UPPSALA
UNIVERSITET

IT 14 051
Examensarbete 15 hp
September 2014

Profiling-Assisted Prefetching with Just-In-Time Compilation

Jonatan Waern

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Profiling-Assisted Prefetching with Just-In-Time Compilation

Jonatan Waern

The Decoupled Access/Execute(DAE) approach is a method to reduce the energy consumption of task-based programs, based on dividing tasks in two phases where the first phase prefetches data at a low CPU frequency and the following phase performs computation at a high CPU frequency. The goal of this project is to extend this approach to sequential programs and examine the benefits of optimising the access phase to better suit the architecture the program runs on, and the program input. By using a Just-In-Time compiler to dynamically optimise the program and by utilising profiling tools to obtain runtime information, we have examined the possible benefits of the DAE approach on sequential programs with optimised access phases. We compared the benefits with the cost of dynamic optimisation by testing the method on selected benchmarks from the SPEC CPU 2006 suite. The results indicate that for many types of sequential programs dynamically optimised DAE is suitable, but care must be taken when determining how to optimise the access phases.

Handledare: Alexandra Jimborean
Ämnesgranskare: Konstantinos Sagonas
Examinator: Olle Gällmo
IT 14 051
Tryckt av: Reprocentralen ITC

Contents

Contents i

List of Figures iii

Abbreviations iv

1 Introduction 1
- 1.1 Background . . . 1
  - 1.1.1 Dynamic Voltage and Frequency Scaling . . . 1
  - 1.1.2 The Decoupled Access Execute Approach . . . 2
- 1.2 Problem Definition . . . 2
- 1.3 Solution . . . 3

2 Method 5
- 2.1 Profiling . . . 5
- 2.2 Static DAE Generation and Marking . . . 6
- 2.3 Optimisation . . . 7
  - 2.3.1 JIT design . . . 8
  - 2.3.2 JIT analysis . . . 10
- 2.4 The Power Model . . . 10

3 Related Work 11

4 Results 12
- 4.1 Benchmark Selection . . . 12
- 4.2 Architecture . . . 13
- 4.3 Offline Profiling . . . 13
  - 4.3.1 mcf . . . 14
  - 4.3.2 milc . . . 15
  - 4.3.3 soplex . . . 16
  - 4.3.4 hmmer . . . 17
  - 4.3.5 libquantum . . . 18
  - 4.3.6 lbm . . . 19
  - 4.3.7 astar . . . 20
- 4.4 Granularity Testing . . . 21
- 4.5 Evaluation . . . 25

5 Conclusions 29

5.1 Conclusions & Future Work 29

Bibliography **32**

List of Figures

2.1 Loop Chunking Layout 7
2.2 Design of JIT Preparation 9

4.1 mcf cache miss% per line and function 14
4.2 milc cache miss% per line and function 15
4.3 soplex cache miss% 16
4.4 soplex cache miss% per line 16
4.5 hmmer cache miss% 17
4.6 hmmer cache miss% per line 17
4.7 libquantum cache miss% per line and function 18
4.8 lbm cache miss% per line and function 19
4.9 astar cache miss% 20
4.10 astar cache miss% per line 20
4.11 Results of granularity tests for time 23
4.12 Results of granularity tests for energy 24
4.13 Results of JIT, Static and Original runs 28

Abbreviations

DAE	**D**ecoupled **A**ccess **E**xecute
JIT	**J**ust **I**n **T**ime
DCE	**D**ead **C**ode **E**limination
DVFS	**D**ynamic **V**oltage **F**requency **S**caling
CPU	**C**entral **P**rocessing **U**nit
IR	**I**ntermediate **R**epresentation
CPI	**C**ycles **P**er **I**nstruction
L1	First Level
LL	**L**ast **L**evel

Chapter 1

Introduction

Compile-time optimisations target various aspects, most commonly enhancing performance or decreasing code size. However, power-efficiency has become an important concern and attracted the research interest of the compiler community [1–5].

1.1 Background

1.1.1 Dynamic Voltage and Frequency Scaling

A well known technique for reducing the energy consumption is utilising Dynamic Voltage and Frequency Scaling (DVFS). DVFS refers to adjusting the frequency or voltage of the processor while the program is running. Because the CPU must stall while loading data from main memory, energy saving can be done by reducing the frequency of the processor during sections of the program where there are a lot of memory-intensive operations. What limits this technique is the intermingling of memory-oriented and compute-oriented instructions, as well as the overhead of changing the CPU frequency.

The frequency of the CPU can be adjusted at runtime to be lower when executing load instructions that must access main memory and to be higher when executing instructions that do not stall the CPU, resulting in a decrease in energy usage [6]. However, because in regular programs memory-bound and compute-bound instructions are mixed, adapting the frequency at per-instruction level is inefficient [7]. A remedy for this is the Decoupled Access Execute approach (DAE) [7, 8].

1.1.2 The Decoupled Access Execute Approach

The DAE approach is a way to improve the energy efficiency of programs by splitting the critical sections into an access and an execute phase. This allows the frequency of the CPU to be lower when loading data and faster when computing values.

Koukos et al. [7] demonstrated a solution to the problem of the overhead of DVFS-adjustments using a DAE approach for task-based parallel applications. In the DAE approach, the tasks on the critical path of the code are split out into an access and an execute version. In the access version data addresses are calculated and prefetched into a CPU-near cache, and in the execute version computations are executed. Thus, the adjustment of the CPU frequency can be done once for the access version, that is memory bound, and once for the compute version, that is compute bound. This process was then automated by Jimborean et al. [8], resulting in a compiler that can statically generate the decoupled access-execute versions for task-based parallel programs.

However, the previous DAE approaches [7, 8] are limited. Designing an efficient access phase at compile-time is challenging, since program's execution is influenced by a number of factors only known at runtime (input data, execution context, memory accessing patterns etc.). Furthermore they are limited to working only with task-based programs.

