbf{dom}(RL)}{C \vdash RL}$$

The changes can mostly be summarised as adding one extra level of indirection each time a lock status is looked up on the heap. The same is true for the mechanised semantics. For example, in the original mechanisation, the lemma showing that stepping a thread cannot cause it to take a lock that is already locked by another thread looks like this:

```
Lemma noDuplicatedLocks :
  forall P t' Gamma l H H' V V' n n' T T' t c F,
    wfProgram P t' ->
    heapLookup H l = Some (c, F, LLocked) ->
    ~ In l (t_locks T) ->
    wfConfiguration P Gamma (H, V, n, T) t ->
    P / (H, V, n, T) ==> (H', V', n', T') ->
    ~ In l (t_locks T').
```

The function `t_locks` extracts the locks held by a thread `T`. In the version extended with region locks, the same lemma instead looks like this:

```
Lemma noDuplicatedLocks : forall
  P t' Gamma l r H H' V V' n n' T T' t c F RL,
    wfProgram P t' ->
    heapLookup H l = Some (c, F, RL) ->
    RL r = Some LLocked ->
    ~ In (l, r) (t_locks T) ->
    wfConfiguration P Gamma (H, V, n, T) t ->
    P / (H, V, n, T) ==> (H', V', n', T') ->
    ~ In (l, r) (t_locks T').
```

Notice that instead of having a single taken lock, the object looked up on the heap has a region map `RL` whose lock related to region `r` is taken. The proof of the lemma is the same as before, except for one extra inversion and an additional call to `congruence`.

This extension increases the size of the mechanised semantics by ∼130 lines.

8.3 Supporting Nullable Types

Null pointers are famously referred to by Tony Hoare as his "billion dollar mistake", referring to the fact that accidentally dereferencing null pointers has been the cause of many bugs since their introduction in the 1960s [13]. One way to reduce the risk of these errors is to have the type system track which references may be **null** during runtime. This section introduces such "nullable types" to OOlong.

We start by extending the syntax of types:

$$t ::= C \mid I \mid C? \mid I? \mid \textbf{Unit}$$

The types $C?$ and $I?$ are given to references which could be **null** valued. In the mechanisation, class and interface types are extended with a boolean flag which tracks if the type is nullable or not:

```
Inductive ty : Type :=
  | TClass : class_id -> bool -> ty
  | TInterface : interface_id -> bool -> ty
  | TUnit : ty.
```

The subtyping rules are extended to allow non-nullable references to flow into nullable ones, but not the other way around:

$$\textsc{t-sub-n}\quad \frac{t <: t'}{t? <: t'?} \qquad\qquad \textsc{t-sub-n-r}\quad \frac{t <: t'}{t <: t'?}$$

Finally, the type checking rule for **null** is updated to disallow non-nullable types (**nullable**(t) is defined to be true for all types $t?$ and for **Unit**):

$$\textsc{wf-nullable}\quad \frac{\vdash \Gamma \qquad \vdash t \qquad \textbf{nullable}\,(t)}{\Gamma \vdash \textbf{null} : t}$$

For simplicity in this presentation, since there are no constructors in OOlong, we require that all field types are nullable (since fields are initialised to **null**). Lifting this restriction is straightforward, for example by providing a list of initial field values when creating new objects: **new** $C(e_1, \ldots, e_n)$.

The mechanised semantics grows by ∼100 lines with the type-level additions shown above. The extension of the subtyping rules requires some proofs to be extended to handle these cases and some minor lemmas to be added (*e.g.,* that if $t_1 <: t_2$ and **nullable**(t_1), then **nullable**(t_2)). Allowing non-nullable types on the heap by adding constructors is possible, but is complicated by Coq's inability to generate effective induction principles for nested data types (*cf.,* [8], Chapter 3.8); the

```
interface Counter {
  add(x : int) : Unit
  get(tmp : int) : int
}
class Cell implements Counter {
  cnt : int
  def init(n : int) : Unit {
    this.cnt = n
  }
  def add(n : int) : Unit {
    let cnt = this.cnt in
      this.cnt = (cnt + n)
  }
  def get(tmp : int) : int {
    this.cnt
  }
}
let cell = new Cell in
let cell2 = (Counter) cell in // (†)
let tmp = cell.init(0) in     // (‡)
finish {
  async {
    lock(cell) in cell.add(1)
  }
  async {
    lock(cell2) in cell2.add(2)
  }
};
cell.get(0)
```

Figure 10: An OOlong program, with added integer support.

new expression would contain a list of expressions, for which no induction hypotheses are automatically generated during proofs by induction. Mechanising this extension with hand-written induction principles is left as future work.

