2005－ARC－161　(7)
2005／1／21

ハイパースレッディング環境における投機的スレッド間の同期手法の提案

本田　大[†]　斎藤　史子[†]　山名早人[‡]

概要　近年、CPU 処理速度と主記憶からのデータ転送速度との間の格差が顕著になってきているため、キャッシュメモリの重要性が高まってきている。特に、非線形なアクセスパターンを示すポインタ遷移プログラムのキャッシュミスが問題視されている。この問題に対し、余剰な CPU 資源を利用して、単数あるいは複数の Helper スレッドを実行させ、キャッシュミスレイテンシを隠蔽する Pre-Execution が提案されている。本論文では、Pre-Execution を前提とした、Helper スレッドとメインスレッドとの間の効率的な同期手法を提案する。さらに、従来方式より prefetch する領域を拡大し、2 次キャッシュの効果を高める手法も検討する。Intel Xeon プロセッサでの実機上で、SPEC2000 181.mcf、300.twolf、Olden ベンチマークの health において、スレッド間の同期をとる手法と非同期による手法で性能評価を行った。結果、平均 29.67%の 2 次キャッシュミスを削減し、181.mcf で 3.26%、health で 1.36%の処理性能高速化を達成できた。

An Efficient Synchronization Scheme Using Speculative Threads on Hyper-Threading Technology

Dai HONDA[†]　Fumiko SAITO[†]　Hayato YAMANA[‡]

Abstract Recently, the gap between CPU processing speed and the data transmission speed from the main memory has lowered execution speed. Thus, data caching technique becomes more important. Particularly in pointer-based programs which have nonlinear access patterns, the cache miss rate is very high. To solve this problem, Pre-Execution has been proposed as a cache miss latency tolerance technique that makes one or more helper threads running in the spare CPU's resources ahead of the main computation. This paper proposes the synchronous technique between main thread and helper thread. Furthermore, this paper examines the expanded prefethes domain in comparison with the conventional Pre-Execution scheme. By expanding the prefetches domain the second level cache miss rate decrease. In SPEC2000 181.mcf , 300.twolf and Olden benchmark health on the Intel Xeon processor , the proposed synchronization technique , in comparison with the asynchronous technique is evaluated. As the result, the average of 29.67% second level cache misses are reduced, and 3.26% of the processing speed is improved in 181.mcf and 1.36% in health.

1　はじめに

近年、CPU とメモリとの間の性能のギャップが年々広がってきている。CPU の処理速度は年率数十%向上しているのに対し、メモリの処理速度は年率数%の向上に留まる。例えば、最近のプロセッサでは、L1 キャッシュアクセスに対し、メインメモリへのアクセスのペナルティは、200 倍近くある。キャッシュメモリはこの差を隠蔽するために実装されている。その結果、キャッシュメモリを有効活用するための最適化は、性能向上に欠かすことができなくなり、盛んに研究されてきている。以下に、キャッシュ最適化に関する研究状況を示す。

キャッシュ最適化には、コンパイラを利用した研究が多く行われている。コンパイラによるキャッシュ最適化には、ループインターチェンジ、ループ融合、ストリップマイニング、ループ・タイリング、ループ・アンローリングなどのループ変換によりアクセスパターンを変更する手法がある。また、単一ループもしくはループ融合されたループを対象に、イタレーション間での配列内および配列間パディング手法も提案されている[1]。

コンパイラによる最適化を適用しても、ポインタを利用した非線形なアクセスが多いプログラムや RDBMS などでは、キャッシュミスが頻発し、メモリアクセス遅延が隠蔽できない。このような、キャッシュミスを頻発するロード命令は、delinquent ロードと呼ばれている[2]。

[†] 早稲田大学大学院理工学研究科
Graduate School of Science and Engineering, Waseda University
[‡] 早稲田大学理工学術院
Science and Engineering, Waseda University

delinquent ロードによる、ロード転送遅延問題を解決する手法の一つとして、スレッドレベル並列性(Thread Level Parallelism)を利用した Pre-Execution[2][3][4][5][6][7]が提案されている。Pre-Execution とは余剰な CPU 資源を利用して Helper スレッドを作成し、メインスレッドの実行を補助する技術である。Helper スレッドの実行結果を、適切なタイミングでメインスレッドが使用できるためには、これらを同期させることが重要になる。

