大域ディペンダブルプロジェクト

～ディペンダブルアーキテクチャグループ　平木研究室～

平木　敬

情報理工学系研究科コンピュータ科学専攻

あらまし　コンピュータは今日では私たちの生活を支える技術であり、コンピュータシステムに対して要求されるディペンダビリティは著しく高くなっている。このような現状において長く使えるコンピュータ、完全にシステムのダウンしないコンピュータへの需要は増している。私達は、本 COE プロジェクトの一環として、突発的な原因不明の部分的なハードウェアの故障が発生する状況において故障とともにハードウェアの性能は徐々に落ちるものの、完全にはダウンしないハードウェアを提案した。これを解決する現在の方法としてはデバイスを多重化して使う方法、現在正常に動いている別のもので代用する等の方法が提案された。私達は、再構成する手法を用い、より多くの故障に耐えられる柔軟なハードウェアを提案した。この機構における速度という側面より、性能を評価した。

1. はじめに

コンピュータは私たちの社会を支える根幹技術の一つであり、その重要性は年を経るごとに増しつつある。社会全体が計算システムに依存する現在、その故障により社会に深刻な被害を与える例も多々ある。非常に長期間の使用に耐え、完全にはシステムが停止しないコンピュータへの要求は増している。そのためには現代のコンピュータシステムを支える CPU、OS,ハードディスクなどの全ての部分要素における対故障性を再考慮する必要がある。本稿では計算の実行部分である CPU について議論する。

CPU における故障は正常な命令実行を行わないという点で長期にわたり安定なシステムを構築する際の最大の障害であり、それに対するさまざまな手法が提案されてきた。

回路の多重化、回路の代用は CPU 故障を回避する方法として、従来から用いられてきた。前者は同じ回路を複数所持し、壊れたときには残った正常動作をする回路を利用するという手法である。後者の手法はある部分の回路が壊れると別の回路を用いて代用を実現し、壊れた機能と同じ役割を手法である。前者の手法は回路サイズにおいて、後者は速度性能の大幅な低下が見られるという欠点を持つ。

これらの状況を踏まえ、私達は回路における再構成の手法を用いることにより、回路に柔軟性を与え、故障においても速度性能の変化をもたらさないハードウェア再構成方式を提唱する。本稿では、実行部分に対する再構成手法を用いて故障回避を行ったときの速度性能の変化をシミュレートし、この機構を実装する際の有用性を評価する。

本稿は次のように構成されている。2 章では従来の手法における問題について説明し、3 章ではそれに対して再構成の手法の説明とそれを具体化するためにモデル化を行う。4 章ではそれを実装、測定し、考察を行う。5 章ではまとめを述べる。

2. 故障と修復

CPU における故障は大きく 2 つに分けられる。一時故障と固定故障である。

前者は電界の不安定さにより回路が保持すべき内部状態が突発的に変化する現象である。特に CPU は事前にある部分の電界が不安定であることがわかっているなどの特別な場合を除いて突発的な内部状態の遷移への耐性をもたないため CPU は停止もしくは正常な実行が行われなくなる。

一方後者は内部回路の断線、急な導通のために内部回路の論理が変化する現象である。この現象が発生すると内部論理が永久的に変化してしまうために以後の全ての計算に異常をきたす。

故障に対する考え方は主に故障を出来るだけ発生させないという立場と故障が発生したときにはどのようにして被害を最小限に食い止めるかという立場の 2 つがある。前者は確かに軽視するべきではないが、本稿では特に後者において重点的に説明する。

故障に対する基本的な手法は多重化であり、最終的な形としては多重化をより圧縮した形態である。一時故障に対するものとしては最終的な形である符号化により回避することが可能であるが固定故障に対しては適用が難しい。システムと

して対故障性を高めるということは壊れたときにためのパスをより増やすことであるともいえる、ただし、それは資源消費を最小化するという目的と相反する。どの場所で、どのくらいの回路共有度を持つのかということも考慮する必要がある

