B-008

選択的関数展開によるソフトウェア高速化の検討
Study of Software Performance Tuning by Selective Inline Function Expansion

加賀　洋渡†　細木　浩二†　石川　誠†

1.　はじめに

近年，組込みソフトウェアの開発では，市場への早期投入のため開発の短期化が求められている．開発現場では，要求される性能を満たすため，開発後期にソースコードの記述変更による性能チューニングを手作業で行うことが多く，具体的にはトレースなどの実行情報を見てボトルネックを判断し，コード修正を行っている．しかしながら，上記の作業過程でソースコードの記述ミスによるバグ混入やボトルネックの判断ミスが発生し，これらが手戻りの原因となって開発期間の長期化に繋がるケースもある．そのため，自動で短期間に実施可能な性能チューニングが求められる．上記に向けた解決手法の一つとして，コンパイラの最適化が挙げられる．コンパイル時に実施される最適化は，ユーザがボトルネックを判断する必要がなく，短期間に自動で実施することが可能である．その反面，一義的な最適化しか行えず，ハードウェアや実行環境を意識しないため，搭載する機器に合わせた最適化が行えない．また別の手法として，組込みシステムの実行情報を用いて動的に最適化を行う手法が報告されている[1]．左記の手法は実行情報を利用するため，実行に即した最適化が可能となるが，実現には専用のハードウェアを追加する必要があり，特に低コスト化が要求される組込みシステムでは適用が困難となる．

本稿では，限られたリソースの中でアーキテクチャを変更せず実施可能な関数展開の自動化に注目した．特に，実行するハードウェアに合わせた最適化を行うため，展開によるコードサイズ増加や展開先のメモリ性能を考慮した選択的展開を提案し，コードサイズと性能向上のトレードオフ両立を図った．

2.　選択的関数展開の検討

2．1　関数展開による性能最適化

通常のコンパイラによる関数展開(inline 展開)では，関数はすべての展開先へ一括展開されるため，多数の箇所からコールされる関数は，関数展開によりコードサイズが増大し，メモリの容量を超える原因となる．また，一般的な組込みシステムは限られたリソースで性能を実現するため，高価で小容量の高速メモリと，安価で大容量の低速メモリを搭載する場合が多い．そのため，高速メモリ上に配置されていた関数が低速メモリ上に展開される場合には，逆に性能が劣化する可能性がある．このように通常のコンパイラによる関数展開はソフトウェアを搭載するリソースに合わせた展開が困難である．この課題に対し，展開先毎に展開(マクロ関数による展開)/非展開(通常関数のコール)を切り替える手法を選択的展開として提案する．本手法の概要を図 1 に示す．

†日立製作所　横浜研究所

従来手法は，コールしているすべての箇所に関数展開を行う．一方，提案手法は，マクロによる展開と，通常の関数コール(非展開)とを切り替える．本手法により，関数展開の適応的な制御が可能となり，前述のように一括展開ではコードサイズがメモリ容量を超えてしまう場合にも，部分的な関数展開を実施することが可能となる．また，低速メモリへの展開を回避でき，性能劣化を防ぐことができる．提案手法では，まず step1 において性能向上への寄与が大きい関数を選択する．次に step2 において，メモリ速度に応じて展開関数を選択し，最終的な展開関数を決定する．以下に step1，step2 の詳細を説明する．

