計算機アーキテクチャ 38－1
（1980. 6. 25）

適応機構を備えたオペレーティングシステムアーキテクチャ

石川千秋，坂村健，前川守
（東京大学理学部情報科学科）

1. はじめに

解こうとする問題の構造を計算機の構造に写像することは一般に難しい問題である。問題解決のためのアルゴリズムとコンピュータを作る電子回路，半導体構造の意味的な差は余りにも大きく，これはGagliardiによってセマンティックギャップ[1]と呼ばれているが、その存在故に解くべき問題を計算機構造に写像することは、現在いくつかの階層構造を経て行われており、これが計算機システムの効率悪化の主要な要因となっている。しかし、この階層的写像法を全くなくすことは、現在のところ考えにくい。そこでより効率の良い計算機構造の実現のためには、問題のアルゴリズムと計算機間の構造のギャップを最小の手間でつなげるような階層構造を見つけることが重要である。これは一般に最適トレードオフの発見問題であり，これは計算機アーキテクチャの本質である。さて、最適トレードオフの発見は解くべき問題のアルゴリズムの検討から始まる。解析的（静的）な手法ですべて解明できればよい。ところが実際にはアルゴリズム，問題の構造の解析を静的に完全に行うことは難しい。問題のほとんどは複雑すぎて静的な解析だけでは不十分で、動的な動作解析により初めてその性質がわかるものに属する。それにもかかわらず、現在の計算機の構造決定法には，このような静的方法しかとられなかった。そのため、無駄が大きく、進歩したハードウェアデバイスを最大限活用するに至っていない。

ここで以上の議論をより現実的例を挙げ考察する。オペレーティングシステムは計算機上で走るプログラムの中でも、最も複雑なもののひとつであり、静的な考察だけでは、最適な構造を決めることはできないように思われる。皮肉なことにオペレーティングシステムは計算機で我々が得たい直接的な解を提供してくれる問題（プログラム）ではないが、計算機を必要とする最大の問題（プログラム）のひとつである。オペレーティングシステムはジョブの要求をシステムリソースに写像するための必要な中間構造であり、そこにオーバヘッドが入ってくる。

このような理由から、オペレーティングシステムの最適構造を考えることは、ユーザジョブの要求をより直接、システムリソースに反映させて、最高性能を引き出すことに通じる。また、問題に適した最適オペレーティングシステムはVLSIベースの高度にパーソナライズされたシステムにとっても必須である[2]。以上のような概念にもとづき、本論文では、動的な情報をもとに、オペレーティングシステムの最適な構造を決定する方法について論ずる。これをオペレーティングシステムのダイナミックチューニングと呼ぶ。従来このような観点からオペレーティングシステムについて述べた論文はないので、まず動的、即ち実行時にならないと決定され得ないような因子についてまとめる。またそこに適応性を持たせた場合の影響について検討する。個別ではあるがいくつかの例を挙げる。さらに個々の適応するための因子間の問題についても考察を行う。

このような異なる因子間にまたがる相乗チューニングにより、システ

ムの性能が静的情報だけをもとに設計された場合に比べて、大中に向上するというのが本論文の結論である。

2. チューニング

過去において、また現在でもそうであるが、我々はシステムの設計，構築にあたっては多くの注意を払ってきた。しかし、ユーザがこれらのシステムをどのように使っているかとか、オペレーティング・システムの設計に限っていえば、ジョブがオペレーティング・システム，広くいえばコンピュータ・システムにどのような要求をしているかということを十分考慮して設計をすすめてきたとは思われない。コンピュータ・システム，オペレーティング・システムの性能が予想と大中に違うことがしばしばあるのは、実際のジョブの要求をオペレーティング・システムの構築の際にうまく取り込めなかったためである。

この論文ではオペレーティング・システムをユーザ・コミュニティーによって種々の変化をするワークロードに適応させることを考えていく。ジョブ（JOB）とはコンピュータ・システムが処理するユーザの問題の総称である。またオペレーティング・システムはこれらのジョブに対してサービスを与えるためのプログラム群である。