1.2 Problem Definition

This work aims to extend the previous DAE implementations by:

- Extend the DAE approach to *sequential* programs, by creating suitable access and execute phases for parts of the programs.
- Use *dynamic* information to optimise the access phase. The behaviour of the access phase can vary greatly depending on factors only known at runtime. How to optimise the behaviour can be guided by profiling,which in this work is done offline by knowing the running architecture and input of the program.
- Estimate the cost of *dynamically* optimising the access phase of the program, by using a Just-In-Time(JIT) compiler to optimise and compile the access phase dynamically based on information obtained prior to running the program.

These contributions will lead to a better understanding of the impact DAE has on complicated sequential programs,as well as how these programs should be optimised to increase the benefits in energy consumption and minimise the overhead in time.

1.3 Solution

When targeting sequential programs for the DAE approach, the memory bound loopsrepresent the main candidates for the method. A targeted memory bound loop is divided into segments (called chunks, see Chapter 2.2), these segments are then treated as tasks that are further divided into an access and execute phase. The access phase is then transformed into a prefetch-only phase by a method similar to the ones used by Jimborean et. al.[8], with the main differences being that the initial access phase contains prefetches for all load instructions in the iteration. This transformation of code is done statically, for efficiency reasons.

The suitable behaviour of the access phase may vary from one architecture to another and may also depend on program input or execution context. It is desirable to optimise the access phase by customising the prefetching code according to these factors. To obtain information about these factors one can profile the program. This can either be done online, during program execution, or offline, unrelated to program execution. In the first case the profiling information can be immediately used for optimisation, but in the later case the information is saved to be used in online optimisations during program execution. In this work the profiling is done offline. In chapter 2.1 it is described in detail what information was collected and how it was used and in chapter 4.3 the results of the offline profiling are presented.

To dynamically optimise and compile the program, a JIT engine is used. The base versions of the access phases are generated statically, thus the JIT will run the base program and only optimise and compile the access phases. Which optimisations are performed and details about how the JIT behaves are described in chapter 2.3.

The framework is outlined in 2.2.

These methods were tried on a number of benchmarks, as described in chapter 4 where the results are also presented. The expectations were that for compute bound benchmarks there would be an overhead in the time taken by the program and small or no energy gains obtained by performing DAE. For memory bound programs the expectation is that there will be small or no overhead in the time taken and a reduction in the energy used by the program.

The data(4.13) shows that in general the overhead in time of applying DAE to sequential programs is small in most cases, both for memory bound and compute bound benchmarks. The overhead of applying dynamic optimisation varies with benchmark, but in most cases it is low. For almost all compute bound benchmarks there were no energy benefits as expected, the cost of prefetching for these benchmarks outweighs the savings obtained as a result. In most cases for memory bound benchmarks we see a breakeven in energy consumption, which is contrary to our expectations. There are two main reasons that could explain why there are no energy gains for a memory bound benchmark:

- The prefetch phase is too heavy, meaning that the energy benefits from prefetching an address are hidden by the cost of calculating the address twice.
- The cost of dynamic optimisation offsets the gains of dynamic optimisation, meaning that the time and energy used to generate the access phases is larger than the energy saved by utilising the DAE approach.

What factors impact the overhead of the access phase and the JIT, as well as what factors affect the gains obtained by the access phase are detailed further in chapter 5.

Extensions of this work include reducing the overhead of the JIT as well as investigating how much the overhead and benefits of dynamically optimised access phases are affected by various factors and variables in the execution.

The natural extension of this work is to move the profiling step online.

Chapter 2

Method

To adapt a sequential program for DAE execution, the targered loops in the program are chosen and divided into segments. Each segment is treated as a task and each task is then split into an access and execute phase, where the access phase prefetches data and the execute phase performs calculations. In order to improve the access phase, it is necessary to identify the load instructions in the original loop that miss in the L1(first level) cache since prefetching these will reduce the time taken in the execute phase. Given this information, removing all the other prefetches and performing deadcode elimination will make the access phase become more lightweight. To enable dynamic optimisation, the execution of the program is wrapped inside a JIT that will compile and optimise the access phases on demand using the dynamic information that was collected statically.

2.1 Profiling

The benefit from the access phase comes from the fact that prefetching data at a lower frequency saves energy while it can later be loaded from a CPU-near cache at a high frequency, the overall performance is maintained since the high-frequency load will be fast. The cost of doing this is that the address for the prefetch needs to be calculated twice. Initially, the DAE phase that is used in this work prefetches once for each address accessed in the targeted loop. This may not always be efficient since not all loads miss in the L1 cache.

Normally, what loads miss in the L1 cache depends on computer architecture and program input. Meaning that it could change on a run-by-run basis of the program. This information is in this work acquired by offline profiling by profiling the program under the same input and computer architecture as it will later be run under.

In order to detect which instructions incur cache misses and, consequently, to determine which addresses require prefetching in the access phase, a cache profiler is used. There are various tools that can accomplish this, but the one chosen for this work is Cachegrind[9], a tool in the Valgrind[10] tool suite. Cachegrind simulates the behaviour of the cache of the current architecture, thus providing very accurate numbers for cache misses at the first and last level of caches. For this work, the misses of interest are mainly in the L1 cache, since achieved by prefetching data so that it is in L1. Comparatively, even if the data is in the LL(Last Level) cache, the CPU will still have to stall to load it.

After obtaining information on the cache misses for a program, it is examined to see how suitable it will be for the DAE approach. Applications that exhibit a high miss rate in the L1 cache in the most time consuming functions of the program[11] are more likely to benefit from decoupled execution. After finding the functions that dominate the execution time and characterising their memory accessing behaviour, the instructions that incur most of the cache misses are noted, if these instructions are within loops it is very likely that the program benefits from DAE. Also, the more concentrated the cache misses are amongst few instructions within a loop, the more likely it is that an optimised access phase will improve performance.

