令和２年度卒業論文

デッドラインを考慮したメモリ削減スケジューリング LMCLF の改善

広島市立大学

情報科学部システム工学科

組込みデザイン研究室

氏名　　新井　諒介

指導教員　　中田　明夫　教授

令和３年１月２８日提出

目次

内容

1. まえがき

1.1 研究背景

スマートフォン，家電製品，医療機器など代表される組込みシステムは大量生産されることが多いため，製造コストの削減は重要な課題である．そのため，組込みシステムの開発目標の一つとして，メモリ消費量を削減することが挙げられる [1]．なぜなら，メモリ消費量を削減することができれば，組込みシステムのメモリ搭載量を削減でき，製造コストの削減に結びつくと考えられるからである．

リアルタイム組込みシステムは，複数の外部入力に対する応答性を向上させるため，複数のタスクを切り替えることによって並行処理するマルチタスクシステムで構成されることが多い．しかしながら，マルチタスクシステムではタスクが切り替わることによって一時停止するとき，タスクにヒープメモリが割り当てられたままとなるため，ヒープメモリ消費量が増加してしまう．従って，一般に，マルチタスクシステムのヒープメモリ消費量は, 同じ機能を実現するシングルタスクシステムのヒープメモリ消費量よりも増加する傾向にある．

マルチタスクシステムのメモリ消費量を静的解析によって削減する手法が提案されている[2][3]．しかしながら，実際に市場に流通しているマルチタスクシステムで動作する製品には，数千個のタスクを扱う製品も存在する．それらの製品において，他のタスクへの切り替えが多くなるとメモリ消費量は増加する．また，タスク単体が持つ状態（ヒープメモリの割り当て箇所）数が少なくても，複数のタスクの並行処理時には各タスクの状態で全ての組み合わせを取り得るため，状態数が爆発的に増加する．よって，既存の静的解析に基づくメモリ削減手法ではそのような解析は困難である．

リアルタイムスケジューリングの研究は多く存在する[4][5][6][7][8][9]. 先行研究では,マルチタスクシステムのヒープメモリ消費量を動的解析によって削減するスケジューリングアルゴリズムである，Least Memory Consumption First(LMCF) スケジューリングが提案されている [10][11][12]．LMCF スケジューリングでは，各タスクの次のステップのヒープメモリの割り当て（以下，消費メモリ増分）は予測可能[13]であると仮定する．プロセッサ数を p とすると，次のステップの消費メモリ増分が小さいタスクから p 個選択し，選択したタスクに優先度を付与する．これにより，ヒープメモリ消費量が最大となる状態（以下，最大メモリ消費状態）を回避可能なタスクセットに対しては, LMCF スケジューリングによって必ず最大メモリ消費状態を回避可能である[11].

一方，LMCF スケジューリングでメモリ消費量の削減はできるが，マルチタスクシステムによる効果（応答性の向上など）が得られない可能性がある．そこで，先行研究では消費メモリ増分だけでなく，残余実行時間と余裕時間を考慮した Least Memory, remaining Computation-time, and Laxity First（LMCLF）スケジューリングを提案されている. LMCLF スケジューリングでは, ($\alpha\times$ 消費メモリ増分 + 残余実行時間 $\times$ 余裕時間)の値が小さいタスクから順に優先度を付与する．(ただし, α は時間とメモリの換算レートであり,設計者が任意に定める.) しかし,メモリ消費量とデッドラインに大きなばらつきがある場合は，事前に適切な α の値を設定するのは困難である.もし α の値が最適でない場合,最適である場合よりもメモリ削減量が減ってしまう可能性がある.本

研究では,そのαのより適切な値をスケジューラ動作中に自動推定する手法を提案する．提案手法により,αを事前に定める必要が無いかつ,αの値が最適でない場合の従来手法よりもメモリ消費量を削減することが期待できる．

1.2 研究目的

LMCLF スケジューリングでは，メモリ増分と余裕時間,残余実行時間に大きなばらつきがある場合αの値を設定するのは困難である.それに加えてαの値が最適でない場合,最適である場合よりもメモリ削減量が減ってしまう.本研究では，そのαのより適切な値をスケジューラ動作中に自動推定することにより LMCLF を改善することを目的とする.提案手法により,αを事前に定める必要が無くかつ,αの値が最適でない場合の従来手法よりもメモリがより削減できることが期待できる．

1.3 研究概要

本研究では,従来の LMCLF よりも最良なαの値の導出法を提案する．提案手法では,スケジューラの各周期毎に次のことを行う．まず，タスクの 2 ステップ後までのメモリ消費量が最小となるようなαの範囲を求め，そこから範囲の上限下限がそれぞれ範囲の条件を満たしているか（上限が負でないかつ下限の値が上限の値を上回っていないか）を調べる．もし満たしている場合，その範囲の下限から上限の間の値をαの値として LMCLF スケジューラを動作させる．提案手法の有効性の評価のため，ランダム生成されたタスクセットに対してシミュレーション実験を行い，提案手法で改善した LMCLF スケジューリングと従来手法の比較評価を行う．

1.4 本論文の構成

本論文の構成について説明する．まず，2 章でマルチタスクシステムの概要や特徴について説明し，3 章でマルチプロセッサシステムの概要の説明と，システムモデルの定義を行う．次に，4 章で先行研究で提案されている消費メモリ増分だけでなく残余実行時間と余裕時間を考慮してヒープメモリ消費量の削減を図る LMCLF スケジューリングについて説明する．その後，5 章では，提案するαの値導出による LMCLF の改善について説明し，6 章では，評価実験について説明する．最後に，7 章で考察を行い，8 章で結論について述べる．

2. マルチタスクシステム

本章では，マルチタスクシステムの概要，マルチタスクシステムにおけるヒープメモリ管理，時間制約の特徴について説明する．最初に，2.1 節で一般的なマルチタスクシステムの動作について説明する．次に，2.2 節でマルチタスクシステムのヒープメモリ管理における特徴について，2.3 節ではマルチタスクシステムの時間制約における特徴について説明する．

2.1 概要

マルチタスクシステムとは，リアルタイム OS（一定時間内に与えられた処理を完了させる制約を持つ OS）が提供する機能の 1 つであり，中でも，OS が複数タスクを切り替えながら処理するシステムのことを指す．

マルチタスクシステムの実行環境では，並行プログラムの手法が利用可能となり，機能をタスクとして分割することで，ソフトウェアの再利用性が向上する．通常の CPU が 1 つのみのコンピュータでは，ある瞬間には 1 つの処理しかできない．しかし，タスクの CPU 処理時間を数 10 ミリ秒程度の短い区間で区切ることで，タスク間で 1 つの CPU をタイムシェアリングすることができる．これによりユーザからは，複数のアプリケーションが同時に処理されているように見える．