これらのジョブはシステムに対する"要求"によって特徴付けることができる。システムをリソースのネットワークと考えてみよう。このリソースの典型的な分類としては、CPU，メモリ・システムを構成するメイン・メモリ，二次記憶用デバイス，I/Oシステムを構成するチャンネル，ディスク、及びドラムなどがある。各ジョブは各リソースに対する要求量とあるリソースを使った後他のリソースを要求する分岐確率によって特徴付けられる。これをジョブ・パターンという。そしてこれは例えば行列網の理論により完全にモデル化しえる。（図1）

システムのコンフィギュレーションを変えることにより、ある特性を持つジョブ・パターンに対応を付けて目標性能を達成していくことをチューニングと定義することができる。システム・コンフィギュレーションとは、オペレーティング・システムで使うポリシー，アルゴリズム，種々のデータ表のサイズなど我々が変えることができるものすべてをいう。そしてこのように考えると、オペレーティング・システムのコンフィギュレーションがジョブ・パターンとうまく対応せずにリソース・ネットワーク中にボトルネックが生ずる時に必要なものがチューニングであるといえる。

リソース・サービスの要求量，リソースの要求頻度は、個々のジョブに本質的に必要なものと、オペレーティング・システムがジョブとコンピュータ・リソースの間に介在するが故に必要な部分とに分けられる。オペレーティング・システムによる部分はオペレーティング・システムの構造，アルゴリズムやパラメータの値、つまりシステムのコンフィギュレーションによって変更できる。チューニングはこれらを変えて，目標達成を妨げるボトルネックを取り除くのである。またこのようなチューニングを行う時には、ジョブをクラム分けして，優先すべきジョブ・クラスのパターンにマッチしたコンフィギュレーションを選び、他のジョブ・クラスの犠牲のもとで、優先ジョブのパフォーマンスを上げることが多い。

やっかいなことにチューニングによって、システム中のボトルネックは取り除かれるが，普通は新たにボトルネックとなるサブ・システムが登場することがある。（図2）しかし、その環境

で目標性能が達成されていれば良いのである。そうでなければチューニングをくり返して、顕著なボトルネックを除去していくことになる。幾度もチューニングを行っても、効果が期待できないことがある。それはジョブ・パターン、特にリソースの要求に大きなかたよりがある場合である。ひとつのリソースに過大な要求をしている場合には、そのリソースの能力を大巾に引き上げる他に性能改善の手はない。ハードウェアのアップグレーディングが必要とされる。

このようにチューニングの限界は常に頭に置いておく必要があろう。しかし実際にはコンピュータ・システムの運管方法がまずく、その限界能力以前の不完全な性能しか引き出していない場合の方が圧倒的に多いように思える。

チューニングの適用可能な場合をいくつか挙げてみよう。チューニングが大きなメリットを発揮するのは、ユーザ・デマンドを処理して目標性能を達成できるだけのリソースがあるにもかかわらず、システムのコンフィギュレーションがジョブ・パターンとうまく対応せずに貧弱な性能しか引き出していない場合がまず挙げられる。さらにこれを発展させて、ワークロードのジョブ・パターンが安定している場合、あるいはユーザがジョブ・パターンを予測できる場合には、何の変更も加えない、現在の一般のシステムよりも高い性能を引き出すようにすることができることも意味する。またワークロードが急激に変化した場合、自動化されたチューニング・プロセスはシステム性能の急低下を防いで、性能の安定化に役立つ。またチューニングのためのモニタ・データ、モデル化の方法は広い意味で新しいコンピュータ・システムを考慮していく上でのデータの枠組みを提供する。次章ではコンピュータ・システムの各部分でどのようなチューニングが可能かについてみていく。

3. チューニングの方法と実例

システムを大きく、CPU, メモリ, I/O, ファイルシステムとに分けて、それぞれがシステムの目標性能の達成のボトルネックとなっている時、どのようなチューニングが可能かを表1について示す。ここでは各サブ・システムを別々のチューニングの対象としているが、実際にはサブ・システムは他のサブ・システムにも影響を及ぼしているのでそれも考慮しなくてはならない。さらに広いレベルをも考えたシステム全体のチューニングについては、次章で考える。以下、例を示しつつ、表に説明を加えていく。