Using the visualisation tool KCachegrind[12] I have mapped instructions in the C or C++ source code to instructions in the LLVM IR(Intermediate Representation), where the ones that result in delinquent loads are marked by metadata. The translation is made by hand by using information about the control-flow-graphs and general knowledge of the code.

2.2 Static DAE Generation and Marking

To generate the initial access phase that will later be optimised, a simplified version of the DAE compiler from [8] is used. This version is designed to generate prefetch

instructions for all data accesses within the loops contained in selected functions of the program. The outermost loops are chunked (sliced) into loop segments, where each segment contains a number of loop iterations. The number of iterations is called the granularity of the chunk. In the cases where program behaviour makes this inefficient(i.e one iteration of the outermost loop accesses more data than the L1 cache can hold), instead the second-to-innermost loop is chunked. As mentioned previously, the targeted functions are determined by what functions dominate program execution time and the cache behaviour of the program. Each chunk is treated as a task and an access and execute phase is generated for each chunk, the initial access phase contains prefetches for *all* the data accesses within the chunk. (See figure 2.1 for an illustration of chunk structure.)

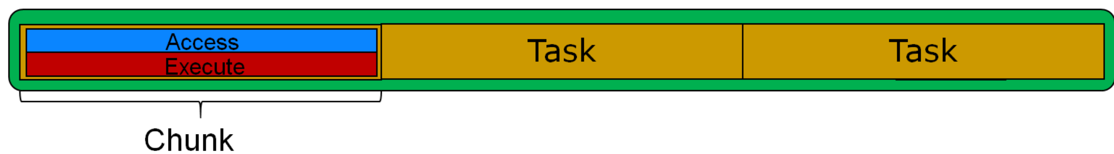


FIGURE 2.1: Illustration of how loop execution is divided into chunks which are treated as tasks, and chunks are further copied into access and execute phases.

Once this version of the code is generated, annotations are added to the prefetch instructions that prefetch addresses expected to be missing in the cache, based on the information obtained by profiling. The version of the code that has annotated instructions is called the "Marked" version of the code.

For the purposes of efficiency, the above steps are done statically before compile time. The unoptimised marked code will always be the same for the program, regardless of architecture or input, and can be generated before running the program. The marking of the code depends on the dynamic information obtained by profiling.

2.3 Optimisation

To remove the unnecessary prefetches and their support code from the "Marked" version of the code, a simple LLVM pass that removes all non-marked prefetches is used. This pass is then followed by the LLVM dead-code-elimination(DCE) pass. Since the access version is completely side-effect free, not storing any result of computations past the duration of the access phase, the DCE pass can remove many unnecessary computations

and branches(See Listing. 2.1 and Listing 2.2 for an example), this results in a lightweight version of the access version. The version of the code that has had unmarked prefetches and unnecessary code removed is called the "pruned" version of the code.

```
define void @prefetch_example(i32 %arg) {
entry:
        %branchselect = icmp sgt i32 %arg, 0
        br i1 %branchselect, label %prefetchbranch, label %funend

prefetchbranch:
        call void @llvm.prefetch(i8* %someaddress, i32 0, i32 0, i32 1)
        br label %funend

funend:
        ret void
}
```

Listing 2.1: Example Code to demonstrate deadcode elimination, note the Prefetch on line 7.

```
define void @prefetch_example(i32 %arg) nounwind readnone {
entry:
  ret void
}
```

Listing 2.2: Example Code to demonstrate DCE, after removing the Prefetch in Listing 2.1 and running a deadcode elimination pass. Note that the branch has been completely removed.

On this pruned version a number of additional test runs were performed to identify the correct granularity values for the chunked loops in the pruned code. For these runs the pruned code was generated offline. In these runs the PAPI interface is used in order to measure the execution time and the energy spent in the access and execute phases of the program. Furthermore a dynamic environment is also emulated, in which the main bus of the computer would be busy with other programs, meaning the hardware prefetcher would prefetch less data thus causing more misses in the L1 cache. To accomplish this, one instance of the program is run on each physical core of the CPU.

2.3.1 JIT design

To generate the pruned version during runtime, it is necessary to optimise and compile the access phases dynamically. This necessitates the use of a JIT engine. In this work I have chosen to work with the LLVM MCJIT engine, since it operates on LLVM IR, can store metadata on IR instructions and can run optimisation passes dynamically. By using JIT callbacks inserted into the code to generate the access phases on demand, and

by compiling the program into a dynamic library to be loaded to the JIT for execution, the overhead of JITing becomes almost only that of optimising and compiling the access phases.

