Universidade Federal de Pernambuco

GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

CENTRO DE INFORMÁTICA

2012.2

Um estudo sobre a transição para arquiteturas multicore em aplicações de código aberto

Trabalho de Graduação

Aluno: Rafael Brandão Lôbo {rbl@cin.ufpe.br}
Orientador: Fernando Jose Castor de Lima Filho {fjclf@cin.ufpe.br}

Recife, 24 de fevereiro de 2013

Universidade Federal de Pernambuco

GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

CENTRO DE INFORMÁTICA

2012.2

Um estudo sobre a transição para arquiteturas multicore em aplicações de código aberto

Trabalho de Graduação

Trabalho de graduação apresentado no Centro de Informática da Universidade Federal de Pernambuco por Rafael Brandão Lôbo, orientado por Fernando Jose Castor de Lima Filho como requisito para a obtenção do Grau de Bacharel em Ciência da Computação.

Aluno: Rafael Brandão Lôbo {rbl@cin.ufpe.br}
Orientador: Fernando Jose Castor de Lima Filho {fjclf@cin.ufpe.br}

Recife, 24 de fevereiro de 2013

Agradecimentos

Agradeço, em primeiro lugar, aos meus pais, pois fizeram um excelente trabalho como pais e são e, grande parte, responsáveis pelo o que eu sou hoje. Agradeço à tia Rosemary que me deixou usar sua internet, permitindo que eu me motivasse a concluir o desenvolvimento de um jogo de pokémon para IRC usado até hoje, essencial para minha formação antes mesmo da minha entrada na faculdade. Agradeço à minha família tão incrível que me ajudou durante todos esses complicados anos de faculdade; só estou aqui hoje porque muitas pessoas acreditaram em mim e, em especial, financiaram a minha permanência em Recife durante os primeiros anos. Em especial, gostaria de agradecer à tia Ednólia que me comove sempre que lembro do apoio que ela deu, à tia Têca e ao tio Ermando pelo contínuo apoio; à minha avó Adalgisa que sempre apoiou mas que infelizmente não está mais presente; à tia Lisieux que me ajudou, mesmo em dificuldade, e sempre me comovia nas despedidas com as orações e carinho dela, entre vários outros familiares. Agradeço ao meu tio Raimundo Brandão Filho que me deu de presente um notebook quando vim morar em Recife, essencial durante todo o curso; ao primo Tayronni por ter me abrigado em sua casa no primeiro semestre da faculdade; ao colega/primo Davi Duarte que está me seguindo desde o jardim de infância (brincadeira!) e hoje moramos juntos em Recife. Aos colegas damaratona de programação que estiveram convivendo comigo nos últimos anos ensinando o quão importante e divertido resolução de problemas pode ser, em especial a Luiz Afonso, Adriana Libório, Arthur Alem, Filipe Martins, Pedro Bello e Hallan Cosmo. Aos colegas do Instituto Nokia de Tecnologia que me ensinaram a importância de se trabalhar com projetos de código aberto e, em especial, a Daker Fernandes, por ter me dado a oportunidade de conhecê-los e pelo convite de participação no projeto Castor. Ao professor Fernando Castor por sua excelente didática de ensino em assuntos tão complicados como concorrência e por motivar o desenvolvimento desse trabalho. E, finalmente, à minha namorada, Carolina Linard, por estar continuamente me aguentando e dando apoio, o que é algo realmente impressionante!

Abstract

Multicore processors get more popular with time and demand that applications make better use of parallelism internally to get more power from this technology otherwise its success may be compromised. Such adoption to parallelism requires profound changes in the architecture of most applications and also a revolution in the way of thinking of software design and development. In this study, it's done an analysis of open source projects that have been changing its own structure to deliver more parallelism, such as WebKit, Qt and Lucene, and observe the motivation behind the changes, project limitations before the changes and in particular the impact they have caused to project's stability. It is also observed ways to reduce this impact, generally associated to test-driven development and open discussions with the project community about the proposed changes, potentially delivering a smoother and less error-prone transition.

Keywords: parallelism, concurrency, multicore, open source, WebKit, Qt, Lucene, lockless programming, non-blocking algorithms, test-driven development.

Resumo

Processadores *multicore* estão cada vez mais populares e demandam que aplicações façam uso de paralelismo internamente para poder aproveitar o poder computacional oferecido pela tecnologia, caso contrário seu sucesso estará comprometido. Essa adesão a uma arquitetura mais paralela requer profundas mudanças na arquitetura da maioria das aplicações existentes e uma revolução na forma como a maioria dos softwares são projetados e desenvolvidos hoje em dia. Neste estudo, uma análise é feita em projetos de código aberto que tem sofrido modificações em sua estrutura para tornar-se mais paralelo, como WebKit, Qt e Lucene, e observar a motivação por trás das mudanças, limitações do projeto antes das mudanças e o impacto que as modificações causaram, em particular, em sua estabilidade. Também é observado formas de redução desse impacto, geralmente associadas a desenvolvimento dirigido por testes e a discussões abertas com a comunidade sobre as modificações propostas, levando a uma transição com menor risco de falhas críticas.

Palavras-chave: paralelismo, concorrência, multicore, código aberto, WebKit, Qt, Lucene, programação concorrente sem locks, algoritmos não-bloqueantes, desenvolvimento dirigido por testes.

Sumário

Índice de Figuras

Índice de Quadros

Capítulo 1 – Introdução

A crescente popularização de processadores *multicore* levanta a necessidade de aplicações que fazem cada vez mais uso de paralelismo internamente; caso contrário, grande parte do poder computacional atribuído a esses processadores será desperdiçado. Esse interesse também é compartilhado pelas empresas que fabricam esses processadores, pois elas entendem que o sucesso dessa tecnologia está diretamente associado ao poder que ela efetivamente dará a aplicações do cotidiano.

Vários projetos de código aberto, ou *open source*, já começaram a adotar mais paralelismo internamente, dada a crescente demanda. É importante aprender com esses projetos porque eles refletem casos reais e atuais de paralelização e podem ensinar bastante sobre o processo de transição para uma arquitetura mais paralela, foco de estudo deste trabalho.

Sabe-se através do histórico de aprendizado de concorrência que esse assunto é muito complicado de ser compreendido por grande parte das pessoas e mesmo especialistas admitem que se trata de um problema complicado, exigindo uma revolução na forma como softwares são desenvolvidos. Pesquisadores estão atualmente se esforçando para tentar compreender qual é a melhor maneira de tornar a maioria dos programadores mais produtivos nestas arquiteturas.

No entanto, concorrência nem sempre é sinônimo de paralelismo; é possível que uma aplicação com um design não apropriado de paralelismo não faça o uso devido desses processadores e possivelmente comprometa sua estabilidade; portanto, é relevante aprender com projetos reais o design de paralelismo escolhido e o impacto que as modificações causaram no projeto.

Objetivos

Objetivos Gerais

Este projeto tem como objetivo geral identificar em projetos *open source* (de código aberto) o grau de complexidade envolvido ao tornar um projeto mais paralelo através da análise, em particular, do impacto das modificações, como identificação de módulos afetados e a frequência de bugs ocorridos após as mudanças.

Objetivos Específicos

Para alcançar o objetivo geral, foi identificado a necessidade de, em cada projeto:

- entender o contexto em que o projeto está inserido;
- entender a estrutura do projeto antes das modificações e verificar suas limitações para então apresentar a motivação que justificaram as modificações propostas;
- analisar o impacto das mudanças em termos de modularidade, se elas afetaram regiões limitadas do código ou não, e utilização de métricas, como linhas de código ou tamanhos de arquivos;
- verificar a frequência e os tipos de bugs encontrados após as modificações propostas terem sido realizadas;
- checar se os resultados, como performance, responsividade, e assim por diante, foram melhorados de modo a corresponder expectativas.

Estrutura do Trabalho

Este documento está dividido em capítulos, sendo este o primeiro capítulo. No segundo capítulo, será contextualizado a necessidade de aplicações darem melhor suporte a processadores com vários núcleos e o que está sendo pesquisado na academia hoje para melhorar o suporte a esta tecnologia, além da apresentação dos termos ao leitor conforme necessário. No terceiro capítulo, será explicado as principais

diferenças entre concorrência e paralelismo e como tirar melhor proveito de paralelismo com programação concorrente. Os capítulos quatro, cinco e seis tratam respectivamente da análise do WebKit, Qt e Lucene, projetos de código aberto sob análise neste estudo. No capítulo sete é feita a conclusão deste trabalho com uma análise crítica deste projeto e com sugestões de trabalhos futuros.

Capítulo 2 – Revolução em Paralelismo

A evolução dos processadores sofreu uma redução na sua velocidade de desenvolvimento, se analisado apenas a velocidade de seu clock, durante a última década. Esse crescimento vinha acompanhando a lei de Moore, que estimava um aumento em duas vezes do número de transistores em circuitos integrados a cada dezoito meses. O motivo por trás dessa desaceleração é tecnológica: velocidades maiores no clock dificultam resfriamento e reduzem sua eficiência energética. Portanto, o que a indústria tem feito nos últimos anos para acelerar processadores foi a inclusão de vários núcleos de processamento dentro do mesmo chip. Estes núcleos são conhecidos como *cores* e processadores que possuem vários deles são chamados *multicore*.

Cada núcleo de um processador é responsável pela execução de instruções em um computador; quanto mais rápido um núcleo puder executar instruções, mais rápido é a sua velocidade em geral, então é natural pensar que um processador com N núcleos seja também N vezes mais rápido que o normal já que ele teria a capacidade de executar N operações simultaneamente; essa capacidade é conhecida como paralelismo a nível de *hardware*. Entretanto, a mera existência de vários núcleos não é suficiente para tornar a execução de programas mais rápida nesta mesma proporção.

Durante a execução de um programa, é possível solicitar que partes de seu código sejam executados ao mesmo tempo; essa execução não precisa ser necessariamente paralela: cada pedaço de código pode ser quebrado em partes menores e cada uma dessas linhas de execução do programa, conhecidas como threads, executariam esses pedaços de forma alternada, estabelecendo, assim, uma relação de concorrência. Um processador *multicore* permite que threads concorrentes possam ser executadas ao mesmo tempo de forma paralela; assim, uma aplicação que faz uso de concorrência pode aproveitar mais recursos de processadores *multicore*. A realidade, porém, é que poucas aplicações hoje fazem esse tipo de execução. Como é

previsto que o aumento da velocidade alcançada por núcleo alcance uma estagnação em pouco tempo por limitações tecnológicas como o aumento da quantidade de transistores em um núcleo e sua proximidade aumentam a quantidade de calor dissipado e seu consumo de energia; por isso, a indústria tem aumentado a quantidade de núcleos nos últimos anos em seus processadores na tentativa de manter seu crescimento e acredita-se que esta tendência seja irreversível [1].