CPUがボトルネックとなっている時、チューニングの対象となるのはオペレーティング・システムのオーバ・ヘッドを減らすことである。これはジョブの要求に対して、システムが不必要に多くのスーパーバイザ・コール(SVC)を行なっていることから起こり得る。例えば少量のメモリ・ブロックをフェッチするかわりに、大きなメモリ・ブロックを1度にフェッチするようにして必要なSVC回数を減らすことができる。さらにSVCの回数はシステム中の他の部分のチューニングで影響を受ける。

例えばI/Oシステムで取り扱う転送ブロック・サイズについて考えてみよう。ブロック・サイズを大きくすると、同じ量のデータを送るのに必要な転送回数は少なくなる。それに応じて、SVCの回数も減り、入出力に必要なサーチ, シーク, チャンネルの終了時間、及びSVCに伴なうCPU時間も少なくなる。ブロック・サイズとCPU時間の関係の例を図3.1に示す。もちろんむやみにブロック・サイズを大きくする

と、ブロック中の無駄な部分が大きくなり、不必要な部分までが転送されることになるので、最適の大きさを選ばなければならない。いずれにせよ、SVCを減らしCPUのロードを軽くすることは可能である。

I/Oシステムがボトルネックとなっている時にはボトルネック・デバイスをなくすためにロード・バランシングの手法が適用できる。Hughes & Moeはこれを取り扱いCPUの使用効率の12%の向上を得た。[2] さらにデバイスがいくつものシリンダに分かれている時、その中でのアームのシーク距離、アームに対する要求の衝突回数を減すことができる。Gurinらは、同一のディスク上のファイルに連続してアクセスがなされると、デバイス間の並列処理の可能性が減り効率が落ちるというモデルを考え、n個のファイルをd個のディスクに割り当てて、今述べたことが起きる可能性を小さくするための手法を論じている。ファイルiの次にファイルjを参照する頻度P_{ij}を測定して、

$$C=\sum_{\text{ディスク}}\sum_{i,j\in(\text{同一のディスク})}P_{ij}$$

という指数を最小にすることを考え、実際のデータをもとに、Cの10~20%程度の減少が得られている。[4]

I/Oシステムでの転送ブロック・サイズの変更は先程CPUに関して述べたようにシステムのオーバーヘッドを減らすのに役立つ。

デバイスのコントロールポリシーを変えることにより、システムのオーバヘッドを減らすことができる。

例えばディスクのコントロールに、FCFS(First Come First Served)とSSTF(Shortest Seek Time Fast)の二つの方法を使い分けることができる。ディスクのロードが軽い時にはFCFSとSSTFの性能に大差はない。この時にはプログラムが短かくて、Xモリ要求量が少なく、CPU時間の要求量も少ないFCFSを使うことによって、システムのオーバヘッドを少なくできる。しかしディスクの負荷が大きい時には、性能の良いSSTFを使うことによって、ディスクがボトルネックとならぬようにすることができる。このようにして、コントロールポリシーもチューニングの対象となり得る。

Xモリ・システムがボトルネックとなっている時には、表に挙げた手法がある。プログラム再構成は、プログラム・モジュール間の参照関係、それぞれの大きさを考えて、モジュールの結合順序を変えて、参照の局所性を高めたり、同一ページ中に関係の高いルーチンを入れてページングの必要量を減らすことである。この方法は手間が比較的かからず、しかもページング・システムのもとでは有効な方法であることが知られている。[3] ロータにこのようなインテリジェンスを持たせて、頻繁に使われるプログラムをワークロードに合わせて、再構成していくことは有効であろう。

またページングシステムのコントロールをプログラムのページ参照列のパターンに合わせて、コントロールすることによって、良好な性能を引き出すことができる。例えばDenningのWSポリシーがある。

ここではページ活動をモニタして、多重度を変えて、Xモリの割り当てを変更するスケジューラのシミュレーションを行ったので、それについて述べる。

システムは図3.2のような閉じた待ち行列網を使ったモデルである。ジョブがページ・フォールトを起こすと、ページング用デバイスに移る。ジョブのモデルはページ・フォールトの多い期間と少ない期間の間を動くセミ・マルコフ過程で記述され、それぞれの期間中で