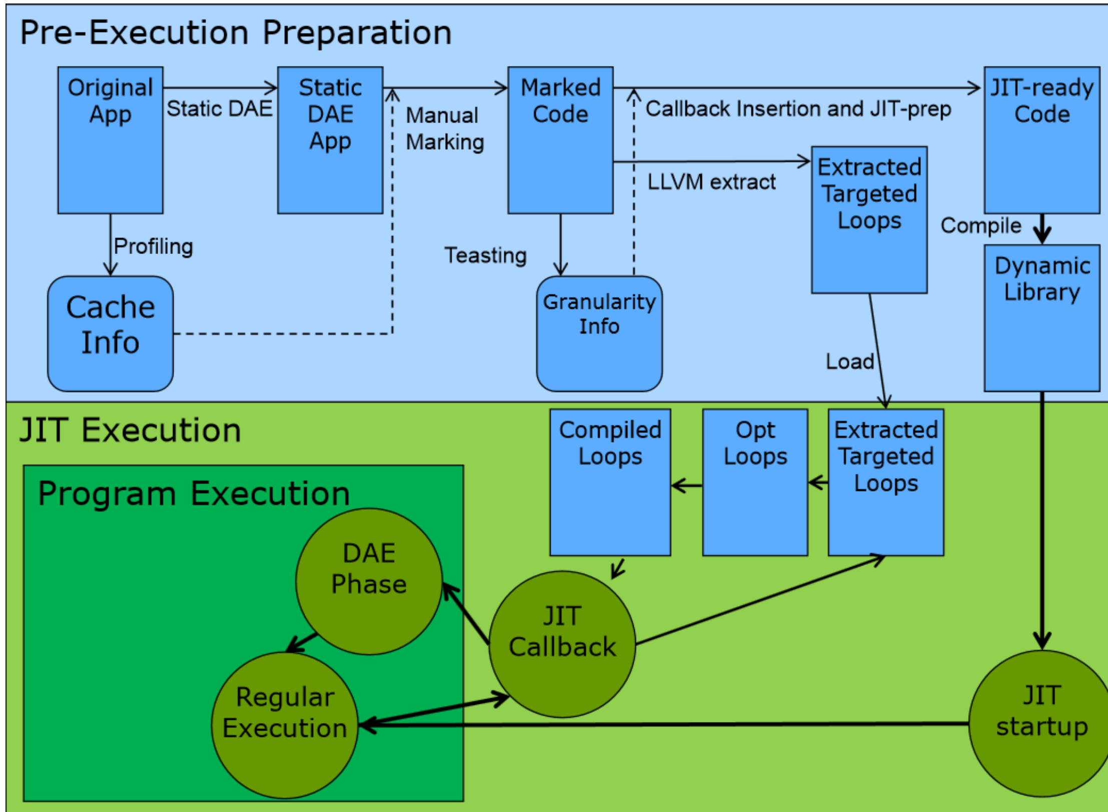


FIGURE 2.2: Flowchart explaining how data is prepared for the JIT, as well as how control flows within the JIT. Squares are Code versions, Circles are stages of execution

The JIT implementation is based on the LLVM MCJIT class. The JIT loads the program to be run as a dynamic library and through callbacks it can generate the Pruned functions from their Marked IR during execution.

In order to reduce the overhead, only the target functions are compiled dynamically, rather than the entire application. This behaviour is achieved by compiling the program into a dynamic library before executing it. Moreover, before compiling to a dynamic library the program must be prepared for the JIT. Since MCJIT only operates on LLVM IR, it cannot modify the code of the dynamic library to setup functionality. To solve this issue, the LLVM IR of the access phase in the targeted loop is extracted using the LLVM-extract tool. Also, the execution of the access phase is replaced by a callback into the JIT, which returns a function pointer that will then be executed to perform the access phase.

On the JIT side, the IR code of the access phase is loaded on startup, at the same time as the dynamic library, which is the main program, is loaded. The "main" function is called with the appropriate input in order to start the execution of the program. Upon receiving a callback, the JIT checks first that it has not resolved the callback for this access phase previously. Upon verifying this, it finds the IR code corresponding to the callback, runs the pass to remove extraneous prefetches together with a DCE pass (thus generation the Pruned version). This version is then JIT compiled to an in-memory executable binary, and a function pointer is stored within the JIT that binds the name of the access phase to name that function pointer, so that in future calls to the same loop it does not need to re-compile the function. Finally it returns the function pointer to the generated access phase, and the program continues execution by calling that function, followed by the execution of the execute version. After the loop is finished, unmodified program execution resumes until another targeted loop is encountered.

2.3.2 JIT analysis

By inserting PAPI callbacks, the cost of modifying and compiling the IR code during execution can be obtained. By inserting one PAPI callback right before the JIT callback and one right after, the time taken for the JIT overhead to setup DAE is shown. The time taken for the JIT to load IR, load the dynamic library, setup its environment and to call the main function is not shown by these timers and is not considered in this work since it should be small. Rather, focus is placed on the JIT overhead for the targeted loops.

2.4 The Power Model

The power model employed to measure the energy consumption is the one described in [8] and [7]. The model approximates power usage based on the metric Instructions per Cycle (IPC) which is collected by the PAPI library. In-depth details are available in [13].

Chapter 3

Related Work

DVFS is a widely used technique to improve program performance[14][15][16][17]. However, the capabilities of DVFS are limited. Koukos et al.[7] suggests the DAE methodology to adapt program structure to better accommodate DVFS. The DAE process was then partially automated by Jimborean et al.[8], which is the basis for most of the methods described in this report.

Cachegrind is a fairly standard tool for analysing programs, used for example by Erdogan and Cao[18] to examine the cache miss ratio of their Virus detecting program and by Kambdadur et al.[19] to examine how their instrumentation affects the cache behaviour. Asaduzzaman and Mahgoub[20] use Cachegrind to examine program behaviour to perform optimisations to programs on mobile devices.

Using a JIT compiler to perform runtime optimisations is a well known possibility[21–23]. Dynamic optimisation is employed by Suganuma et al.[21] when implementing a JIT compiler for the Java language. Dynamic inlining using a JIT is demonstrated by Arnold et al.[22] in their paper about feedback-directed optimisations. Similarly to the end goal of online DAE compiling, Bala et al[23] use dynamic profiling to perform optimisations while running a program through a JIT compiler.

Chapter 4

Results

4.1 Benchmark Selection

To examine the possible gains of optimised DAE, the methods in chapter 2 were applied to a selection of benchmarks from the SPEC CPU 2006 suite[24]. Benchmarks are selected based on information about memory boundness, meaning the number of load instructions in the benchmark, and CPI(Cycles per Instruction) for the benchmarks[25–27]. Both memory bound and compute bound benchmarks were chosen, to see clearly the difference in performance.

The benchmarks are always ran with a given input, and in this work the architecture that the programs are run on is know. In general both a high amount of L1 cache misses and a high CPI are indicative that a benchmark is memory bound. The benchmarks are selected based on previous work[11, 26, 28] but the predictions of behavior are based on the test results for the architecture used in this work.

The benchmarks that were selected are:

- 429.mcf
- 433.milc
- 450.soplex
- 456.hmmer

- 462.libquantum
- 470.lbm
- 473.astar

Benchmarks that were selected and judged as compute bound are marked in blue, benchmarks that seemed memory bound are marked in red. For the rest of the report these benchmarks will usually be referred to without their numeral prefixes.

