大規模ソースコードを対象としたコードクローンの検出と可視化

肥後芳樹[†] リビエリ シモネ[†]
松下誠[†] 井上克郎[†]

コンピュータハードウェアが安価になり，分散処理方式はソフトウェア分析のための現実的な選択肢の 1 つとして用いられるようになった．本論文では，超大規模ソースコードからコードクローンの検出，および可視化を行うシステム D-CCFinder について述べる．D-CCFinder は 80 台のコンピュータを用いた分散型コードクローン検出システムであり，検出されたコードクローン情報は散布図などを用いて可視化される．D-CCFinder は約 4 億行のソースコードから 2 日余りでコードクローン情報を収集し，頻出するコードを容易に特定することができた．

A Code Clone Detection and Visualization for Large-Scale Source Code

Yoshiki Higo,[†] Simone Livieri,[†] Makoto Matsushita[†]
and Katsuro Inoue[†]

The increasing performance-price ratio of computer hardware makes possible to explore a distributed approach at code clone analysis. This paper presents D-CCFinder, a distributed approach at large-scale code clone analysis. D-CCFinder has been implemented with 80 PC workstations in our student laboratory, and a vast collection of open source software with about 400 million lines in total has been analyzed with it in about 2 days. The result has been visualized as a scatter plot, which showed the presence of frequently used code as easy recognizable patterns.

1. はじめに

近年，ソフトウェア分析手法としてコードクローン検出が注目を集めている[13),18)]．コードクローンとは，ソースコード中のある一部分（コード片）のうち，他のコード片と同一または類似しているものを指す．コードクローンはコピーアンドペーストなどのさまざまな理由によりソースコード中に作りこまれる．あるコード片にバグが含まれていた場合，そのコード片のコードクローンすべてについて修正の是非を考慮しなければならない．このようなことから，コードクローンはソフトウェアの保守を困難にしている要因の 1 つであると指摘されている．そのため，コードクローン検出を行うことは，効率的にソフトウェア保守を行うために有効であるといえる．また，コードクローン検出を行うことにより，単純な重複部分の調査だけではなく，ソフトウェアの進化を追うこともできる[14),20)]．

これまでに多くのコードクローン検出法とその適用事例が報告されているが[1),6),12),15),16)]，それらは単一または少数のソフトウェア内でのコードクローンの状態を調査しており，大量のソフトウェア間におけるコードクローン分析は行われていない．

しかし，近年，GNU プロジェクト[9)] や Jakarta プロジェクト[11)] に代表されるように多くのオープンソースソフトウェアが開発されており，その一部は他のソフトウェアでも用いられているとの指摘もされていることから[3)]，大量のソフトウェア間における利用関係を調査することにより，頻繁に再利用されているコードの特定など，有益な情報を取得できるであろう．

オープンソースソフトウェアに限らず，企業の開発部門など，組織規模でコードクローン分析を適用することにより，組織自体のソフトウェア開発プロセスを改善することができるとも思われる．たとえば，或る組織が過去に開発したすべてのソフトウェアからコードクローンを検出することにより，その組織で繰り返し実装されているコードを特定することができる．そのようなコードを社内ライブラリとしてまとめることにより，今後のソフトウェア開発をより効率的に行えるであろう．

† 大阪大学大学院情報科学研究科
Graduate School of Information and Science Technology, Osaka University

しかし，既存の検出法はどれも単一または少数のソフトウェアからのコードクローン検出を目的として提案されているものであり，そのまま大量のソフトウェア群に対して適用することはスケーラビリティの面から難しい．既存手法の中では，字句単位の検出手法を実装しているツール CCFinder[12] が，他の手法に比べスケーラビリティが高いことが知られているが[4]，一度に検出を行うことができる規模の上限は約 500 万行である☆．そこで本論文では，大量のソフトウェア群から短時間でコードクローンを検出するための方法と，検出したコードクローンの可視化について述べる．

本論文では，コードクローン検出対象として，オープンソースオペレーティングシステム FreeBSD[8] 用のソフトウェア集合 Ports システムに含まれている，C 言語で記述された約 4 億行のソースコード（以降，オープンソースターゲット）を用いた．この規模は，CCFinder の検出可能である限界を遥かに超えており，単一のコンピュータ上で一度に検出を行うことは不可能である．そこで，分散処理方式を用いて，検出対象を小さく分割し，各コンピュータに入力として与える．各コンピュータ上で CCFinder を実行し，割り当てられたソースファイルに含まれるコードクローンを検出する．このように対象を小さく分割することにより，CCFinder を用いて短時間でコードクローンを検出することができる．分割したすべてのタスクを単一コンピュータ上で逐次実行した場合も同じ結果を得ることはできるが，すべての検出処理を完了するには 45 日を要すると予測され☆☆，現実的ではない．

提案手法を分散型アプリケーション Distributed-CCFinder（以降，D-CCFinder）として実装した．D-CCFinder を大阪大学基礎工学部情報科学科の学生演習室のコンピュータ 80 台上で実行し，オープンソースターゲット内に含まれるコードクローンを検出したところ，約 2 日で検出を完了することができた．

本研究の成果は以下の通りである．

- 単一あるいは少数のソフトウェアに対する分析手法であったコードクローン検出を分散処理技術を用いることによって応用し，大量のソフトウェア群から短時間でコードクローンを検出する手法を提案した．
- 提案した検出手法をオープンソースターゲットに適用することにより，ソフトウェア間にどのようなコードクローンが存在しているのかを特定することができた．現時点では，可視化により際立った部分に存在するコードクローンを特定したにとどまっている．

