Feedback Compilation for Decoupled Access-Execute Techniques

Wenting Jin

Institutionen för informationsteknologi
Department of Information Technology
Abstract

Feedback Compilation for Decoupled Access-Execute Techniques

Wenting Jin

Software level optimization for compilers has become a major research field. Dynamic Voltage Frequency Scaling (DVFS) technology gives some options for optimization, such as tweaking the voltage or frequency. Improvements in voltage are quickly reaching their limits and hence this thesis work investigates approaches that are focused on dynamic frequency scaling. Finding an optimum with respect to two objectives: performance and energy, has been explored for Decoupled-Access-Execute(DAE) technology with iterative feedback compilations. The goal is to find a close-to-optimum output with a multi-versioned procedure within a limited budget and examine how well and on what aspect DAE can make improvements.

This thesis work focuses on automating the compilation with DAE, including program instruction analysis for automatic selection of initial search parameters.
Acknowledgements

I would like to thank my supervisor, Kim-Anh Tran, and reviewer, Alexandra Jimborean, who have been guiding me throughout the whole period of my thesis work, with their kind help, enormous patience, and professional engagement.

I would also like to give special thanks to my family and friends who have supported me with their expertise on various cases that have mattered greatly for my life.
Contents

Figures & Plots 5

Tables 5

1 Introduction 6
   DAE 6
   Two Phases 6
   Loop Chunking 6
   Decoupled Approach 7
   Multi-Versioned DAE 7

2 Background 8
   Hurdle 8
   Optimum 8
   Energy 8
   Feedback Assisted Iterative Compilation 8
   Evaluating Iterative Compilation 9
   Orchestration Strategies 9
   Multi-Objective Exploration 10

3 Implementation 12
   Overview of the Algorithm 12
   Important Parameters & Input 13
   Orchestration Flow 13
      Step1: Initialization 13
      Step2: Multi-Versioning 14
      Step3: Comparison 15
   Coordinator 17
   Transform & Profile 17
   Strategy 19
   Load Counting & Initial Granularity 21

4 Evaluation 23
   Environment 23
   SPEC CPU2006 23
   Settings 24
   Result Analysis 25
   Discussions 26

5 Conclusions and Future Work 29

6 References 31

Appendices 32

A Example Input File 32

B ReadMe: Feedback Compilation 33
   Prerequisites 33
      DAE 33
      Python 33
      Script 33
      Repository Structure 33
   Getting Started 33
      Run 34
      Result Files 34
Algorithms

1  Feedback Iterative Compilation Flow ........................................ 15
2  Transform & Profile Module .................................................. 18
3  Strategy Module .............................................................. 20
4  Initial Granularity ............................................................. 22
Figures & Plots

1. Overview .................................................. 12
2. Design Modules ............................................. 16
3. 2D Grid Example ......................................... 19
4. 429.mcl .................................................. 35
5. 429.mcl.AllWeights ....................................... 36
6. 429.mcl.Extreme .......................................... 36
7. 433.milc .................................................. 37
8. 433.milc.AllWeights ..................................... 38
9. 433.milc.Extreme ......................................... 38
10. 445.gobmk ............................................... 39
11. 445.gobmk.AllWeights ................................ 40
12. 445.gobmk.Extreme ...................................... 40
13. 450.soplex ............................................... 41
14. 450.soplex.AllWeights .................................. 42
15. 450.soplex.Extreme ...................................... 42
16. 456.hmmer ............................................... 43
17. 456.hmmer.AllWeights .................................. 44
18. 456.hmmer.Extreme ...................................... 44
19. 458.sjeng ................................................. 45
20. 458.sjeng.AllWeights ................................... 46
21. 458.sjeng.Extreme ...................................... 46
22. 462.libquantum ......................................... 47
23. 462.libquantum.AllWeights ............................. 48
24. 462.libquantum.Extreme ................................ 48
25. 470.lbm .................................................. 49
26. 470.lbm.AllWeights ..................................... 50
27. 470.lbm.Extreme ........................................ 50
28. 482.sphinx3 ............................................. 51
29. 482.sphinx3.AllWeights ................................. 52
30. 482.sphinx3.Extreme .................................... 52

Tables

1. Platform .................................................. 23
2. Brief on Benchmarks ................................. 24
3. 429.mcl Compare Table .................................. 35
4. 433.milc Compare Table .................................. 37
5. 445.gobmk Compare Table ............................... 39
6. 450.soplex Compare Table .............................. 41
7. 456.hmmer Compare Table ............................... 43
8. 458.sjeng Compare Table ................................ 45
9. 462.libquantum Compare Table ....................... 47
10. 470.lbm Compare Table .................................. 49
11. 482.sphinx3 Compare Table ............................ 51
1 Introduction

Traditionally CPU:s have been able to lower their power consumption through methods like Dynamic Voltage Frequency Scaling (DVFS), these methods have proven less and less feasible on emerging hardware. To address this shortcoming new compiler techniques have emerged, one such technology is Decoupled Access-Execution (DAE), it finds software level optimizations to save energy while not sacrificing performance. This technique works by re-organizing similar parts of the software into different phases where lower CPU clocks can be efficiently employed depending on the type of the phase.

Previous work within DAE has been heavily dependent on manually tuning parameters to achieve improved efficiency and performance, this work focuses on employing multi-objective optimization to find an optimal configuration within a limited search-time budget.

DAE

DAE is a compiler approach that uses the DVFS technique to reduce the energy expenditure of applications. Hot loops that are identified to be critical on the execution path are the targets for DAE. Such loops can be identified by offline static analysis, as well as annotations that are manually inserted by the programmer in source code.

Two Phases

Code regions for such loops are grained coarsely into two phases: Access and Execute.

Frequency can be scaled down for Access phase, and this region is memory-bound since the processor is mostly waiting for data to be fetched, so the data needed for Execute phase will be residing in the cache ready for computation. The amount of data that is being prefetched is dependent on cache size, so maximizing the amount of data being prefetched can be considered better within the scope of cache size. Data being prefetched may include addresses indicating the location of data. Such multiple data prefetching that may only happen with a certain restricted order in depth, is called the level of Indirection.

Frequency can be scaled up again for the Execute phase once data is ready and closely located in the cache, as this region is compute-bound (a computational problem is considered compute-bound when the processor elements are spending a majority of their time performing computations and not waiting for data to arrive.)

Loop Chunking

Each targeted loop will be split into a suitable number of chunks. In each chunk, there is an Access phase followed by an Execute phase, where each phase contains consecutive iterations.

The size of each chunk is called Granularity which determines how much data is prefetched during the Access phase. An optimal chunk size is supposed to be a best size with respect to the utilization of cache, namely the ratio between cache size and prefetched data size.
Decoupled Approach

Studies [2] using Decoupled Access-Execute (DAE) have shown much potential in improving energy efficiency without performance loss within the research field of compiler and computer architecture [5]. Load instructions take more CPU clock cycles compared to other type of instructions, when the load misses in all cache levels leading to data being fetched from DRAM. The compiler clusters such loads in so called Access phases and the rest of the computation in Execute phases. Creating memory-bound Access phases and compute-bound Execute phases enables high performance at lower-energy expenditure, by adjusting the core's frequency per phase.

Multi-Versioned DAE

A more detailed description and approach on DAE was presented in [6], where multi-versioned decoupled access-execute were experimented with by selecting different indirection levels and loop granularity.

What is still lacking is an automated procedure for selection of granularity and indirection. This paper therefore focuses on the automation part for compilation with DAE, and the output is expected to be a close-to-optimum result from software multi-versioned decoupled access-execute, as well as the auto selection of an indirection level and a loop granularity for such as at find a close-to-optimum solution.

Brief Note on Feedback Compilation with DAE

The feedback compilation scheme in this paper is useful when high-performance and application runtime is the main focus and compilation time is not that relevant. This scheme does compilations and runs the entire application multiple times until version selection is complete. This technique brings additional benefit if there are also requirements on energy efficiency for the application. It is not suitable for applications that only need to provide the answer once, since the compilation time will be much longer than the runtime for usage.

For applications that run multiple times with different inputs, the performance can be quite consistent if an application has low level of indirections as cache misses caused by different input data won’t influence as much as for applications with much higher levels of indirection. The sensitivity to input data can likely increase with higher levels of indirection.
2 Background

Hurdle

Rapid architectural change puts pressure on traditional compiler technology which is based on static analysis and rough estimation on machine models, therefore software level optimizations for compilers is becoming a major research focus.

This is due to the fact that the compiler cannot always statically predict the behavior of the code on a particular hardware. This can either be due to the execution context (problem input, co-executing applications, hardware resource sharing, etc.) or simply because it is difficult to statically estimate the costs of some operations (e.g. memory accesses).

The optimization problem is further complicated by not only because of the ever more complex growing hardware changes, but also because the optimization space is highly non-linear with many local optima.

This work takes a more dynamic approach by leveraging feedback from runtime to identify an optimum that can deliver “good-enough” results.

Optimum

A so called "optimum" can be referred to many different objectives or even a combination of them, to mention a few such as compilation time, runtime, code size, power consumption, etc. where:

- **compilation time** means the time spent to compile and link.
- **runtime** means the time spent to execute.
- **code size** refers to the amount of memory distributed to hold the compiled machine code.
- **power consumption** refers to the amount of energy over time (e.g. Joule per second) that is needed to operate on some application.