9. PROTOTYPE INTERPRETER

In addition the formalised semantics of OOlong, we have implemented a simple interpreter for the language. The purpose of this implementation is twofold: to provide an actual executable semantics, and to minimise the effort of prototyping a future extension of OOlong. It is provided together with the Ott and Coq sources in the OOlong repository [5].

The full implementation is ∼760 lines of OCaml, accompanied by ∼190 lines of comments and documentation. Lexing and parsing (∼90 lines) is implemented using `ocamllex` [15] and Menhir [22]. Type checking (∼190 lines) is based on a bidirectional version of the typing rules which makes typing entirely syntax-directed [21].

Figure 11 shows the bidirectional typing rules. Bidirectional type checking differentiates between *inferring* a type for an expression ($\Gamma \vdash e \Rightarrow t$) and *checking* that an expression has a given expression ($\Gamma \vdash e \Leftarrow t$). Most rules mirror the corresponding rules in Figure 5, but make explicit which types are inferred and which are checked against an existing type. The frame-rule in WF-FJ is exchanged for a requirement

$\boxed{\Gamma \vdash e \Rightarrow t \quad \Gamma \vdash e \Leftarrow t}$ *(Inference/Checking)*

$$\text{BD-INFER-VAR}\quad \frac{\vdash \Gamma \qquad \Gamma(x) = t}{\Gamma \vdash x \Rightarrow t} \qquad \text{BD-INFER-LET}\quad \frac{\Gamma \vdash e_1 \Rightarrow t_1 \qquad \Gamma, x : t_1 \vdash e_2 \Rightarrow t}{\Gamma \vdash \textbf{let}\, x = e_1 \textbf{ in } e_2 \Rightarrow t}$$

$$\text{BD-INFER-CALL}\quad \frac{\Gamma \vdash x \Rightarrow t_1 \qquad \Gamma \vdash e \Leftarrow t_2 \qquad \textbf{msigs}\,(t_1)(m) = y : t_2 \to t}{\Gamma \vdash x.m(e) \Rightarrow t} \qquad \text{BD-INFER-CAST}\quad \frac{\Gamma \vdash e \Leftarrow t}{\Gamma \vdash (t)e \Rightarrow t}$$

$$\text{BD-INFER-SELECT}\quad \frac{\Gamma \vdash x \Rightarrow C \qquad \textbf{fields}\,(C)(f) = t}{\Gamma \vdash x.f \Rightarrow t} \qquad \text{BD-INFER-UPDATE}\quad \frac{\Gamma \vdash x \Rightarrow C \qquad \Gamma \vdash e \Leftarrow t \qquad \textbf{fields}\,(C)(f) = t}{\Gamma \vdash x.f = e \Rightarrow \textbf{Unit}}$$

$$\text{BD-INFER-NEW}\quad \frac{\vdash \Gamma \qquad \vdash C}{\Gamma \vdash \textbf{new}\, C \Rightarrow C} \qquad \text{BD-INFER-LOCK}\quad \frac{\Gamma \vdash x \Rightarrow t_2 \qquad \Gamma \vdash e \Rightarrow t}{\Gamma \vdash \textbf{lock}(x)\,\textbf{in}\, e \Rightarrow t}$$

$$\text{BD-INFER-FJ}\quad \frac{fv(e_1) \cap fv(e_2) = \emptyset \qquad \Gamma \vdash e_1 \Rightarrow t_1 \qquad \Gamma \vdash e_2 \Rightarrow t_2 \qquad \Gamma \vdash e \Rightarrow t}{\Gamma \vdash \textbf{finish}\ \{\ \textbf{async}\ \{\ e_1\ \}\ \textbf{async}\ \{\ e_2\ \}\ \}\,;\, e \Rightarrow t}$$

