キャッシュヒット率向上のための ソースコード自動変換ツールの開発

若山健次† 太田剛‡

静岡大学大学院情報学研究科† 静岡大学情報学部‡

1. 研究の背景

近年の計算機ではCPUと主記憶のアクセス速度の差を緩和するためにキャッシュメモリが導入されており、処理速度を上げるためにはキャッシュを有効に利用することが重要である。

キャッシュヒットを考える際には空間的局所性と時間的局所性を考慮することが重要である。プログラム中では動的メモリアロケーションがよく使われるが、データがメモリ上に散在してしまうため、空間的局所性は薄れてしまい、キャッシュとの相性があまり良くない。そこで本研究では、動的アロケーションを使うプログラムに対してソースコードを変換し、キャッシュヒット率の向上を狙う。

2. 目的

本研究では、動的アロケーションの中でも構造体に着目した。構造体にはよく参照されるフィールドとめったに参照されないフィールドが存在することが多い。そこで、多数のインスタンスを持つ構造体に対して、よく参照されるフィールドをまとめることによってキャッシュヒット率を向上させることを考える。これに関しては、既存の研究でInstance Interleaving手法が既に提案されており、そのための特別なアロケーションライブラリ ialloc が開発されている[1]。ただし、「どの構造体に対して」、「どのようによく参照されるフィールドとめったに参照されないフィールドを分けるのか」を考え、ソースコードを変換することはユーザに任されている。

そこで、本研究では上記の解析を自動で行い、より簡単にInstance Interleavingを適用できるようにするツールを開発したのでその結果を報告する。

3. Instance Interleaving

図1は、よく参照されるフィールドA,Cとめったに参照されないフィールドB,Dをもつ構造体に対してInstance Interleavingを適用した例である。ここではA,Cをまとめてメモリ上の連続番地に割り当てている。

この構造体へのアクセス命令が実行されたとき、キャッシュミスヒットが生じるとメモリからキャッシュへの一定のサイズ(Line size)のデータを読み込むため、Instance Interleavingを適用した後ではよく参照されるフィールドでキャッシュが満たされやすくなり、キャッシュの利用率向上が期待できる点がこの技法のポイントである。

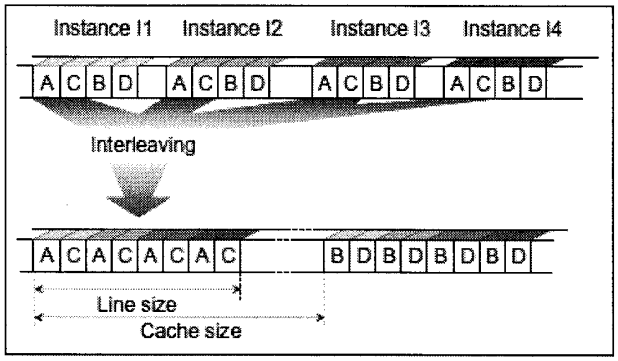

図1　Instance Interleaving(文献[1]より転載)

4. ツールの概要

ツールは以下の3つから構成される。

4.1. 静的解析部

bison、flex用いてソースコードの解析を行い、変換を行う際に必要な情報の取得、動的解析で必要となる情報(何行目で構造体のフィールドが使われているのか、何行目でアロケーション関数が使われているのかといった情報)の取得を行う。

4.2. 動的解析部

テスト・カバレッジ・プログラムであるgcov[2]を用いて各行の実行回数を取得し、どの構造体のどのフィールドがどれくらいの頻度で参照されているのかを解析する。その結果をもとにどの構造体に対してInstance Interleavingを適用するかを判別し、どのようなフィールドの分け方をするのかを決定する。

A source code translation tool to improve cache utilization

†KENJI WAKAYAMA, Graduate School of Informatics, Shizuoka University.

‡TSUYOSHI OHTA, Faculty of Informatics, Shizuoka University.