以上の 2 点から，本論文はソフトウェア分析を分散処理環境を用いて行うことの有用性を示しているといえる．

以降，2 節では D-CCFinder の分散処理モデルを定義し，3 節ではその実装について述べる．4 節ではオープンソースターゲットへの適用について述べ，5 節ではその結果に対する考察を行う．6 節では関連研究について触れ，最後に 7 節では，まとめと今後の課題について述べる．

2. 超大規模ソースコードを対象としたコードクローン検出手法

2.1 検 出 対 象

本論文でのコードクローン検出対象は，オープンソースオペレーティングシステム FreeBSD 用のソフトウェア集合 Ports システムに含まれているソースファイル（オープンソースターゲット）であり，各ソースファイルは 1 つの**プロジェクト**に属している．すべてのプロジェクトは，zip, emacs, apache, windowmaker など，一意に特定可能な名前を持っている．また，本論文では検出対象を C 言語で記述されたソースコードに限定しているが，Java や COBOL などの CCFinder 自体が扱えるプログラミング言語であれば，同様に適用可能である．

共通の特徴を持ったプロジェクトは同じ**カテゴリ**に所属している．たとえば，emacs や vim，gedit などは editors カテゴリに所属している☆☆☆．

ユニットは，全ファイル集合を或るサイズ以下で分割した要素であり，1 つのユニットに含まれるソースファイルは単一プロジェクトからなる場合と複数プロジェクトからなる場合がある．また，ユニットのサイズは，使用するコンピュータの性能によって変えるべきである．

任意の 2 つのユニットで指定されるファイル集合間の対応を**ピース**といい，これが各コンピュータ上で実行される CCFinder への入力となる．もし，ユニットサイズよりも大きなソースファイルが存在した場合は，

☆ CPU:Xeon 2.8GHz, Memory: 4GB を用いた場合

☆☆ CCFinder の検出規模の上限である 500 万行で 4 億行を区切った場合，80 個に分割される．これを 2.3 節のモデルに当てはめると，$\frac{1}{2} \times 80 \times (80+1) = 3240$ 個のタスクが存在することになる．また，これまでの経験から 500 万行からのコードクローン検出には約 20 分を要すると想定し，総検出時間を計算すると 20 分 $\times 3240 = 64800$ 分 $= 45$ 日 となる．

☆☆☆ 各プロジェクトをカテゴリ分けしたのは Ports システムの管理者であり，著者らが行ったわけではない．

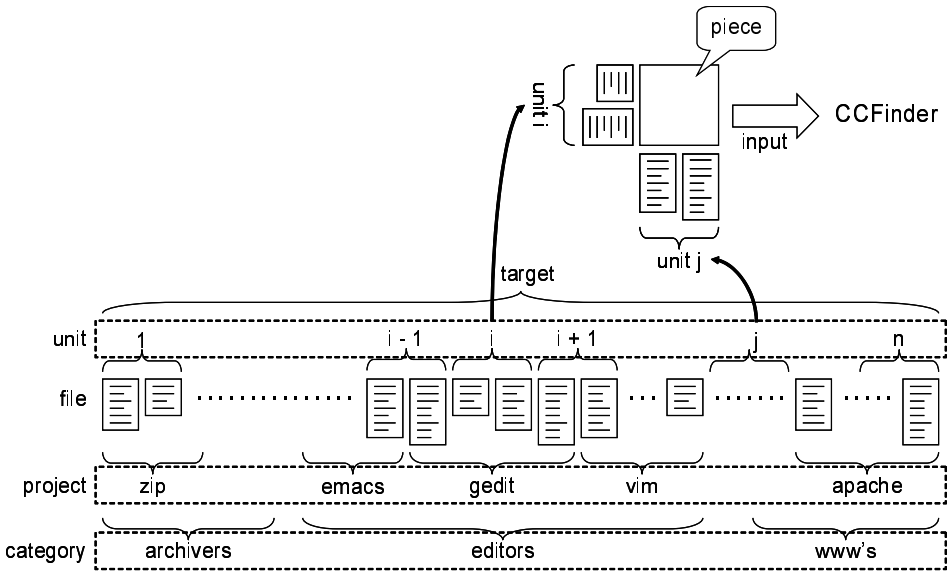


図 1 プロジェクト，カテゴリ，ターゲット，ユニット，ピース間の関係
Fig. 1 Relation between project, category, target, unit, and piece

そのファイル単体で 1 つのユニットを構成する．そのため，サイズの大きなピースが存在することになる．大きすぎるサイズのために CCFinder の実行が失敗してしまった場合でも，ファイルを分割して小さなピースを作成することは行わない．図 1 はプロジェクト，カテゴリ，ユニット，ピース間の関係を表している．

2.2 検出結果の表示

コードクローンには，コメントを除く部分がまったく同一の *exact* クローンと，変数名や関数名などのユーザ定義名が異なる *parameterized* クローンの二種類があるが[2)]，本研究ではこの両方を検出する．

検出対象の規模が非常に大きいため，大量のコードクローンが検出されることが予測できる．そのため，コードクローンの状態を容易に把握するためには，検出結果の抽象化を行う必要がある．提案手法では，コードクローン検出後に，ファイル，プロジェクト，カテゴリの各レベルで抽象化を行う．たとえば，各ファイル間，各プロジェクト間，カテゴリ間の重複の度合いや，単にコードクローンを共有しているかどうかといった抽象化も行う．

