Vol.2019-OS-145 No.9
2019/2/28

WebKit における Unified Builds についての調査

窪田貴文[1]　鈴木勇介[1]　河野健二[1]

概要：巨大なソフトウェアプロジェクトのビルド時間を削減するために，Unified builds はシンプルだが有効な手法である．Unified builds では複数のソースファイルをバンドルすることで，異なるソースファイルから共通にインクルードされているヘッダに関わる処理を削減することができ，その結果，プロジェクト全体のビルド時間が大幅に削減できる．しかしながら，unified builds では一つ一つのコンパイルタスクが大きくなるので，incremental builds への悪影響が存在する．無闇矢鱈とソースファイルをバンドルすると incremental builds において無視できないオーバーヘッドが発生してしまう．WebKit のメーリングリストでは最大 20%（6 秒 → 7 秒）報告されているが，本調査では最大 479%（19 秒 → 110 秒）場合が存在することがわかった．
本研究では，incremental builds を犠牲にせずに unified builds の恩恵が得られるようなビルド方法がないか調査する．調査結果から以下の知見が得られた．1) インクルードするヘッダファイルの類似度が高いソースファイルをバンドルするべきである．2) コンパイル時間に差があるソースファイル同士はバンドルしないほうがよい．3) フロントエンドの処理が主ではないソースファイルはバンドルしないほうがよい．これらの結果をもとに WebKit の unified builds を改良するとフルビルド時間が 2.66% 改善し，incremental builds のオーバヘッドが 479% から 129% まで削減できた．

1. はじめに

ソフトウェアプロジェクトのコードサイズが大きくなるにつれて、コンパイル言語を使用するソフトウェアプロジェクトのビルド時間が問題になってきている．ビルド時間が長いと開発サイクルが遅くなり、開発者の生産性が落ちるという問題がある [15]．WebKit の build bot のログを調べたところ [9]，2018 年 3 月 17 日から 8 月 3 日の間に 1,387 個のビルドタスクが実行された．ビルドボットは 75 日間アクティブであり，15 分以上かかるビルドは，75 日のうち 61 日間で 164 回発生した．このような長いビルドでは、再コンパイルされたソースファイルの数は平均 1,030 であった．

伝統的に，ビルドシステムはビルドをより効率的にし，開発サイクルを加速するために使われてきた．ビルドシステム（GNU Make [5]、CMake [3]、Ninja [6]、Meson [7] など）の主な利点はインクリメンタルビルドである．インクリメンタルビルドでは、ソースファイル間の依存関係を管理し、更新されたファイルとそれらに依存するファイルのみを再コンパイルする．ビルドシステムは、ソフトウェアプロジェクトのますます複雑化する要求を満たすために進化している．メタビルドシステムの一つである CMake の最初のバージョンは 2000 年にリリースされました．Low-level のビルドシステムよりも開発者はより柔軟な構成ができるようになった．Ninja build system は 2012 年に最初にリリースされ，不必要な機能を削除することで非常に高速なインクリメンタルビルドを実現している．最近では，CCache [1] や cHash [11] などのコンパイラキャッシュツールによって，一つのソースファイルのコンパイルをスピードアップすることができる．これらのツールでは以前のコンパイル結果がキャッシュされ，再利用される．

```
#include "API/JSBase.cpp"
#include "API/JSCTestRunnerUtils.cpp"
#include "API/JSCallbackConstructor.cpp"
#include "API/JSCallbackFunction.cpp"
#include "API/JSCallbackObject.cpp"
#include "API/JSClassRef.cpp"
#include "API/JSContextRef.cpp"
#include "API/JSHeapFinalizerPrivate.cpp"
```

図 1. 例: unified source file

大量のソースファイルをコンパイルすることで生じる長いビルド時間を短縮するために，いくつかのソフトウェアプロジェクトでは *unified builds*（または *unity builds*）というテクニックを使用している．たとえば，WebKit [8] はデフォルトのビルド処理として Unified builds を採用し