は、ページフォールトの間隔は、ライフタイム関数の値で決定されるとした。ここで目標性能として、CPUの使用効率を高めることを考えた。このシステムではCPUの使用効率は、閉じた待ち行列網の理論により計算できる。(付録A)それによれば、ジョブに割り当てられるメモリサイズによってCPUの使用効率は変化する。CPUの使用効率を下げないためには、スラッシングが起きないように、しかも多重度を上げる必要がある。そこで解析結果をもとに最適な多重度を求める。それをもとに、ページングが活動をモニタし、スケジューラがCPU待ちのジョブをブロックあるいはリリースして多重度を変えるスケジューラを構成した。多重度を固定したシステムとスケジューラによって多重度を変更するシステムのCPUの使用効率の時間変化のシミュレーション結果の一例を図3.3に示す。

このようなスケジューラによってスラッシングも避けられるし、もしスラッシングに陥っても、その状態から速やかに抜け出ることがわかる。

この例はチューニングの中でも比較的短期間にジョブの特性が大きく変わる場合に、システムの性能が大巾に低下することを防ぐためのダイナミックなチューニングといえる。

4. 因子の相関関係と広域チューニング

4.1 相関関係

前章でオペレーティング・システムの各部にどのようなチューニングを行うことができるかを示した。そこで挙げたいくつかの例からもわかるように、オペレーティングがシステムの一部に変更を加えるとそれは、他の部分にも影響を与える。オペレーティング・システムの一部のチューニングが他にどのような関係を持つかを図4.1に示した。

図4.1は、例えばデータ転送の際のブロック・サイズを変更するとSVCの回数で、CPUにかける負荷が変わり、かつメモリ中のバッファ部分が大きくなったりブロック中の無駄が生じたりして、メモリ・システムの負荷を変えるということを意味する。

図からわかるように、種々のパラメータ、アルゴリズム、あるいはポリシーの変更はシステムの各部に影響を及ぼしていく。チューニングによってボトルネックを除こうとする場合には、この相関についての考察がなされていなければならない。

このようなオペレーティング・システムの各部の相関が、コンピュータ・システムの他の部分のチューニングにどういった効果を及ぼすかについて考えてみる。

計算機のインストラクションセットを応用に合わせて変化させる手法、インストラクション・チューニング[4]を例にとれば、CPUの資源の有効利用を計ることによって、CPU時間の要求量が減ることが考えられる。これによって、CPUがボトルネックとなっている場合に、それを取り除ける可能性がある。また、インストラクション・チューニングはCPUばかりでなく、メモリ参照パターンを変える可能性もある。故に、もし、このメモリ・サブシステムへの影響を考えないとすると、インストラクション・チューニングの持つ能力向上の可能性を最大限に取り出すことはできないだろう。相互に関連したオペレーティング・システムの一部に外部からの変更が加わる場合には、チューニングはその変更による能力向上を引き出すための必須の手順である。

もっと広く、コンピュータ・システムで走るプログラム・モジュールのチューニングを考えるなら、例えばページン

グ・システムの好むページ行動をプログラムにとらせるような方法がある。このようなチューニングは本来ならば、ユーザの要求をシステムに合ったものにすることであって、ユーザのなすべきことであるが、ここでもプログラムのチューニングがオペレーティング・システム全体に与える効果は大きい。そしてシステムの相関を良く知ることが重要と思われる。

4.2 グローバルチューニング

チューニングに使われるワークロードの特性のデータは当然のことながら、チューニングのみならず新システムの設計の際の基本データとなり得る。このデータはソフトウェアとハードウェアのトレード・オフの決定、オペレーティングシステムの基本設計には欠かせない[8]。基本的な意味でシステムの構造を改めていくのに役に立つ。

チューニングの適用できるオペレーティング・システム自身の構築については、チューニングの要求する処方を実行できるだけの、柔軟性が必要とされる。柔軟性を高めるためオペレーティング・システムのファームウェア化の研究も重要であろう。

ミニ、マイクロ・コンピュータの発展に応じて必要となってきたオペレーティング・システムの自動生成の際にも、このような自動チューニング機構を考えておくことが望まれる。

またオペレーティング・システム中のパラメータの変更の効果の評価を、量的パラメータのみならず、質的パラメータの場合についても行えるように、システム・モデルとワークロードのモデルを確立して、洗練されたものにしていくことが必要である。