The source code of, and the inputs for the benchmarks were extracted from the documentation of the benchmark suite. The benchmarks are compiled and run with either the REF or TRAIN inputs. Which inputs were used is detailed with the results presented.

4.2 Architecture

All tests were ran under Ubuntu 13.04 x86_64 on an Intel machine with 4 physical cores, 8 logical cores and 32KB L1 memory with CPU frequency range 1600 MHz to 3200 MHz.

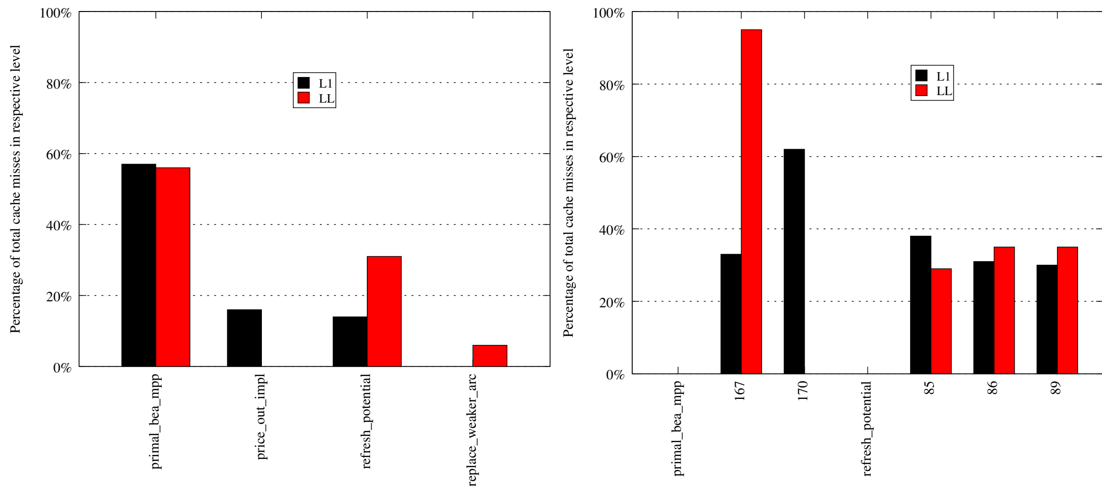
4.3 Offline Profiling

Using Cachegrind information was collected about how much each instruction contributes to the overall cache misses of the benchmark, and in which functions these instructions are. By examining the concentration of the cache misses by function in each benchmark it can be determined how suitable the benchmark is for dynamic optimisation and DAE, as well as which functions within the benchmark to target for loop chunking. The more concentrated within single functions and loops the L1 cache misses of are, the more noticeable the effects of the optimisation. It is also relevant whether the functions containing a large amount of cache misses are the most time consuming functions in the program[11].

Here is the analyses of cache misses, by benchmark, under REF inputs. In the figures, the L1 and LL cache misses are shown either by function or by line of code. In the cases where it is by line of code, it is delimited by the function this line is in, and all line

numbers are relative to the source code for the benchmark and lines that contribute less than 5% of the total cache misses within their function are not shown. Similarly for functions, functions that contribute less than 5% of total cache misses are not shown.

4.3.1 mcf



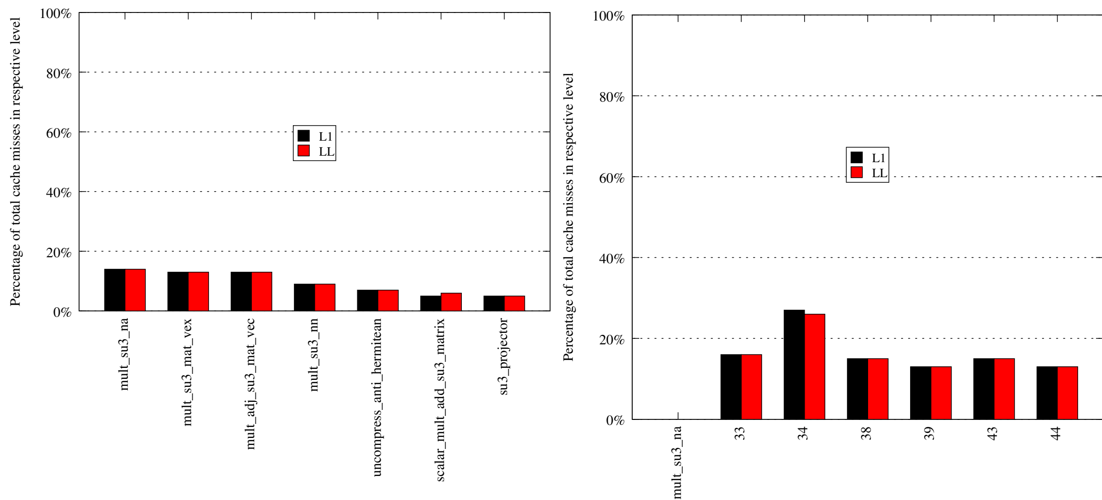
(A) The L1 and LL cache misses per function, as a percentage of total misses.

(B) The L1 and LL cache misses per code line, delimited by function, as a percentage of total misses within that function.

FIGURE 4.1: The cache misses for the mcf benchmark organised by function or line.

The cache profile, described as a the contribution to overall cache misses per function (seen in Fig. 4.1a) shows that primal_bea_mpp and refresh_potential are the functions which are both incurring a large amount of misses and dominate the execution time of the program, so these functions are suitable to target for loop chunking. The cache profile of these functions (seen in Fig. 4.1b) shows that the misses in primal_bea_mpp comes from just two lines, totalling 54% of total program cache misses. This means the effect of optimisation on that function should be large. Refresh_potential is spread out over three instructions, meaning the benefits of optimising it will be smaller. All the listed lines are prefetched.