Energy

Dynamic Frequency Voltage Scaling (DVFS) is one technique that allows voltage and/or frequency to be scaled down (and up) during the execution to achieve a more energy efficient power consumption.

More detailed dynamic power consumption can be expressed as: \( p \sim a \times C \times V^2 \times f \), where \( a \) is the switching activity factor, \( C \) is the capacitance being switched per clock cycle, \( V \) is the supply voltage, and \( f \) is the switching frequency [10].

The correlation between runtime and energy can be described by Energy-Delay-Product (EDP).

\[
EDP = \text{Total Execution Time} \times \text{Total Energy}
\]

This is used to measure efficiency that accounts both for performance and energy.

Feedback Assisted Iterative Compilation

An investigation on compilation using solely profiling information regardless of static information (program analysis, machine models) was described in [7]. The idea
is to ease some pressure off of the compiler so compilation strategies can adapt to architectural change.

Its overall compilation system relies on a collection of the following components:

- Co-ordinator: compiles and runs the transformed program on a target machine, and then collects profile information.
- Strategy Module: in charge of exploration and tells Co-ordinator what the compiler should do based on collected profile information from Co-ordinator.
- Transformation: serves as how to do, this module takes transformation commands from Co-ordinator and returns transformed code.

Transformation oriented and cost oriented are two different basic strategies. With the transformation oriented strategy, a compilation budget (predefined time span) is split into two parts where the first part is used to evaluate each single transformations and each score (worth) is recorded, then the second part of the budget is used to evaluate some combined transformations that are selected by their worth ranking. The final code is given when the budget runs out. This strategy spreads its budget evenly across the whole program. Whereas a cost oriented strategy targets the most-time consuming areas in a program with a custom-defined threshold, so code sections that reach the threshold level become of interest for optimization, and the time consumption information is retrieved from profiling.

Three iterative strategies were presented in [7], and they differ in order & selection & combination of transformations among array padding, loop tiling, loop unrolling.

The results were obtained from using all three strategies on four different platforms and proved that iterative compilation is a viable approach to program optimization, but this requires long compilation runs and a program’s performance to be crucial.

Evaluating Iterative Compilation

It is important to evaluate developed strategies on different platforms with different applications that are featured divergently and with different sets of input data for the same application. A follow-up study [3] from the same authors from above paper [7] is about evaluation on previously implemented strategies, which discussed cross platform performance for all three strategies with 3 full SPEC95 benchmarks using different data sets (training data and reference data), showing the strategies developed were more efficient, with significant reduction in execution time, than using solely full optimizations from native compiler. Conclusively, it proved to be a promising approach towards a more blind generic optimization with feedback directed iterative compilation.

Orchestration Strategies

There can exist more complicated algorithms which are built from simple ones, to orchestrate a series of compilations that keep only certain information of interest as feedback throughout an iterative process [8].

- Exhaustive Search (ES): compile for all the possible combinations from all options, which guarantees a full search space, but this approach is very expensive.
- Batch Eliminations (BE): use one compilation result that applied all possible options as a baseline, then compile with only one option applied and measure improvement against the baseline, for each individual option. This makes it
possible to identify the individual options with negative effects and turn them off all at once, but this approach can not deal with correlating sets of options.

- **Iterative Eliminations (IE):** start with one baseline that all options applied, then compile individually for each option through all possible options, find the option with the most negative effect. Compile a new baseline by removal of the option with the most negative effect, then re-compile for each remaining option. Next it continues such an iterative elimination process until no option with negative effects remains. After which it is then possible to consider the combined positive effect of more than one option, but this process takes longer time compared to BE.

- **Combined Elimination (CE):** starts with the same initial full set of options applied to a baseline, it first runs IE compile and execute to find a list of negative effecting options and update the baseline by removal of the option with the greatest negative effect, it then performs BE for each remaining option with negative effect from the previous list, these are compiled and profiled individually based on the baseline. If options have negative effect they are eliminated. This approach benefits from the iterative process of IE because of it being more careful on the interaction of different options, it also benefits from the greedy approach of BE as it probes and reduces options fast at each iteration.

Clearly the limitation in [8] is the choice of binary only options that are explored, these parameterized options can not be dealt with easily only by switching on and off.

**Multi-Objective Exploration**

The compiler optimization chosen for a specific objective for one program might have conflicting effect if the objective is changed to be another for the same program. Due to the facts of optimization interactions, it is not trivial to find an optimization for a combined aspect involving multiple objectives so that each of them holds a proportional weight, such objectives are application performance, code size, energy consumption, and compilation time.

In paper [4], each objective function is a metric of interest which is presented as one dimension in a Cartesian plane. A result position that gets closer to the Origin means a better optimization. Among each choice of optimization that produces a better result for one objective while not worsening for any other objectives, the best such optimization choices from all objectives form an optimal set of optimizations that each of them varies in different weights for individual objectives. Such an optimal set is called a *Pareto Frontier* (PF).

In the meanwhile, the extended lines from all the worst positions scored by each objective come across at a point that shapes a biggest volume towards the Origin. Such a point is called a *Hyper Volume* (HV) point.

The space between PF and HV becomes the exploration space for optimization. Exploration results can therefore be evaluated through the relative reduction from the HV point. Any strategy that manages to get closer towards the Pareto Frontier can be seen as a better exploration result.

The iterative compilation strategy used in this paper [4] was a genetic algorithm which iterate through different generations that are composed of different populations such that each population holds a certain number of entities. Between each consecutive two generations, entities of interest are archived for each population from the earlier generation, then the new generation is created through
different methods such as migration, mutation and crossover with certain degrees of randomness.

Though some optimal result can be found quite early, this does not necessarily mean a guaranteed time for knowing such an optimality. Only when a pre-set budget has run out can a relative best be defined.
3 Implementation

There are three different target settings that can be specified for compilations, and modifications can be done in Makefile.targets under common settings for compiling with DAE. The possible targets are: ORIGINAL, CAE, and DAE, where

- **ORIGINAL** is simply the original executable binary that will be compiled and generated without any transformations.
- **CAE** on contrast to DAE is featured as Coupled Access-Execute so only loop chunking will be applied for each pair of access-execute without data prefetching.

The whole orchestration (see Algorithm [1]) of feedback iterative compilations is designed with a composition of three modules: Coordinator, Transform & Profile, and Strategy.

- **Coordinator** guides through the process of automated compilations for multi-versioned DAE by passing compilation parameters so a next version can be tested.
- **Transform & Profile** takes charge of executing how the binaries will be compiled.
- **Strategy** decides what to do in searching for a close-to-optimum result, so the agenda of how to explore the search space will be taken care of.

The design pattern of separating different modules is intended to make it possible for future development due to changes in hardware and decision-making without having to re-write the entire orchestration scheme.

The goal is to experiment through multiple compilations with variations on indirection and granularity so that a best DAE combination can be found for an application with its specific set of objectives. This new optimal DAE executable (pseudo best output in Figure [2]) will then be compared with the ORIGINAL version of the same application to gain more insight into the potential advantages and disadvantages of the DAE technique for future work.

**Overview of the Algorithm**

The Algorithm can briefly be divided into three steps:

1. Initialization: generation of setup configurations (metrics, search space, application and file parameters etc.)
3. Comparison: gaining insight into the potential optimality of the version.

![Figure 1: Overview](image-url)
Important Parameters & Input

In order to start an automated procedure, several important parameters need to be specified along with a specific application so that multiple compilations on this specified application will be guided through as desired.

- **objective & weight**: a maximum of two objectives are supported, default is runtime as the first objective and data prefetch rate \(^1\) as the second. Therefore a distributed weight can be set between these two objectives which shall sum up to be 1.0.

- **budget**: as the compilation time can vary greatly among different applications, a discrete budget is used to specify the total number of compilations on target DAE (with an assumption that the user is aware of single compilation time). In the case of a budget lower than the number of indirections of an application, the minimum number of compilations of target DAE will be the maximum indirections found for the application.

- **threshold**: throughout the feedback iterative compilations, there is need for a standard to be compared with as baseline. With the purpose of an adaptive standard that can be used throughout the progressing procedure, the threshold is used to renew such a standard so the baseline will be updated once the improvement has meet the threshold (\(Epsilon\) in Figure 2).

- **granShift**: compilations start with a pre-computed initial granularity (see section Load Counting & Initial Granularity) that is far from obtaining a full image of the search space, granShift is used to conservatively specify the number of bits to shift right, so the space of exploration gets enlarged with a smaller starting granularity. With a non-positive value, there will be no shift on the initial granularity.

Together with above listed information, other information such as the inputs used to run the application, the location of the installed compiler that is featured with DAE, etc. must be specified before running this automated system for any specific applications. A file in json format specifying all such detailed parameters will be used as the input to start feedback iterative compilations(see Appendices A & B).

Orchestration Flow

**Step1: Initialization**

To be able to know the entire search space for generating a DAE executable for an application, it is necessary to gain some information on the suitable range of Loop Chunks (granularity) and the maximal level of indirection.

- Generate and profile ORIGINAL executable.

- Generate a CAE executable with \(maxint^2\) where the loop chunking(granularity) is set to the maximal to get the highest possible amount of cache misses and hence result in a longer runtime.
  - runtime result from running it will be used as denominator for normalizing runtime objective.
  - the "*gran$(maxint).extract.ll" file from generating CAE executable is tracked which will be used for computing initial granularity.

\(^1\)Ratio indicating the actual prefetched data among all data that can be prefetched through the deepest indirection level, which can be considered for energy efficiency, the higher the better.