[1] 慶應義塾大学
Keio Uniersity

ている citewebkit-dev0，そして Chromium [2] は *Jumbo builds* としてサポートしている [4]．Unified builds は複数のソースファイルをまとめてコンパイルすることでコンパイル時間を短縮している．図 **??**は、8 つのファイルを 1 つにマージしている．

この手法では，以下の理由からビルド時間が大幅に短縮される．Unified builds では異なるソースファイル間で共通にインクルードされるヘッダーファイルについての冗長な処理を削減する．ヘッダファイルに複数のソースファイル間で使用されるデータ構造，クラス，テンプレートを定義されている．そのため，複数のソースファイルが共通のヘッダファイルをインクルードすることはよくある．特に C++ では，Unified builds では冗長なテンプレートのインスタンス化を省ける．テンプレートのインスタンス化では，引数の型に応じてコードを生成するため非常に重い処理になる．Unified builds ではコンパイラが複数のソースファイル間でテンプレートのインスタンスを共有できるためコンパイル時間が短縮できる．WebKit では Unified builds により，全体的なビルド時間が 56m：05s から 26m：18s に短縮された．

Unified builds にはこのような利点があるが，インクリメンタルビルドには悪影響が生じる．例えば，同じ unified source file にマージされているソースファイルが更新されたとき，すべてのソースファイルを再コンパイルする必要があるからである．我々の調査によって WebKit では、インクリメンタルビルドは 479 %（19s → 110s）も遅くなる可能性があることがわかった．また， WebKit の開発者メーリングリスト [12] では、Unified builds によって引き起こされるオーバーヘッドについて積極的に議論されている．

本研究では，どのようなソースファイルが同じ unified source file にバンドルされるべきかについて調査する．現在では，WebKit や Chromium では，同じディレクトリに存在するソースファイルを決まった数でバンドルする．しかし，このような ad-hoc かつソースファイルの特徴を無視した方法では，不必要にインクリメンタルビルドが犠牲になってしまう．本研究の目標は．フルビルド時間を減らし，インクリメンタルビルドの時間を増やさないような Unified builds の方法を調査することである．

本研究には大きく 2 つの contributions がある．

- WebKit の Unified builds について調査し，その効率性について分析している．
- また，実験的に WebKit の Unified builds を改善し，フルビルド時間を 2.66% スピードアップし，インクリメンタルのスローダウンをワーストケースで 479% から 129% まで減らした．

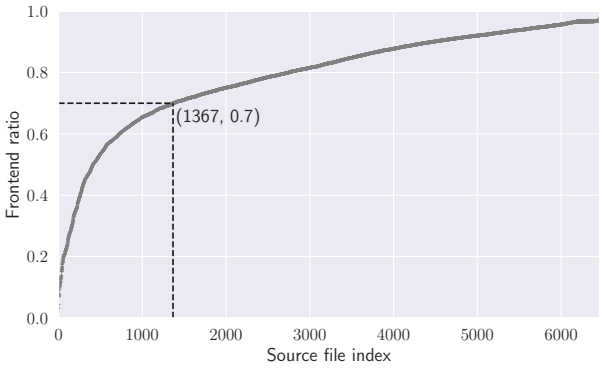


図 2. 各ソースファイルのコンパイル時間におけるフロントエンドの割合

2. 背景

本研究ではまず，ビルド中のコンパイルにおいてどの処理がボトルネックになっているかを調べた．図 2 は各ソースファイルにけるコンパイルでのフロントエンドの処理の割合を示している．図から，79% のソースファイルはフロントエンドの処理時間がコンパイル時間の 70%以上を占めていた．これは，ヘッダファイルに処理が非常におもいことを示している．ヘッダファイルの処理にはファイル読み込み，パース，テンプレートのインスタンス化などフロントエンドにおいて冗長かつおもい処理が含まれるからである．