Sistemas operacionais em geral procuram fazer a execução de processos distintos em diferentes núcleos, mas acredita-se que é possível tornar esse uso ainda mais eficiente. Em processadores multicore heterogêneos, os núcleos apresentam características distintas e alguns deles podem fazer uso intensivo de recursos críticos do hardware dentro do chip. Se o sistema operacional conseguisse identificar quais deles são críticos, ele poderia atribuir a execução de tarefas em um outro núcleo mais simples, fazendo a performance ficar melhor distribuída e reduzindo o consumo de energia [2], pois é conhecido que um núcleo muito mais rápido executando uma tarefa consome mais energia do que vários núcleos mais simples executando a mesma tarefa e terminando no mesmo tempo total [3].

A elaboração de um novo sistema operacional baseado em Linux, proposta em [4], tem como objetivo prover a capacidade de um usuário administrar quais núcleos poderão atuar sobre partições específicas através de uma interface mais amigável do ASMP Linux (ASymmetric Multi-Processor Linux), baseada na interface de arquivos em POSIX. Uma aplicação do usuário poderia pedir que uma tarefa fossa executada em alguma partição específica e daí o sistema operacional iria decidir, dentre os núcleos disponíveis, qual deles executariam tal procedimento; finalmente, o sistema operacional poderia reservar recursos de hardware específicos nessa partição especializada e tornar mais eficiente a execução de alguns tipos de instruções, alcançando melhores performances.

Há também visões pessimistas em relação ao futuro dos processadores *multicore*. John Hennessy alertou ao problema de correlacionar paralelismo com facilidade de uso em computadores essencialmente paralelos. Segundo ele, este é um

dos problemas mais difíceis que a ciência da computação já enfrentou [5]. Ele também levantou a necessidade de que essa revolução em paralelismo venha acompanhada de facilidade de desenvolvimento nesses sistemas de forma eficiente, portável e correta tão quanto tem sido desenvolver em sistemas sequenciais; por exemplo, se programação paralela não for um procedimento tão produtivo, o resultado será o atraso do desenvolvimento de aplicações e consequentemente reduzirá a quantidade de aplicações com capacidade de aproveitar essa arquitetura.

Aplicações

A identificação de aplicações relevantes e com alta demanda de maior poder computacional tem sido uma estratégia bastante usada. Após essa identificação, investimentos são feitos por várias empresas interessadas no sucesso da tecnologia *multicore* para torná-las cada vez mais paralelas, já que esse sucesso não depende da reestruturação de todas as aplicações existentes, apenas das aplicações relevantes [5]. Dentre as otimizações desejadas, pode-se identificar performance, responsividade, segurança e estabilidade.

Grandes empresas, como Microsoft e Intel, estão tentando resolver o problema ao melhorar o suporte a paralelismo em linguagens de programação já conhecidas [1], já que as linguagens de programação existentes que visam facilitar a construção de código paralelizável nunca se tornaram suficientemente populares. Linguagens funcionais, como Erlang e Haskell, têm sido usadas na academia e na indústria para processamento de grandes volumes de dados; linguagens funcionais são baseadas em modelos matemáticos e formais, e acabam diferenciando-se demais das linguagens tradicionais e imperativas, consequentemente eliminando uma grande parte de desenvolvedores que não tem familiaridade com esse modelo.

Mesmo em linguagens com suporte a paralelismo, como Java, a dificuldade em utilizar estruturas mais eficientes de concorrência é evidente. Numa análise em projetos de código aberto, percebeu-se que a maioria das construções usadas por projetos eram de exclusão mútua e de baixo nível como blocos delimitados por

"*synchronized*", desperdiçando oportunidades de usar construções da linguagem de mais alto nível e com menor chance de erros, como a biblioteca "java.util.concurrent" [6].

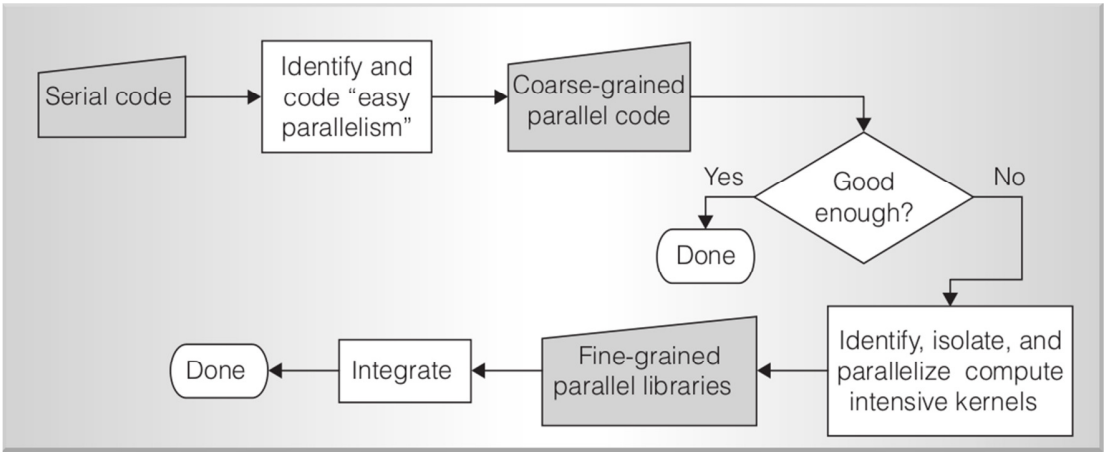

Figura 1 - ciclo de desenvolvimento de código paralelo [7]

No ciclo de desenvolvimento de código paralelo, os desenvolvedores, a princípio, escrevem um código sequencial para sua aplicação; em seguida, é comum poder identificar e encapsular regiões de códigos sequenciais através de um framework de paralelismo sem ter que mudar muito a lógica do código inicial; este processo é conhecido como paralelismo fácil. Entretanto, é possível que essa otimização não seja suficiente e não se obtenha o ganho de performance esperado. Nestes casos, a estratégia é identificar os núcleos de computação intensiva, encapsulá-los em bibliotecas ou frameworks, e daí melhorá-las separadamente, de modo que estas façam melhor uso de paralelismo.

Uma estratégia eficiente que tem sido abordada para reduzir o esforço durante a transição de códigos sequenciais para códigos paralelos consiste na utilização de frameworks que sejam livres de concorrência por design, garantindo que o código executado paralelamente seja determinístico [7]; a vantagem está em evitar os bugs difíceis de se reproduzir, resultantes da concorrência entre threads.

Pesquisadores do Berkley's Par Lab tem trabalhado em arquitetar softwares paralelos com padrões de design, sem linguagens de programação específicas. Sua proposta é a criação de padrões de linguagem que apresentem, com pouca

formalidade, uma forma mais associativa de pensar em um determinado problema, sem a necessidade de ser uma metodologia rígida [5].

Espera-se que esses padrões fomentem a resolução de problemas de forma criativa, instruindo com um vocabulário comum e que capture os principais problemas encontrados durante o design da solução original; dessa forma, esses padrões estariam ajudando a identificar soluções similares para vários outros problemas em diversas áreas. Na prática, padrões realmente úteis dificilmente são inventados, mas extraídos de aplicações bem sucedidas; um exemplo de um padrão interessante será mostrado na análise do Lucene desta monografia.

Um framework poderia ser extraído a partir desses padrões de design e ele faria a ligação entre a linguagem de programação e esses conceitos; através desse framework, seria possível tornar os programadores mais produtivos. O programador de propósito geral faria uso do framework para escrever suas soluções, encontradas com a ajuda dos padrões de design. O programador especialista iria implementar esses frameworks e mapeá-los em diretivas específicas da plataforma de hardware. Dessa forma, o programador de propósito geral poderá criar aplicações usando paralelismo mesmo sem dominar completamente os detalhes da programação em baixo nível.

Desafios

Há muita dificuldade de compreensão em programação paralela, e acredita-se que ela está relacionada a uma tendência natural das pessoas de pensar em uma tarefa como um conjunto de passos lineares e sequenciais [1]. Segundo especialistas, a experiência de ensino de paralelismo sugere que nem todos os programadores são capazes de entender as nuances em torno de softwares paralelos e hardwares paralelos [5]; geralmente, apresentam dificuldades em torno de locks, deadlocks, load balancing, escalonamento de processos, consistência de memória, e assim por diante.

Ao trabalhar com programação paralela, é comum encontrar uma deficiência de ferramentas de debug apropriadas; por exemplo, testes realizados em máquinas

essencialmente paralelas, ou quando envolvem múltiplos processos distintos, ou até o simples caso em que várias threads estão sendo executadas; nestes casos, a capacidade de identificação de bugs é prejudicada devido à enorme complexidade de compreensão do estado em que cada sequência de execução se encontrava e que poderia reproduzir um determinado bug. Facilitar o debug de códigos paralelos é um dos maiores desafios a serem enfrentados nesta área.

De forma similar, torna-se mais complexo a capacidade de testar código paralelo. Durante o desenvolvimento de uma aplicação, é essencial do ponto de vista de engenharia de software que cada modificação venha acompanhada de testes que façam a cobertura do comportamento introduzido pelas mudanças; dessa forma, é possível assegurar que mesmo que o código eventualmente mude, testes detectarão se o comportamento foi preservado ou não. No entanto, a criação de um teste nem sempre é possível ou exige uma enorme complexidade na infraestrutura dos testes para conseguir simular um bug em particular. Na análise do WebKit realizada nesta monografia, será apresentado uma situação complicada de ser testada relacionada ao Scrolling Coordinator e que pode ser estendida a vários outros tipos de aplicação.

Capítulo 3 - Concorrência e Paralelismo

É necessário fazer a distinção entre programação concorrente e programação paralela: a programação concorrente foca-se em resolver problemas onde a concorrência faz parte da sua especificação, enquanto programação paralela resolve problemas em que a concorrência é usada opcionalmente, com o intuito de melhorar performance [7].

Sistemas de interface de usuário, sistemas operacionais, transações de banco online são alguns exemplos de problemas em que a concorrência faz parte de sua especificação; nestes casos, a computação é iniciada por ações não determinísticas, possivelmente concorrentes.