本論文では、Helper スレッドとメインスレッドとの間で効率的な同期処理を行い、メインスレッドの同期処理を削減する Pre-Execution 手法を提案する。これによって、Pre-Execution での同期にかかるオーバヘッドを削減することができる。また、Helper スレッドの起動と待機を、メインスレッドが実行したイタレーション数と Helper スレッドが実行したイタレーションの数の差を計算すること（ヒステリシス）で決定する手法を検証した。

第 2 節で従来手法を紹介し、第 3 節で提案手法、第 4 節で実験方法および評価結果と考察を示す。第 5 節で今後の課題を示して、まとめとする。

2 従来手法

従来の Helper スレッドで実装された Pre-Execution には、静的に実装する手法 [2][3][4][5]と、動的に実装する手法[6][7]に大別できる。

静的に実装する手法の特徴は、既存のハードウェアを変更することなく、Pre-Execution を実現し、性能向上が見込める。文献[2]の手法では、Pre-Execution 専用の命令を独自に作り、プログラムソース内に専用命令を挿入することで、Pre-Execution を実装している。文献[3][4]の手法では、コンパイル後のバイナリコードから Pre-Execution 対象のコードを抽出することで、Helper スレッドが実行する命令を決める。文献[2][3][4]では、Helper スレッド生成時に、Helper スレッドの実行に必要なデータをメインスレッドから渡すことで、スレッド間の同期をとっている。よって、スレッド生成とデータコピーのオーバヘッドがかかる。また、文献[5]の手法では、コンパイル前に Source to Source によるプログラム変換を行うことで Pre-Execution を実装している。文献[5]では、セマフォを使用して、Producer-Consumer 方式でスレッド間の同期を実現している。

動的に実装する手法の特徴は、プログラマやコンパイラに頼らず、実行時の情報とハードウェアの状態から、Pre-Execution を実現できることである。文献[6]の手法では、Helper スレッドによって、求めた値をメインスレッドが再利用する手法も提案されている。文献[7]の手法では、Pre-Execution 専用ハードウェアを提案し、プログラム実行時に、Pre-Execution すべき箇所の特定と Helper スレッドが実行する命令の生成を行う。文献[6][7]では、スレッド間で共有する専用ハードウェアを仮定し、Helper スレッドの実行結果を.Main スレッドがキャッシュミスや分岐予測ミスを緩和するために参照する。

しかし、スレッド生成やセマフォを使用するには、オーバヘッドがかかる。また、専用ハードウェアを仮定すると、チップ容量の問題が生じる。

3 提案手法

第 2 節で述べた従来手法の問題点に対して、我々は、Helper スレッドとメインスレッドとの間の効率的な同期手法を提案する。これによって、スレッドの生成やスレッド間の同期にかかるオーバヘッドを削減できる。

3.1 対象アーキテクチャ

本提案手法は、オンチップマルチプロセッサで、複数の CPU がキャッシュを共有しているアーキテクチャを対象とする。本稿では、その中でも Intel の CPU を対象とした。Intel の SMT(Simultaneous Multi-Threading)アーキテクチャ対応プロセッサは、Hyper-Threading を実装している。Hyper-Threading に対応した CPU では、メインスレッドと Helper スレッド間で実行資源を共有できる。このように、論理 CPU が 2 つあると仮定できるので、1 本のメインスレッドと、1 本 Helper スレッド、つまり合計 2 本のスレッドによる実行を仮定した。

3.2 プログラムのモデル化

Hyper-Threading 技術を利用し、Helper スレッドをメインスレッドに先行して実行させる。Helper スレッドはメインスレッドの処理中に、後で必要となるデータを prefetch することによって、メインスレッドのキャッシュミスに起因するストールを隠蔽する。本手法のモデル図を図 1 に示す。