2.3 分散処理モデル

既存の検出法はどれも単一若しくは少数のソフトウェアからのコードクローン検出を目的として提案されており，そのまま大量のソフトウェア群に対して適用することはスケーラビリティの面から難しい．このため，検出対象を小さなユニットに分割してその組み合わせからピースを作成し，ピース単位で CCFinder を実行することにより，コードクローンを検出する．

図 2 は D-CCFinder の分散処理モデルを表している．検出対象の規模は nu とする．n は分割数であり，u はユニットのサイズを表している．このとき，任意のピースは (i, j) で表すことができる（ただし，$1 \leq i, j \leq n$）．ピース (i, j) に含まれるコードクローンはピース (j, i) に含まれるコードクローンと等価であるため，後者については検出を行わない．これにより，CCFinder を用いてコードクローンを検出しなければならないピースの数は $n(n+1)/2$ となる．なお，ピース内でのコードクローン検出がプロジェクト内のコードクローン検出を含む場合があるが☆，本論文ではプロジェクト間のコードクローンを検出することが目的であるため，プロジェクト内のコードクローンは検出しない．

D-CCFinder は，既存の CCFinder を複数のコンピュータで実行することにより，コードクローン検出を行う．各ピースの演算（コードクローン検出）は他のピースの演算結果にまったく依存しないため，タスクの割り当て処理は単純に行える．まだ一度も調べていないピースをアイドル状態のコンピュータに割り当て，検出結果を回収するだけでよい．

☆ ピース (i, i) や $(i, i-1)$ など，分散処理モデルにおいて主対角線上またはそれに近い位置に存在するピースでは，その垂直方向のユニットと水平方向のユニットに含まれるファイルが同一プロジェクトのものである場合がある．

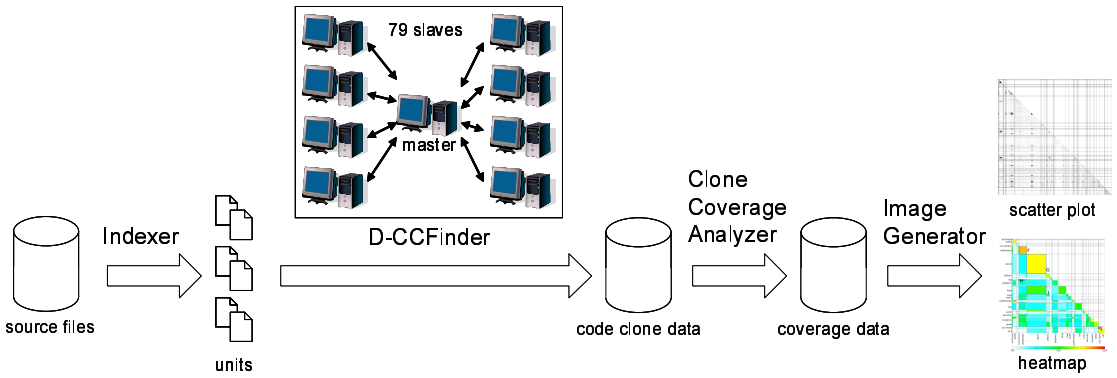


図 3 D-CCFinder の処理の流れ
Fig. 3 Process Overview for Code Clone Analysis using D-CCFinder

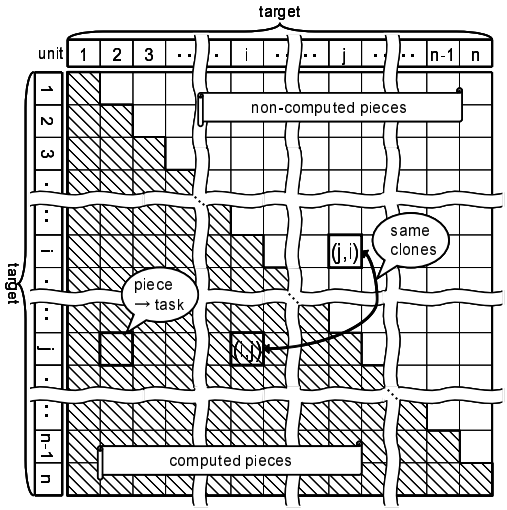


図 2 D-CCFinder の分散処理モデル
Fig. 2 Distributed Processing Model of D-CCFinder

3. D-CCFinder

D-CCFinder はマスタースレーブ型のシステムであり，各スレーブマシン上で CCFinder が実行される．マスターは，スレーブの実行状態を監視し，タスクを割り当てる．マスター・スレーブ間の通信は，Java RMI を用いて行われる．表 1 はマスターと各スレーブマシンの性能を表している，またマスター・スレーブ間は 100Mbps のネットワークで結ばれている．検出対象ソースファイルと検出結果はすべてのマシンがアクセス可能なファイルシステム上に存在し，各マシンは NFS 経由でアクセスする．D-CCFinder は大学の演習室のコンピュータ 80 台を用いて実装されており，1 台がマスター，残りの 79 台がスレーブである．

図 3 に示すように，D-CCFinder には対象ソースファイルに前処理を行うユーテリティ，検出結果を集約するプログラム，および検出結果から散布図などを生成するジェネレータが統合されている．

Indexer 検出対象ソースファイルを走査し，ファイルサイズ，行数，プロジェクト名，カテゴリ名などの情報を収集する．また，ユニットの境界を決定する．

マスターノード スレーブノード上の CCFinder の実行状態を監視する．アイドル状態のスレーブノードを発見した場合は，ユニット境界情報から，CCFinder の入力ファイル（そのユニットに含まれているソースファイルのパスのリスト）を生成し，スレーブノードにタスクを割り当てる．もし割り当てが失敗した場合は，そのタスクは他のスレーブに割り当てられる．