Sistemas de transformações em geral, processamento de sinais, computação científica são alguns exemplos de problemas onde a entrada é mapeada através de uma cadeia de transformações determinísticas a uma saída ou a um conjunto de saídas, constituindo programação paralela.

Considera-se que programação paralela seja muito mais fácil do que programação concorrente porque a paralela raramente precisa fazer uso de código não-determinístico. A arquitetura multicore não aumenta a necessidade de programação concorrente nem a complica, mas certamente aumenta a necessidade de programação paralela.

Uma técnica simples de paralelização é conhecida como "*single instruction, multiple data*" (SIMD). A ideia é pegar uma série de dados de entrada e dividi-las em porções menores. Em seguida, cada porção da entrada pode ser trabalhada de forma independente utilizando o mesmo algoritmo sequencial e determinístico aplicado para a entrada inteira. Essa técnica é frequentemente usada em operações como ajuste de contraste de imagens, onde o efeito pode ser aplicado paralelamente em sub-regiões

distintas da imagem sem problemas. Através desta abordagem o gasto de energia usada na entrega do resultado de uma instrução é amortizada [3].

Existem muitas vantagens em utilizar determinismo em códigos paralelos; uma delas é a capacidade de entendimento de um programa sem ter a preocupação da ocorrência de *interleavings*, *data races*, modelos complexos de consistência de memória, e assim por diante; outra vantagem é capacidade de progressivamente substituir código sequencial por sua versão paralelizada e, ao mesmo tempo, preservar o comportamento original da aplicação; outra vantagem é de que melhores performances podem ser adquiridas já que este modelo reflete melhor a ideia de linhas de execução independentes [8]. Assim, desenvolvedores podem investir mais tempo melhorando a performance do código e menos tempo é desperdiçado investigando bugs não-determinísticos, geralmente causados pela concorrência e que são eliminadas em códigos determinísticos.

Programação concorrente livre de locks

Quando sequências paralelas de execução precisam acessar dados compartilhados, tradicionalmente usava-se locks para sincronizar esse acesso e delimitar regiões críticas de execução para impedir que mais de uma thread executasse simultaneamente o mesmo pedaço de código.

Quando uma thread faz uso de um lock e ele está livre, ela pode prosseguir com sua execução, fechando o lock em seguida. Essa operação de comparar e especificar o novo valor precisa ser feita atomicamente, ou seja, livre de concorrência. Se outras threads tentarem acessar o mesmo lock simultaneamente, elas ficam completamente paradas até que o lock seja liberado.

Infelizmente, a utilização de locks pode implicar em sérios problemas no desenvolvimento de software: bloqueios na execução para as threads que encontram o lock fechado; se o processo que está mantendo um lock fechado quebrar, os demais ficarão esperando para sempre; a utilização não pode ser composta facilmente, ou seja,

não é fácil pegar dois códigos que estão fazendo uso correto de locks e esperar que sua composição também esteja correta [9]; e muitos outros [10]. Em particular, muitas aplicações exigem que, para uma operação em particular ocorrer, é preciso adquirir os locks correspondentes de diferentes níveis da aplicação, dificultando a modularização.

Além disso, a utilização de locks depende de programadores seguirem convenções como lembrar de sempre pegar o lock apropriado quando for fazer uso de memória compartilhada, e estas regras normalmente são escritas como guias aos desenvolvedores, mas nunca são escritas de forma rígida o suficiente de modo a permitir a identificação automática da necessidade do uso de locks [9].

A solução que permite fazer melhor uso de paralelismo com programação concorrente é a utilização de algoritmos não-bloqueantes. Um algoritmo é não-bloqueante se a suspensão de uma ou mais threads não parar o potencial progresso das threads restantes [11]. Esse design pode ser adquirido ao evitar a necessidade de regiões críticas no código.

Um exemplo prático desse conceito sendo aplicado está descrito na análise do Lucene, com uma analogia de crianças no jardim de infância. No WebKit, algoritmos não-bloqueantes são usados largamente em todo o código, em diversos módulos; no Qt também não é diferente. Esses três projetos serão apresentados nos capítulos seguintes.

Capítulo 4 – WebKit

WebKit é um motor de renderização de conteúdo web que integra navegadores populares como Google Chrome e Safari, correspondendo a cerca de 40% da participação no mercado de navegadores web, sendo mais popular que o motor de renderização do Internet Explorer e do Firefox, por exemplo.

Segundo estatísticas, o projeto conta com uma quantidade de commits[1] acima de 2000 por mês em média ao longo do ano de 2012 e esse número tende a aumentar [12], dado que outras empresas passam a usar WebKit em suas aplicações, como o navegador Opera que recentemente anunciou a sua gradual adoção [13].

Ele é responsável por dar funcionalidade de navegação a um navegador web; dentre elas, carregar páginas remotas ou locais; mover-se no histórico de páginas visitadas, como, por exemplo, voltar à página anterior; compreender linguagens como HTML, CSS e JavaScript, entre várias outras; decodificar imagens e vídeos de vários formatos para exibição na página; respeitar diversos protocolos de rede, como HTTP e HTTPS; ser capaz de executar plugins; ou seja, é o núcleo de um navegador web funcional.

No processo de desenvolvimento do WebKit, cada modificação deve vir acompanhada de um teste automático que explore as mudanças de comportamento introduzidas, pois, como a base de desenvolvedores é muito grande, é comum que partes do código mudem completamente com o tempo; os testes ajudam a garantir que o comportamento permanecerá o mesmo ao longo do tempo.

Hoje, o projeto já conta com mais de 35.000 testes automáticos e cada nova modificação introduzida no código deve respeitar essa enorme bateria de testes; caso

[1] http://en.wikipedia.org/wiki/Commit_(data_management)

contrário, a modificação não será aceita ou, mesmo que aceita, pode ser revertida em caso de regressões.

O processo de revisão de modificações no código é também bastante rigoroso[2], normalmente requer que alguém com autoridade para revisar o código e que seja familiar com a área em que as mudanças estão relacionadas manifeste sua aprovação perante as mudanças sugeridas; assim minimiza-se a possibilidade de regressões e também auxilia no aprendizado de novos contribuintes na área através de revisões construtivas, ajudando a direcionar as mudanças na direção certa.

No WebKit, existem diversos *ports* e que são diferentes implementações de partes específicas de seu código; por exemplo, é um *port* que faz a conexão entre o código interno do WebKit e o código específico de um determinado navegador. Dentre os *ports* mais famosos estão o Chromium, usando pelo Google Chrome, e o Mac, usado no Safari, e cada um deles tem suas peculiaridades e expõem funcionalidades do WebKit de forma diferente ou com uma arquitetura diferente.

Em relação a pintura de uma página web, em particular, o código do WebKit identifica na página as unidades básicas de pintura, como retângulos e linhas, através da aplicação de estilos como CSS e elementos HTML básicos e então delega que essas chamadas de pintura de tipos básicos sejam implementadas por códigos específicos de cada *port*. Cada *port* decide a melhor forma de realizar essa pintura, podendo usar bibliotecas externas ou frameworks de acordo com suas necessidades. Assim, o código do WebKit tende a ficar menos complexo e mais focado nos conceitos básicos e compartilhados entre os vários ports.

Com o passar do tempo, diversas áreas do WebKit sofreram modificações com o objetivo de melhorar a responsividade do usuário durante a navegação e adicionar maior suporte a paralelismo tem sido uma estratégia usada em várias áreas para resolver esse problema. A seguir serão apresentadas duas áreas que sofreram modificações e como elas impactaram o projeto.

---

[2] http://www.webkit.org/coding/commit-review-policy.html

Parallel Garbage Collector

Garbage collector, em português, coletor de lixo, é a entidade responsável pela limpeza de dados que se encontravam ocupando espaço na memória mesmo sem ser referenciados por outras regiões do código, tornando-os inúteis e justificando a sua remoção.

Em outubro de 2011, no bug 70995, paralelizou-se o garbage collector presente em um módulo conhecido como JavaScriptCore ao notar que se tratava de um problema do tipo *embarrassingly parallel*[3] e que a maioria das instruções e estruturas usadas naquele código já eram thread-safe[4], com a exceção de *compare-and-swap*[5].

JavaScriptCore é um módulo do WebKit responsável pela execução de código JavaScript, utilizada por vários *ports*, similar ao que o módulo V8 faz no Chromium; no entanto, estas modificações foram protegidas por flags de compilação e só afetaram o *port* Mac naquele momento e, como demonstrado durante a proposta das modificações, melhorou o desempenho em vários benchmarks já existentes.

Analise de impacto das mudanças

Para tentar medir o impacto que esta mudança causou no projeto, foi realizado uma consulta ao sistema de bugs do WebKit procurando por ocorrências em título ou em comentários de todas as seguintes palavras: “parallel”, “GC”; além disso, bugs em qualquer estado deveriam ser considerados (resolvidos, em aberto, inválidos, etc.), filtrando-os a partir da data 14 de outubro de 2011, ou seja, o dia da inclusão das mudanças. Para melhorar a cobertura, outra consulta foi feita nas mesmas condições substituindo “GC” por “Garbage Collector” e procurando pelas palavras “thread” e “GC”. A seguir, um sumário dos bugs relevantes encontrados ordenados por ordem de criação:

---

[3] http://en.wikipedia.org/wiki/Embarrassingly_parallel
[4] http://en.wikipedia.org/wiki/Thread_safety
[5] http://en.wikipedia.org/wiki/Compare-and-swap

Bug ID	Título
70995	The GC should be parallel
73309	[Qt] GC should be parallel on Qt platform
84633	[BlackBerry] Enable parallel GC feature
86672	GC is not thread-safe when moving values between C stacks
90957	[Qt] There are parallel GC related crashes regularly
99331	We should avoid weakCompareAndSwap when parallel GC is disabled
99641	Race condition between GCThread and main thread during copying phase

Quadro 1 - bugs encontrados relacionados a parallel GC

As modificações mais impactantes foram a primeira e a última, pois modificaram diretamente a estrutura do garbage collector. Para calcular o número de linhas resultantes geradas por eles, foi realizado o download do arquivo correspondente às modificações no formato diff, calculado o número de linhas iniciadas pelo símbolo "+" e que não eram seguidas de outro "+" como o número de linhas inseridas, e o número de linhas removidas foi calculado de forma similar, substituindo "+" por "-"; um exemplo desse script é mostrado na figura 2.