マルチタスクシステムは，タスク切り替え時に余分な処理を行う必要があり，キャッシュのミスヒット率の上昇などのコストがかかる．しかし，入出力待ちなどによって，あるタスクの処理が止まっても他のタスクが処理されるため，全体としてのスループットの改善が期待できる．なお，マルチタスクシステムには，CPU の実行権限を全て OS が管理し，強制的に処理の切り替えを行うプリエンプティブマルチタスクと，処理の切り替えが個々のアプリケーションに任せられているノンプリエンプティブマルチタスクが存在する．

マルチタスクシステムの一例を図 1 に示す．図中の四角形はタスクの処理を示し，その中に各タスクが起動した回数を記入する．タスクの優先度は Task 1>Task2>Task 3 とする．上向きの矢印はリリース時刻，下向きの矢印は絶対デッドラインを示し，横軸は時間を示す．

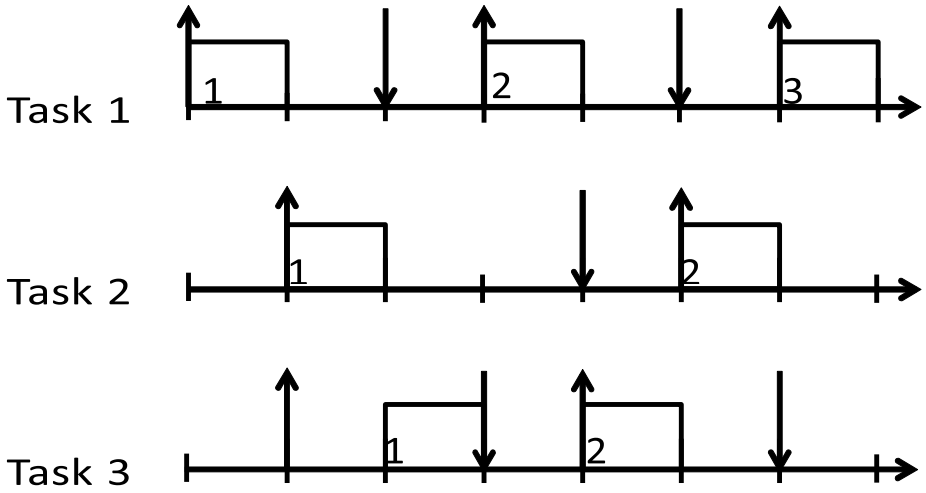

図 1：マルチタスクシステムの一例

2.2 ヒープメモリ管理

各タスクに与えられた優先度によりタスクの切り替えを行うマルチタスクスケジューリングが存在する．ここでは，複数のタスクを並行処理するため，優先度の高い他のタスクへの切り替えが発生すると，切り替えられたほうのタスクは，作業用に確保したメモリを保持したまま一時停止する．その様子を図 2 に示す．ただし，図 2 のタスクの優先度は図 1 と同様，Task 1>Task 2>Task 3 とする．

図 2 より，Task 3 から Task 2 へ切り替わることによって，Task 3 はメモリを確保したまま一時停止する．同様に，Task 2 から Task 1 へ切り替わることによって，Task 2 はメモリを確保したまま一時停止する．このとき，Task 2 と Task 3 の 2 つのタスクで確保された分のメモリを保持し続けなければならない．また，タスク単体が持つ状態数が少なくても，複数のタスクをマルチタスクシステムで扱うと，コンパイル時にタスクの状態数全てを組み合わせたものを生成するため，状態数が爆発的に増加する．

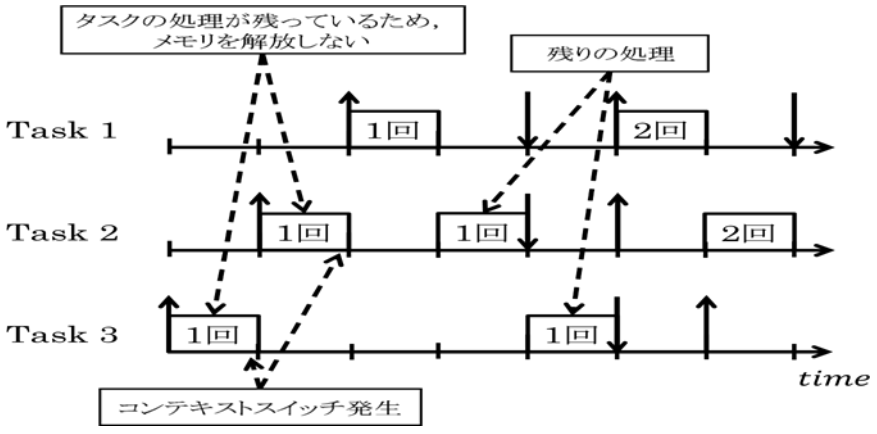

図 2: コンテキストスイッチ発生時における各タスクのメモリ消費状況

この例として，5 つの状態数を持つ 2 つのタスクを図 3，図 4 に示す．図 3，図 4 に示す 2 つのタスクを用いて構成したマルチタスクシステムの状態遷移図を図 5 に示す．図 3，図 4 は，簡単な C 言語プログラムを状態遷移図に変換した図である．状態遷移図の円は，状態を表す．各状態に記載されている番号は，タスクが何番目の状態にあるかを指す．これらの遷移図では，各状態で確保している変数を記載する．また，図中の最初の状態 state1 は初期状態，最後の状態 state5 は終了状態を表している．図 5 の矢印は，左向きは Task 1，右向きは Task 2 がスケジュールされた場合の状態遷移を表す．各状態に記載されている番号は，各タスクが何番目の状態にあるかを指す．

左側の数字は Task 1 の状態，右側は Task 2 の状態とする．

並行処理を行う際には，全てのタスクの状態を組み合わせるため，図 5 のように状態数が増加する．状態を組み合わせた中には，各タスクで確保している変数を同時に保持している状態が存在する．変数を多く確保している状態は，メモリ消費量が大きい動作を行うと考えられる．図 5 では，状態(3,3)と状態(3,4)が最も多く変数を確保しているため，メモリ消費量が大きい動作を行うと考えられる．一般的に，マルチタスクシステムは様々なスケジューリング方針でタスクを切り替える．よって，図 5 の状態(3,3)と状態(3,4)のように，メモリ消費量の大きい状態を通過する可能性を持つ．