\(^2\)https://docs.python.org/2.7/library/sys.html#sys.maxint
• Granularity
The range of granularity is defined by a starting granularity and an ending granularity. The size of the L1 data cache of one given machine sets a common limit to be the maximal for the ending granularity for all applications running on it. However the starting granularity varies from application to application with the fact that the usage of memory and data access behavior can be rather different. Therefore it is necessary to find a theoretical starting granularity - initial granularity.

- if the initial granularity is not computed yet, then compute the initial granularity (see Initial Granularity & Load Counting section) and store it into the "basic info" file.
- else, read "initial granularity" value from "basic info" file.

• Maximum Indirection
The DAE compiler has one important pass: "FKernelPrefetch", which gives information indicating reasons why loads have not been hoisted with status ("Bad", "Red" and "Indir") for a targeted kernel with a given indirection, where "Bad" indicates corrupted data fetch, "Red" indicates redundant data fetch, and "Indir" indicates further unreached level(s) of indirection(s). Starting transformation with a minimal indirection(0) and incrementing by one at a time, we can get knowledge on the maximum existing level of indirection for a targeted kernel when the first time "Indir" shows a zero from observing this kernel's "FKernelPrefetch" status.

It is not difficult to realize that one application has reached to its maximal level of indirection the first time when all its targeted kernels have "Indir" = 0 (equivalent to the sum of all "Indir"(s) is zero).

- if Maximum Indirection not computed yet, set Indirection = 0
  1. Compile with Initial Granularity and Indirection, profile the executable.
  2. Store profiling result into log file.
  3. parse "FKernelPrefetch" text throughout compilation log file and get all corresponding "Indir" text that is within the same line as "FKernelPrefetch" text, get "Indir" value(s).
  4. * if the summation of all "Indir" value(s) is not zero, increment Indirection by 1 and go to step 1.
     * else (the first time all zero(s)), the indirection that is currently used for transformation is found to be the value for Maximum Indirection. Store into "basic info" file.
- else, read "Maximum Indirection" value from "basic info" file.

Step2: Multi-Versioning

Finding best DAE with iterative compilation:

1. pass inputs to Strategy and get a response

2. • if "stop" is not Yes, call Transform & Profile to generate a new DAE executable with new indirection & granularity from Strategy's response and get profiling result which will be fed back to Strategy. Go to step 1.
• if "stop" is Yes, a best DAE combination has been found. Store it into strategy log file.
Step 3: Comparison

Fair comparison: improvement rate with consideration of performance and (or) energy by comparing best found DAE executable and the ORIGINAL version will be stored in log file.

Algorithm 1: Feedback Iterative Compilation Flow for finding the best combination of indirection & granularity for an application with one specific weight setting

1. Generate and profile ORIGINAL executable;
2. Generate a CAE executable with maxint;
3. if initial granularity not found in "basic info" file then
   4. Compute initial granularity with a list of "*gran$(maxint).extract.ll" files;
   else
      6. Read initial granularity from "basic info" file;
   end
8. if Maximum indirection not found in "basic info" file then
   9. indirection=0;
   10. granularity=initialGranularity;
   11. Transform and profile for a DAE executable with granularity & indirection;
   12. Store profiling result into log file;
   13. Parse "FKernelPrefetch" from compilation log file and get all corresponding value of "Indir";
   while sum(all "Indir" value) > 0 do
      14. indirection +=1;
      15. Transform and profile for a DAE executable with granularity & indirection;
      16. Parse "FKernelPrefetch" from compilation log file and get all corresponding value of "Indir";
      17. Store profiling result into log file;
      18. end
   20. maxIndirection=indirection;
   else
      22. Read initial granularity from "basic info" file;
   end
24. Call Strategy with inputs and get response as [stopCompile, variables];
25. [newIndir, newGran] = variables;
26. while stopCompile==No do
   27. indirection=newIndir;
   28. granularity=newGran;
   29. Transform & Profile for a new DAE executable;
   30. Call Strategy with inputs and get response as [stopCompile, variables];
31. [newIndir, newGran] = variables;
32. end
33. Re-compile the DAE compiler without statistics;
34. Transform & Profile for the best found DAE (without statistics);
35. Compute improvement for ORIGINAL and store results into log file;

There is statistical information gathered when compiling DAE executables for the purpose of knowing time and percentages utilized by two phases (compute and prefetch). This information gathering however injects more time to run DAE executables. From a perspective of simplicity during the implementation, modification on the DAE compiler tool was chosen by re-compile of the compiler with disabled command codes for statistics.
Figure 2: Design Modules
Coordinator

The core module Coordinator serves, as its name implies, in coordinating and ensuring that the other modules can accomplish their respective tasks, hence all compilations are managed by the Coordinator, as shown in Algorithm 1.

- Interacts with: Transform&Profile, Strategy modules.
- Responsibility:
  - determine an initial granularity and a maximum granularity so that utilization of the L1 data cache dynamically adapts to the current application with respect to its data load size in the hot path.
  - determine the maximum indirection level of the current application.
  - store computed initial granularity, maximum granularity and maximum indirection into a json formatted “basic info” file.
  - prepare the first column data based on the maximum indirection to be sent to the strategy module.
  - with Transform&Profile: Coordinator passes relevant information to Transform&Profile such as target, indirection, granularity and application, to generate an executable binary file and to get a time record from profiling/running the executable. The time record will be used to compute a score by Coordinator, and both time record and score will be saved to a log file.
  - with Strategy: Coordinator passes relevant information to Strategy such as maximum indirection, initial granularity, maximum granularity, current indirection, current granularity, current profiling result, budget, threshold, weight log file and data type. So Strategy will give a response to Coordinator with a decision that to either stop compiling or to continue with new values for indirection and granularity. In the case of a stop, the Coordinator will halt the iterative compilation procedure and give an output regarded as optimal for DAE.

Transform & Profile

This module is in charge of the interaction between an application and the hardware that the application will run on. Transform & Profile serves as it is named, it takes care of transformation commands to compile and generate executable files as well as profile runs the generated executable files to get timing records, as shown in Algorithm 2.

- Interacts with: Coordinator module.
- Responsibility:
  - receives relevant parameters such as target, indirection, granularity and application from Coordinator to compile for a specific executable binary file and return a generated file name to Coordinator module.
  - receives relevant parameters such as repeat, input args and application’s executable file name from Coordinator, to profile/run the application with its input args repeatedly. The recorded timing results will be returned to Coordinator.

4Performance counter on “cycles” and “instructions” are the options used to gather profiling information, although the statistics result currently is not practically used.
Algorithm 2: Transform & Profile Module

1 function Transform

\[(target\_file, target\_path, indir, gran, transTarget, appName, app\_path, exe\_path)\];

**Input:**
- `target_file` – refers to `Makefile.targets` which assists to compile for DAE executables
- `target_path` – the location of `target_file`
- `indir` – indirection to be used for compilation
- `gran` – granularity to be used for compilation
- `transTarget` – one choice of transformation target among `ORIGINAL`, `CAE` and `DAE`
- `appName` – the name of the application repository which is the entrance to application source code
- `app_path` – the path to `appName`
- `exe_path` – the desired location to dump all generated executable files

Application repository shall have a sub-directory named as `src` and one `Makefile` for the application inside `src`.

Application repository shall reside under `sources` in DAE compiler repository.

**Output:** a list that contains an executable file name

2 Change target variable setting in `Makefile.targets` to be `transTarget`.
3 Run `Make -C app\_path`.
4 Return executable file name;

5 function Profile \((exeFile, exeArg, exePath, repeat)\);

**Input:**
- `exeFile` – the executable file name
- `exeArg` – the parameters/input for running the executable
- `exePath` – the location of the generated executable file
- `repeat` – the number of repeated runs to profile the executable

**Output:** a list that contains a float which is an average runtime from the executable

6 Profile with tools (`taskset` and `perf`) on the inputs.
7 Return runtime of the executable;
8
9 ;
**Strategy**

DAE features *indirection* and *granularity*, the space for searching an optimum for these two variables will create a 2D grid (see Figure 3). Considering the limited capacity of the L1 data cache, the actual representation of search records can make use of a fixed size matrix. Once the first column with the minimal granularity has been filled for all possible indirections, strategy module is called. The search pattern is influenced by a decision making module that uses a 1-directional search pattern at its core. Further changes/implementations of a new strategy will require updates on this decision making module (Algorithm 3).

- Interacts with: Coordinator module.

- Responsibility:
  - receives relevant parameters such as maximum indirection, initial granularity, maximum granularity, current indirection, current granularity, current time record, budget, threshold, weight log file and data type.
  - compute/adjust total number of actual compilations. The default number of compilations might be higher than the budget if the budget is less than the maximum indirection.
  - 1-directional search: Once the best indirection in the first column is determined (with the same starting granularity, the indirection level that gives the highest score computed with considerations of weighted objectives), the entire row will be explored. With adequate budget, more rows will be explored. Selection on rows among the first column is simply based on descending order of the scores. Searching will be terminated once the budget runs out or the entire space has been explored. Searching in this way makes it possible to examine DAE technique on finding the optimal indirection level with respect to energy efficiency in contrast to performance.
  - baseline & decision: For each search step, Strategy updates baseline in case the improvement has reached the threshold, and returns a signal of either Continue or Stop for compilation, new indirection and new granularity will be returned together with Continue.

