A Compiler-automated Array Compression Scheme for Optimizing Memory Intensive Programs

Anuja Tupe - 1100227
Mike Lai - 82982
Swati Patankar - 1087351
COEN 259 - Compilers
March 17, 2015

Acknowledgement

We would like to thank Professor Ming-Hwa Wang for his guidance, his constant supervision and for providing us with the necessary information regarding the project. We would like to thank the authors of the IEEE papers using which we could get a broader view of the topic. It would have not been possible without the kind efforts of the teammates.

Table Of Contents

- Acknowledgement
- Table Of Contents
- List of Tables and Figures
- Abstract
- Introduction
  - Objective
  - What is the problem?
  - Why is this project related to this class?
  - Why we think our approach is better?
  - Statement of the problem
  - Area or scope of investigation
- Theoretical Basis and Literature Review
  - Definition of the problem
  - Related Research
  - Our Solution to Solve this Problem
- Hypothesis (or goals)
- Methodology
  - Generating/ Collecting Input Data
  - Solving the Problem
    - Language Used
    - Algorithm Design
    - Tools Used
    - Generating Output
- Implementation
  - Code
  - Design Document and Flowchart
    - Flowchart for the Parser
    - Pseudocode for the Benefit Model
- Data Analysis and Discussion
  - Output Generation
  - Output Analysis
  - Comparison of Output against Hypothesis
- Improvements for the original Paper
  - Double Array Compression (DAC)
  - Modified Double Array Compression (DAC)
  - A new signed-DDAC
- Conclusions and Recommendations
  - Summary and Conclusions
  - Recommendations for Future Studies
- Bibliography
- Appendices
  - Program Flowchart
  - Program Source Code with Documentation
  - Input/Output Listing

List of Tables and Figures

Figure number	Figure Name	Page Number
Figure - 1	Framework of the automated compression scheme	6
Figure - 2	Flowchart for the Parser	12

Abstract

In this project proposal, we present a compiler automated array compression scheme for optimizing memory intensive programs. It discusses various ways and techniques to carry out array compression and the cost-benefit model which is used at the run-time by the compiler to make decisions regarding which compressed version of the program should be executed or should the uncompressed version of the program should be executed to get the best results. The scheme is data dependent. The scheme chooses the techniques for compression based on the genre of the array (the actual data in the array) .

Introduction

Objective

The objective is to reduce the memory bandwidth consumption and thereby improve the execution speed and reduce the time required by the program to transfer the data between memory and processor. Also, making a programmer's life easier to automate the compression part for them instead of manually carrying out data compression techniques.

What is the problem?

Research shows that the memory bus on the multicore chips is a major performance bottleneck for many numerical applications. Several studies manually applied compression to make the programs faster. Few experiments showed that though the bandwidth reduction can be quite high in many cases, the overhead introduced by the compression operations are significant. In certain cases, manual compression may actually degrade the performance.

Why is this project related to this class?

The term 'compilers' speaks for itself. It is related to the class because we will be building a compiler for a subset of a language which will apply compression techniques and decide which version to execute at runtime.

Why we think our approach is better?

We develop a scheme in which the compression decision is made adaptively based on the cost-benefit analysis with parameters obtained from the input program. So, based on the benefits which can be reaped after applying the method we decide whether the compressed version of the program should be used or the uncompressed version of the program should be used.

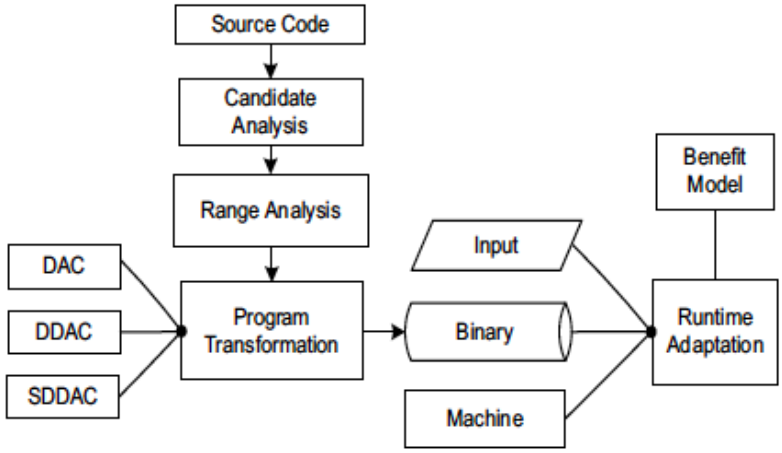

Figure 1 - Framework of the automated compression scheme

We think our approach is better because we are trying to automate a manual process of compressing arrays for optimizing memory intensive programs. Our approach is better because it saves a lot of

programmer's time which would be used for carrying out manual compression. Hence, making his/her life easier.