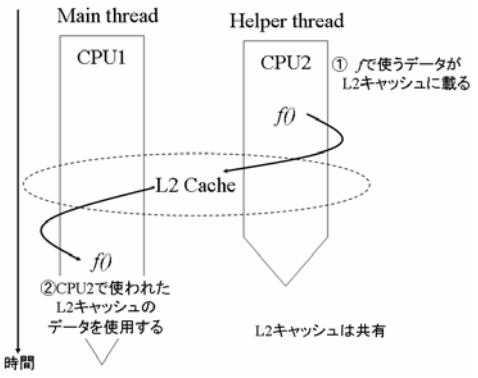

図 1:本手法のモデル図

メインスレッドに対し、Pre-Execution 用の Helper スレッドを１本作成する。Helper スレッドはデータをキャッシュに載せるためだけに使用する。そのため、Helper スレッドでは、メインスレッドに影響を与えるストア命令を発行しない。これにより、メインスレッドにおける実行の正確性を保証し、かつ、Helper スレッドにおける命令数を削減できる。

3.3 プログラムの実行フロー

本手法では、Pre-Execution の適用対象をプログラムの最内ループとする。メインスレッドが Pre-Execution の適用対象ループに入るまでは、シングルスレッドのみで実行する。Pre-Execution 対象ループに入ってからは、Helper スレッドを作成し、2 スレッドによる並列処理を開始する。アプリケーションの動作を図 2 に示す。

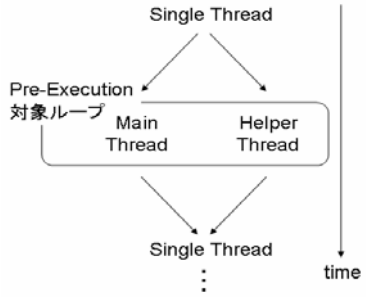

図 2:アプリケーションの実行フロー

3.4 Helper スレッドの生成

Helper スレッドの作成には、WIN32API の CreateThread()を用いて実装した。Helper スレッドが実行する命令は、delinquent ロードと依存関係がある命令のみで構成する。

3.5 スレッド間の同期モデル

Helper スレッドの実行がメインスレッドの実行より、先行しすぎないために、スレッド間で同期をとる必要がある。なぜなら、メインスレッドが必要とするデータを使用する前に、Helper スレッドによって、prefetch しなければならないからである。本手法の同期モデルを図 3 に示す。

図3におけるThread Distance（以下TD）とは、Helperスレッドと Mainスレッドとのイタレーションの距離である。ここでいうイタレーションの距離とは、同一ループ内におけるメインスレッドが実行中のイタレーションより、Helperスレッドが何イタレーション先行しているかを示す。TD_{MAX}とは、Helperスレッドが Mainスレッドに比較して先行しすぎないようにするために静的に設定したTDの値である。また、TD_{MIN}とはHelperスレッドを待機状態から復帰させるために静的に設定した値である。つまり、Helperスレッドは、TDがTD_{MAX}以上になった際に実行を停止し、待機状態に入る。その後、待機状態でTDを計算し、TDがTD_{MIN}以下になった際に再びHelperスレッドの実行を開始させる。2 次キャッシュに載る最適なTD_{MAX}で実装することで、Pre-Execution適用範囲を拡大し、prefetchの量を増加させたことによる、速度向上が見込める。

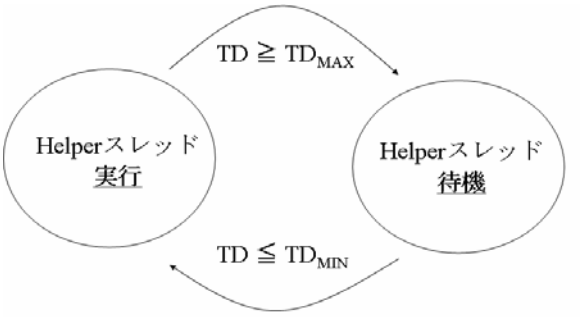

図3:スレッド間の同期手法

図 4 にTD_{MAX}=5 とTD_{MIN}=2 の場合での、本提案手法におけるHelperスレッドの待機と起動させるタイミングを示す。最初はTDが 5 になるまで、Helperスレッドと Mainスレッドは並列に実行し続ける。TDがTD_{MAX}である 5 以上になった場合、Helperスレッドは Pre-Executionしすぎないために待機する。Helperスレッドが次に起動するのは、Main スレッドが Pre-Execution対象ループのイタレーションを実行し続けた後にTDの値がTD_{MIN}である 2 以下になった場合に起動する。後は、TDが 5 になるまで、Helperスレッドと Mainスレッドを並列に実行させ続ける。この手法によって、従来の毎イタレーション同期を取る手法に比べ、同期オーバヘッド、すなわち待機と起動の回数を削減することができ、速度向上が見込める。また、TD_{MAX}の値を大きくしすぎると、L2 キャッシュにprefetchしたデータが収まらなくなるために、逆に性能低下が発生すると考えられる。