2.1 Unified Builds

Unified builds は先程あげたヘッダファイルによって生じる冗長な処理を削減できる．Unified builds では複数のソースファイルを一つにバンドルすることで，共通にインクルードしているヘッダファイルの処理結果を共有できるからである．図 3 は Unified builds の概略を示している．図では共有しているヘッダファイルが多い s1 と s2 がバンドルされ，コンパイラにタスクとして渡される．WebKit [8] と Chromium [2] では Unified builds を採用している [4, 13]．その結果，ビルド時間が大幅に短縮され，Webkit では 53% (65m:05s → 26m:18s) 削減され，Chromium では 66% (2h:33m:40s → 52m:24s) 削減された．

Unified builds の欠点はその性質からインクリメンタルビルドに悪影響を与えることである．これは，Unified builds によって複数のソースファイルファイルが一つにまとめられるため，たとえ一つのソースファイルだけを更新したとしてもすべてのソースファイルを再コンパイルしてしまうからである．図の例だと s1 だけを更新したとしても s2 のコードも再コンパイルされてしまう．Webkit の開発者のメーリングリストではインクリメンタルビルドのスローダ

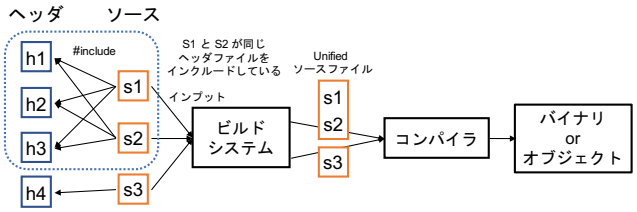


図 3. Unified builds の概略

ウンは 20% (6s → 7s)　と報告されているが，実際には 479% (19s → 110s) となるケースが存在した．

2.2 Motivation

前述の通り，Unified builds は有効だが，その効率性についての定量的調査がまだなされていない．その結果，unified builds のさまざまな側面がまだ明らかにされておらず，開発者たちが unified builds を使う際の障害になっている．例えば，unified builds とインクリメンタルビルドにはトレードオフが存在するが，実際に定量的にビルド時間を比較されてはいない．また，すでに採用している場合でも ad-hoc な経験則に基づいて使用されている．しかしながら，そのようなヒューリスティックは最適でなくビルド時間の削減が限定的で，インクリメンタルビルドを犠牲にしている．

本研究では unified builds について定量的な調査を行う．WebKit においての unified builds の効率について様々な設定を用いて解析する．そして，インクリメンタルビルドに影響を与える unified builds の特性について調査する．最後に調査結果に基づいて，インクリメンタルビルドに悪影響を与えずにビルド時間を削減できるような unified strategy を示す．

3. 調査

3.1 Research Questions

本研究では 5 つの research questions について調査し，unified builds について明らかにされていないヒューリスティックについて明らかにする．

- **RQ1: ヘッダの類似度はどの程度 unified builds に影響を与えるのか？**
- **RQ2: どのようなソースファイルの組み合わせが unified builds に適していて，適していないのか？**
- **RQ3: バンドルするソースファイルは一つの directory に限定すべきなのか?**
- **RQ4: バンドル数はいくつがいいのか？**
- **RQ5: WebKit においてどのくらい unified builds は改善するのか？**

3.2 Metrics

調査には次のような指標を用いた．

- **ヘッダの類似度．** 計算には各ソースファイルでインクルードされるヘッダファイル群の Jaccard 係数を使用した．

$$Similarity(s1, s2) = \frac{Headers(s1) \cap Headers(s2)}{Headers(s1) \cup Headers(s2)}$$