5. 結論

本論文では適応機構を備えたオペレーティング・システムのアーキテクチャの構築を目指して、適応の可能性のある因子と適応可能性について論じた。適応可能性のある因子は多く、それらは互いに影響を及ぼす。しかし、その関係を考慮しつつ、各相のチューニングをくり返すことによって、静的に設計されたオペレーティング・システムに比べ性能を改善することができることが結論された。本文でも述べたように、このような動的オペレーティング・システム・アーキテクチャはVLSIに代表されるパーソナルコンピュータにとってパーソナライズ(適応)化技術として重要であるばかりでなく、専用化された オペレーティング・システムの自動生成とも深い関係がある。

このようにチューニングは単なる性能向上の手法としてだけではなく、将来のアーキテクチャと、それに沿ったソフトウェアの製作にも影響を与えるデータを提供する。

御討論いただいた藤田哲也君に感謝したい。

(参考文献)

1. U.D. Gagliardi
2. Hughes and Moe "A structural approach to computer performance analysis" NCC '73
3. R. Snyder "On a priori program resturucturing for virtual computer systems" in (5)
4. N.N. Gurin et al. "A heuristic approach to file allocation problems" in (5)
5. Proc. 2nd Int'l Symp. Operating Systems Theory and Practice. (ed.) D. Lanciaux
6. K. Sakamura et al. "Automatic tuning of computer architecture" NCC '79
7. 坂村 健「超LSIベース・アーキテクチャ」bit 共立 1980.8.
8. Maekawa et al. "Performance adjustment of an APL interpreter" EUROMICRO '79
9. R.M. Shardt "An MVS tuing approach" IBM Syst. J., vol.19, no.1, 1980

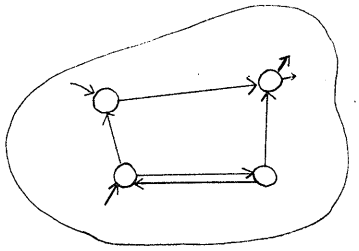
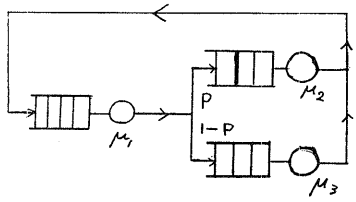


図2.1(a) SYSTEM MODEL AS NETWORK OF RESOURCES



(b) QUEUEING NETWORK MODEL EXAMPLE

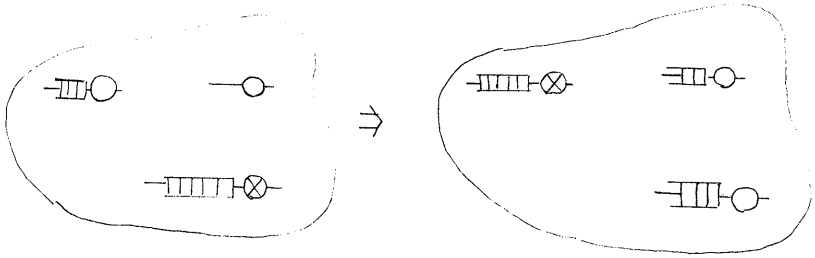


図 2.2 REMOVAL OF BOTTLENECK AND THE APPEARANCE OF NEW ONE

注1. チューニングの可能性.

CPU

- reduce the number of SVCs (Supervisor calls) by increasing the data transfer block size, by decreasing paging activity and etc.

I/O subsystem

- balance the load by
  - reconfiguration of the connections of the channels, control units and devices
  - reallocation of files over devices to remove bottleneck device or to decrease the seek distance and arm contention.
- change the device control policy to strike the trade-off point among the program size, CPU time demand and the efficiency of the control.
- reduce I/O device request by using virtual I/O instead of real I/O, thus passing information inside the main memory.
- increase the transfer block size and reduce the number of I/O SVCs and overhead(search, seek, channel startup time.)
- make frequently used program modules , such as library, commands and etc., resident in main memory to decrease the I/O device usage for the linking process.

Memory subsystem

- reduce the number of swapping by making some programs or parts of programs non-swappable at the cost of free storage area.
- use page replacement algorithm which works well with the referencing pattern of the user job. Example. Working Set Policy.
- restructure programs
  - place the modules near the modules which they make reference to (or) make reference to them.
  - place these modules in one page frame if possible.
  - make user jobs' referencing pattern to what paging system favors.
- regulate the degree of multiprogramming so that the programs can receive enough main memory to run with producing few page faults.
- make OS program module data area just as large as required by the workload, no more.