4.3.2 milc



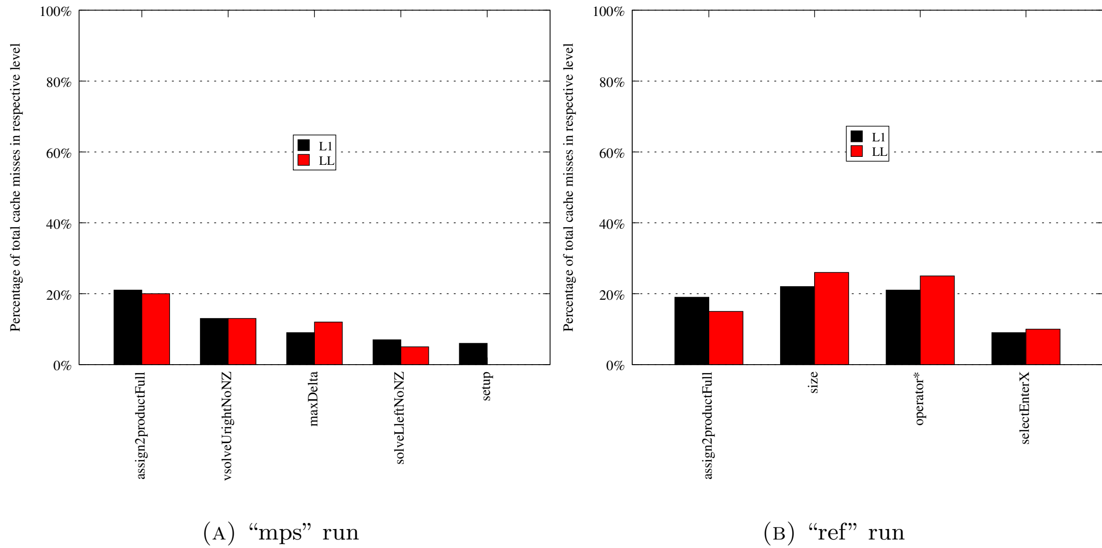
(A) The L1 and LL cache misses per function, as a percentage of total misses.

(B) The L1 and LL cache misses per code line, delimited by function, as a percentage of total misses within that function.

FIGURE 4.2: The cache misses for the milc benchmark organised by function or line.

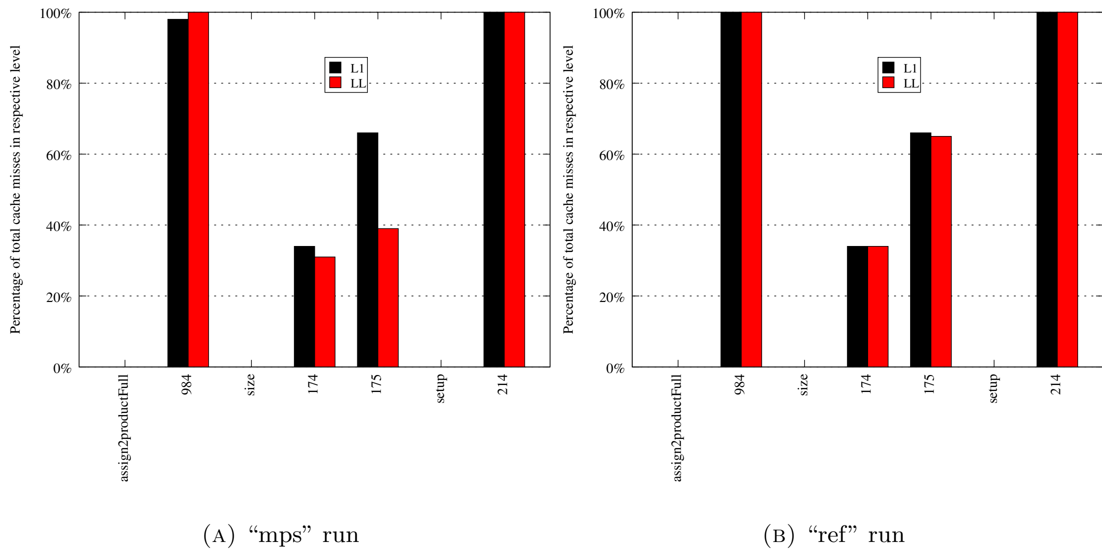
The cache profile, described as the contribution to overall cache misses per function (seen in Fig. 4.2a) shows that misses are concentrated in the functions that dominate program execution time. The spread of cache misses over several functions makes the benchmark not very well suited for dynamic optimisation as multiple access phases would be compiled dynamically. I choose to only target mult_su3_na, it's cache profile (seen in Fig. 4.2b) shows that within the function the misses are also very spread out, meaning the effects of optimising the access phase will be small. All the listed lines are prefetched.

4.3.3 soplex



(A) “mps” run (B) “ref” run

FIGURE 4.3: The L1 and LL cache misses for the soplex benchmark per function, as a percentage of total misses for both the “mps” and the “ref” run.