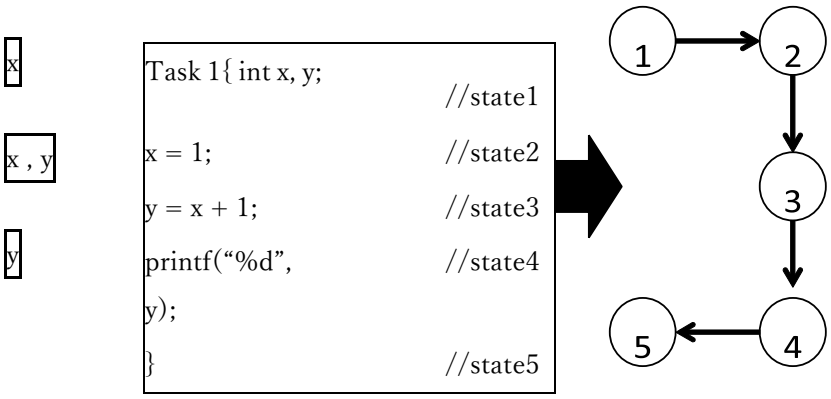

図 3：Task1 の状態遷移

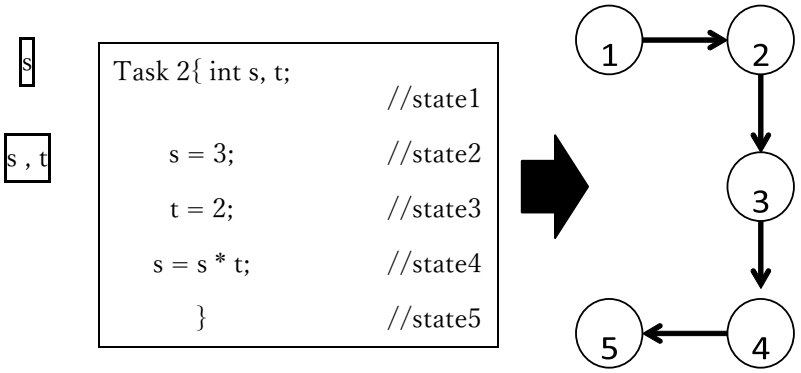

図 4：Task 2 の状態遷移

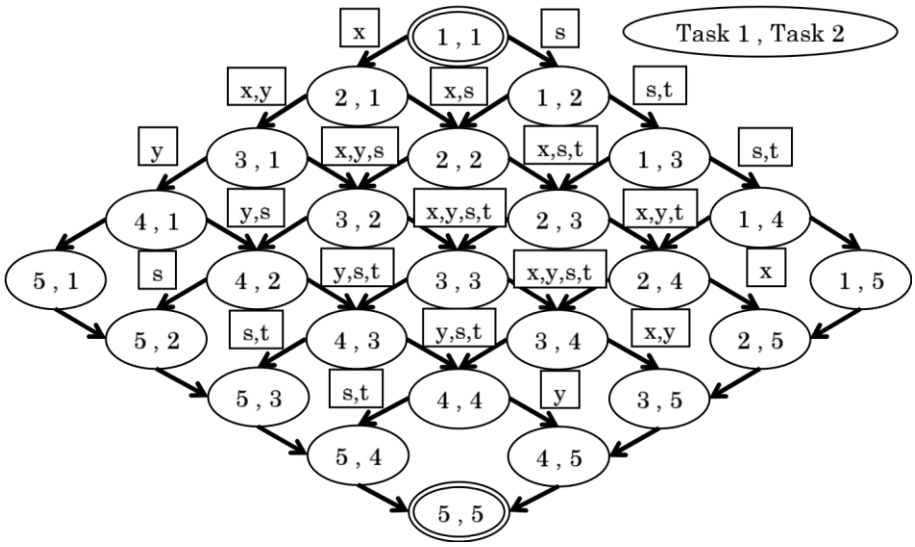

図 5: Task 1 と Task 2 の状態遷移

2.3 時間制約について

システムの中には，処理がデッドラインまでに終了しなかった場合，システムにとって致命的なダメージを受けるものがある．そのため，システムの設計・開発の際に，システムの時間制約を満たすという保証が求められることがある．そこで，システムの時間制約が満たされることを保証するための技術が必要となる．この技術には，各タスクの最悪実行時間を導出するための技術と，複数のタスクに関して，処理の順番を考えるリアルタイムスケジューリング理論がある．

各タスクの最悪実行時間の導出には，あらゆる入力に対する処理の実行時間の導出が必要である．しかし，これはキャッシュや分岐などの機構によって，処理の実行時間の導出が困難となっている．現実的には，処理の実行時間を複数回測定し，結果に安全係数を見込んで，最悪実行時間を定める方法を用いる．

リアルタイムスケジューリング理論とは，複数のタスクがデッドラインを持つ場合に，それらをスケジューリングする方法と,各タスクが時間制約を満たすかを数学的に証明するための方法を与えるものである．

最後にマルチタスクシステムの特徴をまとめる. マルチタスクシステムにおける各タスクは，タスクが切り替わることで待ち状態となり,メモリを確保し続けてしまう.それにより,全体のメモリ消費量が増加してしまう. また, 複数のタスクに時間制約がある場合に,各タスクが時間制約を満たすかどうかを数学的に証明する方法が存在することがマルチタスクシステムの特徴であるといえる.

3. マルチプロセッサシステム

本章では，マルチプロセッサシステムの概要，マルチプロセッサシステムにおけるスケジューリングについて説明する．3.1 節では，一般的なマルチプロセッサシステムの特徴を説明し，3.2 節では，システムモデルを定義し，3.3 節では，マルチプロセッサシステムにおけるスケジューリングについて説明する．

3. 1 概要

マルチプロセッサシステムとは，コンピュータに複数の CPU を搭載して一連の処理を行うシステムのことである．マルチプロセッサシステムは，CPU ごとに異なる処理を並行できるという特徴がある．マルチプロセッサシステムには，複数の CPU で 1 つの主記憶を共有する密結合と，1 つの CPU につき 1 つの主記憶で構成される疎結合がある．また，各 CPU が対等でどの CPU でも同じ処理を行えるものを対称型マルチプロセッサ（SMP：Symmetric Multiprocessing）といい，CPU ごとに役割が決められている（異なる機能を持つ）ものを非対称型プロセッサ（ASMP：Asymmetric Multiprocessing）という．