スレーブノード マスターノードから与えられた入力ファイルを用いてコードクローン検出処理を行う．検出対象ファイルはスレーブノードのローカル

表 1 マスター・スレーブノードの性能
Table 1 Spec. of the Master and Slave Nodes

CPU	Pentium IV 3GHz
メモリ	1 GBytes
オペレーティングシステム	FreeBSD 5.3-STABLE
利用可能 HDD 領域	40 ～ 50 GBytes

表 2 オープンソースターゲットのサイズ
Table 2 Characteristics of the open source target

カテゴリ数	45
プロジェクト数	6,658
.c ファイル数	754,552
総行数	403,625,067
総容量	10.8 GBytes

ファイルシステムにコピーされ，その後コードクローン検出が行われる．検出処理完了後もローカルファイルシステム上のコピーは削除されず，次回以降の検出処理のキャッシュとして利用される．

Clone Coverage Analyzer D-CCFinder の出力から，ファイル，プロジェクト，およびカテゴリレベルのコードクローンカバレッジを算出する．

Image Generator Clone Coverage Analyzer が生成した定量データから，散布図やヒートマップを生成する．

4. 実　　験

4.1 概　　要

本論文での実験対象は，オープンソースオペレーティングシステム FreeBSD[8] 用のソフトウェアの集合である Ports システムに含まれているソースファイル（オープンソースターゲット）である．表 2 に対象ソースファイルの規模，表 3 にカテゴリを示す．

オープンソースターゲットは同じプロジェクトの複数のバージョンを含んでいる場合がある．たとえば Apache web server の場合は，1.3，2.0, 2.1, 2.2 の 4 つのバージョンが含まれている．これは，古いバージョンを必要としているユーザやシステムとの下位互換性を保つためである．このように，複数のバージョンが含まれる場合は，それらの間で非常に多くのコードクローンが検出されることが予測される．

表 3 オープンソースターゲットのカテゴリ一覧
Table 3 Categories in the open source target

Index	Name	Index	Name
1	accessibility	24	math
2	arabic	25	mbone
3	archivers	26	misc
4	astro	27	multimedia
5	audio	28	net-im
6	benchmarks	29	net-mgmt
7	biology	30	net-p2p
8	cad	31	net
9	comms	32	news
10	converters	33	palm
11	databases	34	polish
12	deskutils	35	print
13	devel	36	science
14	dns	37	security
15	editors	38	shells
16	emulators	39	sysutils
17	finance	40	textproc
18	ftp	41	www
19	graphics	42	x11-clocks
20	irc	43	x11-fm
21	java	44	x11-fonts
22	lang	45	x11
23	mail		

また，コードクローンの状態を定量的に表すためにメトリックス $Coverage(M_0, M_1)$ を定義する．このメトリックスは，2 つのモジュール M_0 と M_1 がどの程度類似しているかを表すものであり，次式を用いて表現される．

$$Coverage(M_0, M_1) = \frac{LOC(C_{M_1}(M_0)) + LOC(C_{M_0}(M_1))}{LOC(M_0) + LOC(M_1)}$$

ただし:
M_0, M_1: ファイル，プロジェクト，またはカテゴリ，
$C_{M_1}(M_0)$: M_0 の中で，M_1 とコードクローンである部分，
$LOC(x)$: x の行数．

4.2 結　　果

最小一致字句数☆を 50，ユニットサイズを 15MBytes☆☆に設定し，D-CCFinder を実行した．実行されたタスクの総数は，269,745 個である．図 4 は全体の散布図を表している．この散布図では，1 ピクセルあたり 200x200 ファイルを表している．200x200 ファイル間で 1 つでもコードクローンが存在する場合は，点を描画している．ファイルレベルでの $Coverage(M_0, M_1)$ の平均は 4%であり，もしこのような縮退を行っていない場合は，これより遥かに点の少ない散布図になると思われる．

図 4 の特徴的な部分を枠で囲んでいる．これらの部分に対して，より詳細に調査を行った．

A この部分のコードクローンは php4 と php5 のソースコードが流用されていることを表している．図から読み取れるように，さまざまなカテゴリのプロジェクトに流用されている．

B この部分には X11 関係の 4 つのカテゴリが存在しており，それらの間で多くのコードクローンが検出された．その多くは X Window System の中心

☆ 最小一致字句数とは，CCFinder がコードクローンを検出する際に用いる閾値である．CCFinder はこの値以上の字句を持つコードクローンを検出する．著者らはこれまでの経験から，新規でコードクローン検出を行う際の閾値として 30 を用いているが，今回の実験対象は同じソフトウェアの複数のバージョンを含むことを考慮し 50 とした．

☆☆ ユニットサイズが大きいほどタスク数が減るため，大きいほうがよいのであるが，演習室のマシンスペックを考慮した結果，この値を用いた．実際に，ユニットサイズをこれよりも大きい値にして D-CCFinder を実行したところ，同じソフトウェアの異なるバージョン部分のピースなど，非常に多くのコードクローンを含んでいる部分において，メモリ不足により CCFinder の実行に失敗してしまい，正常にすべてのピースの検出処理を終えることができなかった．