$$\text{BD-CHECK-NULL}\quad \frac{\vdash \Gamma \qquad \vdash t}{\Gamma \vdash \textbf{null} \Leftarrow t} \qquad \text{BD-CHECK-SUB}\quad \frac{\Gamma \vdash e \Rightarrow t' \qquad t' <: t \qquad e \neq \textbf{null}}{\Gamma \vdash e \Leftarrow t}$$

Figure 11: Bidirectional typing rules.

that the free variables in two parallel **async**s are disjoint (BD-INFER-FJ). Notably, the type of **null** is never inferred, but can be given any well-formed type (BD-CHECK-NULL). Checking any other expression against some type t amounts to inferring a type t' and seeing if t' is a subtype of t (BD-CHECK-SUB). We omit rules for syntax-directed subtyping, which is implemented as a simple traversal of the interface hierarchy. Since we only ever type check static programs, there are no rules for the dynamic expressions ι and $\textbf{locked}_\iota\{e\}$.

The actual evaluation of programs (~290 lines) is a more or less direct translation of the dynamic rules in Section 4. The interpreter evaluates entire configurations one step a time, until the program terminates or deadlocks. Non-terminating programs (including programs with livelocks) are not detected by the interpreter, but will run forever. Non-deterministic choice is implemented using a "scheduler" function which decides whether to run the left or the right thread in a fork. This function is customisable, allowing for deterministic scheduling as well. There is no parallelism in the interpreter itself.

To allow for *slightly* more interesting programs to be written, we have also extended the interpreter with support for integers and addition. This extension adds ~50 lines of code. Other than the obvious additions to parsing, type checking and evaluation, the typing rule BD-CHECK-NULL must also be updated with the premise $t \neq \text{int}$ so that **null** does not inhabit the integer type. Figure 10 shows an example program written with this extension. Ignoring the integers, it is also a syntactically correct OOlong program according to the formal semantics. It shows some of the ways that the implementation (and formalism) is kept simple. For example, a function must take exactly one argument, even when it is not used (*cf.,* method `get`). Similarly, there is no sequencing without variable binding, so the **Unit** result of `cell.init(0)` (‡) must be bound to a variable. Note also that the `Cell` object must be aliased (†) in order to be used by both threads in the subsequent fork (the upcast to `Counter` is not necessary, but is included to show all features of the language). All of these things are of course easily remedied, but the point of the interpreter is not to provide a smooth programmer experience, but to keep the *implementation* simple.

Interpreting the program in Figure 10 gives the following output:

```
Ran for 31 steps, resulting in
([], 3)
Heap:
{
 0 -> (Cell, {cnt -> 3}, Unlocked)
}
Variables:
{
 cell -> 0
 cell2 -> 0
 cnt -> 0
 cnt#4 -> 1
 n -> 0
 n#0 -> 1
 n#2 -> 2
 this -> 0
 this#1 -> 0
 this#3 -> 0
 this#6 -> 0
 tmp -> null
 tmp#5 -> 0
}
```

The result `([], 3)` represents a single thread with an empty set of held locks and the expression `3` (*cf., T* in Figure 6). The heap maps integers ("addresses") to objects, in this case a single unlocked `Cell` object, with a field `cnt` of value 3. The "stack" is represented as a map from variables to values ("addresses", integers, or **null**). Since variable names are never recycled, fresh variable names are sometimes generated to avoid clashes (*e.g.,* `this#3`). These names may differ between different runs due to non-determinism in the scheduling of threads. For debugging purposes, the interpreter supports printing a representation of the state as above after every evaluation step.

The reason for implementing the interpreter in OCaml rather than in Coq, where correspondence to the formal semantics could be proven directly, is partly to be able to make the implementation simpler and partly to lower the threshold for a potential user. An OOlong program is not guaranteed to terminate, and since Coq requires all functions to be total, writing an interpreter would require tricks to work around this (*e.g.,* limiting evaluation to a maximum number of steps). We also believe that it is easier for someone without prior experience to approach OCaml than it is to approach Coq. We leave implementing a formally verified interpreter in Coq for future work.