4.3. 変換部

解析結果をもとに実際にソースコードの変換を行い、Instance Interleaving を適用する。

5. 実験結果

5.1. 実行環境

CPU は PentiumM 1400Mhz、L1 キャッシュは 64KB、L2 キャッシュは 1MB の 8way セットアソシアティブである。

5.2. 実行結果

以下の 4 種のプログラムで実験した。

① 単連結リスト：単純にデータの追加、検索、削除のみを行う。

② S-Btree：文字列を Key とする二分木に対して、単純にデータの追加、検索、削除のみを行う。

③ Health：Olden Benchmark suite[3]内のプログラムの 1 種で、Columbian Health care Simulation プログラム。重連結リストを用いている。

④ Tsp：Olden Benchmark suite[3]内のプログラムの 1 種で、partitioning アルゴリズムと closest point heuristic を用いて巡回セールスマン問題を解くプログラム。二分平衡木を用いている。

	original	Converted
L1miss	4,875,292,458	4,581,822,047
L2miss	4,513,076,376	293,444
User time	10m58s	1m28s

表 1. 単連結リスト

	original	Converted
L1miss	5,432,901	9,079,861
L2miss	2,492,390	3,796,748
User time	1m0s	1m5s

表 2. S-Btree

	original	Converted
L1miss	198,756,663	237,530,669
L2miss	149,789,072	68,303,924
User time	26.7s	16.3s

表 3. Health

	original	Converted
L1miss	48,986,053	34,543,843
L2miss	30,971,949	22,919,875
User time	48.1s	42.2s

表 4. Tsp

単連結リストでは、L2 ミスを 1/15000 程度まで減らすことができ、実行時間を 1/10 程度まで減少させることができた。Health では L1 ミスは増えてしまったが、L2 ミスを減らすことができた。L1 よりも L2 のほうが、ミスペナルティのコストが大きいので速度アップには十分な効果があった。Tsp でも L1 ミス、L2 ミスともに減少し、実行速度も上がった。

一方、S-Btree では L1 ミス、L2 ミスともに増加し、実行時間も遅くなってしまった。

6. 考察と今後の課題

S-Btree が遅くなってしまった理由は、参照されやすいフィールドに配列が使われていたことにあると考えられる。ライブラリの仕様上、フィールドに配列が使われる場合は宣言部をポインタに変更し、別途 malloc を用いて割り当てるため、必要なデータにアクセスするまでの回数が一回増えてしまう。さらに、そのデータのメモリは malloc で別途確保したものであるために空間的局所性は Instance Interleaving を施しても向上しない。つまり、処理が増えるのに対してキャッシュヒット率は向上しなかったために遅くなってしまったと考えられる。

連結リストに対して二分木は L2 ミスの減少率が少ない。これはメモリ確保した構造体に対するアクセスの仕方が影響しているのだと考えられる。Instance Interleaving では参照され難いフィールドを他のインスタンスの参照されやすいフィールドで満たすのだが、このとき確保したインスタンスの順に詰め込んでいく。これによって空間的局所性が改善される場合とは、インスタンスへのアクセス順がメモリ確保の順とある程度関連性がある場合である。実際に、多数のインスタンスに対して完全にランダムにインスタンスを選びアクセスした場合は Instance Interleaving を適用するとかえって遅くなってしまった。

現在は、各構造体フィールドの使用頻度のみで参照されやすいフィールドと参照されにくいフィールドを切り分けているが、今後は構造体がどのように使われるのかといったことまで含めた解析を行う必要がある。

7. 参考文献

[1] D.N.Truong, F.Bodin, A.Seznec : "Improving cache behavior of dynamically allocated data structures" , International Conference on Parallel Architectures and Compilation Techniques, 1998.

[2] gcov：テストカバレージプログラム http://www.sra.co.jp/wingnut/gcc/gcc-j.html#Gcov (2010 年 1 月 11 日)

[3] Olden Benchmarks : http://www.irisa.fr/caps/people/truong/M2COct99/Benchmarks/Olden/Welcome.html (2010 年 1 月 11 日)