本手法では、同期をとるためにMainスレッドでの実行済みイタレーション数をグローバル変数としてインプリメントした。Mainスレッドは、グローバル変数に現在まで実行したイタレーションの数を書き込む。このグローバル変数の値とHelperスレッドの実行済みイタレーション数の差が、静的に設定した値TD_{MAX}以上になると、HelperスレッドはPre-Executionを停止し待機する。次に、Helperスレッドは、グローバル変数とHelperスレッドが実行済みであるイタレーション数の

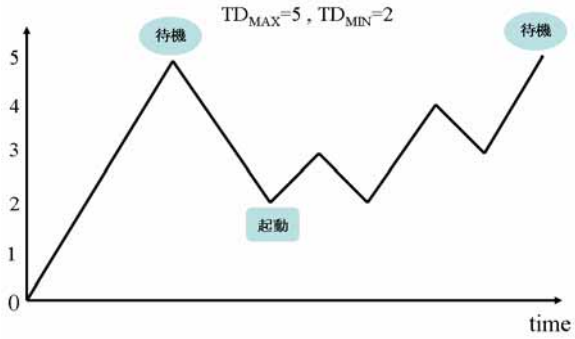

図 4:本提案手法における Helper スレッドの待機と起動

差が、静的に設定したTD_{MIN}の値以下になると、Helper スレッドを待機中から復帰させる。この手法が有効である理由は、Helperスレッドの命令数がメインスレッドに比べ短く、Helperスレッドが常に先行して実行できるためである。なお、スレッドの待機はスピンロックで実装している。

4 実験と評価

本節では、第 3 節で説明した手法の有効性を検証するための評価環境と実験結果を示す。アプリケーションをシングルスレッドで実行すると、2 次キャッシュミスが頻発するアプリケーションを評価対象とした。この評価対象アプリケーションに対し、シングルスレッドで実行した場合と本手法を適用した場合の 2 次キャッシュミス数を計測し評価を行った。また、対象アプリケーションをシングルスレッドで実行した場合と本手法を適用した場合とのプログラム全体の実行時間を求め、速度向上率を算出することによって評価を行った。

4.1 評価環境

実行環境を表 1 に示す。2 次キャッシュミスを測定するツールとして、VTune Performance Analyzer 7.1 を使用した。実行時間の計測には、timeGetTime()関数を使用した。

表 1：実行環境

CPU	Xeon 2.4GHz
L1 Data Cache	8KB,4-way-set associative 32-byte Line
L2 Data Cache	512KB,8-way-set associative 64-byte Line
Main Memory	1GB
OS	Windows XP Professional SP2
Compiler	Intel C++ Compiler 8.0.48
Compile Option	/Zi /Zd /QaxN /O3
Link Library	winmm.lib

実行環境における、キャッシュミスにおけるアクセスレイテンシを表 2 に示す。この評価には、Cache Burst[8]を用い、10 回の平均をとることで求めた。

表 2：キャッシュミスレイテンシ

L1 Cache Access	1 Cycle
L2 Cache Access	18 Cycle
Main Memory Access	213 Cycle

L2 キャッシュへのアクセスレイテンシと比較して、メインメモリへのアクセスは、レイテンシが約 11.8 倍となっている。そこで、メインメモリへのアクセスを減らすことで計算機の速度向上を図る。

4.2 評価対象

表 3 に本手法で評価したプログラムを示す。

表 3：実験対象アプリケーション

ベンチマーク	アプリケーション	入力セット
SPEC INT 2000	181.mcf	ref
	181.twolf	ref
Olden	health	5 Level 2,048 Node

今回の実験の評価プログラムとして、SPEC2000 から、181.mcf、300.twolf を選択した。また、Olden ベンチマークから health を選択した。VTune による解析結果から、これらのプログラムは、シングルスレッドで実行した際に、2 次キャッシュミスが多く発生するアプリケーションであることが判っている。

2 次キャッシュミス数の大部分は、一部の命令に起因している。181.mcf の場合、キャッシュミスが頻繁に発生する命令は、双方向リストにおけるポインタの参照先アドレスを対象としたロード命令である。181.mcf における、delinquent ロードの例を図 5 に示す。図 5 では、表4に示す最も2次キャッシュミス数が多い関数を例としてとりあげた。

```
Long price_out_impl()
{
    for(i=0;i<MAX;i++)
    {
        while( arcin )
        {
            tail = arcin->tail;
            if(tail->time + arcin->org_cost < latest)
            {
                arcin = tail->mark;
            }
            ...
        }
    }
}
```

Cache Miss!!