まず多重化とは単純に全ての回路について予備回路をつむということである。この手法における長所としては 3 つ挙げられる。まず一つ目としては理解しやすい。これにより他のシステムに比べより多くの人間がこのシステムについて考察を行うこととなり、より多くの具体例を得ることが出来るということが予期される。二つ目としては呼び回路と本来の回路はほぼ独立であるということである。つまり、本来の回路の如何を問わず、予備回路は独立に考えることが出来るためスケーラビリティーにおいて優れている。第 3 の長所として実装面での容易さがあげられる。比較的システムが単純であること、その各部分回路の独立性より、よりシステムのバグを容易に取り除くことが出来る。

確かにこの手法はすばらしいものの、全ての点において優れているとはいいがたい。というのはこの手法は大きく資源を消費してしまうという欠点がある。純粋にある CPU の N 倍の回路をつむことは出来るが呼び回路として使われるのはそのうちの一部であり、非常に資源としては無駄の多いものである。

この手法は消費資源のあまり問題にならない場合は非常に有用である。予備回路をつむという本質的な手法であり、一方それは非常に資源消費が激しいという観点からすると消費資源を抑えつつ予備のパスを作る他の手法に目を向ける必要がある。

2.1. 代用

代用とはあるファンクションユニットが壊れたときには別のファンクションユニットを利用して演算を行うという手法である。具体的には排他的論理積は論理和、論理積、否定を使うことにより実装される。また、積は加算、シフト、即値命令を使って実装できる。このように各演算は他のファンクションユニットを使って演算することができ、互いに依存関係を結ぶことが出来る。この手法の長所としてはあまり資源を消費せずに回路の修復を行うことが出来る。つまるところ、演算部分には新たに回路の追加をする必要が無く、データの流れをコントロールするための資源を追加するだけでよい。

ただしこの手法にも短所が存在する。各演算は多数の計算パスを持つこととなるのだが、それらの独立性が小さいためいくつかのユニットが故障すると連鎖的な再起呼び出しのため性能が劇的に変化する危険性がある。

3. 再構成による故障修復

3.1. 再構成における一般論

この手法は全ての回路を汎用性のある回路で構成し、それを組み合わせることにより全体としての回路を作る。もしある汎用回路が壊れたときには別の使われていない汎用回路を使用することにより故障を回避する。また、仮に全ての汎用回路が使用中であり、かつある汎用回路が壊れたときには内部の汎用回路の組み合わせを再編成し、あまり重要と判断されない回路を汎用回路の形に分解し、それを壊れた部分にあてがうという手法である。

ただし、内部回路を再構成するにあたって、再構成以前よりも機能が落ちることは必然であり、何らかの形で機能を補完する必要がある。これにあたっては前述の回路の代用を用いることとなり、その意味で再構成の手法は代用の手法を含むと言うことが出来る。

利点としては内部回路を自由に組み立てることによる高い柔軟性である。もし実行に必要不可欠なファンクションユニットが壊れた場合には別のあまり重要ではないファンクションユニットを失う代わりに全体の機能としてはうまく動作させることが出来る。仮に部分的な故障が発生したとしても最終的にはある一部分を切り捨てることによりシステムとして復帰することが出来る。一部分を切り捨てるために徐々に性能としては低下するものの高い資源利用率で回路を組むことが出来る。

欠点としては汎用回路をどのようにすべきかという問題がある。高い資源利用率を保ちつつ、しかも高い柔軟性を得るデバイスを提案することは難しい。

3.2. 実験のモデル

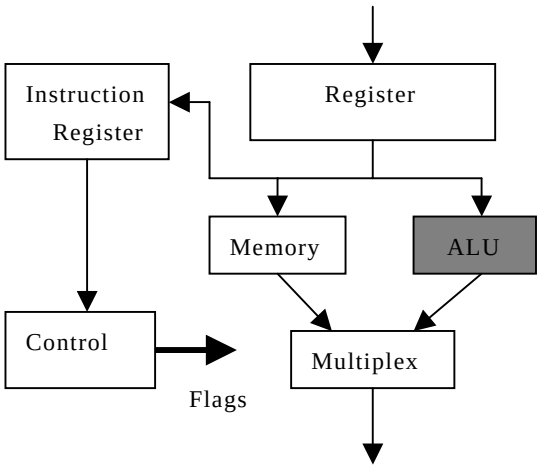
私達は再構成手法における速度性能をシミュレートしたいのだか、それを行うためには問題のモデル化を行う必要がある。