Statement of the problem

Manual compression techniques result into wastage of time. Although experiments suggest that they guarantee compression benefits , the overhead in applying compression operations is very significant and it might neutralize the benefits obtained after compression.

Area or scope of investigation

- We will be implementing two of the array compression techniques mentioned in the paper i.e. DAC (Double Array Compression) and DDAC (Delta Double Array Compression).
- We will be improving the DDAC method to some extent.
- Future scope - If time permits, we will also be building a cost benefit model which can be used determine which compressed version to select at run-time.

Theoretical Basis and Literature Review

Definition of the problem

Memory bus on multi core chips is a major performance bottleneck for many numerical applications. We investigate methods of compression in order to reduce memory bandwidth consumed by integer or pointer arrays.

Related Research

Some of the previous studies on this topic are -

1. Optimization of sparse matrix-vector multiplication on emerging multicore platforms by Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K.,and Demmel, J.

2. Optimizing sparse matrix-vector multiplication using index and value compression by Kourtis, K., Goumas, G., and Koziris, N and

3. Accelerating sparse matrix computations via data compression by Willcock, J. and Lumsdaine, A.

The disadvantage of these is the they manually applied compression to index arrays used in sparse matrix-vector multiplication. Experiments showed that, while the bandwidth reduction can be quite high in many cases, the overhead introduced by the compression operations can also be significant. Branch mispredictions and loop overhead may also be increased. Performance of numerical kernels can be sensitive to these factors.

Our Solution to Solve this Problem

Instead of using manual compression we use compiler automated compression scheme. We have three encoding methods used in our adaptive compression scheme.

Hypothesis (or goals)

Our hypothesis that we the change that we have introduced in the DDAC method will actually improve it. The improvement over DDAC is given below.

Improvement over the delta double array compression(DDAC) -

The original code(before DDAC) -

```
for ( i = 0; i < N; i++) {
  b[i] += a[i] * 2;
}
```

The original code(after DDAC) -

```
int prev = 0;
for (i = 0; i < N; i++) {
  int t = newa[i];
  if (t == 0xFFFF) t = a[i];
  else  t += prev;
  prev = t;
  b[i] += t * 2;
}
```

The original scheme has 2 constraints -

1. Since the scheme is storing the delta to the physical memory, so no random access is allowed.
2. The scheme assumes that the elements of the array are either decreasing or increasing, so the delta can only be positive only or negative only.

We think there is no way around constrain (1), but we find it not necessary to have constrain (2). We will come up with an implementation that does not have constrain (2). To prove that works, we will create some artificial test input set to feed to the our new scheme.

For example - one test input set can be something like this:
raw data: -10, 0, 10, 15, 12, 128, 60, 0...
delta: 0, 10, 10, 5, -3, 116, -68, -60…

The original DDAC would not allow it but our new scheme does.

Methodology

Generating/ Collecting Input Data

The input to the compiler will be java files having arrays as one of the main components as our compressions scheme revolves around compressing the arrays which will be used in the input files.

Solving the Problem

Language Used

We will be using C as the language of implementation. The purpose of the project is to build a compiler with these array compression scheme. As we have already have had a hands on experience in using C for building parser and a scanner along with Flex and Bison in our previous programming assignments, we thought it would be the best language to opt for.

Algorithm Design

- Input java files which will have arrays
- Compiler creates compressed versions of the java input file using the DAC and DDAC compression techniques
- Compiler at runtime based on the cost-benefit analysis model i.e. based on the elements in the array and some other factors, make decisions about the compressed version which should be executed or whether the original program should be executed.

Tools Used

We will be using Flex and Bison - tool for generating scanners and tool for generating parsers.

Generating Output

The input for compiler will be any java program having a long array. Our output will be compressed versions of the java input files which will try optimizing using our array compression techniques.

Implementation

Code

The implementation is divided into two parts. The parser which adds additional code to the array initialization and array references part of the original file which is fed to the parser. The parser produces a new file which is modified after integrating the new array references. The second part is the benefit model which checks the array which is passed to it and decides which compression method to use or just use the original array depending on the nature of the array elements.

The parser code is in files,

- parser.l
- parser.y

The test files for parser are,

- input1
- input2
- input3

The parser generates new code and stores it in shorter file

The benefit model code is in files,

- top.cc
- ArrayModel.cc
- ArrayModel.h

The benefit model can be tested using

- prototype*.cc - * represents any number from 1 to 4. We have 4 prototype files which when run give you an idea of how the benefit model will react to arrays having a particular nature.