An experimental search pattern is implemented which is 2-directional. For non-completed rows, the row with an indirection of a best score selected from the rightmost fully-filled column will be explored rightwards until the end of the row. This procedure repeats until the entire table is filled under an adequate budget. Evaluation of this 2-directional search pattern will be left out for future work.

<table>
<thead>
<tr>
<th>Granularity →</th>
<th>0,4</th>
<th>0,8</th>
<th>0,16</th>
<th>0,32</th>
<th>0,64</th>
<th>0,128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indirection ↓</td>
<td>1,4</td>
<td>1,8</td>
<td>1,16</td>
<td>1,32</td>
<td>1,64</td>
<td>1,128</td>
</tr>
<tr>
<td></td>
<td>2,4</td>
<td>2,8</td>
<td>2,16</td>
<td>2,32</td>
<td>2,64</td>
<td>2,128</td>
</tr>
</tbody>
</table>

Figure 3: 2D Grid Example: each cell represents a 2-tuple (*indirection, granularity*). Top-left cell indicates zero-prefetch and initial granularity of 4.
Algorithm 3: Strategy Module

function Strategy (maxInd, initGran, maxGran, weight, threshold, budget, 
proRes, curInd, curGran, dumpFile, dataType)

Input :
- maxInd – maximum indirection found
- initGran – initial granularity found
- maxGran – maximum granularity found
- weight – weights for 2 objectives
- threshold – criteria for updating the current baseline
- budget – one criteria for stopping the iterative compilation
- proRes – contains runtime and prefetch rate obtained from profiling current executable compiled with curInd & curGran
- curInd – current indirection
- curGran – current granularity
- dumpFile – log file that stores all information for an application and used by strategy module
- dataType* – if dataType is not within predefined group, search pattern change to be 2-directional

Output: {"stopCompile":decisionStop, "var":[indirection, granularity]}

1 Compute/adjust total number of actual compilations based on the given budget;
2 Load log file to get number of finished compilations;
3 if the entire first Column of the search space has been explored for the first time
   then
   Store a first baseline by selecting the best scored (indirection, granularity) combination;
   else
   Get the current baseline;
   Store profiling result for the current DAE executable and update the log file.;
4 end
5 Update baseline if current profiling results gives a better score whose improvement is larger than threshold;
6 Budget control: check if the given budget has run out;
7 Granularity control: check if current granularity is larger than maximum granularity (this is only error state check);
8 decisionStop=Yes, if either budget or granularity control is True;
9 if dataType is among "test", "train" and "ref" then
   (1-directional search pattern);
   Get all row numbers that is not completely filled;
   If there is incomplete row(s), get a best scored indirection among the scores from the first column with incomplete row(indirection) numbers;
   if all rows are completed then
   decisionStop=Yes
   else if just finished a row where granularity=maximum granularity then
   Get a best scored indirection among the scores from the first column with incomplete row(indirection) numbers;
   else
   newIndirection=current indirection;
   newGranularity=current granularity * 2;
   end
10 else
11 (2-directional search pattern);
12 //NOT in the scope of this thesis work, detailed description therefore skipped;
13 end
14 Return{"stopCompile":decisionStop, "var":[newIndirection, newGranularity]};
Load Counting & Initial Granularity

The DAE compiler tool takes one indirection value and one granularity value for a whole application, therefore only one granularity value can be chosen for compilation and there is currently no choice of granularity per kernel.

To be able to quantify the number of load instructions and approximate size of these loads for an application, a CountLoad LLVM pass has been implemented, which focuses on getting the size of loads in hot loops that were identified statistically. The CountLoad pass works as follows:

- Under the scope of an application, the pass checks each function, and identifies if a function has the suffix that of a marked kernel
- If the function is identified as a marked kernel:
  - Iterates over all the blocks in the marked kernel
  - Iterates over all the instructions in a block
  - All instructions are checked, if an instruction is a load instruction, the size for this type of load is computed in the total number of bytes
  - All the acquired load sizes under a marked kernel get summed together
- Stores load size in byte for each targeted kernel.

Since this CountLoad pass checks through every instruction under all targeted kernels, it can likely happen to be an overestimation since:

- there is no exclusion on branch cases so all loads from different branches are all included
- method from LLVM used is by default with data alignment padding⁵, the collected size might be larger than the actual data size
- without consideration of the fact that a same cache line can hold multiple load data at the same time, so there can exist loads that have been counted more than once

The load counting information is used to calculate an initial granularity (see Algorithm 4).

- Get the L1 data cache size
- Get the largest load size among all target kernels
- Compute initial granularity, the result of the L1 data cache size divided by the largest load size is then rounded up to the closest power of two value.

This theoretical initial granularity on one hand is essentially the maximum kernel granularity for the kernel with the largest load size, since the L1 data cache is fully filled with multiples of loaded data for this kernel. On the other hand, this initial granularity can as well be a minimum for the application with multiple targeted kernels, and used as a starting point to increase the granularity during the exploration for the application.

Since there is no assumption that can be made whether any arbitrary application contains only one targeted kernel, the computed theoretical initial granularity is believed to be minimal.

⁵`uint64_t`getTypeAllocSize: Returns the offset in bytes between successive objects of the specified type, including alignment padding. [http://llvm.org/docs/doxygen/html/classllvm_1_1DataLayout.html](http://llvm.org/docs/doxygen/html/classllvm_1_1DataLayout.html)
Algorithm 4: Initial Granularity

function initGranularity (fileList, appName, logRepo);

Input:
- fileList – a list of 
  
  "*gran$(maxint).extract.ll"
  filenames for targeted kernels
- appName – the name of the application
- logRepo – a subdirectory named with 
  
  log$_{(appName)}$
  under application’s
  dump repository for storing compilation information

Output:
[initGran, maxGran]

2 Use lscpu tool to get CPU information and store into a cache file;
3 Parse "L1d cache" text and get its size in the same line from cache file;
4 Transform "L1d cache" size into byte unit;
5 maxGran=|L1d size$/8;
6 foreach llfile in fileList do
7   Call CountLoad LLVM pass with input of llfile appName and logRepo;
8   Get load size and append into a load log file;
9 end
10 Find a largest load size, maxLoadSize=largest load size;
11 if maxLoadSize=0 then
12   initGran=2
13 else
14   initRaw=L1d size/maxLoadSize;
15 end
16 end
17 Return [initGran, maxGran];
4 Evaluation

Environment

The experiments are carried out on the alpha machine at: alpha.it.uu.se (see Table 1).

<table>
<thead>
<tr>
<th>Operating System:</th>
<th>Linux alpha 3.16.0-4-amd64</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#1 SMP Debian 3.16.7-ckt11-1+deb8u4</td>
</tr>
<tr>
<td></td>
<td>(2015-09-19) x86_64 GNU/Linux</td>
</tr>
<tr>
<td>Architecture:</td>
<td>x86_64</td>
</tr>
<tr>
<td>CPU op-mode(s):</td>
<td>32-bit, 64-bit</td>
</tr>
<tr>
<td>Byte Order:</td>
<td>LittleEndian</td>
</tr>
<tr>
<td>CPU(s):</td>
<td>8</td>
</tr>
<tr>
<td>On-line CPU(s) list:</td>
<td>0-7</td>
</tr>
<tr>
<td>Thread(s) per core:</td>
<td>2</td>
</tr>
<tr>
<td>Core(s) per socket:</td>
<td>4</td>
</tr>
<tr>
<td>Socket(s):</td>
<td>1</td>
</tr>
<tr>
<td>NUMA node(s):</td>
<td>1</td>
</tr>
<tr>
<td>Vendor ID:</td>
<td>GenuineIntel</td>
</tr>
<tr>
<td>CPU family:</td>
<td>6</td>
</tr>
<tr>
<td>Model:</td>
<td>42</td>
</tr>
<tr>
<td>Model name:</td>
<td>Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz</td>
</tr>
<tr>
<td>Stepping:</td>
<td>7</td>
</tr>
<tr>
<td>CPU MHz:</td>
<td>1600.000</td>
</tr>
<tr>
<td>CPU max MHz:</td>
<td>3400.0000</td>
</tr>
<tr>
<td>CPU min MHz:</td>
<td>1600.0000</td>
</tr>
<tr>
<td>BogoMIPS:</td>
<td>6820.16</td>
</tr>
<tr>
<td>Virtualization:</td>
<td>VT-x</td>
</tr>
<tr>
<td>L1d cache:</td>
<td>32K</td>
</tr>
<tr>
<td>L1i cache:</td>
<td>32K</td>
</tr>
<tr>
<td>L2 cache:</td>
<td>256K</td>
</tr>
<tr>
<td>L3 cache:</td>
<td>8192K</td>
</tr>
<tr>
<td>NUMA node0 CPU(s):</td>
<td>0-7</td>
</tr>
</tbody>
</table>

Table 1: Platform

SPEC CPU2006

The evaluation of feedback iterative compilation with DAE is based on a selection of C and C++ applications from SPEC CPU2006 [1]. These benchmarks have been classified into three different categories from previous work [9]. The classification into the categories Memory bound (MEM), Compute bound (COM) and Medium compute and memory bound (MIX) was based on two metrics: L1 cache misses and Cycles Per Instruction (CPI).

There are 12 benchmarks provided that are suitable for evaluation with targeted functions and loops that are all marked from previous work [9].

All experiments were done with training data sets (train) provided together with benchmarks, and the executions of the benchmarks were done with the same inputs respectively.