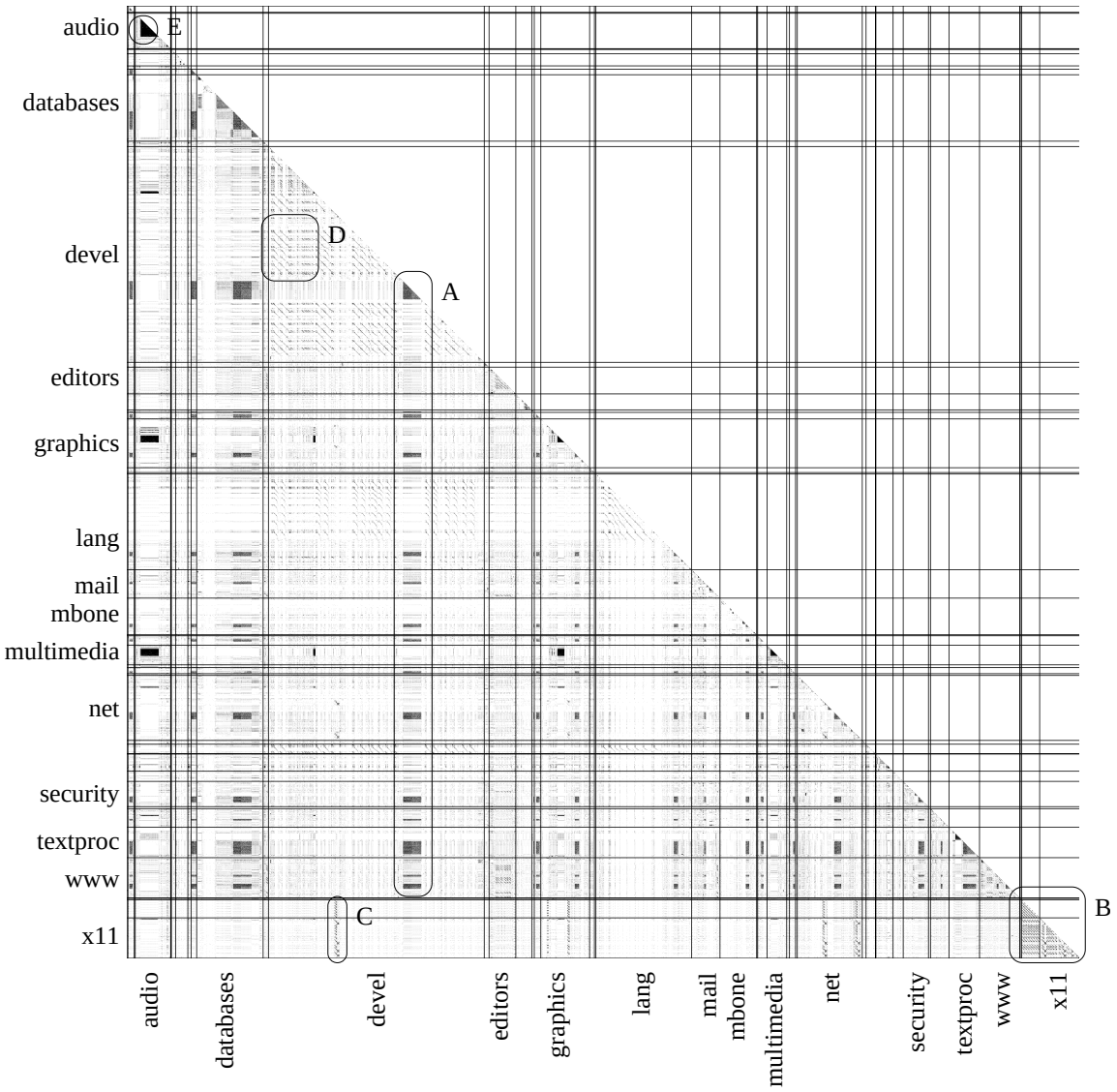


図 4 オープンソースターゲット全体の散布図
Fig. 4 Scatter Plot of Inter-Project Code Clone Coverage for the open source Target

的な処理を行っている部分からのコピーであった．

C imake は make に代表されるビルドツールの一種であり，X Window System の一部となっているソフトウェアである．このソフトウェア自体はカテゴリ devel に属しているが，X11 関係のソフトウェアの多くがこのコピーを所持していた．

D カテゴリ devel の大部分で一様なパターンが現われていた．これはソフトウェア binutils のコードの流用によるものである．binutils はリンカやアセンブラなどその他オブジェクトファイルやアーカイブを扱うためのソフトウェアツール群であり，カテゴリ devel に属するソフトウェアがこのツールを流用しているのは納得できる結果である．

E カテゴリ audio 内に存在するマルチメディアフレームワーク gstreamer とその複数のプラグインが多数の同一ファイルを所持していた．

上記のコードクローンを所有しているファイルに対して $Coverage(M_0, M_1)$ を計測したところ，ほとんどが 100%であった．つまり，これらのファイルはある一部ではなく，ファイル全体がコードクローンになっていることを表している．すでに述べたように図 4 の 1 ドットは 200x200 ファイルを表しているため，特定の 2 つのプロジェクト間でのみコードクローンになっている部分を発見することは難しい．

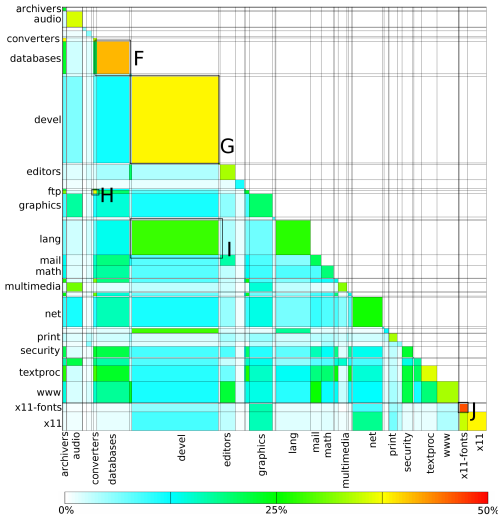


図 5 オープンソースターゲット全体のカテゴリレベルでのヒートマップ

Fig. 5 Color heatmap for the code clone coverage of the open source target (category view)