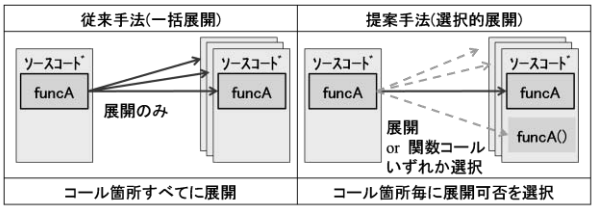

図 1. 選択的展開の概要

2．2　展開効率を考慮した関数の選択［step1］

関数展開によるコードサイズ増加を抑制するため，性能向上効果の高い展開先を選択する．本稿では，コードサイズ増加に対する性能向上を展開効率と呼ぶ．以下に展開効率の高い展開先を選択する手順の詳細を述べる．一般に，関数展開を行うと関数コールおよび付随するレジスタ退避/回復命令分のコードサイズが減少し，その一方で関数サイズ×展開回数分のコードサイズが増加する．但し，静的参照回数 N=1 の場合，対象関数の通常関数定義が不要となり，関数の実体が展開されるのは一箇所のため，結果として関数コールオーバーヘッド分のコードサイズが減少する．そのため，N=1 の場合には無条件に展開する．N≧2 の場合には，関数コールオーバーヘッドの総和時間の大きい関数，すなわち実行回数の多い関数に注目する．図 2 に従来手法と提案手法の展開効率の比較を示す．図 2 は上位関数①～④と下位関数 A,B の参照関係を示している．数字は関数の実行回数を表しており，従来手法では下位関数 A が 50 回，下位関数 B が 70 回実行されることを示している．なお，以下では下位関数 A,B のコードサイズを同一とし，即ち，展開により増加するコードサイズは関数 A,B で同一と仮定する．これに伴い展開効率を展開先数に対する関数実行回数として簡易化すると，従来手法では関数 A の展開効率が 50/3，関数 Bの展開効率が 70/2 となり，関数 B が展開候補として選択される．一方，提案手法では実行回数を細分化し，下位関数と上位関数の参照関係毎の実行回数を考慮することが可能である．そのため，

従来手法と同じく展開先を2つとした場合，実行回数の多い上位関数④－下位関数 B，上位関数③－下位関数 A の組合せを選択した場合の展開効率が最も高く((60+30)/2)，提案方式では従来方式よりも効率よく展開を行うことができる．なお，実行情報から展開効率を算出するため，トレースから実行回数を抽出する．

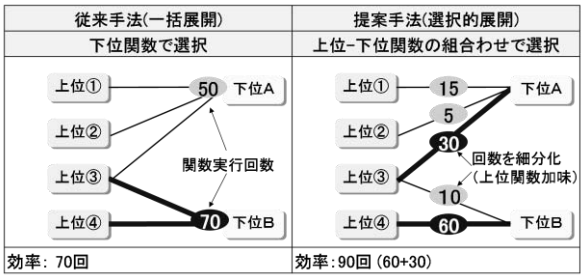

図 2. 展開効率の比較

2．3　メモリ速度を考慮した展開関数の選択 [step2]

前述のように，組込みシステムでは高速メモリと低速メモリが搭載されることが多く，関数がどちらのメモリ上で実行されるかにより実行時間が異なる．そのため，低速メモリ上に関数展開された場合には展開前より性能が低下するケースがあり得る．そこで，関数コールのオーバーヘッド削減時間よりも展開による増加時間が大きい場合には関数展開を行わない．具体例として，メモリ種別を考慮しない場合(図3 左側)は実行回数 30 回と 60 回の参照関係が選択され，展開効率は 90 となるが，上位関数③が低速メモリの場合，オーバーヘッド削減時間よりも処理時間の増加が大きくなる可能性があり，低速メモリと高速メモリの速度の差によって展開先として適切でない場合があり得る．そのため，展開先が低速メモリとなる場合を除外し，次に展開効率の高い候補を選択する(図3 右側)．