Design Document and Flowchart

Flowchart for the Parser

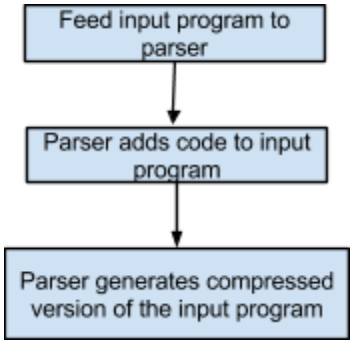

Figure 2 - Flowchart for the Parser

Pseudocode for the Benefit Model

The pseudo code for evaluating the original array if DAC can be used:

```
void evalute(int* array, size_t size) {
  for(size_t i=0; i<size; i++) {
    write(i, array[i]);
  }
}

void write(int idx, int val) {
  if(val >= 0xffff) {
    if the larege array is full {
      declare DAC cannot be used;
    }
    else {
          fill up the array;
      update virtual index to physical index mapping;
      Record bank 0 is used for this entry;
    }
  }
  else {
    if the small array is full {
      declare DAC cannot be used;
    }
    else {
          fill up the array;
      update virtual index to physical index mapping;
      Record bank 1 is used for this entry;
    }
  }
}
```

The pseudocode for evaluating the original array if SDDAC can be used:

```
void evaluateDDAC(int* array, size_t size) {
  for(size_t i=0; i<size; i++) {
    write(i, array[i]);
  }
}

void write(int idx, int val) {
  prev_ = idx==0 ? 0 : prev_;
  int diff = val - prev_;

  if(diff>0xefff) {
    declare DAC cannot be used;
```

```
  }
  else if(diff<0x8000) {
    declare DAC cannot be used;
  }
  else {
    fill up the small array;
  }
  prev_ = val;
}
```

Data Analysis and Discussion

Output Generation

For generating output for our project we were required to source different patterns of data. We wanted to convey the practical application of this implementation and so we used statistical data for out testing. We sourced most of the test data from wikipedia.

Output Analysis

Consider the following test data:

- 1, 32777, 18, 19
  Here, since the delta is much larger than what a short can accommodate, signed DDAC cannot be chosen. Instead DAC is implemented.

- 45,59,51,62,59,54,42,59,49,47,52,63,40,53,61,47,54,58,53,32,61,39,51,37,43,53
  In this data pattern, the delta between consecutive elements can be accommodated in a short and hence signed DDAC is implemented.

- 110256,93879,42026,2929,240,43531,604,321,4228,22589
  Since this data pattern satisfies none of our conditions to use array compression, we use the original memory allocation.

Comparison of Output against Hypothesis

Our hypothesis was to establish reduction in memory bandwidth by means of array compression.
Now consider an example: If we have an array of 8 integers, without compression the array would occupy 8 integers x 4 bytes each i.e. 32 bytes. With our compression scheme, if we allow for 2 elements to overflow into byte size, then we occupy 8 elements of type short x 2 bytes + 2 elements of type integer which is a total of 24 bytes.
This agrees with our hypothesis of reducing memory bandwidth and footprint.

Improvements for the original Paper

Double Array Compression (DAC)

The original DAC proposed reduces the memory bandwidth but increases the total memory footprint. It reduces the memory bandwidth by making copies of the original array. Our proposed new scheme assumes a fixed amount of data that fit into the short data types and the rest fit the original data type.

For example:

Original: int array[8]

index	Value of int
0	0xf
1	0xff
2	0xfff
3	0xffff
4	0xffffff
5	0xffff
6	0xfff
7	0xf

Modified Double Array Compression (DAC)

There are two arrays. One is an array of short and the other is an array of int.

index	Value of short (bank 0)
0	0xf
1	0xff
2	0xfff
3	0xffff
4	0xffff
5	0xfff
6	0xf

index	Value of int (bank 1)
0	0xffffff

It is a design decision to allocate how big each array is. One can allocate the same number of entries for the small array (array of short) and the big array holds up to 20% of the original size. If this predefined assumption is violated, DAC cannot be used. The whole scheme relies on the fact that the array values are constant. Therefore, the original array can be analyzed in order to tell if DAC works.

Since we are using two arrays for DAC and the indexing for the two arrays are independent, we need a bank ID and a mapping table to map the original index to the actual physical index used in each array. Like the tables above shown, the array of short is bank 0 and the array of int is bank 1. Notice that the entry in the original array indexed by 4 is now entry 0 of bank 1.