10. CONCLUSION

We have presented OOlong, an object calculus with concurrency and locks, with a focus on extensibility. OOlong aims to model the most important details of concurrent object-oriented programming, but also lends itself to extension and modification to cover other topics. A good language calculus should be both reliable and reusable. By providing a mechanised formalisation of the semantics, we reduce the leap of faith needed to trust the calculus, and also give a solid starting point for anyone wanting to extend the calculus in a rigorous way. Using Ott makes it easy to extend the calculus on paper and get usable LaTeX figures without having to spend time on manual typesetting. The prototype implementation offers an entry point for the more engineering-oriented researcher looking to experiment with new language features.

We have found OOlong to be a useful and extensible calculus, and by making our work available to others we hope that we will help save time for researchers looking to explore concurrent object-oriented languages in the future.

11. ACKNOWLEDGMENTS

We thank the anonymous reviewers for helpful feedback and suggestions for the original paper. The project is financially supported by the Swedish Foundation for Strategic Research (FFL15-0032), and by the Swedish Research Council Project 2014-05545, SCADA: Scalable Data for Pervasive Parallelism

12. REFERENCES

[1] N. Amin, S. Grütter, M. Odersky, T. Rompf, and S. Stucki. The Essence of Dependent Object Types. In *A List of Successes That Can Change the World*. Springer, 2016.

[2] A. Banerjee and D. A. Naumann. Secure Information Flow and Pointer Confinement in a Java-like Language. In *CSFW*, volume 2, page 253, 2002.

[3] G. M. Bierman, M. Parkinson, and A. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical report, University of Cambridge, Computer Laboratory, 2003.

[4] R. Bocchino. An Effect System And Language For Deterministic-By-Default Parallel Programming, 2010. PhD thesis, University of Illinois at Urbana-Champaign.

[5] E. Castegren. Coq and Ott sources for OOlong. `https://github.com/EliasC/oolong`, 2018. GitHub.

[6] E. Castegren and T. Wrigstad. Reference Capabilities for Concurrency Control. In *ECOOP*, 2016.

[7] E. Castegren and T. Wrigstad. OOlong: An Extensible Concurrent Object Calculus. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 1022–1029. ACM Press, 2018.

[8] A. Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013.

[9] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *ACM SIGPLAN Notices*, volume 33, pages 48–64. ACM, 1998.

[10] B. Delaware, W. R. Cook, and D. Batory. Fitting the pieces together: a machine-checked model of safe composition. In *ESEC-FSE*. ACM, 2009.

[11] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *ACM SIGPLAN Notices*, volume 35, pages 219–232. ACM, 2000.

[12] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 171–183. ACM, 1998.

[13] T. Hoare. Null References: The Billion Dollar Mistake, 2009. Talk at QCon.

[14] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *TOPLAS*, 23(3), 2001.

[15] INRIA. Lexer and parser generators (ocamllex, ocamlyacc). `https://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html`, 2018. Tool tutorial.

[16] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *TOPLAS*, 28(4), 2006.

[17] J. Mackay, H. Mehnert, A. Potanin, L. Groves, and N. Cameron. Encoding Featherweight Java with assignment and immutability using the Coq proof assistant. In *FTfJP*, 2012.

[18] J. Östlund and T. Wrigstad. Welterweight Java. In *TOOLS*. Springer, 2010.

[19] A. M. Peters, D. Kitchin, J. A. Thywissen, and W. R. Cook. OrcO: a concurrency-first approach to objects. In *OOPSLA*, 2016.

[20] B. C. Pierce, A. A. de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hritcu, V. Sjöberg, and B. Yorgey. *Software Foundations*. Electronic textbook, 2017. Version 5.0.

[21] B. C. Pierce and D. N. Turner. Local Type Inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000.