Benchmarking is successful on 9 out of 12 applications. Brief findings for each benchmark can be found in Table 2 with respective initial granularity (InitGran),
maximum indirection (MaxIndir) and time spent for thorough exhaustive search for one weight setting in minute (Exhaustive(min)), number of grids in the search space, theoretical total time \( T \) of feedback compilation for one weight setting in minute (Total(min)), as well as classified categories.

The unsuccessful ones are 464.h264ref, 471.omnetpp and 473.astar, which all require certain modifications. Such modifications can be changes on source code of the benchmarks or adaptive changes in `Makefile.defaults` of DAE compiler. Using DAE compiler as a tool readily given and deployed throughout this thesis, the author leaves out further modifications on DAE compiler.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Category</th>
<th>MaxIndir</th>
<th>InitGran</th>
<th>Exhaustive(min)</th>
<th>#Grid</th>
<th>Total(min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>429.mcf</td>
<td>MEM</td>
<td>3</td>
<td>256</td>
<td>38</td>
<td>48</td>
<td>15.83</td>
</tr>
<tr>
<td>433.milc</td>
<td>MEM</td>
<td>2</td>
<td>64</td>
<td>25</td>
<td>36</td>
<td>14.58</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>COM</td>
<td>4</td>
<td>64</td>
<td>16</td>
<td>60</td>
<td>9.33</td>
</tr>
<tr>
<td>450.soplex</td>
<td>MEM</td>
<td>15</td>
<td>64</td>
<td>25</td>
<td>192</td>
<td>14.58</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>COM</td>
<td>6</td>
<td>32</td>
<td>9</td>
<td>84</td>
<td>6</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>COM</td>
<td>4</td>
<td>32</td>
<td>337</td>
<td>60</td>
<td>224.67</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>MIX</td>
<td>3</td>
<td>64</td>
<td>16</td>
<td>48</td>
<td>9.33</td>
</tr>
<tr>
<td>470.lbm</td>
<td>MEM</td>
<td>2</td>
<td>64</td>
<td>52</td>
<td>36</td>
<td>30.33</td>
</tr>
<tr>
<td>482.sphinx3</td>
<td>MIX</td>
<td>7</td>
<td>32</td>
<td>7</td>
<td>96</td>
<td>4.67</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>COM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>MIX</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>473.astar</td>
<td>MIX</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 2: Brief on Benchmarks

**Settings**

As mentioned in previous section (**4. Implementation**), notable parameters that can significantly affect the searching results are **threshold**, **budget** and **weights**.

The threshold used to update the baseline is 0.05, which means the baseline for comparison is updated to be a DAE version that scores with a 5% improvement against the baseline.

The budget used to indicate the number of compilations is greater than the amount needed for traversing through the entire search space, therefore the results can be seen as an exhaustive search for all benchmarks.

Different weight settings for two objectives (runtime, prefetch rate) vary stepwise with 0.1 from 0.0 to 1.0, therefore there are 11 respective optimal DAE combinations found (i.e. \((0.0,1.0),(0.1,0.9),...,(1.0,0.0)\)).

With a higher weight for runtime, there is more focus on performance. Analogously, a higher weight for prefetch indicates more focus on energy/EDP so to dispatch more time spent in the Access phase.

Since the sum of two objectives is always 1, one just needs to refer to the weight of one objective. The notation for example "weight 0.7" in the rest of this paper will always refer to the runtime that is weighted for 0.7 and by default prefetch rate is the other part subtracted from one which is weighted for 0.3.

Minor impacts on the searching results may also be traced back to the specified number of **repeats** for profiling. In this evaluation, **repeat** is set to 3.

\(^{6}\text{calculated with MaxIndir and InitGran, based on Exhaustive(min)}\)
Result Analysis

All detailed running results shall be found under specified dump repository (specified in input file) coupled with a completed orchestration of feedback compilation.

For the purpose of analysis, for each benchmark, one table and three graphs are generated for a closer observation.

- Data presented in the table shows the difference in runtime and prefetch between different (weight) versions of the best found DAE executables during the Multi-Versioning step. The last column “Found At” is in the shape of a nominator/denominator, indicates the number of compilations done getting to the optimal point among total adequate number of compilations for an exhaustive search.

To estimate the actual time spent on searching, multiplications shall be done between the found number of compilations and the exhaustive time (Table 2) divided by the total adequate number of compilations.

- The figure named only with benchmark shows the results from feedback compilations for all 11 different weights with their respective best found (granularity, indirection) values, with a horizontal line indicating the MaxIndir and a vertical line indicating InitGran for the benchmark. Note that multiple weights can result at a same optimal point for best combination of granularity and indirection.

- The figure named with benchmark followed by “AllWeights” shows an overview at Comparison step the runtime performance of the best DAE version for all weight settings with respective reference of ORIGINAL. ORIGINAL does not depend on the weights or on the granularity / indirection, theoretically it is sufficient to compare with one ORIGINAL for all weights. The feedback compilation scheme is implemented with focus on orchestration for one weight setting, therefore for different weights there exist multiple respective ORIGINAL executables which shall theoretically all have a very similar runtime behavior.

- The figure named with benchmark followed by ”Extreme” gives a more detailed comparison on the extreme weight setting points (0.0 and 1.0) at the Comparison step. The best found DAE is annotated with runtime data, prefetch data, and the improvement rate is compared with its ORIGINAL which is also annotated with runtime data in its horizontal line.

Analysis Example with 429.mcf

Figure 429.mcf shows a log-scaled granularity and indirection. The point at (2,2) indicates a combination of indirection=4 and granularity=2 is the best for weight 0.7, the point at (2,0) indicates a combination of indirection=4 and granularity=0 is the best for weight 0.8, weight 0.9, and weight 1.0, whereas the point at (5,2) indicates a combination of indirection=32 and granularity=2 is the best for weight 0.0.

The computed initial granularity is 64 and the maximal indirection is found to be 3.

Table 429.mcf Compare Table shows the best found DAE for respective weights during the multi-versioning exploration stage. For instance, for weight 0.3, the best (granularity, indirection) combination is (4, 1) which gives a runtime of 16.2229 seconds with prefetch rate of 67.6%, this best combination can be found at the 5th out of total 48 compilations for an exhaustive search.
Figure 429.mcf.AllWeights shows from left to right each weights with its runtime of the best DAE version during the Comparison step, referenced by its original performance (horizontal line) in the same vertical range. All 11 different weights seem to have stable ORIGINALs. For this benchmark, it seems that there is notable slowdown for all best DAEs compared to their original.

Figure 429.mcf.Extreme shows a closer look on the two extreme weights annotated with details from 429.mcf.AllWeights.

For weight 0.0, its best found combination (granularity=32, indirection=2) gives a runtime of 15.86 seconds with prefetch rate of 82%, this best DAE runtime performed a slowdown and reached 79.78% from the original runtime of 12.6532 seconds.

For weight 1.0, its best found combination (granularity=4, indirection=0) gives a runtime of 15.86 seconds with prefetch rate of 23%, this best DAE runtime performed a slowdown and reached 79.74% from the original runtime of 12.6644 seconds.

Benchmark 429.mcf was marked as memory-bound, the runtime performance on the two extreme weights are very close, but the prefetch rate differs greatly. Increased data prefetching rate is believed to be more energy efficient for memory-bound applications. This extreme points comparison result helps to show the advantage of DAE technique of being energy efficient without much performance loss.

Discussions

• InitGran: is computed initial granularity a minimal or maximal?
  – MEM:
    429.mcf and 450.soplex both have all their best DAE weights found with lower granularity than InitGran.
    433.milc has one weight point much higher than InitGran but all others are lower.
    470.lbm has four starting weights that are found at higher granularity than InitGran.
  – COM:
    445.gobmk has four points found at granularity higher than InitGran.
    456.hmmer has four points found at granularity not higher than InitGran.
    458.sjeng has two points found at granularity that is not higher than InitGran.
  – MIX:
    462.libquantum has three points lower than InitGran.
    482.sphinx3 has one point lower but rest higher.

Observations on Figures with only benchmark names give the following conclusion:

There exists a higher probability that computed initial granularity lies closer to be maximal for memory bound applications. For compute bound and medium compute and memory bound applications, there is no clear conclusion that can be drawn, as neither of these two categories show any consistent trend.

• Indirection&Granularity: is increased focus of prefetch/performance reflected?
MEM:
429.mcf gets better performance with a higher granularity and a higher
indirection at weight 0.0 than at weight 1.0.
433.milc gets worse performance with a lower granularity and the same
indirection (at MaxIndir) at weight 0.0 than at weight 1.0.
450.soplex gets a very insignificant performance improvement with a lower
granularity and a higher indirection at weight 0.0 than at weight 1.0.
470.lbm gets better performance with a higher granularity and the same
indirection (at MaxIndir) at weight 0.0 than at weight 1.0.

COM:
445.gobmk gets worse performance with a lower granularity and a higher
indirection at weight 0.0 than at weight 1.0.
456.hmmer gets worse performance with a lower granularity and a higher
indirection (at MaxIndir) at weight 0.0 than at weight 1.0.
458.sjeng gets worse performance with a lower granularity and a higher
indirection (at MaxIndir) at weight 0.0 than at weight 1.0.

MIX:
462.libquantum gets worse performance with a lower granularity and a
higher indirection at weight 0.0 than at weight 1.0.
482.sphinx3 gets a very insignificant performance improvement with a
higher granularity and the same indirection (at MaxIndir) at weight 0.0
than at weight 1.0.