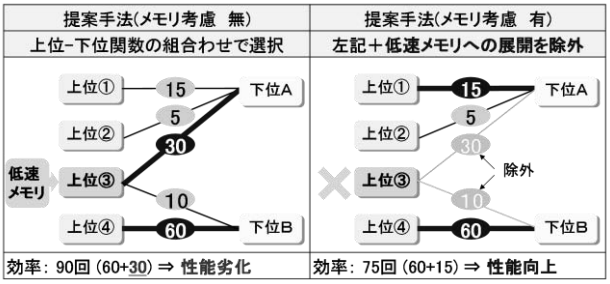

図 3. メモリ種別を考慮した展開効率の比較

2．4　判定条件の詳細

上述の step1，step2 について判定条件を表 1 に示す．ここで，T_{OH} は該当関数を通常関数としてコールした場合のオーバーヘッド時間，T_{expand}，T_{call} はそれぞれ該当関数をマクロ関数により展開した場合の処理時間，通常関数をコールした場合の処理時間を示す．step1 では，T_{OH} の大きい関数から順に展開する．今回の適用では，step2 の判定条件を簡略化し，展開先が同種もしくはより高速メモリの場合($T_{expand} = T_{call}$，$T_{expand} < T_{call}$)には，常に判定条件が成立するため展開し，展開先が低速メモリの場合($T_{expand} > T_{call}$)は，展開しても劣化のない場合($T_{expand} < T_{call} + T_{OH}$)のみ展開する．

表 1. 選択的展開の判定条件

	判定条件
step1	N=1 または，N≧2 かつ $\Sigma T_{OH} > t$ (t:閾値)
step2	$T_{expand} \leqq T_{call}$ (展開先が高速メモリの時) $T_{expand} < T_{call} + T_{OH}$ (展開先が低速メモリの時)

3.　評価実験

3．1　実験条件

提案手法について，高速メモリ(SRAM，128Kbyte)と低速メモリ(DRAM，580Kbyte)を搭載した組込みシステムを対象とし，スループット[IOPS]とコードサイズ[byte]を評価項目として比較した．

3．2　選択的展開による性能向上効果

提案手法を組込みソフトウェアに適用した際の性能向上効果および，コードサイズ削減率を表 2 に示す．提案手法では 14.4%の性能向上を確認した(表 2「step1+step2」の項)．なお，コードサイズが最適化前と比べて 127,788−123,596=4192byte 減少している．そのため，今回対象としたソースコードの参照関係を調査したところ(図4)，参照回数が 1 回のみの関数が約 50%存在しており，関数展開によるコード削減効果が大きく表れていることが判明した．そこで，適用により空いた高速メモリに，関数コールオーバーヘッド時間の長い関数を低速メモリから移動させた[step3]．再度確認を行ったところ，更に3%の性能向上を確認し，最適化前と比べて17.4%の性能向上を実現することができた(表 2「step1+step2+step3」の項)．

表 2. 性能及びコードサイズの比較

	最適化前	最適化後	
	(基準)	step1+step2	step1+step2+step3
ｽﾙｰﾌﾟｯﾄ [IOPS]	6594	7542	7742
性能向上率 [%]	−	14.4	17.4
ｺｰﾄﾞｻｲｽﾞ [byte]	127,788	123,596	127,156

IOPS : Input Output Per Second

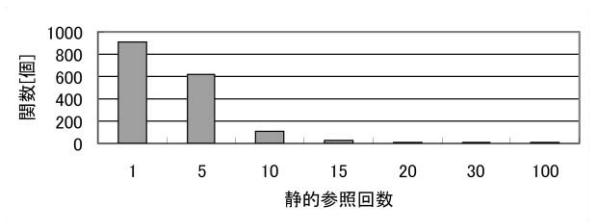

図4. 対象ソースコードの静的参照回数の分布

4.　おわりに

組込みシステムに合わせた性能最適化の選択的展開を提案し，17.4%の性能向上を確認することができた．提案手法はメモリのアクセス速度と容量を考慮し，関数展開の適応切換えを自動で行うことが可能で，CPU への依存性がなく，汎用性の点でも有利である．

参考文献

[1] 請園 智玲，田中 清史，”ハードウェア解析システムによるバイナリコードの動的最適化” 情報処理学会研究報告，ARC，Vol.2005，No. 120，pp.7−12，2005