- **フロントエンド比率．** あるソースファイルのコンパイルにおいてフロントエンドの処理が占める割合．コンパイラのタイマレポート (`-ftime-report`) の機能を利用した．
- **コンパイルスピードアップ．** unified builds によるコンパイル時間のスピードアップは次のように計算した．

$$Speedup(s1, s2) = \frac{CT(s1) + CT(s2)}{CT(unify)}$$

- **インクリメンタルビルドへのペナルティ．** unified builds によるインクリメンタルビルドのスローダウンのペナルティは次のように計算した．

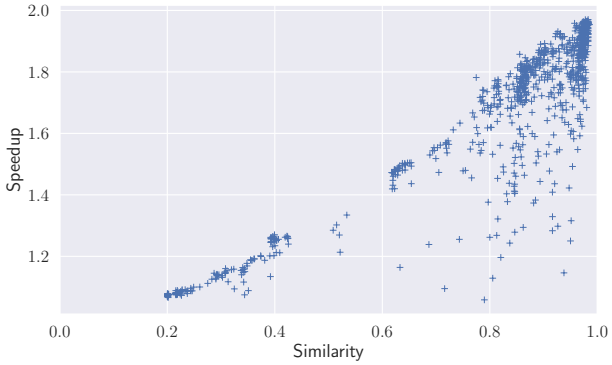
$$Penalty = \frac{CT(unify)}{CT(s1)} \times 0.5 + \frac{CT(unify)}{CT(s2)} \times 0.5$$

4. 調査結果

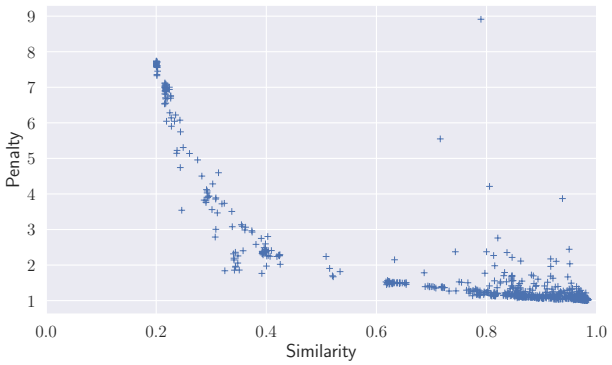
4.1 RQ1: ヘッダの類似度はどの程度 unified builds に影響を与えるのか？

ヘッダの類似度がどの程度 unified builds のパフォーマンスに影響を与えるかを調べるために，Webkit の中からランダムに 1 つに選び，その他のソースファイルと unified builds をし，その時のコンパイルのスピードアップとインクリメンタルビルドへのペナルティを計算した．

図 4 は結果を示している．図から基本的にヘッダの類似度が高いほうが unified builds によって得られるコンパイルのスピードアップが高いことがわかる．例えば，類似度が 0.9 以上だと平均 1.85 のスピードアップが得られる．一番いいケース (`jit/JITStubRoutine.cpp` との場合) で 1.97 ものスピードアップが得られた．これはインクリメンタルビルドにとって良い効果をもたらす．なぜなら，unified builds によって得られるスピードアップが高ければ高いほど，インクリメンタルビルドへの悪影響が小さくなるからである．たとえば，`API/JSBase.cpp` と `jit/JITStubRoutine.cpp` の場合，インクリメンタルビルドのペナルティは 1.02 はだけである．これらのことから，正しく unified builds を行えばインクリメンタルビルドに悪影響を与えずにコンパイルのスピードアップが得られることがわかる．

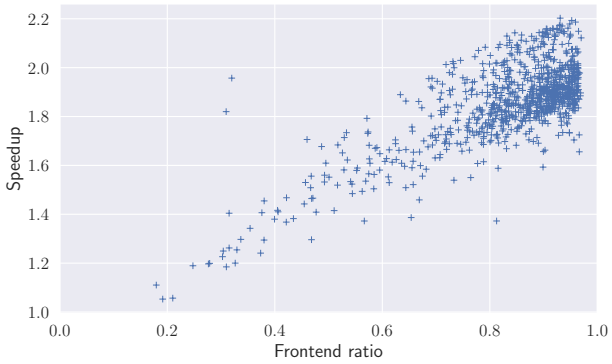


(a) コンパイルのスピードアップ (higher is better).