[22] F. Pottier and Y. Régis-Gianas. What is Menhir? `http://gallium.inria.fr/~fpottier/menhir/`, 2018. Tool website.

[23] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, volume 46. ACM, 2011.

[24] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable Units of Behaviour. In L. Cardelli, editor, *ECOOP 2003*, volume 2743 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003.

[25] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: Effective Tool Support for the Working Semanticist. In *ICFP*, 2007.

[26] R. Strňisa. *Formalising, improving, and reusing the Java Module System*. PhD thesis, St. John's College, United Kingdom, May 2010.

[27] M. van Dooren and W. Joosen. A modular type system for first-class composition inheritance, 2009.

APPENDIX

A. OMITTED RULES

This appendix lists the rules for deadlocked states, exception propagation, and the helper functions used in the main article. They should all be unsurprising but are included for completeness.

A.1 Blocking

The blocking property of a configuration holds if all its threads are either blocking on a lock or are done (*i.e.,* have reduced to a value). This property is necessary to distinguish deadlocks from stuck states.

$\boxed{\mathit{Blocked}(\mathit{cfg})}$ *(Configuration is blocked)*

BLOCKED-LOCKED
$$\frac{V(x) = \iota \quad H(\iota) = (C, F, \mathbf{locked}) \quad \iota \notin \mathcal{L}}{\mathit{Blocked}(\langle H;\, V;\, (\mathcal{L}, \mathbf{lock}(x)\,\mathbf{in}\,e)\rangle)}$$

BLOCKED-DEADLOCK
$$\frac{\mathit{Blocked}(\langle H;\, V;\, T_1\rangle) \quad \mathit{Blocked}(\langle H;\, V;\, T_2\rangle)}{\mathit{Blocked}(\langle H;\, V;\, T_1 \mid\mid T_2 \rhd e\rangle)}$$

BLOCKED-LEFT
$$\frac{\mathit{Blocked}(\langle H;\, V;\, T_1\rangle)}{\mathit{Blocked}(\langle H;\, V;\, T_1 \mid\mid (\mathcal{L}, v) \rhd e\rangle)}$$

BLOCKED-RIGHT
$$\frac{\mathit{Blocked}(\langle H;\, V;\, T_2\rangle)}{\mathit{Blocked}(\langle H;\, V;\, (\mathcal{L}, v) \mid\mid T_2 \rhd e\rangle)}$$

BLOCKED-CONTEXT
$$\frac{\mathit{Blocked}(\langle H;\, V;\, (\mathcal{L}, e)\rangle)}{\mathit{Blocked}(\langle H;\, V;\, (\mathcal{L}, E[e])\rangle)}$$

A.2 Exceptions

Exceptions terminate the entire program and cannot be caught. The only rule that warrants clarification is the rule for exceptions in evaluation contexts which abstracts the nature of an underlying exception to avoid rule duplication (DYN-EXCEPTION-CONTEXT). For readability we abbreviate **NullPointerException** as **NPE**. When we don't care about the kind of exception we write **EXN**.

$\boxed{\mathit{cfg}_1 \hookrightarrow \mathit{cfg}_2}$ *(Exceptions)*

DYN-NPE-SELECT
$$\frac{V(x) = \mathbf{null}}{\langle H;\, V;\, (\mathcal{L}, x.f)\rangle \hookrightarrow \langle H;\, V;\, \mathbf{NPE}\rangle}$$

DYN-NPE-UPDATE
$$\frac{V(x) = \mathbf{null}}{\langle H;\, V;\, (\mathcal{L}, x.f = v)\rangle \hookrightarrow \langle H;\, V;\, \mathbf{NPE}\rangle}$$

DYN-NPE-CALL
$$\frac{V(x) = \mathbf{null}}{\langle H;\, V;\, (\mathcal{L}, x.m(v))\rangle \hookrightarrow \langle H;\, V;\, \mathbf{NPE}\rangle}$$

DYN-NPE-LOCK
$$\frac{V(x) = \mathbf{null}}{\langle H;\, V;\, (\mathcal{L}, \mathbf{lock}(x)\,\mathbf{in}\,e)\rangle \hookrightarrow \langle H;\, V;\, \mathbf{NPE}\rangle}$$