図 5:delinquent ロード位置

表 4:本手法の実験結果

	関数名	TD_{MAX}	TD_{MIN}	提案手法適用前の2次キャッシュミス数	提案手法適用後のキャッシュミス数	2次キャッシュミス削減率
SPEC2000	price_out_impl()			2,230,461,558	1,497,031,268	32.88%
181.mcf	compute_red_cost()	4	3	268,263,348	162,418,341	39.46%
300.twolf	new_dbox_a()	3	2	1,114,842,354	758,598,612	31.95%
Olden:health	check_patient_waiting()	非同期		2,102,176,440	1,925,713,844	8.39%

.

本実験では、Pre-Execution の実装は、2 次キャッシュミス数が最も多い関数中の最内ループに対して適用した。

4.3 評価結果

本手法で最も効果が得られたTD_{MAX}とTD_{MIN}の組み合わせ時における 2 次キャッシュミス数と削減率を表 4 に示す。表 4 に示すように、すべての関数において、2 次キャッシュミス数を削減できている。

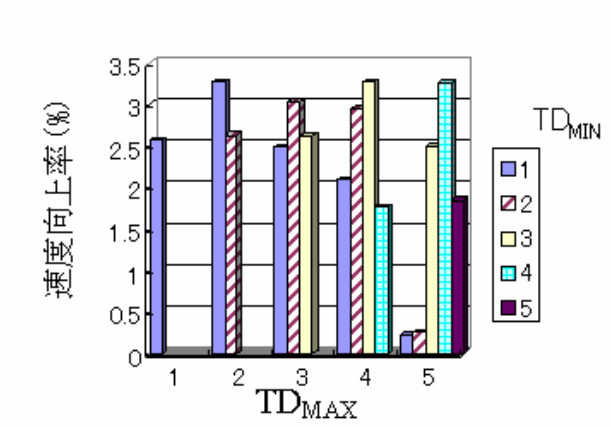

図 6:速度向上率　SPEC2000 181.mcf

図 6 にSPEC2000 181.mcfに本手法を適用した結果を示す。図6はPre-Executionを適用しない場合を基準としたときの、本手法を適用した場合の速度向上率を示している。図 6 より、TD_{MAX}=4、TD_{MIN}=3 のとき、最も高い速度向上率が得られた。181.mcfでは、Helperスレッドの方がメインスレッドに比べ、実行コードが短いので、図6 に示すように、提案手法が有効に機能していることがわかる。TD_{MAX}とTD_{MIN}の差が大きい場合、Helperスレッドがprefetchしたデータが 2 次キャッシュからはずれ、Pre-Executionの効果が低減したと考えられる。

図7 にSPEC2000 300.twolf に本手法を適用した結果を、図8 にOlden health に本手法を適用した結果を示す。しかし、300.twolf や health に本手法を適用した場合、約 15～19%速度低下した。300.twolf では、Pre-Execution 対象関数での最内ループのイタレーション回数が数回しかなく、スレッドの起動にかかるオーバヘッドが大きかったため速度低下が生じたと考えられる。また、health で速度低下が生じた原因として、Pre-Execution 対象関数でのメインスレッドの実行コード数が非常に少ないため、Helper スレッドの起動･停止にかかるオーバヘッドが相対的に大きくなったためであると考えられる。