3. 2 システムモデル

本研究で取り扱うシステムモデルは，[17]で定義されているものと同様である．最小リリース間隔をT_i,最悪実行時間をC_i(Worst-case Execution Time, WCET）,相対デッドラインをD_iとし，このモデルにおけるタスクτ_i=(T_i, C_i, D_i)はタスクセット TS に含まれる（τ_i∈TS）．タスクτ_i は一連のジョブを呼び出し,各ジョブは，前のジョブと少なくともT_i時間単位だけ間隔をとる．また,タスクのひとつのジョブは並列に実行されないと仮定する．

タスクτ_iの時刻tにおける相対デッドラインと残余実行時間をそれぞれD_i(t), C_i(t)とし,タスク τ_iの時刻tにおける余裕時間をL_i(t)とし,式（1）から計算する．

$$L_i(t) = D_i(t) - C_i(t) \qquad (1)$$

タスク数の合計は n とし，システムの利用率を $U_{sys} = \Sigma\ \tau_{i\in} TS\, C_i/T_i$ とする．また，p 個のプロセッサが利用可能であるとし，プロセッサの性能は同等のものとする．タスクセットを以下のように定義する．

定義 **1.** タスクセット

$n(\geq 1)$ を任意の自然数とする．任意の $i \in \{1,...,n\}$ に対して，タスク τ_i は 1 ステップ目から e_i（1 以上の任意の自然数）ステップ目まで状態を変化させる有限状態機械であるとし，τ_i の $j(1 \leq j \leq e_i)$ ステップ目の状態を s_{ij} とする．

タスクの集合（以下，タスクセットと呼ぶ）を $TS = \{\tau_1, \tau_2,..., \tau_n\}$ とする．$i \in \{1,...,n\}$ に対して，$j_i(1 \leq j_i \leq e_i)$ を自然数とし，TS の状態は各タスクの状態の直積$(s_{1j1}, s_{2j2}, s_{3j3},..., s_{njn})$であると定義する．$TS$ の初期状態はすべてのタスクの初期状態

（1 ステップ目の状態 s_{i1}）の直積，TS の最終状態はすべてのタスクの最終状態（e_i ステップ目の状態 s_{iei}）の直積と定義する．

時刻 t における τ_i の状態が s_{ij} の時，状態 s_{ij} の τ_i の相対デッドラインと残余実行時間をそれぞれ，$D(s_{ij})$ と $C(s_{ij})$ とする．

T_i に対して，$\Sigma 1_{\leq j \leq e_i - 1}\, m(s_{ij}) = 0$ を満たすような各状態 s_{ij} への整数値 $m(s_{ij})$の割り当てを状態 s_{ij} における「消費メモリ増分」と呼ぶ．

4. ヒープメモリと実時間制約を共に考慮したスケジューリング手法 LMCLF

この章では，先行研究[3]で提案されている消費メモリ増分だけでなく，残余実行時間と余裕時間を考慮した新しいスケジューリングアルゴリズムを示す．

図 14 の 3 つのタスクのスケジューリング問題を考える．τ_1，τ_2，τ_3 のパラメータはそれぞれ，$\tau_1=(24,4,24)$，$\tau_2=(19,6,19)$，$\tau_3=(18,4,18)$ とする．この時，各タスクの余裕時間はそれぞれ $L_1=20$，$L_2=13$，$L_3=14$ となる．図 14 のタスクセットを LMCF スケジューリングによってスケジュールした時の総メモリ消費量の時系列変化を図 15 に示す．同様に，LLF スケジューリングによってスケジュールした時の総メモリ消費量の時系列変化を図 15 に示す．LMCF スケジューリングでは，最悪メモリ消費量が 25 であるのに対して，LLF スケジューリングでは，最悪メモリ消費量が 20 である．このことから，LMCF スケジューリングが LLF スケジューリングよりもメモリ削減できない場合が存在する．これは，残余実行時間が短いタスクがメモリを確保したまま一時停止してしまうことが原因であると考えられる．

したがって，消費メモリ増分だけでなく，残余実行時間と余裕時間も考慮したスケジューリング手法を提案する．このスケジューリング手法を，Least Memory, remaining Computation-time, and Laxity First (LMCLF) スケジューリングと呼ぶ．LMCLF スケジューリングでは，次のスケジューリングサイクルにおいて，$\theta_i(s_{ijj})$ を式 (5) から計算し，$\theta_i(s_{ijj})$ の値が小さいタスクから順に優先度を付与する．

$$\theta i(siji) = \alpha \times m(siji) + Ci(siji) \times Li(siji) \quad (\alpha > 0) \tag{5}$$

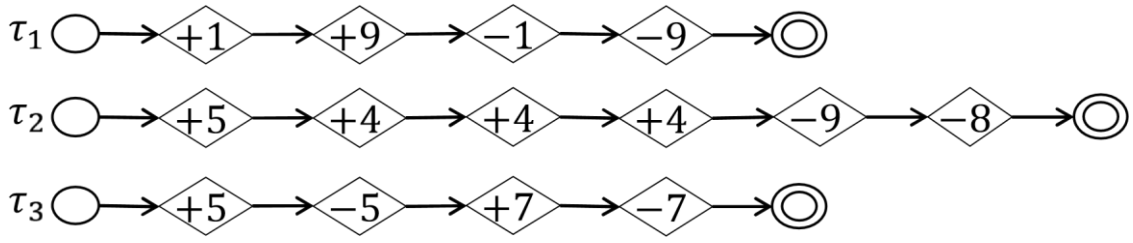

図 6: 3 つのタスクのスケジューリング問題

($\boldsymbol{\alpha}=$ **1**)		
1. MCI (LMCF)	0615142591090 0109201791090	25
2. L (LLF)	0109201791090 066171427170	20
3. $C \times L$	0109142591090	20
4. $\alpha \times \mathrm{MCI} + C$	061016059131780	27
5. $\alpha \times \mathrm{MCI} + L$		25
6. $\alpha \times \mathrm{MCI} + (C \times L)$ (LMCLF)		17

αC *L* .

図 7：総メモリ消費量の時系列変化

$\theta_i(s_{iji})$ を式（5）から計算する際に，用いられるαは，メモリと時間の換算レートでありαは，設計者が任意に定めるパラメータである．αの値をどのようにして決定するか，最適値が存在するかどうかが本研究である．

5. ヒープメモリと実時間制約を共に考慮したスケジューリング手法 LMCLF の改善

この章では，文献[3]のスケジューリング手法 LMCLF のパラメータαのより適切な値を自動的に導出する提案手法について説明する.提案手法は 1 プロセッサ環境を想定している.