Figura 2 - script para calcular linhas adicionadas em arquivos diff

No primeiro e no último bug, foram inseridos 915 e 78 linhas respectivamente, totalizando 993 linhas inseridas no projeto. Mesmo considerando que cada patch contenha inserções de linhas que não sejam exatamente de código, a grandeza deste número ajuda a entender a complexidade do código que foi inserido.

Durante a resolução do bug 86672 foi detectado que parte do garbage collector na verdade não era thread-safe; no entanto, o problema até então era teórico, pois não tinha sido possível reproduzir um bug com as características relatadas.

No Qt conseguiram reproduzir diversas quebras de execução aleatoriamente durante a execução de alguns dos testes automáticos. No bug 90957, uma série de pilhas de execução no momento da quebra foram coletadas na tentativa de ajudar a determinar a natureza do bug, levando a conclusão que se tratava de um problema de memória compartilhada e concorrência. No entanto, por não entenderem exatamente como o garbage collector paralelo funcionava nem qual era seu comportamento esperado, solicitaram ajuda dos autores do garbage collector em paralelo e eles reportaram esse bug no sistema de bugs privado da Apple para investigação. Apesar deste bug não ter sido resolvido oficialmente, recentemente foi comentado que as quebras não eram mais reproduzíveis no Qt, possivelmente foram consertadas através de outro bug de forma indireta.

Resultados

Nenhum dos bugs listados inseriu novos testes automáticos, apenas mostraram que a melhoria de performance foi significativa utilizando benchmarks populares[6], e outros *ports* passaram a fazer uso dessa otimização em seguida, mesmo sem entender como o garbage collector paralelo funcionava.

É possível que ainda existam bugs na implementação, como foi percebido por alguns *ports* ao longo do tempo, mas não está claro como ter certeza de que eles não serão introduzidos acidentalmente no futuro sem a execução de testes que evitem essas regressões.

Threaded Scrolling

No WebKit, é possível que ao carregar uma página a execução de vários scripts torne a navegação mais lenta, prejudicando a interação do usuário ao fazer *scrolling*; *scrolling* é o ato de descer ou subir a visibilidade de uma página em uma certa quantidade de pixels.

---

[6] http://www.webkit.org/perf/sunspider/sunspider.html

Uma estratégia utilizada por vários *ports* para otimizar a execução de animações, transições e transformações 3D é conhecida como Accelerated Compositing, fazendo a pintura da página ser dividida em camadas sobrepostas de modo que cada camada seja pintada separadamente e só depois compostas para gerar a imagem final.

Com as camadas pintadas previamente, a simples tarefa de composição pode ser feita em uma thread ou processo distinto de forma independente, fazendo uso da GPU para obter melhores performances [36]; assim, mesmo que um script esteja sendo executado, a unidade de composição, sabendo previamente que tipos de animações estão acontecendo ou transições, pode simulá-las de forma independente.

Uma ação do usuário, em particular *scrolling*, poderia ser executada rapidamente pelo compositor das camadas se ele soubesse antecipadamente do acontecimento do evento, fazendo a simples movimentação das camadas no sentido solicitado; em seguida, esse evento seria enviado ao núcleo de execução da página, já que ela pode fazer uso do evento também via JavaScript ou através de elementos especiais na página.

No entanto, é difícil fazer *scrolling* de forma correta quando a página contém elementos fixos na tela e que se movimentam junto com a área visível do usuário, necessitando formas de identificar e armazenar as informações de quais são as camadas que seguem esse comportamento e como elas se comportam durante a movimentação do ponto de vista do compositor.

Em virtude dessa necessidade, desenvolvedores do *port* Mac introduziram o conceito de Scrolling Coordinator com o papel de coordenar quais camadas eram fixas e como elas deveriam se comportar durante um evento de *scrolling*, provendo as informações relevantes à unidade de composição e sendo executado em uma thread distinta.

O primeiro *commit* encontrado possui a hash iniciada por 632add855 e foi realizado no dia 15 de dezembro de 2011; depois, uma série de modificações relacionadas foram sendo aplicadas em um curto espaço de tempo, o que aumenta a

possibilidade de que esse mecanismo já estivesse implementado em um *branch* privado e só naquele momento estivesse sendo integrado no repositório público do WebKit.

Essas mudanças foram protegidas por uma flag de compilação e não afetaram os demais ports; porém, com o tempo, o Chromium manifestou interesse e também começou a trabalhar nesta área, incluindo suas implementações específicas e levantando questionamentos de como essas informações deveriam ser expostas à unidade de composição[7], optando por uma arquitetura diferente da usada pelo Mac.

Análise de impacto das mudanças

No momento que esta análise foi realizada, a hash do commit atual era iniciada por "14635c474dfb" e os arquivos relacionados estavam localizados no diretório "Source/WebCore/page/scrolling", com um total de 37 arquivos, contribuindo com 5444 linhas de código; dessa quantidade, 388 linhas eram específicas do Chromium e 1568 linhas específicas do Mac; estes resultados foram coletados com o uso do script mostrado na figura 3.

wk_scrolling_compare.bash — Shell

```
1  git checkout 14635c474dfb
2  SCROLLING_DIR="./Source/WebCore/page/scrolling"
3  echo "Total files: `find $SCROLLING_DIR -type f | wc -l`"
4  echo "Total LOC: `find $SCROLLING_DIR -type f -exec cat '{}' \; | wc -l`"
5  echo " * Chromium: `find $SCROLLING_DIR/chromium/ -type f -exec cat '{}' \; | wc -l`"
6  echo " * Mac: `find $SCROLLING_DIR/mac/ -type f -exec cat '{}' \; | wc -l`"
```

Figura 3 - script para calcular números de arquivos e linhas de código

Para avaliar o impacto causado pelas mudanças, consultas foram realizadas ao sistema de bugs do WebKit procurando por ocorrências de bugs em que qualquer uma das seguintes palavras ocorreram em comentários no bug ou no seu título: "scrollingcoordinator", "scrollingstate", "scrollingtree", "scrolling" e "threaded_scrolling"; dentre esses bugs, qualquer estado de resolução seria considerado, como por exemplo

[7] https://bugs.webkit.org/show_bug.cgi?id=71385

bugs resolvidos, inválidos, pendentes, e assim por diante; e aplicando a filtragem de resultados a bugs criados a partir do dia 15 de dezembro de 2011.

A consulta resultou em 88 bugs encontrados, sendo relevante notar que 8 deles possuíam a palavra “crash” no título, indicando bugs críticos. Fazendo essa mesma busca, mas agora considerando apenas os bugs que continuam em aberto, restaram apenas 4 bugs, o que permite fazer uma análise mais detalhada de cada um deles.

No bug 89576, cujo título era “Scrolling tree should be testable”, foi abordado que, em alguns casos, é necessário simular um processo lento sendo executado em uma página web e também simular simultaneamente um evento de scroll proveniente do usuário da aplicação de modo que o uso imediato do evento pela aplicação ocorra, fazendo o usuário achar que o scroll realmente ocorreu e só depois ter o evento processado pela página em si; neste estado intermediário em que a interface está realizando scrolling de forma independente, é necessário investigar o estado de sua ScrollingTree e observar o comportamento dela; no entanto, por haver diferenças de implementação de como a ScrollingTree é representada em diferentes ports, não existe ainda um padrão de como deveria ser sua representação textual, caso esta seja a estratégia preferida, inviabilizando a criação de testes automatizados que possam ser executados independentemente do port. Este bug foi criado em junho de 2012 e sua última modificação em outubro do mesmo ano, ou seja, há mais de seis meses não existe atividade nesta área, permitindo observar que muitos bugs podem estar escondidos e comportamentos estão passíveis de modificações com o tempo sem a capacidade de capturar essas regressões, tornando a observação humana e a realização de testes manuais necessários.

No bug 106858, notou-se que durante a execução de testes em modo debug, o código do WebKit pode quebrar ao encontrar uma condição inválida estabelecida por um assert. Um assert é normalmente colocado por desenvolvedores que, ao entenderem o código, sabem que determinadas condições são necessárias para garantir sua corretude; falhar em alguma dessas condições pode significar bugs reais,

pois apesar do usuário final não utilizar a aplicação compilada em modo debug, sem quebrar ao passar por condições inválidas, se o código está num estado não suportado, os resultados tornam-se imprevisíveis. Teoricamente a execução em modo debug não deveria influenciar em nada nos resultados dos testes; no entanto, barras de scroll estavam sendo apresentadas diferentemente. Outro desenvolvedor conseguiu localizar o assert que levou à quebra da execução normal e isso pode ajudar na identificação da solução. Esse bug não parece ser tão crítico quanto o anterior dado que ele é reproduzível apenas em uma plataforma específica e ainda em modo debug, podendo ser uma falha de implementação específica do Chromium nessa plataforma.

No bug 107426, cujo título era “[CoordinatedGraphics] CSS sticky elements are blinking while scrolling”, discutiu-se como a solução do problema do *scroll* imediato através do sistema do Coordinated Graphics estava ultrapassada, possivelmente com vários erros, e que deviam fazer uso também do Scrolling Coordinator, solução adotada pelos principais ports. Coordinated Graphics é outra área do WebKit que delega e sincroniza pintura entre dois processos, utilizado por *ports* como Qt e EFL. Como o problema não reflete um problema no Scrolling Coordinator e sim no Coordinated Graphics, este bug não é relevante para esta análise, mas ajuda reforçar como diferentes *ports* podem ter soluções diferentes para o mesmo problema dentro do WebKit.

E, finalmente, no bug 109826, sendo o mais recente até a escrita deste documento, cujo título é “REGRESSION (r142505?): Crashes in WebCore::ScrollingStateNode::appendChild when using back/forward buttons”, é descrito um problema crítico do navegador Safari que quebra ao pressionar botões de navegação como “página anterior” ou “próxima página” com uma versão recente do WebKit; também foi identificado que este bug pode ter envolvimento com modificações recentes na área do Scrolling Coordinator e que foram introduzidas sem a inserção de novos testes.

Resultados

O resultado dessas mudanças é percebido de forma diferente para cada port, já que cada um tem diferenciação em sua implementação. Em particular, o navegador Safari, desenvolvido pelo Mac, tem apresentado excelentes resultados, como relatado em [33]; segundo a análise realizada, a frequência de atualizações da tela por segundo durante um evento de scroll chegou a duplicar, dando a sensação de uma rolagem muito mais suave. É importante notar que não está claro se foram exatamente estas mudanças que provocaram uma melhora tão significante na performance visto que o projeto, por ser muito extenso, pode ter sofrido melhorias em várias outras áreas também.