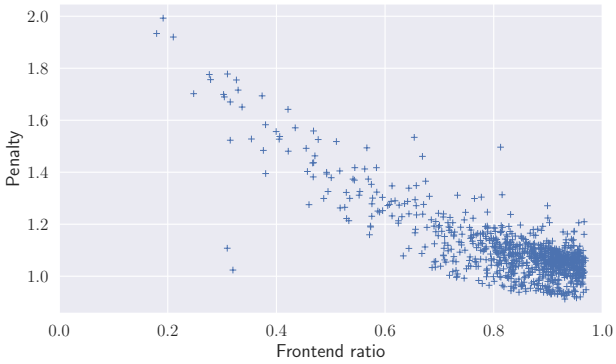


(b) インクリメンタルビルドのペナルティ (lower is better).

図 4. `API/JSBase.cpp` とその他のファイルとの unified builds でのヘッダの類似度が unified builds のパフォーマンスに与える影響



(a) コンパイルのスピードアップ (higher is better).



(b) インクリメンタルビルドのペナルティ (lower is better).

図 5. フロントエンドの比率が unified builds のパフォーマンスに与える影響

4.2 RQ2: どのようなソースファイルの組み合わせが unified builds に適していて，適していないのか？

Unified builds がうまくいかないケースは大きく分けて 2 つある．一つは，コンパイル時間に差があるソースファイルを unified builds をする場合である．例えば，`API/JSBase.cpp` and `parser/Parser.cpp` の場合である．この 2 つを unified builds をするとヘッダファイルの類似度が 0.94 高いにもかかわらずスピードアップはたった 1.15 である，これは 2 つのソースファイルのコンパイル時間に差があるからである．`API/JSBase.cpp` には 4.29s かかるが，`parser/Parser.cpp` には 28.81s もかかる．このとき，unified builds のパフォーマンスゲインが長い方のコンパイル時間で限定される．

もう一つのケースは，フロントエンドの処理がドミナントではないソースファイルに unified build しても効果は限定的である．Unified builds では主にヘッダファイルによるフロントエンドの冗長な処理を削減するので，フロントエンドの処理が少ないソースファイルのコンパイルには unified builds は向いていない．

これを示すために，図 4 と同様な実験をほかのソースファイルにも実行し，ソースファイルのフロントエンドの比率をもとに分類した結果が図 5 である．この図によると、フロントエンドの比率は、コンパイルのスピードアップとインクリメンタルビルドのペナルティに寄与している．フロントエンドの比率が 1.0 になると，コンパイルのスピードアップはほぼ直線的に 2.0 になります．フロントエンド率が高いソースファイルに unified builds を適用すると、インクリメンタルビルドのペナルティが小さくなることも示している．ただし，フロントエンドの比率が低いと、コンパイルのスピードアップは限定的でペナルティが増加し、インクリメンタルビルドの速度が遅くなります。

4.3 RQ3: バンドルするソースファイルは一つの directory に限定すべきなのか?

現在，WebKit では unified builds の範囲は同じディレクトリ内に限定されている．そこで，各ディレクトリ間のヘッダファイルの類似度を計算すると図 6 のようになった．図からたとえ，同一ディレクトリ内でなくても類似