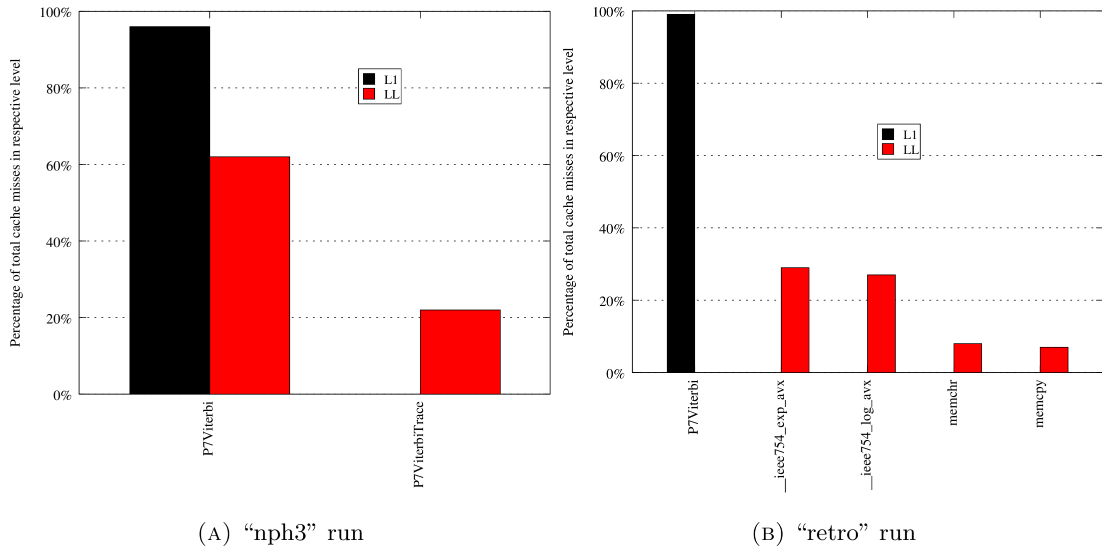
(A) “mps” run (B) “ref” run

FIGURE 4.4: The L1 and LL cache misses for the soplex benchmark per code line, delimited by function, as a percentage of total misses within that function for both the “mps” and the “ref” run.

The soplex benchmark is usually run two times with different inputs[29]. The cache profile for each of these runs, described as the contribution to overall cache misses per function (seen in Fig. 4.3) shows that the only function that contributes largely to the cache misses of both runs is assign2productFull. Based on what functions dominate execution time, it makes sense to target that function, as well as the functions size

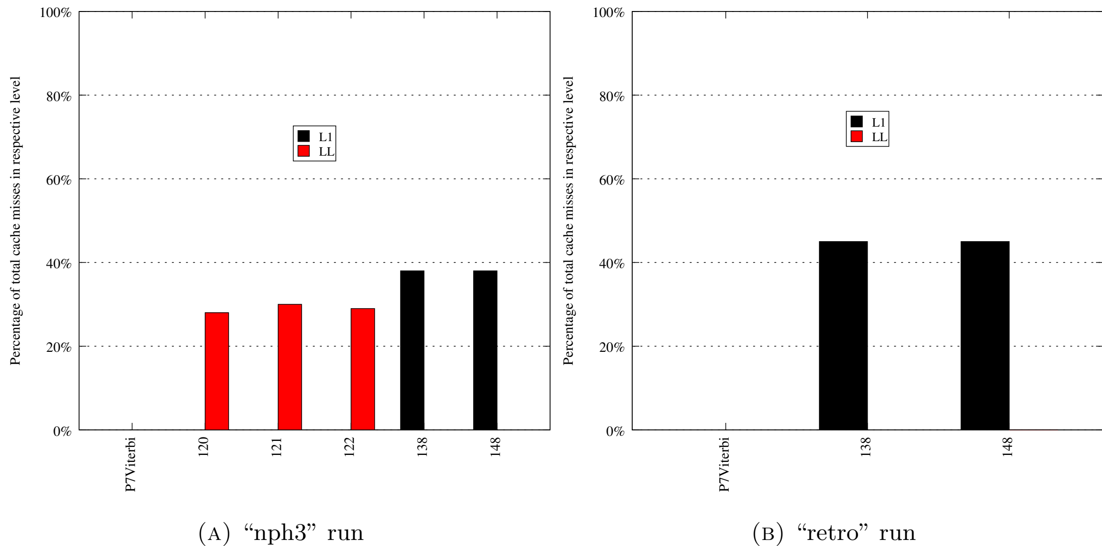
and setup. The cache profiles of the functions (seen in Fig. 4.4) shows that the cache misses they generate are concentrated to a few lines of code, meaning that the effects of optimisation should be noticeable. All the listed lines are prefetched.

4.3.4 hmmer



(A) "nph3" run (B) "retro" run

FIGURE 4.5: The L1 and LL cache misses for the hmmer benchmark per function for both "nph3" and "retro" run, as a percentage of total misses.



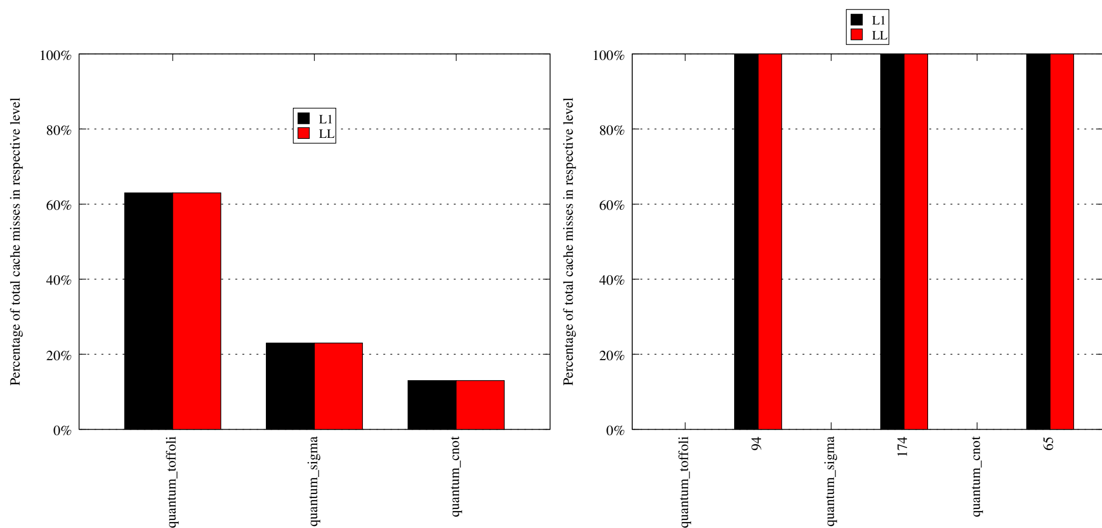
(A) "nph3" run (B) "retro" run

FIGURE 4.6: The L1 and LL cache misses for the hmmer benchmark per code line, delimited by function, as a percentage of total misses within that function.