図 5 はカテゴリレベルでの $Coverage(M_0, M_1)$ のヒートマップを表している．この図から，主対角線上，つまりカテゴリ内のカバレッジがカテゴリ間のカバレッジに比べ高いことが分かる．異なるカテゴリ間の場合は，$Coverage(M_0, M_1)$ の値が 25%を超えている箇所は少ない．次に，図 5 の特徴的な部分にどのようなコードクローンが存在していたのかを示す．

F カテゴリ database の値が 41%と非常に高かった．これには 2 つの理由がある．1 つめの理由はこのカテゴリに属するいくつかのソフトウェアは，複数バージョンのソースコードが存在したこと，2 つめの理由は，ruby や php など異なるプログラミング言語向けにデータベースドライバが提供されている点であった．前者の理由により，ファイルの一部分のコードクローンが多く存在し，後者の理由により，ファイル全体のコードクローンが多く存在した．

G カテゴリ devel の値が 38%であった．このカテゴリにはプロジェクト binutils や gcc の複数のバージョンが含まれており，$Coverage(M_0, M_1)$ の値を押し上げていた．

H カテゴリ ftp と converters の間の値が 37%であった．これらのカテゴリに含まれる複数のソフトウェアがプロジェクト php4 と php5 のソースコードを流用しているため，$Coverage(M_0, M_1)$ の値が高くなっていた．

I カテゴリ lang と devel の間の値が 28%であった．これはカテゴリ devel 内に複数バージョンのプロジェクト gcc が存在しているためであり，このコードはカテゴリ lang に含まれるプロジェクトでも流用されていた．

J カテゴリ x11-fonts の値が 46%と最も高い数値であった．このカテゴリに属しているソフトウェアは少数であり，X Window System からのコードの流用が多く（7 箇所）行われていたため，このような高い数値になっていた．

5. 考　　察

5.1 分散環境下での分析について

表 4 に示すように，80 台のコンピュータ上で D-CCFinder を実行した結果，約 51 時間でコードクローン検出を完了することができた．理論上は，80 台のコンピュータを用いることにより 12 時間で検出が完了するはずであるが，実際にはネットワークトラフィックやマスター・スレーブ間の同期，CCFinder の出力の後処理などのため 51 時間を要した．この検出速度は単一のコンピュータ上で行った場合の約 20 倍である．現在，各コンピュータは 100BASE のスイッチで接続されているので，ギガビットスイッチなど，高速なネットワーク環境を導入することにより，検出速度の向上が見込まれる．

現在の実装では，Clone Coverage Analyzer と Image Generator は著者らの研究室内のワークステーション☆上で実行される．単一のワークステーションで実行されるため，これらの処理を完了するためには長い時間を必要としている．しかし，コードクローンの検出処理と同様に，これらの計算をスレーブノードに分割して割り当てることにより，処理速度の向上を図ることができる．

D-CCFinder をクラスタ計算機やグリッド計算機[7]で実現することも可能であろう．クラスタ計算機で実現する場合は，ネットワークの遅延などが減少し，全

表 4 Time elapsed
Table 4 実行時間

Indexer	22 分
D-CCFinder	51 時間
散布図	
Clone Coverage Analyzer	23 時間
Image Generator	4 時間
ヒートマップ	
Clone Coverage Analyzer	70 時間
Image Generator	2 分

☆ CPU:Xeon 2.8GHz, Memory: 4GB

体の効率の向上が期待される．グリッド計算機では，大量の入出力データの効率的な分配・回収方法を実現する必要があろう．

5.2 CCFinder について

CCFinder は，広く使われている実用的なツールである．本論文では，単純な分散処理モデルを用いることによって，単一コンピュータ用のアプリケーションである CCFinder を用いて，大規模ソフトウェアから短時間でコードクローンを検出することに成功した．しかし，単一コンピュータ用のアプリケーションを用いることに起因する問題点も存在した．

D-CCFinder 実行前は，コードクローン検出対象ソースファイルは，ある 1 つのマシン☆上にのみ存在する（このマシンを以降，データノードと呼ぶ）．各スレーブノード上で CCFinder が実行されるため，各ピースからコードクローンを検出する前に，必要なファイルをスレーブノードに転送する必要がある．スレーブノードが 79 台あり，これらすべてが必要なファイルをデータノードから取得するため，データノードのネットワークトラフィックが非常に大きく，D-CCFinder のボトルネックになっている．特に，各スレーブノードがローカルストレージにキャッシュを持っていない検出処理の開始直後は，データノードのネットワークトラフィックは最大になる．

現在の実装では，タスクの割り当ては単純にアイドル状態のノードに対して行っている．しかし，どのスレーブノードがどのファイルをキャッシュとして持つかを管理することにより，ソースファイルの総転送量が減少し，より短時間で検出処理を完了できるであろう．

5.3 散布図について

散布図による可視化により，大量のソフトウェア間に含まれるコードクローンを容易に特定することができた．仮にそれらのソフトウェアの開発者がこのコードクローンの存在をもっと早くに把握していた場合は，ライブラリなどのより再利用しやすい形でまとめられていたかもしれない．

今回の対象には，同一プロジェクトの複数のバージョンが存在していたため，それらの間で大量のコードクローンが検出されており，他のコードクローン情報を隠している（目立たないようにしている）と思われる部分があった．複数バージョンが存在しているプロジェクトについては，最新バージョンのみを使うなどの工夫をすることによって，より興味深い結果が得られるかもしれない．