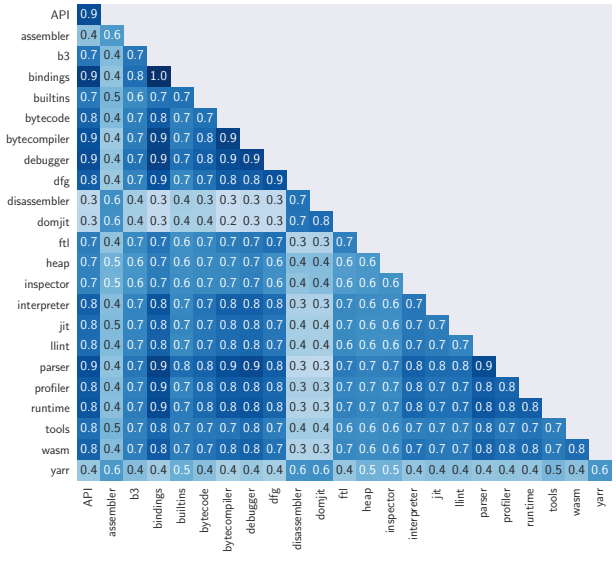


図 6. ヘッダファイルの類似度

度の高いディレクトリの組み合わせがあることがわかる．よって，unified builds は一つのディレクトリに限定しなくてもよい．

4.4 RQ4: バンドル数はいくつがいいのか？

ここでは，unified builds における適したバンドル数について議論する．Unified builds のパフォーマンスはソースファイルのフロントエンドの比率に依存するため，適切なバンドル数もフロントエンドの比率が関係してくる．表 1 はフロントエンドの比率が異なる 3 っつのソースファイルについてバンドル数を変えて unified builds のパフォーマンスを測っている．表から，フロントエンドの比率が高いソースファイルに関してはバンドル数を多くしても unified builds がスケールしていることがわかる．しかし，比率が低い場合は逆にスケールしていないことがわかる．

4.5 RQ5: WebKit においてどのくらい unified builds は改善するのか？

ここでは既存の Webkit の unified builds を改善できるか実験する．調査結果から unified builds のパフォーマンスを向上させるために，次のような unifiy ルールを設定する．

- ヘッダファイルの類似度が 0.9 以上のソースファイル同士のみをバンドルする．
- コンパイル時間に 50% 以上の差がある場合はバンドルしない．
- フロントエンドの比率が 0.2 以下のソースファイルはバンドルしない．
- 別ディレクトリのソースファイルもバンドルする．
- フロントエンドの比率によって最大バンドル数を変える．

表 2 がルールありとなしとで unified builds を実行したときのパフォーマンスの比較結果である．ここでは *Guided* はルールありを示しており，*WebKit* は既存の unified builds を示している．カッコは最大バンドル数を示している．表からわかるとおり，Guided は WebKit に対してすべてのケースでインクリメンタルビルドのスローダウンを削減できている．ビルド時間においてバンドル数が 16 と 32 の場合において負けているが，これは WebKit はインクリメンタルビルドを犠牲にしてビルド時間を短縮しているからである．また，WebKit(8) と Guided(32) 比較するとバンドル数を多くしているのにもかかわらず，インクリメンタルビルドの悪影響を抑えながらビルド時間を短縮できていることがわかる．以上のことから unified builds は適切に行えば，インクリメンタルビルドへの悪影響を抑えながらビルド時間を短縮できることがわかる．

5. 関連研究

私たちの知る限りでは、本研究はインクリメンタルビルドの減速を考慮した unified builds の効率に関する最初の調査である．しかし，いくつか Unified builds の必要性を理解するのを助けとなる既存の技術が存在する．

Build Systems: ビルドシステム (e.g., GNU Make [5], CMake [3], Ninja [6], Meson [7] など)　はソフトウェア開発にとって欠かせないものである．しかし，ソフトウェアプロジェクトのコードサイズの増加とともに既存のインクリメンタルビルドのみをターゲットとするデザインではスケールしなくなってきている．

本研究では，将来的な unified builds を組み込んだビルドシステムの新たなデザインの手助けとなるデータと指針を示している．

Implement new compilers: ヘッダファイルの冗長な処理を解決する一つの手段として，コンパイラの再設計がある．Zapcc [10]　は新しいコンパイラであり，内部でコンパイル結果をキャッシュし再利用することでソースファイル間でヘッダファイルの処理内容を共有できる．Zapcc は非常に有効だが，既存のコンパイラに大規模な変更が必要になる．本研究では，コンパイラの改変を一切必要ない．