第一の問題点は CPU をどのようなアーキテクチャにするかである。ここでは RISC32 ビットアーキテク

チャ、SISD とする。確かに再構成手法は現在の主流であるスーパースカラ、パイプライン、分

岐予測を行うことは原理的に可能かもしれないが、ここでは他の二つに対するモデルとしての立場を際立たせるために前述のとおりとする。

第二に再構成の立場を明確にする必要がある。CPUの内部を大きく2つに分ける方法としてコントロールパス、データパスという分類分けを行う。コントロールパスとは CPU の各部分を制御する信号が流れる部分であり、データパスとは実際に計算するべきデータとしての信号が流れる道筋のことである。ただし、ジャンプ命令のようにデータパスの値が CPU の部分回路の制御につながるもの、逆に即値命令のように本来部分を制御すべきインストラクションよりデータパスへと流れるものもあり、両者の厳密な区分はできない。ここではレジスタ、各ファンクションユニット、そして歌旅レジスタへと戻るパスをデータパスとし、この部分に限定した再構成を考える。



第三にどのような再構成の単位をとるのかも定める必要がある。再構成の単位を Functional Block とし、ここでは入力 3 ビット、出力 1 ビットの RAM とする。Functional Block を組み合わせることにより任意の論理回路を作ることが出来る。たとえば 1 ビットの Full-Adder は 2 つに Functional Block より作ることが出来る。同様に 16 ビットの Full-Adder は 32 個の Functional Block より作ることが出来る。CPU における他の Function Unit においても同様に作ることが可能であり、それらを組み合わせることにより実行部分を形成することが可能である。

第四にどのような再構成の手法をとるのかも考える必要がある。つまり、もしある部分が故障したときにはどのようにしてその機能を補うべきかということである。ここでは予備の Functional Block をつむのではなく、壊れたときにはべつのファンクションユニットを壊して Functional Block の形に戻し、それを再配布するという手法をとる。これは故障とともに実行部分が動的に組替えられ、ファンクションユニットが減少していくことを意味する。どのくらいの故障の場合にどのファンクションユニットが使えるのかについても議論の余地のある問題であるが、ここではより多くの Functional Block を占有しているものから順に壊していくという戦略の下、表のとおりとする。(表挿入、必須)

また、故障がどのように発生するかも考慮する必要がある。ここでは再構成の単位である Functional Block においてこしょうが発生する場合を仮定するのだが、それがいつ壊れるのかを決定する必要がある。たとえば、Functional Block が参照されるたびに故障するのか、参照され、回路の電位が変化したときに壊れるのか、それとも時間の経過とともに無条件に故障するのか、その仮定により結果も異なる。ここではプログラムの実行時間に対し故障の発生率は十分小さいと仮定し、プログラムの実行時に故障は発生しないと考える。つまるところ、故障が既に発生した時点でプログラムを実行し、その実行時間がどのように変化するかを測定する。

4. 予備評価

4.1. 実験の手法

この機構の速度性能の評価を行うことは難しい。現在プログラムが動く際には CPU を支援するさまざまなプログラムの影響のため実際にどのような計算を行っているのかを調べることは不可能である。私達はこの機構における実行を追跡するためのシミュレータを作り、その上でそれぞれのプログラムを実行した。それぞれのプログラムにおいて必要な命令をトレースし、それより実行速度を計測した。

実際にこのシミュレートを使用して計測するためには次のような過程を経ることとなる。

(1) C,C++ 言語で作成したプログラムを g++2.95.3 でアセンブリ言語レベルまでコンパイルする。ただし、ターゲットアーキテクチャは SPARC であり、コンパイルオプションは-mflat(レジスタウインドウを無視)、-S である。

(2) (1)で生成した SPARC アセンブリ言語をシミュレータ用のコードへと変換する。これは SPARC アーキテクチャに依存した命令を今回の実験のモデルに対応させるためであ

り、これは独自に開発したものである。

(3) (2)により生成されたコードをシミュレータ上で実行

4.2. 具体的な実装

シミュレータの基本的な構造は図のとおりである。それぞれの命令はシーケンシャル実行であり、メモリアクセス時間等は無視する。それぞれのユニットはそれぞれの参照回数を計算し、終了時に表示する。