BLOCK SIZE	200	1,000	2,000	4,000	6,000
CPU TIME	73.6	21.6	14.3	10.6	9.3
CHANNEL TIME	95.2	46.9	38.4	36.5	33.3

(9)より.

図 3.1 Block size and processing time of same amount of data.

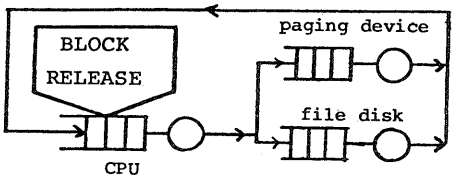


図 3.2 シミュレーション・システム

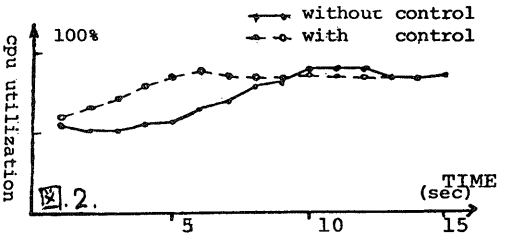


図.2.

図 3.3 シミュレーション結果の1例.

(付録)

$$P(\vec{n}) = 1/g(N,M) * B_{cpu}(n_M) W_{cpu}^{n_M} \prod_{i=1}^{p} (W_i \tau_p)^{n_i} \prod_{j=p+1}^{p+f} (W_j \tau_f)^{n_j}$$

(システムの状態確率を与える式)

$\vec{n} = (n_1, \ldots, n_p, n_{p+1}, \ldots, n_{p+f}, n_M)$

$n_1, \ldots, n_p$: # of jobs at paging devices.

$n_{p+1}, \ldots, n_{p+f}$: # of jobs at file disks.

n_M : # of jobs at CPU

$$g(N,M) = \sum_{k=0}^{N} g(n-k, m-1) B_m(k) W_m^k$$

$$B_M(k) = B_{cpu}(k) = \prod_{i=1}^{k} 1/C_{cpu}(i)$$

$C_{cpu}(i)$: the processing speed of the cpu stations when there are i jobs.

$$B_j(k) = (W_j \tau_f)^k \qquad p+1 \leq j \leq p+f$$

$$B_i(k) = (W_i \tau_p)^k \qquad 1 \leq i \leq p$$

W_j : the routing probability of the j-th disk.

τ_f : the mean service time of the file disk.

$W_i = W_{cpu} / L \cdot f_i$

W_{cpu}: the mean inter-I/O time.

f_i : the routing probability of the i-th paging device.

τ_p : the mean service time of the paging device.

図 4.1 チューニング因子の相関関係図

TUNING THERAPY	I/O SUBSYSTEM	MEMORY SUBSYSTEM	CPU
file reallocation	decrease of arm contentions. load balancing of channels, control units and devices.		
reconfiguration of connections of channels, control units and devices.	same as above.		
making program modules memory resident.	decrease of channel, device utilization.	request for free storage.	
change of transfer block size.	decrease of I/O SVC overhead, (search, seek, channel start up time.)	change of the buffer area	Frequency of I/O SVC.
Change of device control algorithm	yes.	change of program size.	change of CPU TIME request.
paging algorithm	swapping, paging frequency	yes.	paging, swapping SVC.
change of page frame size.	amount of data transferred at page fault time.	yes.	frequency of SVC
CPU scheduling, control of degree of multiprogramming	swapping, paging frequency.	memory partition.	frequency of swapping, paging, process switch SVC.
use of virtual I/O instead of real I/O	decrease of I/O device utilization and overhead.	firtual I/O area request.	decrease of I/O channel interrupts.
placing of program modules, program restructuring.	decrease of I/O activity due to paging.	yes.	frequency of paging SVC.
allocate to OS programs just as much data area as required by workload, no more.		yes.	
instruction tuning.		change of memory referencing pattern.	decrease of load due to better utilization of CPU resources.
program module tuning. (program restructuring,	elimination of unnecessary I/O.	yes, probably.	elimination of unnecessary calculation, use of better algorithms, decrease of SVC and et