DYN-EXCEPTION-ASYNC-LEFT
$$\frac{}{\langle H;\, V;\, \mathbf{EXN} \mid\mid T_2 \rhd e\rangle \hookrightarrow \langle H;\, V;\, \mathbf{EXN}\rangle}$$

DYN-EXCEPTION-ASYNC-RIGHT
$$\frac{}{\langle H;\, V;\, T_1 \mid\mid \mathbf{EXN} \rhd e\rangle \hookrightarrow \langle H;\, V;\, \mathbf{EXN}\rangle}$$

DYN-EXCEPTION-CONTEXT
$$\frac{\langle H;\, V;\, (\mathcal{L}, e)\rangle \hookrightarrow \langle H';\, V';\, \mathbf{EXN}\rangle}{\langle H;\, V;\, (\mathcal{L}, E[e])\rangle \hookrightarrow \langle H';\, V';\, \mathbf{EXN}\rangle}$$

A.3 Helper Functions

This section presents the helper functions used in the formalism. Helpers **methods** and **fields** are analogous to **msigs**, and we refer to the mechanised semantics for details [5].

$$\mathbf{vardom}(\Gamma) = \{x \mid x \in \mathbf{dom}(\Gamma)\}$$

$$\mathbf{msigs}(I) = \begin{cases} \mathit{Msigs} & \text{if } \mathbf{interface}\ I\{\mathit{Msigs}\} \in P \\ \mathbf{msigs}(I_1) \cup \mathbf{msigs}(I_2) & \text{if } \mathbf{interface}\ I\ \mathbf{extends}\ I_1, I_2 \in P \end{cases}$$

$$\mathbf{msigs}(C) = \{\mathit{Msig} \mid \mathbf{def}\ \mathit{Msig}\{e\} \in \mathit{Mds}\} \text{ if } \mathbf{class}\ C ...\{\_\ \mathit{Mds}\} \in P$$

$$\mathbf{msigs}(t)(m) = x : t_1 \to t_2 \quad \text{if} \quad m(x : t_1) : t_2 \in \mathbf{msigs}(t)$$

$$\mathbf{heldLocks}(T) = \begin{cases} \mathcal{L} & \text{if } T = (\mathcal{L}, e) \\ \mathbf{heldLocks}(T_1) \cup \mathbf{heldLocks}(T_2) & \text{if } T = T_1 \mid\mid T_2 \rhd e \end{cases}$$

$$\mathbf{locks}(e) = \{\iota \mid \mathbf{locked}_\iota\{e'\} \in e\}$$

$$\mathit{distinctLocks}(e) \equiv |\mathbf{locks}(e)| = |\mathbf{lockList}(e)|$$
where $\mathbf{lockList}(e) = [\iota \mid \mathbf{locked}_\iota\{e'\} \in e]$

ABOUT THE AUTHORS:

Elias Castegren is currently a postdoc at KTH Royal Institute of Technology in Stockholm, working in the Model-based computing systems group with David Broman, developing tools for creating efficient and composable domain specific languages. He received his PhD from Uppsala University, working with Tobias Wrigstad developing type systems for concurrency control. After his PhD, he worked at Microsoft Research in Cambridge for a three month internship on compiler-based side-channel mitigation together with Matthew Parkinson. Elias is a member of ACM and SIGPLAN and has served on artifact evaluation committees for PLDI, OOPSLA, and ECOOP.

Tobias Wrigstad is a lecturer in computer science at Uppsala University. His main topics are programming languages and software engineering, and the interaction between them. He is currently excited by scripting languages, concurrent and parallel garbage collection, and types for lock-free programming, data layout and data-race freedom. Dr. Wrigstad is a member of the ACM, SIGPLAN, SIGCSE and AITO. He regularly serves on the program committees of conferences like OOPSLA and ECOOP, and has been organizer of conferences and summer schools in Sweden as well as internationally. He received the Dahl-Nygaard Junior Prize in 2012, a teacher of the year award in 2017, and was appointed excellent teacher in 2018.