提案する具体的なアルゴリズムを図 8 に示す.タスク$\tau_i\ (i = 1, \dots, n)$の時刻$t(t = 0,1)$における残余実行時間 c_i(t),余裕時間$L_i(t)$,及び$q(q = 0,1)$ステップ目におけるメモリ増分を$m_i(q)$とする.また$\tau_j, \tau_k, \tau_l$も$i$と同様であるとする．まず,図 8 のアルゴリズムの 5 行目から 11 行目でタスクτ_iおよびτ_jが$m_i(i) < m_j(j)$となる場合$m_i(i) < m_j(j)$となるようにαの範囲の上限を求め,$m_i(i) > m_j(j)$の場合$m_i(i) > m_j(j)$となるように α の範囲の下限を求める.次に,12 行目から 22 行目で$m_k(k) < m_l(l)$であれば$m_k(k) < m_l(l)$となるようにαの範囲の上限を求め,$m_k(k) > m_l(l)$であるとき$m_k(k) > m_l(l)$となるようにαの範囲の下限を求める.この時,2 回連続で同じタスクを選択している場合そのタスクのそのタスクの 1 ステップ後のαの範囲を求める．次に,22 行目から 32 行目で今まで求めた α の範囲の上限下限を満たすαが存在するなら，その時選択されていたタスクの最悪メモリ消費量を求める．最後に 33 行目で先ほど求めた最悪メモリ消費量が今まで求めた最悪メモリ消費量よりも小さい場合，今まで求めた上限下限を満たす値を求めてそれをαの値の候補とする.これを繰り返してαの値を決定する.具体的な決定方法は，求めた α の上限下限の値を足して 2 割った数値を α の値として更新していくというものである.もし上限がない場合下限の値を α の値とする.

- 入力:タスク τ i(i=1,…,n)の時刻 t(t=0,1)における残余実行時間 c_i(t),余裕時間 L_i(t),及び q(q=0,1)ステップ目におけるメモリ増分 m_i(q)
- 出力:LMCLF でスケジュールした場合に,2 ステップ後のメモリ消費量が最小となるような α

1．i=1~n まで以下を繰り返す
    1. 全ての j∈{1,2,…,n}\{i}に対して以下を繰り返す
        1. もし m_i(i)<m_j(j) ならば（ m_i(i) α +C_i(i)*Li<m_j(j) α +C_j(j)*L_j(j) かつ m_j(j)>m_i(i)）
            1. α の上限(c_i(i)*L_i(i)-c_j(j)*L_j(j))/(m_i(i)-m_j(j))を求める
        2. さもなければ（m_i(i) α +C_i(i)*L_i(i)<m_j(j) α +C_j(j)*L_j(j) かつ m_j(j)<m_i(i)）
            1. α の下限(c_j(j)*L_j(j)-c_i(i)*L_i(i))/(m_i(i)-m_j(j))を求める
    2. k=1~n まで以下を繰り返す
        1. 全ての l∈{1,2,…,n}\{k}に対して以下を繰り返す
            1. もし m_k(k)<m_l(l)ならば
            （m_k(k) α +C_k(k)*L_k(k)<m_l(l) α +C_l(l)*L_l(l) かつ m_l(l)>m_k(k)）
                1. α の上限(c_k(k)*L_k(k)-c_l(l)*L_l(l))/(m_l((l)-m_k(k))を求める
            2. さもなければ
            （m_k(k) α +C_k(k)*L_k(k)<m_l(l) α +C_i(i)*L_i(i) かつ m_l(l)<m_k(k)）
                1. α の下限(c_l(l)*L_l(l)-c_k(k)*L_k(k))/(m_k(k)-m_l((l))を求める
        2. 今まで求めた α の上限下限を満たす α が存在するならば
            1. もし m_i(i+1)>0 ならば
                1. 最悪メモリ消費量 max(m_i(i),m_i(i)+m_i(i+1))を求める
            2. さもなければ
                1. 最悪メモリ消費量 max(m_i(i),m_i(i))を求める
            3. さもなければ
                1. もし m_k(k)>0 ならば
                    1. 最悪メモリ消費量 max(m_i(i),m_i(i)+m_k(k))を求める
                2. さもなければ
                    1. 最悪メモリ消費量 max(m_i(i),m_i(i))を求める
            4. 今まで求めた最悪メモリ消費量よりも小さいならば
                1. 今まで求めた α の上限下限を満たす α を求めて,求める α の候補とする

図 8：提案手法のアルゴリズム

6. 評価実験

6.1 実験目的・方法

提案手法がαの値が最適でないときの従来手法よりもメモリ削減できるか否かを確認することを実験目的とする.実験方法に関してはランダムに生成した 100 個のタスクセットに対してスケジューリングを行い従来手法と提案手法の最悪メモリ消費量を比較する.

6.2 タスクセットの生成方法

本実験では，文献[3]における評価実験と同様に，文献［5］に基づいたタスクセットの生成方法を用いる．文献［5］のタスクセット生成方法は，文献［4］で提案されているタスクセット生成方法に基づいており，文献［4］のタスクセット生成方法は様々な研究の評価実験において用いられている（例えば，[5][6][7][8]）．提案手法がどのようなタスクセットでも対応できることを示すため上記のタスクセット生成法と同様の方法でメモリ使用量とプロセッサ利用率のパラメータを変更する．
2 組のタスクセットを生成するため，以下のパラメータ群を与える．

- 1 組目のタスクセット
    1. プロセッサ数 P1
    2. [－100000,100000]と[－1000000,1000000]のそれぞれの一様分布で決定した各タスクの消費メモリ増分の時系列変化
    3. パラメータ 0.7 と 0.9 のそれぞれの指数分布で決定した個々のタスクのプロセッサ利用率 $\delta_i = (C_i/T_i)$
    4. [100,1000] の一様分布で決定した,最小リリース間隔 T_i
       （ただし，本実験ではデッドラインに焦点をあてるため，$T_i = D_i$ する.）
    5. 与えられた δ_i と T_i から算出した実行時間 C_i
- 2 組目のタスクセット

    1. プロセッサ数 P1
    2. [－1000000,1000000]の一様分布で決定した各タスクの消費メモリ増分の時系列変化
    3. パラメータ 0.9 の指数分布で決定した個々のタスクのプロセッサ利用率 $\delta_i = (C_i/T_i)$
    4. [100,1000] の一様分布で決定した,最小リリース間隔 T_i
       （ただし，本実験ではデッドラインに焦点をあてるため，$T_i = D_i$ する.）
    5. 与えられた δ_i と T_i から算出した実行時間 C_i

2組のパラメータ群に対して，以下の Step に従い，100 個のタスクからなるタスクセットを 2 組生成する．

Step 1 初期値として，$p+1$ 個のタスクを生成する．

Step 2 スケジュール不可能なタスクセットを取り除くために，生成されたタスクセット$\Sigma\ \tau_{i\in TS}\ \delta_i \leq p$）を満たすかどうかを判定する [9].