C++ Modules: ヘッダファイルの冗長な処理を解決するために，C++ に モジュール機能 [14] が導入されつつある．しかし，依然としてモジュールの仕様が決定されておらず多くのプロジェクトが対応できていない．Unified builds は現在，大規模なプロジェクトのコンパイル時間を短縮できる現実的なソリューションとなっている．

表 1. バンドル数の変化よる unified builds のパフォーマンス比較

ソースファイル	フロントエンドの比率	コンパイル時間	Unified builds				
			2	4	8	16	32
JSBase.cpp	0.90	4.26	4.36	4.43	4.46	4.61	4.97
LiteralParser.cpp	0.52	10.59	7.84	8.87	11.08	14.90	20.80
Parser.cpp	0.18	28.62	57.10	84.98	124.82	183.88	307.50

表 2. Unified builds のパフォーマンス

ラベル (最大バンドル数)	# of unified ソースファイル	フルビルド時間 (-j16)	インクリメンタルビルドのスローダウン			
			mean	P90	P99	the worst case
WebKit (8)	665	26m 21s	21%	58%	126%	479% (19s → 110s)
Guided (8)	861	25m 39s	5%	22%	62%	129% (24s → 55s)
WebKit (16)	416	22m 47s	41%	91%	214%	638% (16s → 118s)
Guided (16)	651	23m 09s	13%	41%	88%	140% (20s → 48s)
WebKit (32)	294	21m 08s	74%	147%	300%	793% (14s → 125s)
Guided (32)	556	21m 58s	27%	69%	124%	145% (22s → 54s)

6. まとめ

本研究は unified builds の包括的な調査を行っっている．WebKit を詳細に分析した結果、4 つの知見が得られた．WebKit の unify ルールを改善したしかにビルドパフォーマンスが向上するか実験をおこなった．実際に，バンドル数が 8 のときビルド時間が 2.66% 減少し，インクリメンタルビルドのオーバーヘッドも減少している．

これらの調査結果が、よりインテリジェントな unify ルールを導き出すのに役立ち、将来のビルドシステム、コンパイラ、モジュールシステムに関する議論の基礎として役立てることを願う．

謝辞 本研究は，JST, CREST, JPMJCR1683 の支援を受けたものである．

参考文献

[1] : ccache — Compiler cache. https://ccache.samba.org/.
[2] : The Chromium Projects. http://www.chromium.org/.
[3] : CMake. https://cmake.org/.
[4] : Jumbo / Unity builds. https://chromium.googlesource.com/chromium/src/+/lkgr/docs/jumbo.md.
[5] : Make - GNU Project - Free Software Foundation. https://www.gnu.org/software/make/.
[6] : Ninja, a small build system with a focus on speed. https://ninja-build.org/.
[7] : The Meson Build system. https://mesonbuild.com/.
[8] : WebKit. https://webkit.org/.
[9] : WebKit Build Central. https://build.webkit.org/.
[10] : Zapcc – A (Much) Faster C++ Compiler. https://www.zapcc.com/.
[11] Dietrich, C., Rothberg, V., Füracker, L., Ziegler, A. and Lohmann, D.: cHash: Detection of Redundant Compilations via AST Hashing, *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)* (2017).
[12] Garen, G.: [webkit-dev] Growing tired of long build times? Check out this awesome new way to speed up your build... soon (HINT: It's not buying a new computer) (2017). https://lists.webkit.org/pipermail/webkit-dev/2017-August/029508.html.
[13] Miller, K.: [webkit-dev] Unified source builds: A new rule for static variables (2017). https://lists.webkit.org/pipermail/webkit-dev/2017-August/029465.html.
[14] Reis, G. D., Hall, M. and Nishanov, G.: A Module System for C++ (Revision 4) (2016). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0142r0.pdf.
[15] Sathyanathan, P. W., He, W. and Tzen, T. H.: Incremental Whole Program Optimization and Compilation, *Proceedings of the 2017 International Symposium on Code Generationand Optimization (CGO)* (2017).