また，散布図自体をより正確に生成する必要がある．現在の生成方法では，速度とサイズを重視しているため，精度が悪い（1 ピクセルが 200x200 ファイルを表している）．このため，対象全体の状態を大まかに把握することができるが，小さなプロジェクト間のコードクローンの状態を把握することはできなかった．

☆ このマシンは著者らの研究室内にあるネットワーク接続ストレージ（Network Attached Storage）である．

6. 関 連 研 究

超大規模ソースコードからのコードクローン検出の発想は，メガソフトウェアエンジニアリング[10] からのものである．メガソウトウェアエンジニアリングとは，既存のソフトウェア分析技術を，広く組織全体やオープンソースのソフトウェア群に対して適用することである．既存のソフトウェア技術は本来，個々のソフトウェアに対して適用するものであるが，組織規模で適用することにより，組織自体のソフトウェア開発プロセスを改善することができる．たとえば，組織（企業や企業内の開発部門）で過去に開発されたすべてのソフトウェアからコードクローンを検出することにより，その組織で繰り返し実装されている頻出コードを特定することができる．そのようなコードを社内ライブラリとしてまとめることにより，今後のソフトウェア開発をより効率的に行うことができると思われる．コンピュータハードウェアの値下がりと，パフォーマンスの向上により，メガソフトウェアエンジニアリングは現実に可能になってきている．

さまざまなコードクローン検出手法が提案されている．Baxter らは抽象構文木を用いた検出手法を提案しており[1]，Ducasse らは行単位での検出ツールを作成している[6]．またプログラム依存グラフを用いた検出手法も提案されている[15]．

著者らはこれまでにも CCFinder を用いてコードクローン検出を行ってきており，このツールに関する知識を持っているため，本実験でも CCFinder を用いた．CCFinder はこれまでに提案されている検出手法の中でもスケーラビリティが高く，本実験の目的にあったツールである．また，fingerprint 技術[16] を用いたコードクローン検出も高いスケーラビリティを実現できるため，超大規模からのコードクローン検出に向いていると考えられる．

オープンソースソフトウェアに対してのコードクローン検出・分析は既に行われている[5),19]．しかし，それらは 1 つのプロジェクト内のコードクローン検出にとどまっており，本研究のように大量のソフトウェア群からコードクローン検出を行ってはいない．

7. ま と め

本論文では，超大規模から短時間でコードクローンを検出する手法を提案した．提案手法を分散システム D-CCFinder として実装し，オープンソースオペレーティングシステム FreeBSD 用のソフトウェア集合である Ports システムに含まれるソースファイルに対して適用した．約 4 億行の C 言語で記述されたソースコードから 51 時間でコードクローン検出を完了することができ，散布図やヒートマップを用いて大まかにコードクローンの状態を把握することができた．

著者らは，このような大規模ソースコードからのコードクローン検出が，近年問題になってきている著作権違反に応用できると考えている[17)]．たとえば，ソフトウェアライセンスの 1 つである GPL でライセンスされた著作物は，その派生著作物に対しても GPL でライセンスされなければならない．コードクローン検出および可視化を行うことにより，GPL でライセンスされたソフトウェアと他のライセンスを持つソフトウェアが高い類似度であることが判明した場合は，著作権違反の疑いを指摘することができる．このように，或るソフトウェアと過去に開発されたソフトウェアや外部で開発されたソフトウェア間のコードクローンを調査することによって，そのソフトウェアに著作権違反のコードが存在しているかどうかを調査することができる．

本論文は，ソフトウェア分析を分散環境で行うことの有用性を示している．D-CCFinder は超大規模ソースコードからコードクローン検出を行うための，単純で実用的なシステムであり，既存のネットワーク環境を用いて実装されている．

現在の D-CCFinder はプロトタイプシステムであり，5 節で述べたように多くの課題を残している．また，今後は，CCFinder ではなく，fingerprint 技術を用いてコードクローン検出を行うことも検討しており，よりパフォーマンスの向上が見込まれる．

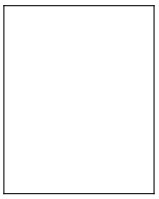
謝辞 大学の演習室を利用するにあたり協力していただいた大阪大学 大学院基礎工学研究科の田島滋人氏，ならびに情報科学研究科の小泉文弘氏に感謝する．本研究は一部，文部科学省リーディングプロジェクト「e-Society 基盤ソフトウェアの総合開発」の委託に基づいて行われた．また，日本学術振興会 科学研究費補助金 基盤研究 (A)(課題番号:17200001) および萌芽研究 (No.18650006) の助成を得た．