Step 3 もし満たさなければ，そのタスクセットを破棄し，Step 1 に戻る．
もし満たせば，そのタスクセットを実験するタスクセットに含め，
そのタスクセットに新しいタスクを一つ追加して，Step 2 に戻る．

本実験で用いた計算機の CPU は，Intel(R) Xeon(R) CPU E5-2643v2 @3.50GHz ,メモリ容量は 16305MB であり OS は，Windows 10 Pro 64 bit である．

6. 3 実験結果

6.2 節のタスクセット生成法で生成された 2 組のタスクセットに先行研究[3]の手法で α =1000,100,1と与えたときと,提案手法の時とで実験を行った結果の最悪メモリ消費量と最悪メモリ消費量の平均,最大,最小,標準偏差とデッドラインミスの回数をそれぞれ従来手法と提案手法について表にまとめ,最悪メモリ消費量とデッドラインミス回数についてグラフにした.1 組目のタスクセットのものをそれぞれ図 9 ,図 10.2 組目のタスクセットのものをそれぞれ図 11,図 12 に示す.

	平均	最大	最小	標準偏差	デッドラインミス
α =1000	1248683	4843012	271985	631627.3	42
α =100	1120848	4843012	271985	589024.2	0
α =1	1112681	4843012	271985	595594.2	0
提案手法	1118658	4843012	271985	574690.8	0

図 9：従来手法と提案手法の比較表(1 組目のタスクセット)

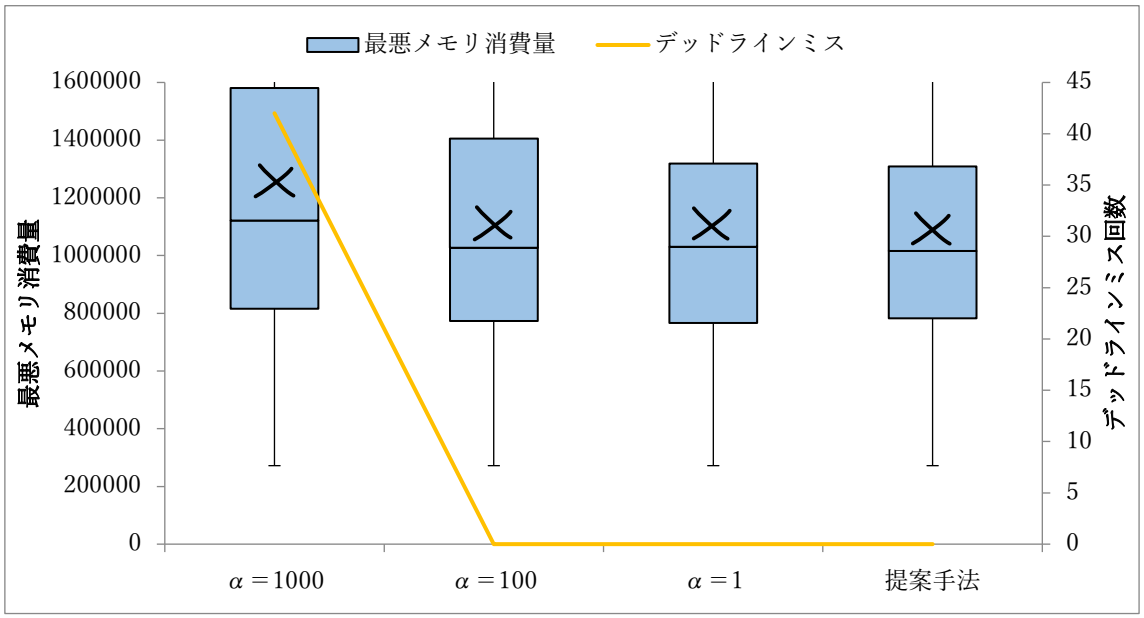

図 10：従来手法と提案手法の比較(1 組目のタスクセット)

	平均	最大	最小	標準偏差	デッドラインミス
α ＝1000	13493600	38248797	3924143	6567006	45
α ＝100	13456945	38248797	3924143	6555177	45
α ＝1	12565536	38248797	3924143	5884108	7
提案手法	12259112	38248797	3924143	5891194	0

図 11：従来手法と提案手法の比較表(2 組目のタスクセット)

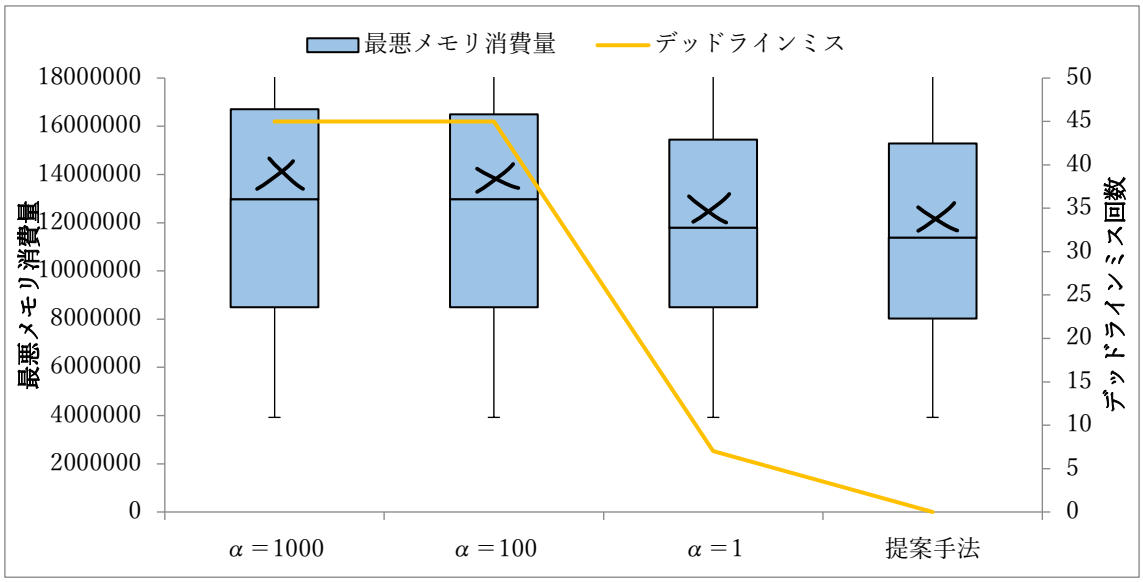

図 12：従来手法と提案手法の比較(2 組目のタスクセット)