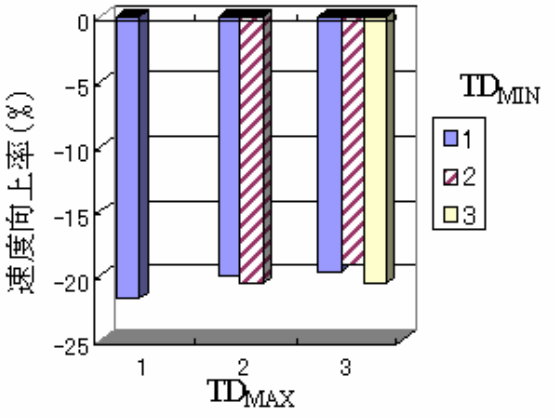

図 7:速度向上率 SPEC2000 300.twolf

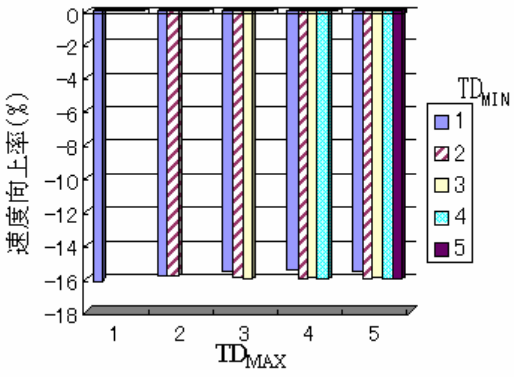

図 8:速度向上率 Olden health

つまり、Helper スレッドがメインスレッドに追いつかれたため Pre-Execution の効果が得られず、同期にかかるオーバヘッド分だけ速度低下が生じたと考

えられる。よって、Olden health において非同期で Pre-Execution を実行したところ、本手法を適用しない場合と比較して、1.36%の速度向上率が得られた。ここで、非同期での実行とは、同期のためのTD$_N$を設定せず、Helperスレッドが終了するまで実行することである。

5 おわりに

本稿では、Main スレッドと Helper スレッド間の同期手法に関する提案と実装を行い、かつ適用範囲を拡大した Pre-Execution 手法を Intel Xeon プロセッサ上で検証した。結果、2 次キャッシュミス数を平均 29.67%削減し、実行時間を最大 3.26%短縮することができた。

今後は、スレッド間のオーバヘッドを軽減するためにも、新しい同期法の考案が必要だと考える。特に、同期あるいは非同期に Helper スレッドを起動させるかを自動的に判別するようなコンパイラが必要とされる。さらに、現在の Hyper Threading 技術を拡張し、論理CPU を増やすことで、割り当てる Helper スレッドの数も CPU 台数に対応させ、Pre-Execution の効果をより高めていきたい。

謝辞

本研究の一部は、日本学術振興会 21 世紀 COE プログラム「プロダクティブ ICT アカデミア」の支援により行われた。

参 考 文 献

[1]石坂 一久, 小幡 元樹, 笠原 博徳, ”配列間パディングを用いた粗粒度タスク間キャッシュ最適化”, 情報処理学会論文誌, Vol. 45, No.4 ,pp.1063-1076, April, 2004

[2]C. K. Luk:"Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors”, In Proc. of 28th Intl. Symp. on Computer Architecture (ISCA) , pp.40-51 , 2001

[3]J.Collins,H.Wang,D.Tullsen,C.Hughes,Y-F.Lee , D.Lavery , J.Shen : "Speculative Precomputation : Long-range Prefetching of Delinquent Loads",In Proc. of 28th Intl. Symp. on Computer Architecture (ISCA) , p p. 14-25, 2001

[4]Steve S.Liao,Perry H.Wang,Hong Wang , Gerolf Hoflehner , Daniel Lavery , John P.Shen : ” Post-Pass Binary Adaptation for Software-Based Speculative Precomputation”, In Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI) ,pp.117-128, 2002

[5]Dongkeun Kim,Donald Yeung:”Design and Evaluation of Compiler Algorithms for Pre-Execution”,In Proc. of 9th Intl. Conference on Architectual Support for Programming Languages and Operating System(ASPLOS) ,pp.62-73,2002

[6]Amir Roth and Gurindear S.Sohi:"Speculatie Data-Driven Multithreading ",In Proc. of 7th Intl. Symp. on High Performance Compute Architecture(HPCA), pp.37-48,2001

[7]C.Zilles,G.Sohi:"Execution-based prediction using speculative slices", In Proc. of 28th Intl. Symp. on Computer Architecture (ISCA) ,pp.2-13,2001

[8]Cache Burst 32
http://user.rol.ru/~dxover/cburst/