参 考 文 献

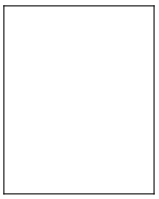
1) Baxter, I.D., Yahin, A., Moura, L., Anna, M. and Bier, L.: Clone Detection Using Abstract Syntax Trees, *Proc. of International Conference on Software Maintenance '98*, Bethesda, Maryland, pp.368–377 (1998).
2) Bellon, S. and Koschke, R.: A Comparison of Automatic Techniques for the Detection of Duplicated Code, Technical report, Institute for Software Technology, University of Stuttgart (2003).
3) Brown, A.W. and Booch, G.: Reusing Open-Source Software and Practices: The Impact of Open-Source on Commercial Vendors, *Proc. of the 7th International Conference on Software Reuse*, Lecture Notes in Computer Science, Vol.2319, Austin, Texas, Springer, pp.123–136 (2002).
4) Burd, E. and Bailey, J.: Evaluating Clone Detection Tools for Use during Preventative Maintenance, *Proc. of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation*, pp.36–43 (2002).
5) Casazza, G., Antoniol, G., Villano, U., Merlo, E. and Penta, M.D.: Identifying clones in the Linux kernel, *Proc. of the First IEEE International Workshop on Source Code Analysis and Manipulation*, Florence, Italy, IEEE Computer Society Press, pp.92–100 (2001).
6) Ducasse, S., Rieger, M. and Demeyer, S.: A Language Independent Approach for Detecting Duplicated Code, *Proc. of the International Conference on Software Maintenance '99*, Oxford, England, pp.109–118 (1999).
7) Foster, I.: What is the Grid? A Three Point Checklist (2002). available at `http://www-fp.mcs.anl.gov/~foster/Articles/WhatIsTheGrid.pdf`.
8) FreeBSD Project: `http://www.freebsd.org/`.
9) GNU Project: `http://www.gnu.org/`.
10) Inoue, K., Garg, P., Iida, H., Matsumoto, K. and Torii, K.: Mega Software Engineering, *Proc. of the 6th International PROFES (Product Focused Software Process Improvement) Conference*, Oulu, Finland, pp.399–413 (2005).
11) Jakarta Project: `http://jakarta.apache.org/`.
12) Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A multi-linguistic token-based code clone detection system for large scale source code, *IEEE Transactions on Software Engineering*, Vol.28, No.7, pp.654–670 (2002).
13) Kapser, C. and Godfrey, M.: Improved Tool

Support for the Investigation of Duplication in Software, *Proc. of the 21st International Conference on Software Maintenance*, Budapest, Hungary, pp.25–30 (2005).
14) Kim, M., Sazawal, V., Notkin, D. and Murphy, G. C.: An empirical study of code clone genealogies, *Proc. of the 10th European software engineering conference*, Lisbon, Portugal, pp.187–196 (2005).
15) Krinke, J.: Identifying Similar Code with Program Dependence Graphs, *Proc. of the 8th Working Conference on Reverse Engineering*, Suttgart, Germany, pp.301–309 (2001).
16) Li, Z., Lu, S., Myagmar, S. and Zhou, Y.: CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code, *IEEE Transaction on Software Engineering*, Vol.32, No.3, pp. 176–192 (2006).
17) Livieri, S., Higo, Y., Matsushita, M. and Inoue, K.: Very-Large Scale Code Clone Analysis and Visualization of Open Source Programs Using Distributed CCFinder: D-CCFinder, *Proc. of the 29th International Conference on Software Engineering*, Minneapolis, Minnesota, pp.106–115 (2007).
18) Rajapakse, D.C. and Jarzabek, S.: An Investigation of Cloning in Web Applications, *Proc. of the 5th International Conference on Web Engineering*, Lecture Notes in Computer Science, Sydney, Australia, Springer, pp.252–262 (2005).
19) Uchida, S., Monden, A., Ohsugi, N., Kamiya, T., Matsumoto, K. and Kudo, H.: Software Analysis by Code Clones in Open Source Software, *The Journal of Computer Information Systems*, Vol.XLV, No.3, pp.1–11 (2005).
20) Yamamoto, T., Matsushita, M., Kamiya, T. and Inoue, K.: Measuring Similarity of Large Software Systems Based on Source Code Correspondence, *Proc. of the 6th International PROFES (Product Focused Software Process Improvement) Conference*, Oulu, Finland, pp. 530–544 (2005).

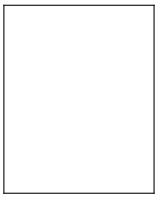
(平成？年？月？日受付)
(平成？年？月？日採録)



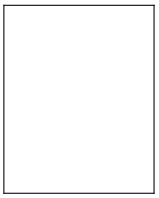
肥後　芳樹（正会員）
平成 14 阪大基礎工情報中退．平成 18 同大大学院博士後期課程修了．現在同大情報科学研究科コンピュータサイエンス専攻助教．博士（情報科学）．ソースコード分析，特にコードクローン分析やリファクタリング支援に関する研究に従事．電子情報通信学会，IEEE 各会員．



リビエリシモネ
平成 15 伊パドヴァ大学工学部コンピュータ工学科卒．現在大阪大学情報科学研究科博士後期課程に在学．アスペクト指向プログラミングやコードクローン分析の研究に従事．



松下　　誠（正会員）
平成 5 年大阪大学基礎工学部情報工学科卒業．平成 10 年同大学大学院博士後期課程修了．同年同大学基礎工学部情報工学科助手．平成 14 年大阪大学大学院情報科学研究科コンピュータサイエンス専攻助手．平成 17 年同専攻助教授．平成 19 年同専攻准教授．博士（工学）．ソフトウェア開発環境，リポジトリマイニングの研究に従事．日本ソフトウェア科学会，ACM 各会員．



井上　克郎（正会員）
昭和 54 年大阪大学基礎工学部情報工学科卒業．昭和 59 年同大学大学院博士課程修了．同年同大学基礎工学部情報工学科助手．昭和 59〜61 年ハワイ大学マノア校情報工学科助教授．平成元年大阪大学基礎工学部情報工学科講師．平成 3 年同学科助教授．平成 7 年同学科教授．平成 14 年大阪大学大学院情報科学研究科コンピュータサイエンス専攻教授．工学博士．ソフトウェア工学の研究に従事．日本ソフトウェア科学会，電子情報通信学会，IEEE，ACM 各会員．