No Chromium, iniciaram a implementação da suavização de scroll há pouco tempo, como anunciado em [34], aproximadamente no mesmo período em que começaram a trabalhar com Scrolling Coordinator; portanto, é improvável que tenham sido essas mudanças que provocaram a suavização nesse *port*.

Desde julho de 2012, um bug foi reportado por vários usuários ao perceberem que a performance do Google Chrome durante scroll estava muito mais lenta do que o Safari em várias páginas, principalmente na plataforma MacOSX Mountain Lion [35], além de relatos de vários bugs visuais; porém, desenvolvedores declararam estar trabalhando para melhorar essa performance em várias áreas distintas do código, mostrando a grande importância da suavidade do scroll em um navegador web e o impacto que ele causa diretamente nos usuários.

Capítulo 5 - Qt

Qt é um framework de desenvolvimento de aplicações multi-plataforma largamente utilizado no desenvolvimento de aplicações com interface gráfica; sua base de código é escrita em C++, mas também faz uso de extensões da linguagem com o propósito de facilitar sua escrita, gerando o código correspondente às extensões utilizadas antes da compilação; seu design é modularizado, separando seus componentes em hierarquias baseadas em suas dependências, e essa separação é clara na organização dos diretórios do projeto.

Quando o desenvolvimento de aplicações para plataformas móveis tornou-se o grande foco, percebeu-se que a maioria das aplicações escritas em Qt sofriam degradações de performance durante animações na interface enquanto o usuário a deslizava, gesto comum nessas plataformas. A degradação acontecia em virtude da aplicação estar fazendo algum trabalho intensivo durante a própria pintura, pois alguns elementos, mais complexos, poderiam demorar mais para serem pintados, tornando a animação pouco suave.

A solução proposta foi separar o trabalho intensivo de pintura de um elemento em uma thread distinta e, quando a thread principal estivesse pronta para fazer a pintura do quadro visível ao usuário, ela faria uso desse elemento já pintado ou pintaria alguma coisa padrão caso ainda não estivesse pronta [14]. Essa estratégia é similar ao que o Google Maps faz em seu site, onde o usuário pode fazer deslizamentos no mapa mesmo que partes dele ainda não estejam prontas para serem exibidas.

Numa comparação entre o modelo anterior e o novo modelo, com a pintura realizada em uma thread distinta, e a vantagem ficou clara: a taxa de 60 quadros por segundo era mantida apenas no novo modelo, enquanto que o anterior chegava a cair a uma taxa inferior a 10 durante um deslize da tela que ocasionava o aparecimento de novos elementos [14].

O desempenho obtido ao utilizar o sistema de pintura acelerado por hardware do Qt não estava satisfatório e após várias tentativas de melhorá-lo sem obter sucesso, decidiram dar um passo atrás e rever todo o modelo de pintura imperativa que existia no Qt. Outros projetos estavam usando modelos de pintura baseados em grafos para pintura acelerada por hardware; esse tipo de pintura vinha crescendo em demanda com o passar dos anos.

Dessa necessidade, surgiu o Qt Scene Graph, um novo modelo de pintura em que uma cena é representada por uma árvore de unidades bem básicas de pintura. Neste novo modelo, tornaria mais fácil o uso direto de OpenGL 2.0 pela API para realização de efeitos que antes eram complicados de serem alcançados, como um sistema de partículas e transformações 3D em geral [15].

Percebendo o grande potencial no Scene Graph, a nova versão do Qt foi baseada nesse novo modelo de pintura e com a visão de que os desenvolvedores deveriam usar QML[8], a linguagem declarativa criada pelo Qt, enquanto o uso de C++ ficaria restrito à criação de novos elementos que seriam visíveis no contexto dessa linguagem declarativa; ela ficou conhecida como Qt 5 e teria como um dos principais objetivos fazer melhor uso da GPU para melhoria da experiência gráfica do usuário, tornando a nova versão dependente de OpenGL para funcionar [16].

QML Scene Graph

Scene Graph é uma árvore de nós que representam unidades básicas necessárias à pintura, sendo as mais importantes GeometryNode, TransformNode e ClipNode [17]: GeometryNode seria um nó usado para representar todo o conteúdo visível, contendo apenas informações básicas como malha poligonal e seu material (por exemplo, uma textura); TransformNode identificaria uma transformação matricial a ser aplicada em seus nós filhos; e ClipNode definiria os limites visíveis dos seus filhos.

[8] http://en.wikipedia.org/wiki/QML

Teoricamente, qualquer elemento visível pode ser representado por esse modelo simplificado.

No momento da pintura dos nós, a entidade que ficará responsável pela pintura pega a árvore para realizá-la, com uma clara separação da pintura do resto do código. Esse compositor é livre para fazer as otimizações que ele quiser em relação a pintura, oferecendo uma implementação padrão e uma API não pública que dá ao usuário a capacidade de fazer sua própria otimização quando a padrão não for suficiente às suas necessidades. Sua implementação pode ser feita em qualquer thread que tenha um contexto OpenGL associado a ela, mas quando um Scene Graph for associado a uma thread e a um contexto, ele não poderia mais ser movido [18].

Desde o início do seu desenvolvimento, já pensavam na possibilidade de mover a execução de animações em uma thread dedicada, mas, devido à forma como QML funcionava na época, a estratégia foi abandonada e só retomada muito tempo depois, com a preparação para a versão 5.1 do Qt [19].

Mesmo deixando as animações continuarem sendo executadas na thread principal, conhecida como GUI Thread, a proposta do novo modelo ainda poderia representar uma grande melhoria, pois as animações seriam coordenadas pela pela Render Thread, thread em que a renderização realmente aconteceria.

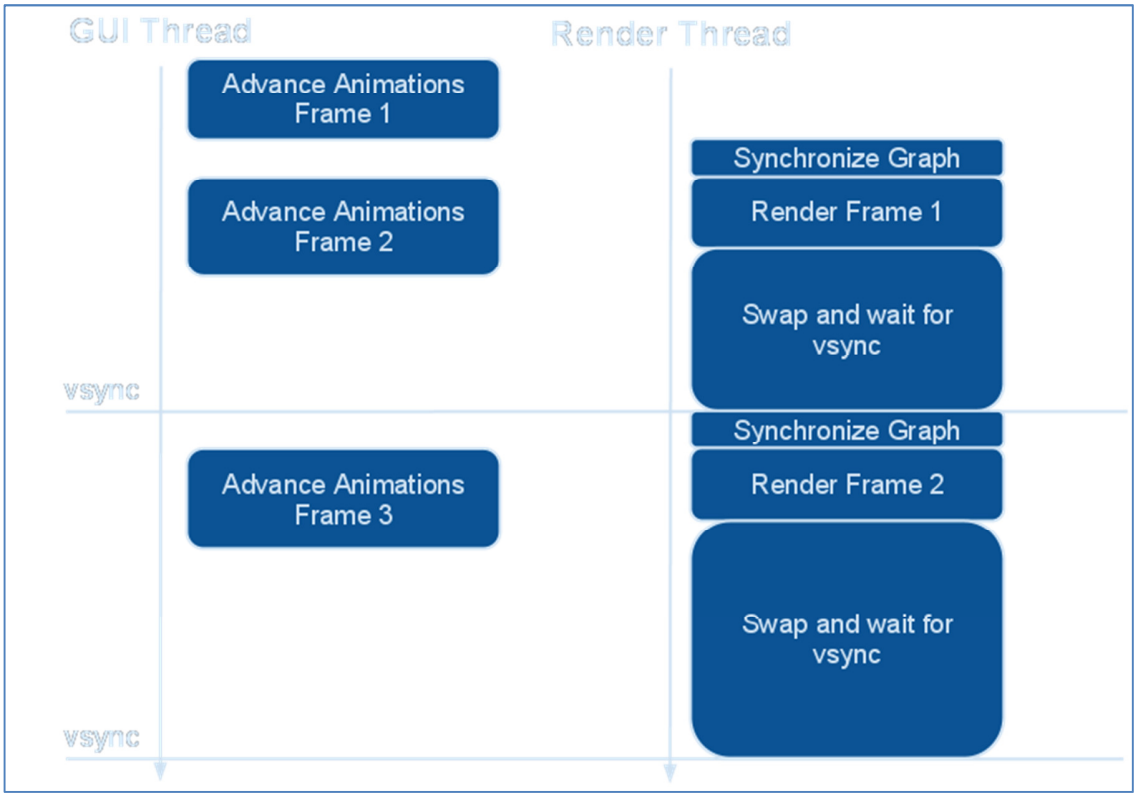

Figura 4 - modelo de pintura em thread distinta usando SceneGraph [18]

No diagrama, é possível ver que a GUI Thread executa as animações respeitando os intervalos de pinturas que são coordenados pela Render Thread; assim, a GUI Thread só precisaria ajustar as posições do quadro atual pensando no seu estado final representado pelo "vsync", isto é, o momento em que o monitor do computador atualiza o que é visível, sobrando mais tempo para a realização de outras tarefas, como ações do usuário ou execução de código em JavaScript. Mesmo para máquinas com apenas um núcleo no processador, essas modificações seriam um ganho, pois as animações poderiam ser executadas menos frequentemente do que o modelo anterior, onde um temporizador controlava cada atualização de quadro durante uma animação.

Análise de impacto das mudanças

Para avaliar o impacto que as mudanças relacionadas causaram no projeto, inicialmente foi feito o download do código fonte, sendo a *hash* do último *commit*

iniciada por “eeaef0d8f50”, no submódulo do Qt conhecido como “qtdeclarative”, e a parte do código relevante está no subdiretório “src/quick/scenegraph/”. Existiam 68 arquivos nesse subdiretório, totalizando 19417 linhas de código; dentro deles existiam outros dois, “coreapi” e “util”, e cada um deles contribui com 6124 linhas e 4594 linhas respectivamente.