Observations on tables for all benchmarks give the following conclusions:
For all 9 benchmarks, there is a clear trend of positive correlation for weight
0.0 (more focus on energy) with higher indirection.
There is much consistent pattern with higher probability for compute bound
applications to have higher granularity with focus on runtime performance,
whereas no similar conclusion can be made for the other two categories.

Comparison: how close towards ORIGINAL version on runtime?

MEM:
429.mcf gets an insignificant difference at both extreme weight points and
achieve about 80% of the ORIGINAL performance.
433.milc gets an insignificant difference at both extreme weight points
and achieve about 96% of the ORIGINAL performance.
450.soplex gets an insignificant difference at both extreme weight points
and achieve about the same performance as the ORIGINAL with
insignificant variation.
470.lbm gets an insignificant difference at both extreme weight points and
achieve about 83% of the ORIGINAL performance.

COM:
445.gobmk gets an insignificant difference at both extreme weight points
and achieve about the same performance as the ORIGINAL with
insignificant variation.
456. hmmer gets an insignificant difference at both extreme weight points and achieve about the same performance as the ORIGINAL with less than 2%.

458. sjeng gets an insignificant difference at both extreme weight points and achieve about 92% of the ORIGINAL performance.

– MIX:

462. milc gets an insignificant difference at both extreme weight points and achieve about 47% of the ORIGINAL performance.

482. sphinx3 (results with large variation give low credibility) shows an improved performance for both extreme weight points.

Observations on figures with benchmark on extreme points give the following conclusion:

Both memory bound and compute bound applications show some promising DAE performance results that differ by no more than 20% of the ORIGINAL versions. No conclusion can be made basis on the inadequate data for medium compute and memory bound categories.

Results from feedback iterative experiments may shine a glint of more confidence on DAE technology for its goal on energy efficiency whilst without much performance loss.
5 Conclusions and Future Work

Results summary

Within the scope of running feedback compilation successfully on 9 benchmarks with DAE technique, there is trend showing that majority of the best DAE combinations from memory-bound applications have their computed initial granularity closer towards the maximal and have their indirection closer towards the maximum indirection.

Higher indirection values get chosen in the experiment results for lower weight settings (less focus on runtime), and higher granularity values get chosen for compute-bound benchmarks. This correlates well with the idea that DAE technique is supposed to deliver a high performance at low energy costs.

For memory-bound benchmarks and compute-bound benchmarks, majority of their best DAE versions achieved some runtime performance differences that are within 20% of their original results.

Conclusions

This proposed solution is able to find optimal DAE parameters for the multi-criterion objective of performance and improved energy efficiency. This makes it feasible to optimize targeted kernels of general applications and hence achieve improved energy efficiency with minimal required user input.

Future Work

The performance results from benchmarking with train inputs already vary within a rather limited small range, this can create noise on performance analysis for profiling.

Ideally, as the inputs from the benchmarks are classified into three categories: small (test input), medium (train input), and large (ref input), the result from running one benchmark with feedback compilation using test input shall identify the right configuration, and this configuration should be validated whether it provides good result when running the same benchmark using ref input. This needs to be studied further.

One potential improvement can be the usage of a much increased increased repeat setting with removal of one minimal and one maximal records, so hopefully the averaged result from profiling can be more trustworthy.

For a fair comparison in the end of the whole process, multiple trials have been done without (the necessity of) documentation, these show a very small difference on the performance records compared to keeping statistics (instead of spending more time&energy in re-compiling the DAE compiler).

Potential work for future continuation can be:

- The search strategy can be further improved/explored.
- Modify the source code of DAE compiler, to generate a shared object (.so file) for non-statistics purposes and at the same time re-compiling the compiler. This can help to reduce the redundant compilation time on the DAE compiler during the feedback process.
- Improve the Makefile.defaults for DAE compiler, so applications with multiple levels of subdirectories can be also swiftly and (or) successfully compiled.
• The current implementation is carried out with principles of engineering, refinement on the implementation and code could happen when there is time budget.

• The criteria for being optimal currently is based on runtime performance and energy efficiency. There can be other factors/objectives selected for exploring the optimal, for instance, compilation time and / or code size, etc.

This is a preliminary study to fine tune the compilation parameters of DAE, there are still some questions to be further discussed and explored in the future.

Based on the two factors that tuning indirection helps to prefetch data for less idle time on the high frequency computing unit, and tuning granularity helps to get data ready for reduced cache misses, this together with the pre-knowledge on hot loops of an application, it is believed that there is no necessity to al ways run the entire application to compare with its original, if the code region that is irrelevant to DAE transformations (remains the same as the original) is relatively large.

Different input data surely affects the results for each individual compilation. The focus should move towards quantified results that are at an average/consistent level of the measuring metrics.

Once there is observable/notable difference compared to using DAE technique, the input to the same application can therefore be considered useful for feedback compilations. The behavior of the same application with larger inputs is believed to be predictable, according to the principles of induction methodology.

The compilation parameters for DAE might not necessarily remain the same as the problem input changes, if the application is featured with much higher level of indirections, although this statement requires further experiments and validation.
6 References


Appendices

A  Example Input File

```json
{
    "#": 0,
    "0": {
        "dataType": "ref",
        "compilerPath": "/home/daedal/compiler/projects/",
        "targetFile": "Makefile.targets",
        "targetPath": "/home/daedal/sources/common/DAE/",
        "benchmarks": ["myBenchmark"],
        "benchArgs": {"myBenchmark": ""},
        "benchmarkPath": "/home/daedal/sources/",
        "objective": 2,
        "weight": [1.0, 0],
        "threshold": 0.05,
        "budget": 10,
        "repeat": 3,
        "dumpAt": "/home/feedResult/",
        "granShift": 0
    },
    "1": {
        "dataType": "ref",
        "compilerPath": "/home/daedal/compiler/projects/",
        "targetFile": "Makefile.targets",
        "targetPath": "/home/daedal/sources/common/DAE/",
        "benchmarks": ["myBenchmark"],
        "benchArgs": {"myBenchmark": ""},
        "benchmarkPath": "/home/daedal/sources/",
        "objective": 2,
        "weight": [0.5, 0.5],
        "threshold": 0.05,
        "budget": 10,
        "repeat": 3,
        "dumpAt": "/home/feedResult/",
        "granShift": 2
    },
    "2": {
        "dataType": "ref",
        "compilerPath": "/home/daedal/compiler/projects/",
        "targetFile": "Makefile.targets",
        "targetPath": "/home/daedal/sources/common/DAE/",
        "benchmarks": ["myBenchmark"],
        "benchArgs": {"myBenchmark": ""},
        "benchmarkPath": "/home/daedal/sources/",
        "objective": 2,
        "weight": [0.0, 1.0],
        "threshold": 0.05,
        "budget": 10,
        "repeat": 3,
        "dumpAt": "/home/feedResult/",
        "granShift": 1
    }
}
```
B  ReadMe: Feedback Compilation

This tool is designed to orchestrate a series of compilation process so a close to optimum executable result will be found within some specified budget for programs/applications. Currently it supports up to 2 objectives: execution time and data prefetching rate.

Prerequisites

DAE

Make sure DAE framework is installed and working on legitimate applications.

Python

Python2 is installed, version preferable 2.7

Script

Make sure the mandatory 4 files are in this script repository:

* `args.json` - example input file where all input arguments are specified for a successful compilation

* `countLoad.cpp` - source file of a LLVM pass that gathers load information from an application, so a starting value for granularity is decided dynamically

* `feedbackCompilation.py` - script that orchestrates a whole iterative procedure for compilations with feedback information on up to 2 objectives.

* `Makefile` - file used to compile and generate a shared object from `countLoad.cpp`

Repository Structure

Analogous hierarchy as small benchmark from DAE: For example:

$(path to benchmarks)/$(benchmarkName)/$(src)

$(path to benchmarks)/$(benchmarkName)/$(bin)

Getting Started

Some details need to be specified in a `json` file, example file `args.json` can be used for reference as a starting point.

* `#` - integer or a list of integers, specify arguments for the script.

(1) If is a positive integer, the first $(#)$ argument settings will be run, which means there must exist at least $(#)$ number of argument settings.

(2) If is a non-positive integer, all the existing arguments settings will be run.

(3) If is a list of integers, corresponding argument index(indices) will be run.

* `dataType` - string, specify input type for an application so strategy searching pattern might be applied differently. Choose among `train`, `test` and `reference`.

* `compilerPath` - string, path to the compiler projects which consist of DAE
* **targetFile** - string, name of the DAE makefile for target settings, by default: Makefile.targets

* **targetPath** - string, path to the DAE makefile for target settings

* **benchmarks** - list of string, benchmark/application names

* **benchArgs** - dictionary, for each existing benchmarks, specify respective arguments(and path to arguments if not in the same directory) for executions.