7. 考察

図 9,10 から 1 組目のタスクセットで 100 回実験した際の最悪メモリ消費量の平均は α=1000(従来手法)が 1248683 で一番大きくなり, α=1(従来手法)が 1112681 で一番小さくなった.提案手法は 1118658 となった.最悪メモリ消費量が最大になるときはどの α の値でも同じで最小となるときも同じとなった.さらに α=1000(従来手法)では 42 回, α=100(従来手法) α=1(従来手法),提案手法は 1 度もデッドラインミスをしていない.以上の結果から 1 組目のタスクセットの実験においては最適に近い従来手法の値と同じくらいのメモリ削減量とデッドラインミス回数を達成できていることがわかる.

図 11,12 から 2 組目のタスクセットで 100 回実験した際の最悪メモリ消費量の平均は α=1000(従来手法)が 1349300 で一番大きくなり,提案手法が 12259112 で一番小さくなった.最悪メモリ消費量が最大になるときはどの α の値でも同じで最小となるときも同じとなった.以上の実験結果から提案手法は従来手法よりも手法として優れていることがわかる.さらに α=1000(従来手法),100(従来手法)では 45 回, α=1では 7 回デッドラインミスしているが,提案手法は 1 度もデッドラインミスをしていない. 以上のパラメータ変更後の実験結果から提案手法は従来手法よりも手法として優れていることがわかる.よって,今回の実験においては提案手法 αの自動推定は有効であることがわかる.

8. あとがき

本研究では,先行研究[3]で提案されていたヒープメモリと実時間制約を共に考慮したスケジューリング手法 LMCLF で使用されるパラメータαのより適切な値を自動的に導出する方法を提案した.評価実験により，提案手法は事前にαの値を設定しなくても，平均的に従来手法とほぼ同等かそれ以上のメモリ削減量やデッドラインミス回数を達成できることがわかった．しかし,今回は実験においてスケジューラが 1 プロセッサ環境下しか想定していないためこれを 2 プロセッサ環境下でも対応できるようにさせていくのが今後の課題となる.

参考文献

[1]R. Zurawski, “Embedded Systems Handbook, Second Edition: Embedded Systems Design and Verification”, CRC Press, 2009.
[2] Y. Machigashira and A. Nakata, “An improved LLF scheduling for reducing maximum memory consumption by considering laxity time”, In Proc. of 12th Int. Symp. on Theoretical Aspects of Software Engineering, pp. 144–149, IEEE Computer Society Press, 2018.
[3] 町頭優輝, 中田明夫, 「ヒープメモリ確保・解放量と実時間制約を共に考慮しマルチプロセッサシステムのメモリ消費量を削減するリアルタイムスケジューリング」, 電子情報通信学会ソフトウェアサイエンス研究会報告(SS2019), 信学技報(SS2019–45), pp. 25–30, 2020.
[4] T. P. Baker, “Comparison of Empirical Success Rates of Global vs. Partitioned Fix-Priority and EDF Scheduling for Hard Real Time”, Technical Report TR-050601, Department of Computer Science, Florida State University, pp. 1–14, 2005.
[5] J. Lee, “Time-Reversibility for Real-Time Scheduling on Multiprocessor Systems”, IEEE Transactions on Parallel and Distributed Systems, Vol. 28, No. 1, pp.230-243, 2017.
[6] J. Lee and I. Shin, “EDZL Schedulability Analysis in Real-Time Tasks”, IEEE Transactions on Software Engineering, Vol. 39, No. 7, pp. 910–916, 2013.
[7] J. Lee, A. Easwaran, and I. Shin, “Laxity Dynamics and LLF Schedulability Analysis on Multiprocessor Platforms”, Real-Time Systems, Vol. 48, Issue 6, pp. 716–749, 2012.
[8] J. Lee, A. Easwaran, and I. Shin, “Maximizing Contention-Free Executions in Multiprocessor Scheduling”, In Proc. of 17th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 235–244, 2011.
[9] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, “Proportionate Progress: A Notion of Fairness in Resource Allocation”, Algorithmica, Vol. 15, Issue 6, pp. 600–625, 1996.

謝辞

本研究を行うにあたってご指導いただいた，中田明夫教授に心より感謝申し上げます．また，本研究へ貴重な助言をいただいた村田佳洋准教授，双紙正和准教授，島和之准教授，佐藤康臣助教に心より感謝いたします．