Através do sistema de bugs do Qt[9], várias buscas foram realizadas nos componentes “QtQuick: SceneGraph” e “Declarative (QML)”, onde o primeiro é a área desejada e o segundo é a parte do Qt que mais usa esse novo modelo de pintura. Dentre as várias buscas, a mais efetiva foi a que buscava por bugs com a palavra “crash” e qualquer outra palavra começada por “thread” ou “renderer”, ou simplesmente se encontrada uma palavra começada por “seg”; essa busca de texto avaliou comentários, sumários e descrição de bugs e restringiu-se o escopo a bugs que afetassem qualquer versão do Qt igual ou superior a 5; dessa busca, 61 resultados de bugs foram encontrados, onde apenas 3 deles eram inválidos, 2 precisavam de mais informações para avaliar se eram mesmo bugs, 9 foram fechados por não ser mais possível sua reprodução, 4 marcados como duplicatas, 3 marcados como fora de escopo do projeto, e finalmente 9 bugs ainda estavam abertos.

De todos os bugs encontrados, pelo menos 12 deles foram detectados por testes automáticos já existentes, incluindo testes automáticos de API que existiam para cada elemento QML e também testes em aplicações de exemplo usando QML; pelo menos 4 deles aconteciam ocasionalmente. Como a política de contribuição do Qt[10] não exige a criação de bugs para cada modificação feita no projeto, o número de bugs pode ser muito maior do que o calculado nesta análise.

[9] http://bugreports.qt-project.org/

[10] http://qt-project.org/wiki/Qt-Contribution-Guidelines

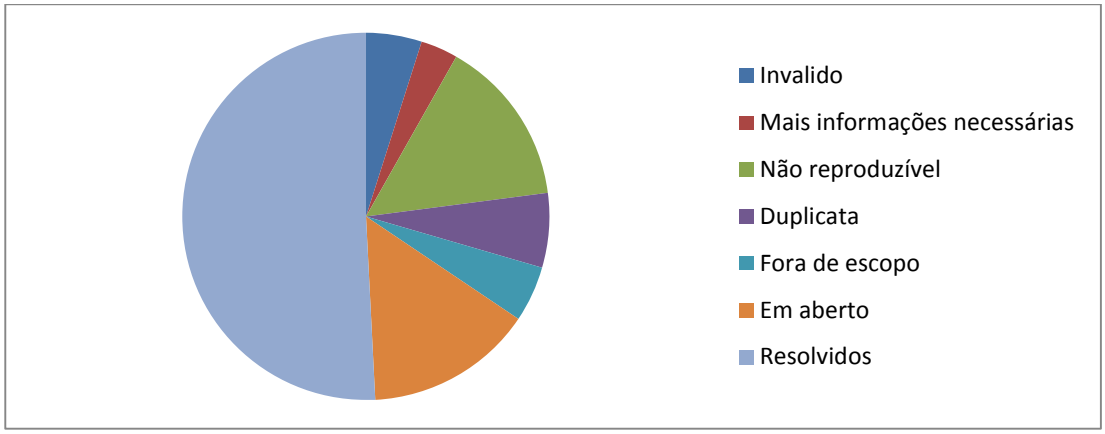

Figura 5 - diagrama mostrando estado de bugs encontrados

Resultados

Apesar do grande esforço de reescrever parte do código de todos os elementos QML para fazer uso do Scene Graph, os resultados foram satisfatórios; no lançamento do Qt 5, um vídeo demonstrativo foi publicado[11] mostrando sua capacidade em criar aplicações fazendo extensivo uso de OpenGL na criação de efeitos visuais, possibilitando a rápida criação de sistemas de partículas, deformações em superfícies planas, melhor renderização de texto e vários outros novos recursos.

Nos experimentos analisados pela comunidade na época das mudanças, foi mostrando que na melhor configuração era possível alcançar a taxa de 250 quadros por segundo, enquanto que o melhor alcançado pela versão antiga era de apenas 100 quadros por segundo [18].

[11] http://blog.qt.digia.com/blog/2012/12/19/qt-5-0/

Capítulo 6 – Apache Lucene

Apache Lucene é um projeto open source de uma biblioteca que realiza a atividade de busca de informações relevantes em um grande escopo de documentos, provendo uma API para inserir documentos para indexação e para realizar busca em cima da base de dados, sendo utilizado em vários serviços na web; originalmente foi escrito em Java, mas já existem outras implementações em linguagens como Python, PHP e C++, organizados em projetos separados [20].

O mecanismo de indexação de documentos do Lucene funciona a partir do uso de uma estrutura de dados com lista de índices invertidos[12] que mapeia termos à suas ocorrências; essa estrutura é gerada para cada campo de um documento configurável pelo usuário, como “título”, “autor”, “sumário” e assim por diante; com as palavras mapeadas aos documentos em que são encontradas, a complexidade da busca torna-se linear, podendo ser executada rapidamente.

Em sistemas de busca, é crucial que a obtenção de resultados seja feita da forma mais rápida possível, caso contrário, o usuário da aplicação pode desistir da busca e abandonar o serviço; enquanto isso, a atualização imediata de documentos na base de índices não é tão importante na maioria dos casos, embora desejável em alguns, como no Twitter que em 2010 passou a usar Lucene como ponto de partida ao seu sistema de buscas e desejava realizar a indexação de novas mensagens em no máximo 10 segundos; o projeto, no estado em que se encontrava na época, não era suficiente para alcançar esses resultados, então fizeram várias modificações, entre elas melhoraram a performance do garbage collector e optaram por estruturas de dados e algoritmos que evitassem o uso de locks; mas prometeram contribuir de volta ao projeto open source [21].

[12] http://en.wikipedia.org/wiki/Inverted_index

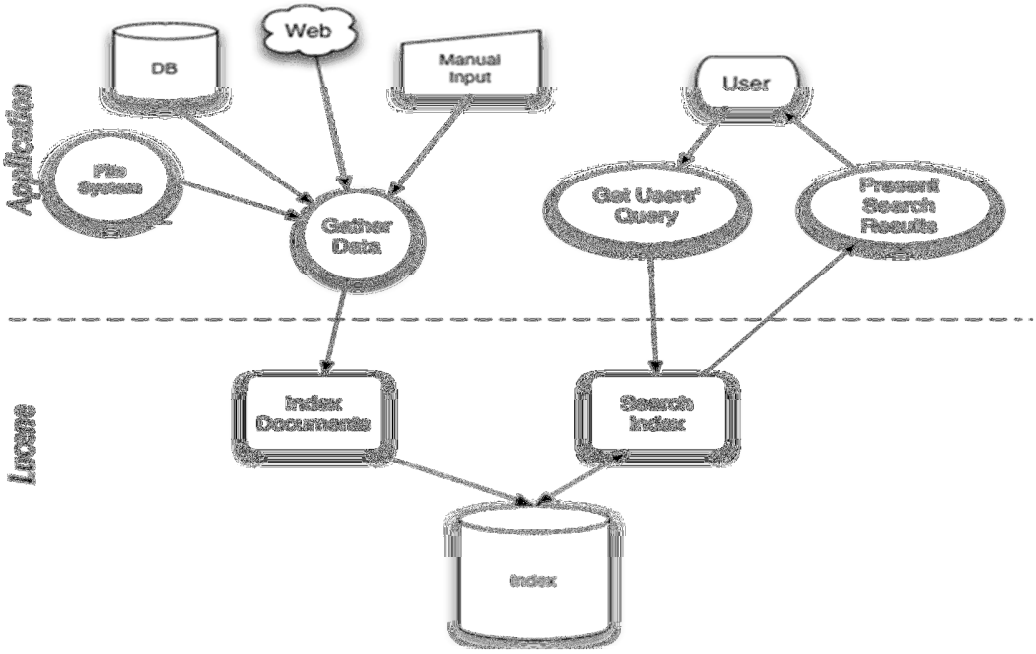

Figura 6 - arquitetura do Apache Lucene [22]

Durante o processo de indexação, cada documento é indexado unicamente na sua estrutura de índices invertidos, e depois os resultados são unidos em uma só estrutura, seguindo a estratégia de dividir para conquistar, calculando resultados de dados menores e fazendo a união desses resultados para gerar resultados maiores; a forma como essas operações são feitas e a heurística por trás da união desses segmentos pode ser controlada pelo usuário através da API do Lucene definindo a política das uniões [23].

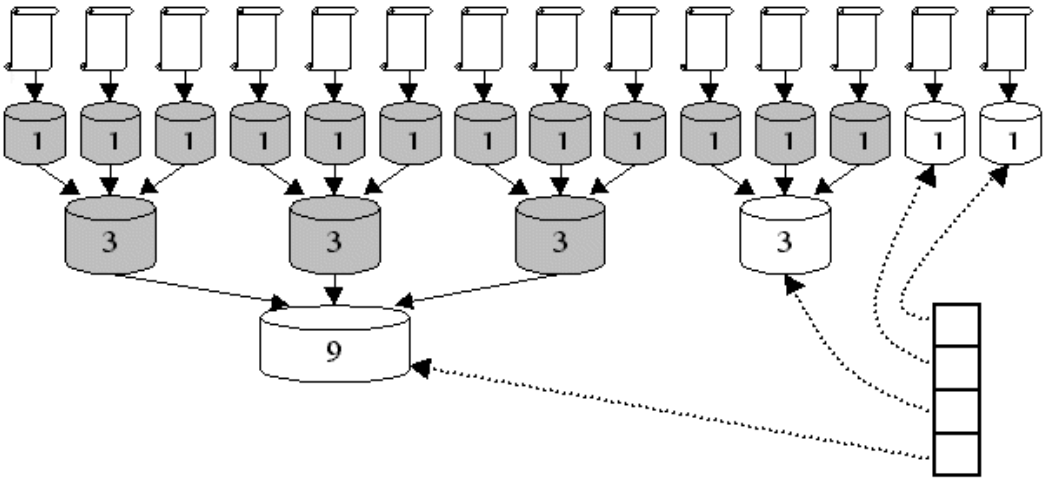

Figura 7 - diagrama com segmentos em união, cinzas foram excluídos [22]

Dentre as mudanças ocorridas na versão 4 do Lucene, lançada em 2012, foi escolhida para esta análise a que ficou conhecida como “Concurrent Flushing” devido a quantidade de conteúdo encontrado nas pesquisas e também pela sua grande relevância dentro deste tema.

Concurrent Flushing

Antes dessas mudanças, durante uma indexação, as threads trabalhavam em indexar os documentos paralelamente, cada um guardando em memória os índices invertidos obtidos, mas quando um limite pré-estabelecido de memória era alcançado, todas as threads precisavam parar suas atividades para que os fragmentos pudessem ser unidos e esse resultado obtido seria escrito no disco rígido; essa escrita poderia acontecer enquanto as threads voltavam a trabalhar em indexar documentos, mas a parada era prejudicial.