* **benchmarkPath** - string, path to the repository that contains benchmarks/applications

* **objective** - integer, number of objectives, maximum value can be set is 2, the first objective is execution time, and the second objective is data prefetch rate

* **weight** - list of float, for each objective, the proportional weight needs to be specified. All weights must sum up to be 1.0

* **threshold** - float, the value for updating a baseline which is used for comparison if improvement meets the threshold

* **budget** - integer, the number of compilations that can be afforded to run this script

* **repeat** - integer, the number of repetitions on profiling each executable binary

* **dumpAt** - string, path to a repository where all resulting files can be dumped from feedback compilation

* **granShift** - integer, the number of right shifts so to have a starting granularity that is smaller than computed initial granularity, conservative fashion. For instance, with a granShift=2 on computed initGran=64, the starting granularity for compilations will be gran=16

Run

```python2 feedbackCompilation.py args.json```

Result Files

All output from feedback compilation will be dumped under specified dumpAt repository, where common information will be found directly under this repository.

Benchmark specific file that holds basic information(indirection, initial granularity and maximum granularity) that can be shared for different running occasions is saved in: dump.$(benchmark)/log.$(benchmark)_strategy_basicInfo.json

Benchmark specific results such as overview on searching pattern and results for different settings from each compilation will be found in .txt file under subdirectory: dump.$(benchmark)/plot.$(benchmark & setting).$(date&time).txt

Two further repository will be found that contains respectively executable files and load info from llvm loadCount pass: exe.$(benchmark) and info.$(benchmark)

======== contact weji3183@student.uu.se =======
C  Benchmark Plots

Figure 4: 429.mcf

Table 3: 429.mcf Compare Table

<table>
<thead>
<tr>
<th>Weight (run-time)</th>
<th>Granularity</th>
<th>Indirection</th>
<th>Runtime (in seconds)</th>
<th>Prefetch Rate</th>
<th>Found At</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>32</td>
<td>2</td>
<td>14.370387539</td>
<td>0.8217</td>
<td>19/48</td>
</tr>
<tr>
<td>0.1</td>
<td>16</td>
<td>2</td>
<td>14.643145087</td>
<td>0.8145</td>
<td>18/48</td>
</tr>
<tr>
<td>0.2</td>
<td>8</td>
<td>1</td>
<td>15.115641658</td>
<td>0.7579</td>
<td>6/48</td>
</tr>
<tr>
<td>0.3</td>
<td>4</td>
<td>1</td>
<td>16.222946383</td>
<td>0.676</td>
<td>5/48</td>
</tr>
<tr>
<td>0.4</td>
<td>4</td>
<td>1</td>
<td>16.215476741</td>
<td>0.6759</td>
<td>5/48</td>
</tr>
<tr>
<td>0.5</td>
<td>4</td>
<td>1</td>
<td>16.210870018</td>
<td>0.6762</td>
<td>5/48</td>
</tr>
<tr>
<td>0.6</td>
<td>4</td>
<td>1</td>
<td>16.207997434</td>
<td>0.6757</td>
<td>5/48</td>
</tr>
<tr>
<td>0.7</td>
<td>4</td>
<td>2</td>
<td>16.166101163</td>
<td>0.6896</td>
<td>16/48</td>
</tr>
<tr>
<td>0.8</td>
<td>4</td>
<td>0</td>
<td>15.860074015</td>
<td>0.2321</td>
<td>5/48</td>
</tr>
<tr>
<td>0.9</td>
<td>4</td>
<td>0</td>
<td>15.860033319</td>
<td>0.2326</td>
<td>5/48</td>
</tr>
<tr>
<td>1.0</td>
<td>4</td>
<td>0</td>
<td>15.828483144</td>
<td>0.2331</td>
<td>5/48</td>
</tr>
</tbody>
</table>
Figure 5: 429.mcf.AllWeights

Figure 6: 429.mcf.Extreme
Figure 7: 433.milc

Table 4: 433.milc Compare Table

<table>
<thead>
<tr>
<th>Weight (run-time)</th>
<th>Granularity</th>
<th>Indirection</th>
<th>Runtime (in seconds)</th>
<th>Prefetch Rate</th>
<th>Found At</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>2</td>
<td>2</td>
<td>4.39522188</td>
<td>0.2132</td>
<td>3/36</td>
</tr>
<tr>
<td>0.1</td>
<td>2</td>
<td>2</td>
<td>4.391947231</td>
<td>0.2145</td>
<td>3/36</td>
</tr>
<tr>
<td>0.2</td>
<td>32</td>
<td>2</td>
<td>4.139123078</td>
<td>0.192</td>
<td>7/36</td>
</tr>
<tr>
<td>0.3</td>
<td>16</td>
<td>2</td>
<td>4.136386744</td>
<td>0.1916</td>
<td>6/36</td>
</tr>
<tr>
<td>0.4</td>
<td>16</td>
<td>2</td>
<td>4.132503013</td>
<td>0.1921</td>
<td>6/36</td>
</tr>
<tr>
<td>0.5</td>
<td>64</td>
<td>2</td>
<td>4.130066614</td>
<td>0.1933</td>
<td>8/36</td>
</tr>
<tr>
<td>0.6</td>
<td>64</td>
<td>2</td>
<td>4.123716199</td>
<td>0.1933</td>
<td>8/36</td>
</tr>
<tr>
<td>0.7</td>
<td>4096</td>
<td>2</td>
<td>4.110761065</td>
<td>0.1923</td>
<td>14/36</td>
</tr>
<tr>
<td>0.8</td>
<td>8</td>
<td>2</td>
<td>4.122907394</td>
<td>0.1927</td>
<td>5/36</td>
</tr>
<tr>
<td>0.9</td>
<td>8</td>
<td>2</td>
<td>4.114811238</td>
<td>0.1936</td>
<td>5/36</td>
</tr>
<tr>
<td>1.0</td>
<td>32</td>
<td>2</td>
<td>4.10479317</td>
<td>0.1915</td>
<td>7/36</td>
</tr>
</tbody>
</table>
Figure 8: 433.milc.AllWeights

Figure 9: 433.milc.Extreme
445.gobmk: Best DAE Comparison for All Weights

<table>
<thead>
<tr>
<th>Weight (run-time)</th>
<th>Granularity</th>
<th>Indirection</th>
<th>Runtime (in seconds)</th>
<th>Prefetch Rate</th>
<th>Found At</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>2</td>
<td>4</td>
<td>0.010743509</td>
<td>0.2368</td>
<td>5/60</td>
</tr>
<tr>
<td>0.1</td>
<td>2</td>
<td>4</td>
<td>0.010782218</td>
<td>0.2366666667</td>
<td>5/60</td>
</tr>
<tr>
<td>0.2</td>
<td>2</td>
<td>4</td>
<td>0.011173586</td>
<td>0.2365333333</td>
<td>5/60</td>
</tr>
<tr>
<td>0.3</td>
<td>2048</td>
<td>4</td>
<td>0.0111268985</td>
<td>0.1787333333</td>
<td>38/60</td>
</tr>
<tr>
<td>0.4</td>
<td>16</td>
<td>1</td>
<td>0.010619656</td>
<td>0.1770666667</td>
<td>52/60</td>
</tr>
<tr>
<td>0.5</td>
<td>16</td>
<td>0</td>
<td>0.011123582</td>
<td>0.1647</td>
<td>8/60</td>
</tr>
<tr>
<td>0.6</td>
<td>16</td>
<td>0</td>
<td>0.009586005</td>
<td>0.1644</td>
<td>8/60</td>
</tr>
<tr>
<td>0.7</td>
<td>128</td>
<td>0</td>
<td>0.009932822</td>
<td>0.1647666667</td>
<td>11/60</td>
</tr>
<tr>
<td>0.8</td>
<td>16</td>
<td>0</td>
<td>0.009678199</td>
<td>0.1646</td>
<td>8/60</td>
</tr>
<tr>
<td>0.9</td>
<td>1024</td>
<td>0</td>
<td>0.010594004</td>
<td>0.1650666667</td>
<td>14/60</td>
</tr>
<tr>
<td>1.0</td>
<td>256</td>
<td>0</td>
<td>0.009988963</td>
<td>0.1646333333</td>
<td>12/60</td>
</tr>
</tbody>
</table>

Table 5: 445.gobmk Compare Table
Figure 11: 445.gobmk.AllWeights

Figure 12: 445.gobmk.Extreme
Figure 13: 450.soplex

450.soplex[MEM]: best found points
([Granularity,Indirection], annotated with respective weights)

<table>
<thead>
<tr>
<th>Weight (run-time)</th>
<th>Granularity</th>
<th>Indirection</th>
<th>Runtime (in seconds)</th>
<th>Prefetch Rate</th>
<th>Found At</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>2</td>
<td>12</td>