最後に，お世話になった広島市立大学組込みデザイン研究室の皆様にお礼申し上げます．

付録

6章で説明した実験で用いたプログラムのうち,文献[3]からの変更部分を以下に掲載する.

```
*優先度関数値が小さい順にソートする関数*/
void LMCLF(){

  double priority_func[S];                    //優先度関数値格納変数（作業用）
  double memory=0,minmemory=MAX;                    //メモリ消費量格納変数
  double priority_func1=0,priority_func2=0;
  int i = 0, j = 0, k = 0,l = 0;                    //カウント用変数
  double alphauppermin=MAX,alphalowermax=0;    // α の範囲最大最小
  double alphauppermin sav=MAX,alphalowermaxsav=0;    // α の範囲最大最小(保存用)
  double tempalpha1=0,tempalpha2=0;  //仮の α の上限下限格納関数
  double WCETLaxity1=0,WCETLaxity2=0; //残余実行時間×余裕時間
  double randma1=0,randma2=0;  // α ×消費メモリ増分
  int keta1=0,keta2=0; // α ×消費メモリ増分と残余実行時間×余裕時間の桁数を引いたもの
  int besti,bestk;    // 最小メモリとなる i と k を記憶

    pthread_mutex_lock(&mutex);
    alphadiff=0;
          /*換算レート α の決定*/
  for(i=0;i<TN;i++){
    alphauppermin=MAX,alphalowermax=0;
      if(state[i] == 1){
        for(j=0;j<TN;j++){
          if(state[j] == 1){
            if(i==j){
            }else if(rand_memory[i][step[i]] < rand_memory[j][step[j]]){
                    // m(i) α +Ci*Li<m(j) α +Cj*Lj && m(j)>m(i) --> Ci*Li-Cj*Lj<(m(j)-m(i)) α  -->  α >(Ci*Li-Cj*Lj)/(m(j)-m(i))
              tempalpha1=(((task_data[i].WCET - step[i]) * task_data[i].Laxity_Time)-((task_data[j].WCET - step[j]) *
                          task_data[j].Laxity_Time))/((rand_memory[j][step[j]])-(rand_memory[i][step[i]]));
              WCETLaxity1=(((task_data[i].WCET - step[i]) * task_data[i].Laxity_Time) + ((task_data[j].WCET - step[j]) *
                           task_data[j].Laxity_Time))/2;
              randma1=(fabs(tempalpha1)*(fabs(rand_memory[i][step[i]]) + fabs(rand_memory[j][step[j]])))/2;
                    if(alphalowermax<tempalpha1){
                      alphalowermax=tempalpha1;
                      keta1=get_digit(randma1) - get_digit(WCETLaxity1);
                     }else{
                     }
                }else{ // m(i) α +Ci*Li<m(j) α +Cj*Lj && m(j)<m(i) --> (m(i)-m(j)) α <Cj*Lj-Ci*Li -->  α <(Cj*Lj-Ci*Li)/(m(i)-m(j))
                    tempalpha2=(((task_data[j].WCET - step[j]) * task_data[j].Laxity_Time)-((task_data[i].WCET - step[i]) *
                                task_data[i].Laxity_Time))/((rand_memory[i][step[i]])-(rand_memory[j][step[j]]));
                    WCETLaxity1=(((task_data[j].WCET - step[j]) * task_data[j].Laxity_Time) + ((task_data[i].WCET - step[i]) *
                                task_data[i].Laxity_Time))/2;
                    randma1=(fabs(tempalpha2)*(fabs(rand_memory[i][step[i]]) + fabs(rand_memory[j][step[j]])))/2;
```

```
            if(alphauppermin>tempalpha2){
                alphauppermin=tempalpha2;
                keta1=get_digit(randma1) - get_digit(WCETLaxity1);
            }
    }
  }
}
if(!(alphauppermin>0 && alphalowermax < alphauppermin)){
    continue;
}
fprintf(stderr,"%lf <= alpha <= %lf for scheduling task %d first¥n",alphalowermax,alphauppermin,i+1);
alphauppersminsav=alphauppermin; alphalowermaxsav=alphalowermax;
for(k=0;k<TN;k++){
    alphauppermin=alphauppersminsav; alphalowermax=alphalowermaxsav;
    if(state[k] == 1){
        for(l=0;l<TN;l++){
        if(state[l] == 1){
        if(k==l){
         }else if(rand_memory[k][(k==i)?(step[k]+1):step[k]] < rand_memory[l][(l==i)?(step[l]+1):step[l]] ){
          // m(k) α+Ck*Lk<m(l) α+Cl*Ll && m(l)>m(k) --> Ck*Lk-Cl*Ll<(m(l)-m(k)) α  -->  α>(Ck*Lk-Cl*Ll)/(m(l)-m(k))
              tempalpha1=(((task_data[k].WCET - (k==i)?(step[k]+1):step[k]) * task_data[k].Laxity_Time)-((task_data[l].WCET –
                          (l==i)?(step[l]+1):step[l])*task_data[l].Laxity_Time))/((rand_memory[l][(l==i)?(step[l]+1):step[l]])-
                          (rand_memory[k][(k==i)?(step[k]+1):step[k]]));
              WCETLaxity2=(((task_data[k].WCET - step[k]) * task_data[k].Laxity_Time) + ((task_data[l].WCET - step[l]) *
                           task_data[l].Laxity_Time))/2;
              randma2=(fabs(tempalpha1)*(fabs(rand_memory[k][step[k]]) + fabs(tempalpha1)*fabs(rand_memory[l][step[l]])))/2;
              if(alphalowermax<tempalpha1){
                  alphalowermax=tempalpha1;
                  keta2=get_digit(randma2) - get_digit(WCETLaxity2);
               }else{
               }
               }else{
            // m(k) α+Ck*Lk<m(l) α+Cl*Ll && m(l)<m(k) --> (m(k)-m(l)) α<Cl*Ll-Ck*Lk    -->    α<(Cl*Ll-Ck*Lk)/(m(k)-m(l))
                  tempalpha2=(((task_data[l].WCET - (l==i)?(step[l]+1):step[l]) * task_data[l].Laxity_Time)-((task_data[k].WCET
                              -(k==i)?(step[k]+1):step[k])*task_data[k].Laxity_Time))/((rand_memory[k][(k==i)?
                              (step[k]+1):step[k]])-(rand_memory[l][(l==i)?(step[l]+1):step[l]]));
                  WCETLaxity2=(((task_data[l].WCET - step[l]) * task_data[l].Laxity_Time) + ((task_data[k].WCET - step[k]) *
                               task_data[k].Laxity_Time))/2;
                               randma2=(fabs(tempalpha2)*(fabs(rand_memory[l][step[l]])+fabs(tempalpha2)*
                                       fabs(rand_memory[k][step[k]])))/2;
                  if(alphauppermin>tempalpha2){
                          alphauppermin=tempalpha2;
```

```
                                keta2=get_digit(randma2) - get_digit(WCETLaxity2);
                            }
                        }
                    }
                }
                if(alphauppermin>0 && alphalowermax < alphauppermin){
                    fprintf(stderr,"%lf <= alpha <= %lf for scheduling task %d and then task %d¥n",alphalowermax,alphauppermin,i+1,k+1);
                    if(i==k){
                        if(rand_memory[i][step[i]+1]>0){
                            memory=rand_memory[i][step[i]] + rand_memory[i][step[i]+1];
                        }else{
                            memory=rand_memory[i][step[i]];
                                                        }
                    }else{
                        if(rand_memory[k][step[k]]>0){
                            memory=rand_memory[i][step[i]] + rand_memory[k][step[k]];
                        }else{
                            memory=rand_memory[i][step[i]];
                        }
                    }
                    if(minmemory>memory){
                        minmemory=memory;    besti=i; bestk=k;
                        if(alphauppermin<MAX){
                            if(((keta1+keta2)/2)>0){
                                alpha=((alphalowermax + alphauppermin)/2)/(pow(10,(keta1+keta2)/2));
                            }else{
                                alpha=((alphalowermax + alphauppermin)/2);
                            }
                        }else{
                            if(((keta1+keta2)/2)>0){
                                alpha=(alphalowermax)/(pow(10,(keta1+keta2)/2));
                             }else{
                                alpha=(alphalowermax);
                            }
                        }
                    }
                }
            }
        }
    }
}
```