Em uma analogia, Simon Willnauer explica o mesmo fenômeno envolvendo crianças no jardim de infância [24]: um jogo em que o professor pede às crianças que peguem de uma pilha de peças de Lego um punhado de peças e entregue ao professor; este segurará os punhados de peças de vários alunos até que não seja possível segurar mais peças em sua mão; neste momento, o professor vira-se e, com as peças coletadas, separa por cores antes de guardar em compartimentos.

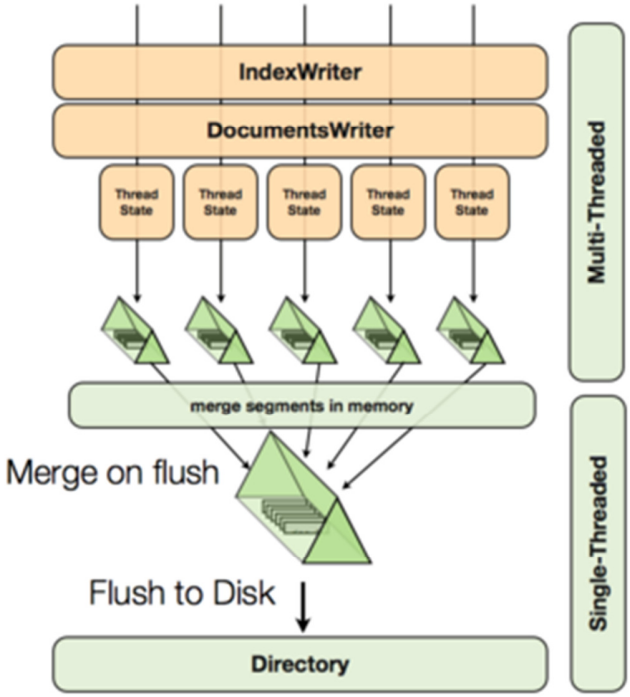

Figura 8 - gargalo antes do concurrent flushing [25]

Após as mudanças [13], Lucene passou a usar internamente a classe DocumentsWriterPerThread, ou abreviadamente DWPT, e que poderia fazer uma operação de *flush*, isto é, uma atualização permanente nos dados, no seu próprio segmento privado. Na mesma analogia das crianças no jardim de infância, o professor teria solicitado que elas ordenassem previamente as cores das peças que estavam segurando e já entregassem esse resultado ao professor deixando no chão as peças organizadas e, sem esperar pelo professor, procurassem novas peças para repetir o processo; assim, o professor, em seu próprio ritmo, poderia coletar as peças e juntá-las para formar blocos maiores de peças organizadas [24].

[13] https://issues.apache.org/jira/browse/LUCENE-3023

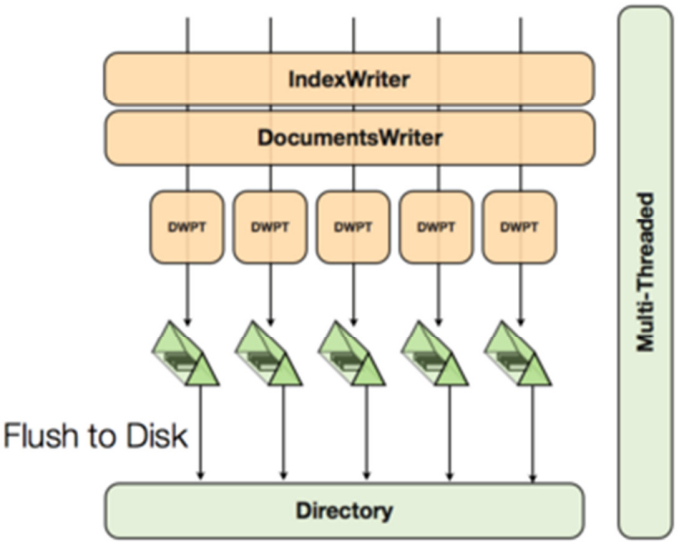

Figura 9 - DWPT escrevendo em seus próprios segmentos privados [25]

Durante uma atualização de documento na base de dados, o que deveria acontecer internamente era a inserção atômica do documento acompanhada da remoção de todas referências anteriores do mesmo documento; note que esta ordem de deleção antes da inserção precisa ser respeitada para manter a consistência na base de índices; a remoção era feita atomicamente e todas as threads removiam suas referências do documento, mas no novo modelo desejava-se algo que fizesse melhor proveito do paralelismo. A solução encontrada, sem fazer uso de locks, foi de manter uma lista encadeada visível entre as threads e usada pelas threads para registrar seus eventos de inserção de documento na fila, cuja inserção acontecia atomicamente por meio de operações do tipo *compare-and-swap*, ou seja, não necessitava de uso de locks.

No exato momento da inserção, a thread deveria observar os elementos anteriores ao elemento atual, varrendo a lista ao contrário, até chegar no último evento de criação do mesmo documento ou no fim da lista, e para cada elemento visitado na lista, a thread deveria fazer a remoção do documento relacionado; dessa forma, é garantido que todas as threads, no momento em que terminarem de inserir documentos, respeitassem as requisições de remoção de documentos recém-criados, mantendo a integridade da base de índices invertidos [24].

Análise de impacto das mudanças

As modificações relacionadas a *concurrent flushing* aconteceram inicialmente em um *branch* separado e no bug 3023 essas modificações foram integradas no repositório principal na revisão cujo número é 1098592. As modificações aplicadas neste bug chegam a mais de 350 kB, e seu arquivo "LUCENE-3023-svn-diff.patch" contém modificações em 69 arquivos no formato ".java", somente na pasta de arquivos relacionados a indexação; destes, 14 foram deletados, 43 modificados e 12 adicionados. Essa informação pode ser obtida utilizando o exemplo de código a seguir e com os parâmetros "1098592 java lucene/index".

svn-stats-summary.bash Shell

```
1  echo "Checking summary for r$1..."
2  svn diff --summarize -c $1 > /tmp/patch
3  cat /tmp/patch | grep .$2 > /tmp/patch2
4  if [ $3 ]; then
5    cat /tmp/patch2 | grep $3 > /tmp/patch3;
6    mv /tmp/patch3 /tmp/patch2
7  fi
8  echo "Num of modified files (.$2): `cat /tmp/patch2 | wc -l`"
9  echo " * Deleted: `cat /tmp/patch2 | grep --regex='^D' | wc -l`"
10 echo " * Modified: `cat /tmp/patch2 | grep --regex='^M' | wc -l`"
11 echo " * Added: `cat /tmp/patch2 | grep --regex='^A' | wc -l`"
```

Figura 10 - script para calcular número de arquivos modificados

Utilizando o script de calcular número de linhas adicionadas disponibilizado na figura 2 com o arquivo de modificações como entrada, obteve-se o número -1302, resultante de 1871 linhas adicionadas e 3173 linhas removidas; ou seja, apesar das modificações terem sido drásticas, no final, o código resultante pode ter ficado menos complexo.

Através do sistema de busca de bugs do Lucene[14], pesquisas foram realizadas no projeto "Lucene: Core", a partir do dia 27 de abril de 2011, época em que

[14] https://issues.apache.org/jira/issues

as modificações estavam sendo aplicadas, cujas *queries* foram “test* && (DWPT || DocumentsWriterPerThread)” e “fail* && (DWPT || DocumentsWriterPerThread)”, ou seja, precisava ter os termos “DWPT” ou “DocumentsWriterPerThread” e também algum termo iniciado por “test” ou “fail”. Nas duas consultas, foram encontrados 19 e 15 bugs com alguns itens em comum; entre os vários bugs, houve falhas em testes já existentes, *deadlocks*, *memory leaks*, falhas na execução do mecanismo de atualização de documentos durante uma indexação, bugs oriundos da quantidade de threads em execução e a quantidade de memória que cada thread pode suportar[15].

Resultados

Membros da comunidade criaram benchmarks[16] utilizando documentos de todas as palavras da Wikipédia em inglês para medir o tempo necessário para que a indexação de todos eles fosse concluída. Esses benchmarks são executados para cada nova revisão do projeto possibilitando a detecção de grandes melhorias ou pioras bruscas de performance ao longo do tempo. Quando as modificações relacionadas foram integradas no repositório principal, esse benchmark passou a usá-lo também, e daí o resultado foi surpreendente: melhora de 265% após alguns ajustes, como número de threads e quantidade de memória por thread, mas na mesma configuração já apresentou melhoria de pelo menos 100%, com o custo de tornar as buscas um pouco mais lentas devido à grande quantidade de segmentos privados usados.

[15] https://issues.apache.org/jira/browse/LUCENE-4182
[16] http://people.apache.org/~mikemccand/lucenebench/indexing.html

Figura 11 - indexação acelerou em 265% com concurrent flushing [26]

Para alcançar estas modificações, houve muitas discussões[17][18], geralmente longas, em bugs durante o desenvolvimento no branch separado, ilustrando o quão complicado é alterar uma arquitetura de uma aplicação, principalmente quando ela é biblioteca de centenas de outras aplicações de modo a causar o mínimo de instabilidade.

Os bugs que foram encontrados depois da inclusão das mudanças não foram frequentes e a bateria de testes existente tem se mostrado eficiente na detecção de regressões como visto nas consultas, pois pelo menos 9 dos 19 bugs foram encontrados através de falhas em testes.

[17] https://issues.apache.org/jira/browse/LUCENE-2293
[18] https://issues.apache.org/jira/browse/LUCENE-2324

Capítulo 7 - Conclusão

Neste trabalho, foram analisados três projetos de código aberto durante modificações em sua estrutura para fazer melhor uso da paralelização, largamente disponível em *hardware* hoje em dia através dos processadores *multicore*.

O WebKit tem sofrido diversas modificações em várias áreas, mas poucas delas afetam drasticamente o seu núcleo compartilhado; normalmente, as mudanças são restritas a alguns navegadores, protegidas por flags de compilação. Essa abordagem de tornar modificações que envolvem paralelização opcionais é sensata, dado a complexidade que esse tipo de programação pode trazer ao sistema e permite fazer comparações do tipo A/B (antes e depois das modificações). Mesmo em porções menos críticas do projeto, como o mecanismo de *parallel garbage collector* usado pelo módulo JavaScriptCore, a flag foi adicionada, e a comparação utilizando benchmarks comuns foi realizada, mostrando que tratava-se de um ganho relevante na performance. Similarmente, o suporte a *threaded scrolling* foi adicionado ao projeto protegido por flags e ativado somente em condições restritas; porém, a análise da frequência de bugs mostrou que o componente Scrolling Coordinator, foco dessas modificações, ainda está em fase de estabilização: pelo menos 9% dos bugs encontrados eram críticos. A quantidade de bugs não é suficiente para mostrar que não vale a pena a transição: outros navegadores passaram a adotar esse componente com implementações distintas, mostrando que essa área tem grande relevância ao projeto.