<td>2.410919411</td>
<td>0.6045</td>
<td>13/192</td>
</tr>
<tr>
<td>0.1</td>
<td>2</td>
<td>12</td>
<td>2.406477206</td>
<td>0.6418</td>
<td>13/192</td>
</tr>
<tr>
<td>0.2</td>
<td>2</td>
<td>14</td>
<td>2.412506142</td>
<td>0.6248</td>
<td>15/192</td>
</tr>
<tr>
<td>0.3</td>
<td>2</td>
<td>14</td>
<td>2.423753543</td>
<td>0.6261</td>
<td>15/192</td>
</tr>
<tr>
<td>0.4</td>
<td>2</td>
<td>14</td>
<td>2.408885815</td>
<td>0.6164</td>
<td>15/192</td>
</tr>
<tr>
<td>0.5</td>
<td>2</td>
<td>12</td>
<td>2.430486469</td>
<td>0.623</td>
<td>13/192</td>
</tr>
<tr>
<td>0.6</td>
<td>8</td>
<td>8</td>
<td>2.419307282</td>
<td>0.5231</td>
<td>139/192</td>
</tr>
<tr>
<td>0.7</td>
<td>8</td>
<td>5</td>
<td>2.426016458</td>
<td>0.537</td>
<td>139/192</td>
</tr>
<tr>
<td>0.8</td>
<td>8</td>
<td>7</td>
<td>2.413259844</td>
<td>0.534</td>
<td>106/192</td>
</tr>
<tr>
<td>0.9</td>
<td>8</td>
<td>12</td>
<td>2.406335104</td>
<td>0.5225</td>
<td>29/192</td>
</tr>
<tr>
<td>1.0</td>
<td>8</td>
<td>4</td>
<td>2.402674777</td>
<td>0.5251</td>
<td>172/192</td>
</tr>
</tbody>
</table>

Table 6: 450.soplex Compare Table
Figure 14: 450.soplex.AllWeights

Figure 15: 450.soplex.Extreme
Table 7: 456.hmmer Compare Table
Figure 17: 456.hmmer.AllWeights

Figure 18: 456.hmmer.Extreme
Table 8: 458.sjeng Compare Table

<table>
<thead>
<tr>
<th>Weight (run-time)</th>
<th>Granularity</th>
<th>Indirection</th>
<th>Runtime (in seconds)</th>
<th>Prefetch Rate</th>
<th>Found At</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>2</td>
<td>4</td>
<td>125.969275342</td>
<td>0.5108</td>
<td>5/60</td>
</tr>
<tr>
<td>0.1</td>
<td>512</td>
<td>4</td>
<td>103.799915167</td>
<td>0.4293</td>
<td>13/60</td>
</tr>
<tr>
<td>0.2</td>
<td>32</td>
<td>0</td>
<td>101.440224456</td>
<td>0.2549</td>
<td>42/60</td>
</tr>
<tr>
<td>0.3</td>
<td>512</td>
<td>0</td>
<td>101.22831378</td>
<td>0.2551</td>
<td>46/60</td>
</tr>
<tr>
<td>0.4</td>
<td>256</td>
<td>0</td>
<td>101.345138576</td>
<td>0.2534</td>
<td>34/60</td>
</tr>
<tr>
<td>0.5</td>
<td>4096</td>
<td>0</td>
<td>101.155775309</td>
<td>0.2549</td>
<td>16/60</td>
</tr>
<tr>
<td>0.6</td>
<td>4096</td>
<td>0</td>
<td>101.324120791</td>
<td>0.2537</td>
<td>16/60</td>
</tr>
<tr>
<td>0.7</td>
<td>128</td>
<td>0</td>
<td>101.260654704</td>
<td>0.2535</td>
<td>11/60</td>
</tr>
<tr>
<td>0.8</td>
<td>32</td>
<td>0</td>
<td>101.288981053</td>
<td>0.2504</td>
<td>9/60</td>
</tr>
<tr>
<td>0.9</td>
<td>2048</td>
<td>0</td>
<td>101.211093645</td>
<td>0.2551</td>
<td>15/60</td>
</tr>
<tr>
<td>1.0</td>
<td>512</td>
<td>0</td>
<td>101.400314813</td>
<td>0.2527</td>
<td>13/60</td>
</tr>
</tbody>
</table>
Figure 20: 458.sjeng.AllWeights

Figure 21: 458.sjeng.Extreme
### Table 9: 462.libquantum Compare Table

<table>
<thead>
<tr>
<th>Weight (run-time)</th>
<th>Granularity</th>
<th>Indirection</th>
<th>Runtime (in seconds)</th>
<th>Prefetch Rate</th>
<th>Found At</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>4</td>
<td>2</td>
<td>11.971150032</td>
<td>0.5062</td>
<td>5/48</td>
</tr>
<tr>
<td>0.1</td>
<td>16</td>
<td>1</td>
<td>5.632583313</td>
<td>0.4958</td>
<td>7/48</td>
</tr>
<tr>
<td>0.2</td>
<td>16</td>
<td>2</td>
<td>5.63291684</td>
<td>0.4952</td>
<td>7/48</td>
</tr>
<tr>
<td>0.3</td>
<td>1024</td>
<td>3</td>
<td>3.256903174</td>
<td>0.4548</td>
<td>46/48</td>
</tr>
<tr>
<td>0.4</td>
<td>1024</td>
<td>3</td>
<td>3.257570608</td>
<td>0.4548</td>
<td>46/48</td>
</tr>
<tr>
<td>0.5</td>
<td>1024</td>
<td>3</td>
<td>3.25769662</td>
<td>0.4548</td>
<td>46/48</td>
</tr>
<tr>
<td>0.6</td>
<td>4096</td>
<td>0</td>
<td>2.96720977</td>
<td>0.3979</td>
<td>37/48</td>
</tr>
<tr>
<td>0.7</td>
<td>4096</td>
<td>0</td>
<td>2.968085892</td>
<td>0.398</td>
<td>37/48</td>
</tr>
<tr>
<td>0.8</td>
<td>4096</td>
<td>0</td>
<td>2.968206811</td>
<td>0.3974</td>
<td>37/48</td>
</tr>
<tr>
<td>0.9</td>
<td>4096</td>
<td>0</td>
<td>2.966082823</td>
<td>0.3979</td>
<td>48/48</td>
</tr>
<tr>
<td>1.0</td>
<td>4096</td>
<td>0</td>
<td>2.96720977</td>
<td>0.3979</td>
<td>48/48</td>
</tr>
</tbody>
</table>

**Figure 22: 462.libquantum**

462.libquantum[MIX]: best found points
(Granularity,Indirection), annotated with respective weights

<table>
<thead>
<tr>
<th>Granularity</th>
<th>Indirection</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>[0.0]</td>
</tr>
<tr>
<td>0.1</td>
<td>[0.2]</td>
</tr>
<tr>
<td>0.2</td>
<td>[0.1]</td>
</tr>
<tr>
<td>0.3</td>
<td>[0.6, 0.7, 0.8, 0.9, 1.0]</td>
</tr>
</tbody>
</table>
Figure 23: 462.libquantum.AllWeights

Figure 24: 462.libquantum.Extreme
Table 10: 470.lbm Compare Table

Weight (run-time) | Granularity | Indirection | Runtime (in seconds) | Prefetch Rate | Found At
--- | --- | --- | --- | --- | ---
0.0 | 256 | 2 | 28.372382394 | 0.1956 | 10/36
0.1 | 128 | 2 | 28.195000148 | 0.1946 | 9/36
0.2 | 128 | 2 | 28.182575102 | 0.1947 | 9/36
0.3 | 128 | 2 | 28.197924637 | 0.1953 | 9/36
0.4 | 2 | 2 | 28.865461524 | 0.0924 | 3/36
0.5 | 2 | 2 | 29.108412296 | 0.0924 | 3/36
0.6 | 2 | 2 | 28.832061873 | 0.0924 | 3/36
0.7 | 2 | 2 | 28.86429242 | 0.0923 | 3/36
0.8 | 2 | 2 | 28.844869999 | 0.0922 | 3/36
0.9 | 2 | 2 | 28.841461668 | 0.0924 | 3/36
1.0 | 2 | 2 | 28.835292249 | 0.0924 | 3/36
Figure 26: 470.lbm.AllWeights

Figure 27: 470.lbmExtreme
Table 11: 482.sphinx3 Compare Table

<table>
<thead>
<tr>
<th>Weight (run-time)</th>
<th>Granularity</th>
<th>Indirection</th>
<th>Runtime (in seconds)</th>
<th>Prefetch Rate</th>
<th>Found At</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>512</td>
<td>7</td>
<td>0.000546413</td>
<td>0.0442</td>
<td>16/96</td>
</tr>
<tr>
<td>0.1</td>
<td>256</td>
<td>7</td>
<td>0.000488592</td>
<td>0.0446</td>
<td>15/96</td>
</tr>
<tr>
<td>0.2</td>
<td>4096</td>
<td>7</td>
<td>0.000547054</td>
<td>0.045</td>
<td>19/96</td>
</tr>
<tr>
<td>0.3</td>
<td>1024</td>
<td>7</td>
<td>0.000539906</td>
<td>0.0451</td>
<td>17/96</td>
</tr>
<tr>
<td>0.4</td>
<td>512</td>
<td>7</td>
<td>0.000206325</td>
<td>0.0539</td>
<td>16/96</td>
</tr>
<tr>
<td>0.5</td>
<td>4096</td>
<td>7</td>
<td>0.000412267</td>
<td>0.044</td>
<td>19/96</td>
</tr>
<tr>
<td>0.6</td>
<td>2048</td>
<td>7</td>
<td>0.000372357</td>
<td>0.0445</td>
<td>18/96</td>
</tr>
<tr>
<td>0.7</td>
<td>2048</td>
<td>7</td>
<td>0.000471612</td>
<td>0.045</td>
<td>18/96</td>
</tr>
<tr>
<td>0.8</td>
<td>4096</td>
<td>7</td>
<td>0.000538096</td>
<td>0.0531</td>
<td>19/96</td>
</tr>
<tr>
<td>0.9</td>
<td>64</td>
<td>7</td>
<td>0.000265589</td>
<td>0.0535</td>
<td>13/96</td>
</tr>
<tr>
<td>1.0</td>
<td>2</td>
<td>7</td>
<td>0.000564596</td>
<td>0.0537</td>
<td>8/96</td>
</tr>
</tbody>
</table>
Figure 29: 482.sphinx3.AllWeights

Figure 30: 482.sphinx3.Extreme