The mapping table will look like for the input array from above:

Original index	Bank index	Index to the new array
0	0	0
1	0	1
2	0	2
3	0	3
4	1	0
5	0	4
6	0	5
7	0	6

Given the simple example above, assuming we allocate 8 entries in the small array and 2 entries in the big array, the total size is 8*2 + 2*4= 24 byes. Whereas the original array is 4*8=32 bytes. In practical cases, we can use an array of char to shore the values that is smaller or the same size as a byte. Then the over saving will be more impressive.

The pseudo code for evaluating the original array if DAC can be used:

```
void evalute(int* array, size_t size) {
  for(size_t i=0; i<size; i++) {
    write(i, array[i]);
  }
}

void write(int idx, int val) {
```

```
  if(val >= 0xffff) {
    if the larege array is full {
      declare DAC cannot be used;
    }
    else {
          fill up the array;
      update virtual index to physical index mapping;
      Record bank 0 is used for this entry;
    }
  }
  else {
    if the small array is full {
      declare DAC cannot be used;
    }
    else {
          fill up the array;
      update virtual index to physical index mapping;
      Record bank 1 is used for this entry;
    }
  }
}
```

If the above test passes, that means DAC can be be use and the arrays in DAC have been filled up. To extract elements from DAC is very strict forward. The following pseudo code shows how it works:

```
int DAC::read(int idx) {
  int val;

  if(bankIdx[idx]==0) {
    physical-index = index[idx];
    read data from bank 0 with physical-ind;
  }
  else {
    physical-index = index[idx];
    read data from bank 0 with physical-ind;
  }

  return val;
}
```

A new signed-DDAC

The original DDAC placed a limitation on how the data in the array is growing(either increasing or decreasing).

We thought this limitation is unnecessary. Therefore we are proposing a new DDAC scheme that uses a signed value for storing the difference. It has been shown that the new scheme works for some preliminary test patterns. More testing need to be done to firmly validate the scheme.

The new signed-DDAC scheme reduces the overall memory footprint by only storing the delta. Assuming the we are using short to store the delta of an int array, we practically reduce the memory footprint by half.

Similar to the limitation for DAC, the signed-DDAC also has its limitation. That is when the delta is wider then the array element size. Therefore, once the values of the array are know, we can evaluate if the signed-DDAC works. The following pseudocode shows the general steps.

The pseudocode for evaluating the original array if SDDAC can be used:

```
void evaluateDDAC(int* array, size_t size) {
  for(size_t i=0; i<size; i++) {
    write(i, array[i]);
  }
}

void write(int idx, int val) {
  prev_ = idx==0 ? 0 : prev_;
  int diff = val - prev_;

  if(diff>0xefff) {
    declare DAC cannot be used;
  }
  else if(diff<0x8000) {
    declare DAC cannot be used;
  }
  else {
    fill up the small array;
  }
  prev_ = val;
}
```

Read access data from the signed-DDAC is simple. The following pseudo code shows that. Since the scheme assumes sequential access(read start from the first element), we need state variable called prev to record the previous decoded values. And it is initialized to 0 when the first element is read.

```
int SDDAC::read(int idx) {
  prev = idx==0 ? 0 : prev;

  int val = smallArray[idx];

  val += prev;
```

```
  prev_ = val;

  return val;
}
```

Conclusions and Recommendations

Summary and Conclusions

We successfully implemented the parser and the benefit model for the project. We proposed improvements to the original paper. The benefit model implemented by us is different from the one presented in the paper. One of the drawbacks of the paper was that it increased the footprint by a great extent. In our project, we have also tried reducing the memory footprint by changing the benefit model.

Recommendations for Future Studies

Our future scope includes linking the parser and the benefit model to make it a whole project. We also propose to improve the benefit model for random access of the array elements. At the moment, the benefit model only works for sequential access of array elements. We could improve that by letting the benefit model know before hand whether there are any random accesses to the array which is to be compressed.

Bibliography

- A Compiler-automated Array Compression Scheme for Optimizing Memory Intensive Programs by Lixia Liu and Zhiyuan Li, Purdue University, West Lafayette, IN 47907, ACM
- Using Data Compression for Increasing Memory System Utilization , Ozcan Ozturk, Member, IEEE, Mahmut Kandemir, Member, IEEE, and Mary Jane Irwin, Fellow, IEEE
- Compiler Optimization on Instruction Scheduling for Low Power, Chingren Lee, Jenq Kuen Lee, TingTing Hwang, Dept. of Computer Science, National Tsing-Hua University, Dept. of Information Management, National Chi-Nan University, IEEE
- Reducing Memory Requirements of Resource-Constrained Applications, P. UNNIKRISHNAN, IBM Toronto Lab, G. CHEN, Microsoft Corporation, M. KANDEMIR, The Pennsylvania State University, M. KARAKOY, Imperial College and I. KOLCU, University of Manchester, ACM