Os desenvolvedores do Qt entenderam que seu sistema de pintura possuía sérios problemas de performance ao realizar pintura acelerada por hardware através da GPU, cuja demanda aumentava com o tempo, e decidiram repensar seu modelo totalmente, adotando o novo Qt Scene Graph. Essa decisão provocou mudanças drásticas e a reimplementação de partes de vários componentes para que funcionassem no novo modelo. Os resultados são satisfatórios: as aplicações escritas através de QML, a linguagem declarativa criada pelo Qt, podem adicionar uma série de

efeitos gráficos que antes seriam muito complexos de alcançar e com boa performance. A análise dos bugs apontam que ainda existem falhas na implementação, algumas vezes críticas como quebras de execução, mas a nova versão do Qt tem tido uma boa recepção pela comunidade.

O processo de indexação do Lucene tornou-se mais rápido ao aplicar técnicas de paralelização minimizando a necessidade de locks com o seu *concurrent flushing*. Através do monitoramento constante do projeto, percebeu-se que com a inclusão das modificações, a vazão de dados no seu módulo indexador aumentou consideravelmente; e, pela pequena quantidade de bugs encontrados após as mudanças, percebe-se o grau de maturidade das modificações, introduzidas no repositório principal após um longo período de desenvolvimento em um *branch* separado e com uma série de longas discussões entre contribuintes sobre qual seria a melhor estratégia e como evitar problemas de concorrência em vários casos, refletindo a importância de discussão na comunidade e a dificuldade para se encontrar uma boa solução.

Após este estudo, ficou mais clara a dificuldade de se fazer uma transição desse porte em um projeto que já é usado por muitas aplicações. É difícil que bugs não ocorram após mudanças tão significantes na estrutura dos projetos, mas se o projeto já contar com uma política razoável de criação de testes a cada nova modificação, com o tempo, uma grande quantidade de testes estará disponível e com eles será possível capturar regressões, principalmente durante mudanças tão importantes como as estudadas neste trabalho.

Também foi observado em todos os projetos a dificuldade de criação de testes de várias modificações durante essa transição; alguns comportamentos estabelecidos nas modificações exigiam o conhecimento do estado entre várias threads ou processos ao mesmo tempo e isso é difícil de ser reproduzido numa ferramenta de execução de testes apropriada, dado que não há como controlar o estado de todos eles numa simulação de um caso de uso real, normalmente simulada por testes automáticos.

Observa-se que, em particular, a área de testes ainda está muito defasada no contexto de paralelização. O maior desafio é poder simular em casos de testes reais o comportamento de componentes paralelos na aplicação e poder identificar se seu comportamento está de acordo com sua especificação. É importante também que estes testes sejam fáceis de serem escritos e entendidos por terceiros. Testes muito complexos, pelo contrário, dificultam a manutenção do código e é preciso ter cuidado para que não reflitam situações em que um usuário jamais poderia realmente alcançar.

A análise realizada neste trabalho poderia ter sido mais aprofundada se houvesse uma maior disponibilidade de tempo para realizá-la; portanto, é possível que alguns dados avaliados aqui ainda estejam superficiais e não conclusivos. No entanto, como o objetivo era identificar a dificuldade relacionada à migração de sistemas para tirar melhor proveito de paralelização, espera-se que esta análise seja suficiente no propósito de refletir o grande desafio que é realizar esse tipo de transição e ainda manter a estabilidade do projeto.

Uma proposta de trabalho futuro é tentar elaborar formas simples de testar códigos paralelos e aplicar esses conceitos em projetos reais. Outro trabalho possível é explorar em detalhes os designs de projetos reais e compreender quais decisões foram tomadas para se obter melhor uso de paralelismo sem comprometer a legibilidade do código nem sua estabilidade.

Em particular, tenho interesse de entender melhor as decisões de design do Chromium e de outros *ports* do WebKit e elaborar um estudo compreensivo sobre elas em trabalhos futuros.

Referências

1. INSIDE STORY. The Economist. **Parallel Bars**, 2011. Disponivel em: <http://www.economist.com/node/18750706>. Acesso em: 27 de novembro de 2012.

2. LI, T. et al. International Symposium on High Performance Computer Architecture. **Operating System Support for Overlapping-ISA Hetereogeneous Multi-core Architectures**, 2010. pp. 1-12.

3. YE, W.; HEIDEMANN, J.; ESTRIN, D. **An Energy-Efficient MAC Protocol for Wireless Sensor Networks**. INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. New York: IEEE. junho 2002.

4. BETTI, E. et al. **Operating System's Support for Multicore Systems: Current and Future Trends**. Technical Report SPRGTV-TR001 SPRG - TOR VERGATA, System Programming Research Group, Dept. of Computer Science, System and Industrial Engineering, University of Rome "Tor Vergata". [S.l.]. 2007.

5. ASANOVIC, K. et al. **A View of the Parallel Computing Landscape**, Communications of the ACM, v. 52, n. 10, p. 56-67, outubro 2009.

6. TORRES, W. et al. **Are Java Programmers Transitioning to Multicore? A Large Scale Study of Java FLOSS**. SPLASH Workshops 2011. New York: ACM. 2011. p. 123-128.

7. CATANZARO, B. et al. **Ubiquitous Parallel Computing from Berkeley, Illinois, and Stanford**, Micro, IEEE, v. 30, n. 2, p. 41-55, abril 2010.

8. BOCCHINO, R. L. et al. **Parallel Programming Must Be Deterministic by Default**. Proceedings of the First USENIX conference on Hot topics in parallelism. Berkley, California: [s.n.]. 2009. p. 4.

9. SUTTER, H.; LARUS, J. **Software and the Concurrency Revolution**, Queue, v. 3, n. 7, setembro 2005.

10. WIKIPEDIA. Wikipedia. **Lock (computer science)**. Disponivel em: <http://en.wikipedia.org/wiki/Lock_(computer_science)>. Acesso em: 22 de fevereiro

de 2013.

11. WIKIPEDIA. Wikipedia. **Non-blocking algorithm**. Disponivel em: <http://en.wikipedia.org/wiki/Non-blocking_algorithm>. Acesso em: 23 de fevereiro de 2013.

12. ADLER, D. Ohloh. **WebKit**. Disponivel em: <http://www.ohloh.net/p/WebKit>. Acesso em: 7 de fevereiro de 2013.

13. OPERA. Opera Press Releases. **Opera gears up at 300 million users**, 2013. Disponivel em: <http://www.opera.com/press/releases/2013/02/13/>. Acesso em: 14 de fevereiro de 2013.

14. SLETTA, G. Qt Graphics and Performance – Generating Content in Threads. **Qt Blog**, 21 de janeiro 2010. Disponivel em: <http://blog.qt.digia.com/blog/2010/01/21/qt-graphics-and-performance-generating-content-in-threads/>. Acesso em: 10 de fevereiro de 2013.

15. KALLAND, K. The convenient power of QML Scene Graph. **Qt Blog**, 2011. Disponivel em: <http://blog.qt.digia.com/blog/2011/03/22/the-convenient-power-of-qml-scene-graph/>. Acesso em: 17 de fevereiro de 2013.

16. KNOLL, L. Thoughts about Qt5. **Qt Blog**, 2011. Disponivel em: <http://blog.qt.digia.com/blog/2011/05/09/thoughts-about-qt-5/>. Acesso em: 17 de fevereiro de 2013.

17. SLETTA, G. A Qt Scenegraph. **Qt Blog**, 2010. Disponivel em: <http://blog.qt.digia.com/blog/2010/05/18/a-qt-scenegraph/>. Acesso em: 17 de fevereiro de 2013.

18. SLETTA, G. QML Scene Graph in Master. **Qt Blog**, 2011. Disponivel em: <http://blog.qt.digia.com/blog/2011/05/31/qml-scene-graph-in-master/>. Acesso em: 18 de fevereiro de 2013.

19. SLETTA, G. Render Thread Animations in Qt Quick 2.0. **Qt Blog**, 2012. Disponivel em: <http://blog.qt.digia.com/blog/2012/08/20/render-thread-animations-in-qt-quick-2-0/>. Acesso em: 15 de fevereiro de 2013.

20. WIKIPEDIA. Lucene, 2013. Disponivel em: <http://en.wikipedia.org/wiki/Lucene>. Acesso em: 19 de fevereiro de 2013.

21. BUSCH, M. Twitter Engineering: Twitter's New Search Architecture. **Twitter Engineering Blog**, 2010. Disponivel em: <http://engineering.twitter.com/2010/10/twitters-new-search-architecture.html>. Acesso em: 19 de fevereiro de 2013.

22. SENTHILVEL, G. Code Project. **Lucene Search Programming**, 2011. Disponivel em: <http://www.codeproject.com/Articles/272309/Lucene-Search-Programming>. Acesso em: 19 de fevereiro de 2013.

23. MCCANDLESS, M. Changing Bits. **Visualizing Lucene's segment merges**, 2011. Disponivel em: <http://blog.mikemccandless.com/2011/02/visualizing-lucenes-segment-merges.html>. Acesso em: 19 de fevereiro de 2013.

24. WILLNAUER, S. Trifork. **Lucene indexing gains concurrency**, 2011. Disponivel em: <http://blog.trifork.nl/2011/05/03/lucene-indexing-gains-concurrency/>. Acesso em: 20 de fevereiro de 2013.

25. WILLNAUER, S. Trifork. **Gimme all resorces you have - I can use them!**, 2011. Disponivel em: <http://blog.trifork.nl/2011/04/01/gimme-all-resources-you-have-i-can-use-them/>. Acesso em: 20 de fevereiro de 2013.

26. MCCANDLESS, M. Changing Bits. **265% indexing speedup with Lucene's concurrent flushing**, 2011. Disponivel em: <http://blog.mikemccandless.com/2011/05/265-indexing-speedup-with-lucenes.html>. Acesso em: 21 de fevereiro de 2013.