SPARC アセンブリ言語をシミュレータ用のコードへと変換するプログラムは両者のアーキテクチャ的な構造的相違を吸収するものである。具体的にはディレイドスロット、アドレス計算時における加減算の割り当ての変化等である。

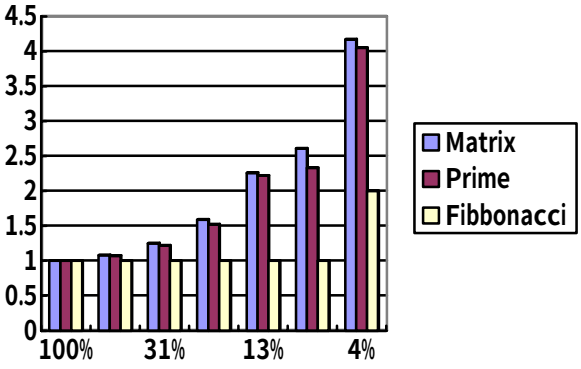
4.3. 実行プログラムの説明

実行を行ったプログラムは以下のとおりである。

(1) MATRIX　正方行列の積の演算である。これは和席の演算が集中する。

(2) PRIME　素数判定の計算である。和、積、商に演算が集まる。

(3) FIBBONACCI　フィボナッチ数の演算である。和、積に演算が集中する。



5. 結果

部分回路が壊れるとともに回路を再構成する手法におけるプログラム実行速度はグラフのようになる。

グラフ XX においては確かに速度性能の低下が見られる。ただし速度性能の低下の現れ方はプログラムにより大きく変化する。Prime、Matrix においては Functional Block が失われるにつれ、徐々に速度性能が低下する。とくに、基本的な命令である。加減算に集中するフィボナッチ数列においてはその大半の Functional Block が失われてもなお通常の速度のままである。また、汎用回路の柔軟性が功を奏し、回路のうち 10 パーセント程度の故障においては劇的な性能の変化をもたらさない。

しかし、この実験で用いられているプログラムは現在の商用に使われている CPU の上で動かされているプログラムと比較するにはあまりにも単純であり、かつ非常に規模の小さいものであると断言せざるを得ない。更なる大規模なプログラムの実験結果を必要としている。

6. まとめ

本報告ではファンクションユニットの回路の固定故障における障害を回避するために柔軟な回路合成の可能である再構成の手法を述べた。、また、障害回避に伴う速度性能の変化についてシミュレーションにより評価を行った。従来の 2 つの手法に比べより低いレベルからの復帰を行うこの手法は速度の大きな低下をもたらさないという点で有用であること示された。

参考文献

[1] Alfredo Benso, Silvia Chiusano and Paolo Prinetto ”A Self-repairing Execution Unit for Microprogrammed Processors “, IEEE/micro Sep-Oct 16-22

[2] Weaver D.L and Germond, T., editors. The SPARC architecture manual : version 9. Prentice Hall, Englewood Cliffs, NJ, 1994

[3] W. Mangione-Smith and B. Hutchings “Configurable computing: The road ahead ,” in Reconfigurable Architectures: High Performance by Configure (R.Hartenstein and V. Prasanna, eds.), Microsystems Engineering Series, (Chicago), pp. 81-96, IT Press, 1997. Proceedings of the Reconfigurable Architectures Workshop (RAW ’97)

[4] M.J. Wirthlin and B.L. Hutchigs. A Dynamic Instruction Set Computer. IEEE Workshop on FPGAs on Custom Computing Machines. April 1995.

[5] System C Version 2.0 User’s Guide, 2000. Available at www.systemc.org

[6] Scott Hauck. “the Future of Reconfigurable Systems”, in Proceedings of the 5th Canadian Conference on Field Programmable Devices, June 1998.

[7] K. Compton, S. Hauck, “Configurable Computing: A Survey of Systems and Software”, Northweastern University, Dept. of ECE Technical Report, 1999.

[8] Reiner W. Hartenstein, Micheal Herz, Thomas offmann, Ulrich Nageldinger. “On Reconfigurable Co-processing Units”, Proc. Of Reconfigurable Architectures Workshop: International Parallel Processing Symposium, 1998, pp. 67-72.

[9] MIPS Technologies, Inc. MIPS R4000/R4400 Microprocessor User Manual, Seconod Edition, October 1994.