Appendices

Program Flowchart

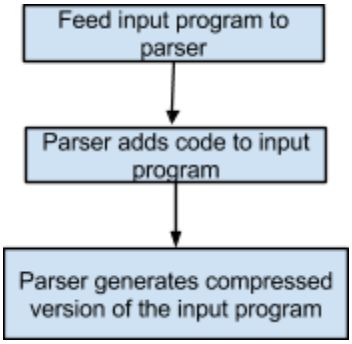

Figure 2 - Flowchart for the Parser

Program Source Code with Documentation

```
#include <assert.h>
#include "ArrayModel.h"

//#define DEBUG

void
DAC::fill(int* array, size_t size) {
  for(size_t i=0; i<size; i++) {
    write(i, array[i]);
  }
}

void
DAC::write(int idx, int val) {
  if(val >= kLimit) {
    if(intMem_->full()) {
      arrayNotFit_ = arrayNotFit_ | true;
    }
    size_t phyIdx = intMem_->idx();
    bankIdx_[idx] = 0;
    phyIdx_[idx] = phyIdx;
    intMem_->element(idx, val);
    printf("bank=0\n");
  }
  else {
    if(shortMem_->full()) {
```

```
      arrayNotFit_ = arrayNotFit_ | true;
    }
    size_t phyIdx = shortMem_->idx();
    bankIdx_[idx] = 1;
    phyIdx_[idx] = phyIdx;
    shortMem_->element(idx, val);
    printf("bank=1\n");
  }
}

int
DAC::read(int idx) {
  int val;

  if(bankIdx_[idx]==0) {
    val = intMem_->element(idx);
    printf("bank=0\n");
  }
  else {
    val = shortMem_->element(idx);
    printf("bank=1\n");
  }

  return val;
}

void
DDAC::fill(int* array, size_t size) {
  assert(size <= org_.size());
  assert(size <= dup_.size());
  for(size_t i=0; i<size; i++) {
    write(i, array[i]);
  }
}

void
DDAC::write(int idx, int val) {
  prev_ = idx==0 ? 0 : prev_;
  int diff = val - prev_;

  if(diff >= kLimit) {
    dup_[idx] = kLimit;
    org_[idx] = val;
    arrayNotFit_ = arrayNotFit_ | true;
  }
  else {
    dup_[idx] = val - prev_;
  }
```

```
  prev_ = val;
}

int
DDAC::read(int idx) {
  prev_ = idx==0 ? 0 : prev_;
  int t = dup_[idx];

  if(t==kLimit) {
    t = org_[idx];
  }
  else {
    t += prev_;
  }
  prev_ = t;

  return t;
}

void
SDDAC::fill(int* array, size_t size) {
  assert(size <= org_.size());
  assert(size <= dup_.size());
  for(size_t i=0; i<size; i++) {
    write(i, array[i]);
  }
}

void
SDDAC::write(int idx, int val) {
  prev_ = idx==0 ? 0 : prev_;
  int diff = val - prev_;
#ifdef DEBUG
  printf("%s: prev=%d(0x%x) idx=%d val=%d (0x%x) => diff=%d (0x%x)\n",__func__,
    prev_,prev_,idx,val,val,diff,diff);
#endif
  if(diff>=kPosLimit) {
    printf("Over\n");
    dup_[idx] = kPosLimit;
    org_[idx] = val;
    arrayNotFit_ = arrayNotFit_ | true;
  }
  else if(diff<=kNegLimit) {
    dup_[idx] = kNegLimit & 0xffff;
    org_[idx] = val;
    printf("Under: org[%d]=%d\n", idx, org_[idx]);
    arrayNotFit_ = arrayNotFit_ | true;
  }
```

```
  else {
    printf("In Range: diff=%d\n", diff);
    dup_[idx] = diff;
  }
  prev_ = val;
  printf("\n");
}

int
SDDAC::read(int idx) {
  prev_ = idx==0 ? 0 : prev_;
  int val = dup_[idx];
#ifdef DEBUG
  printf("%s: prev=%d idx=%d val=%d (0x%x)\n",__func__,
    prev_,idx,val,val);
#endif
  if(val==kPosLimit || val==(kNegLimit&0xffff)) {
    val = org_[idx];
    printf("get val from org[%d]=%d\n",  idx, org_[idx]);
  }
  else {
    val += prev_;
  }
  prev_ = val;
  printf("\n");
  return val;
}
```