The hmmer benchmark is usually run two times with different inputs[29]. The cache profile for each of these runs, described as a the contribution to overall cache misses per

function (seen in Fig. 4.5) shows that the function P7Viterbi dominates the L1 cache misses in both runs. This function also dominates program execution time, thus that function is targeted. The cache profile for P7Viterbi (seen in Fig. 4.6) shows that the L1 cache misses are concentrated on just two lines of code, meaning that the effects of optimisation should be noticeable. Lines 138 and 148 are prefetched.

4.3.5 libquantum



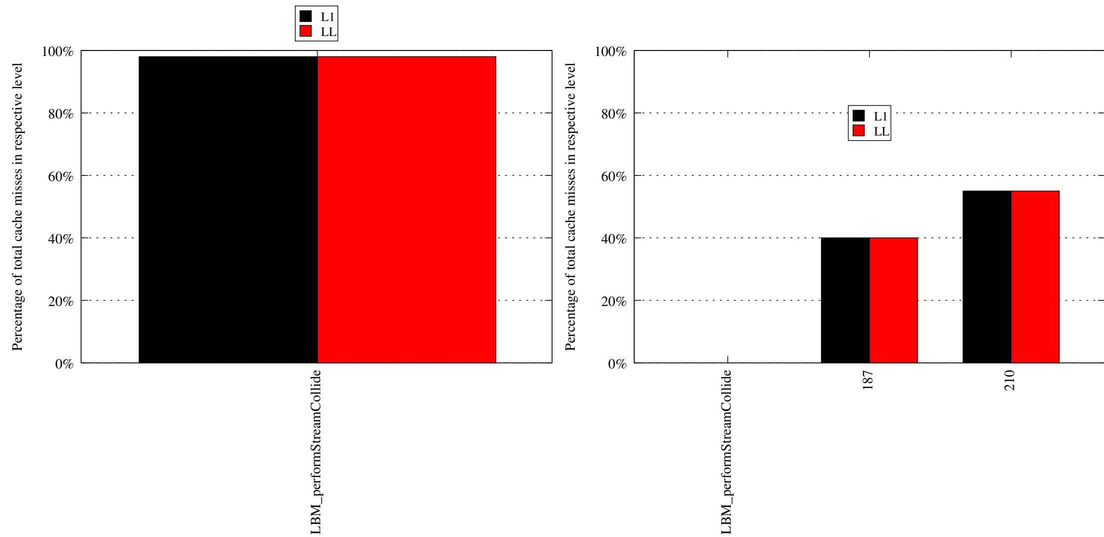
(A) The L1 and LL cache misses per function, as a percentage of total misses.

(B) The L1 and LL cache misses per code line, delimited by function, as a percentage of total misses within that function.

FIGURE 4.7: The cache misses for the libquantum benchmark organised by function or line.

The cache profile, described as the contribution to overall cache misses per function (seen in Fig. 4.7a) shows that the cache misses are concentrated within the functions that dominate the execution time of the program. The misses are centred around three functions, which is not entirely optimal for dynamic optimisation as there would be multiple access phases to compile dynamically. Nevertheless quantum_toffoli, quantum_sigma and quantum_cnot are suitable targets. The cache profiles of these functions (seen in Fig. 4.7b) shows that within the functions the misses are concentrated to just one line in each function, meaning the effect of dynamic optimisation should be noticeable. All the listed lines are prefetched.

4.3.6 lbm



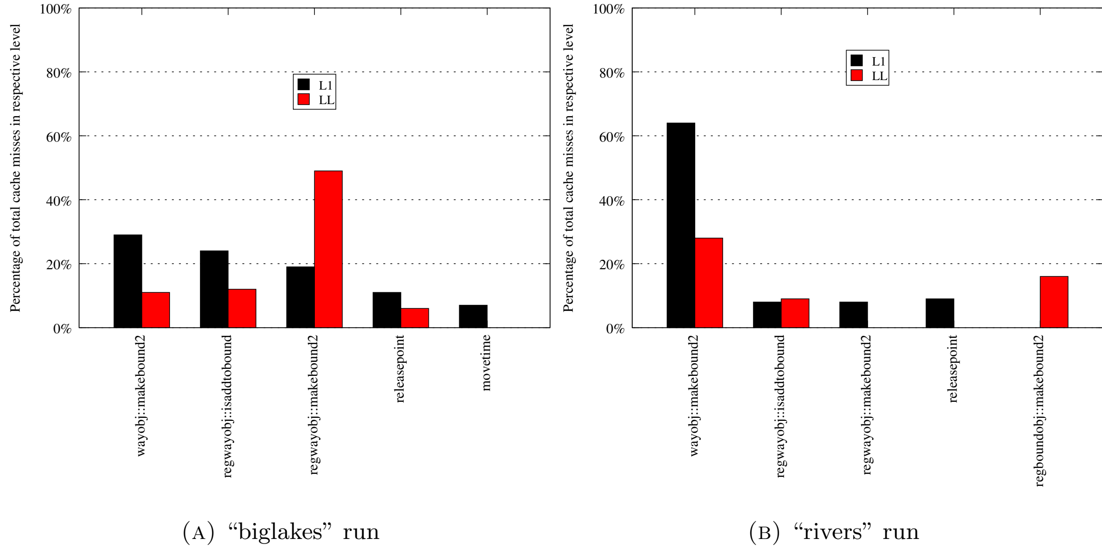
(A) The L1 and LL cache misses per function, as a percentage of total misses.

(B) The L1 and last LL cache misses per code line, delimited by function, as a percentage of total misses within that function.

FIGURE 4.8: The cache misses for the lbm benchmark organised by function or line.

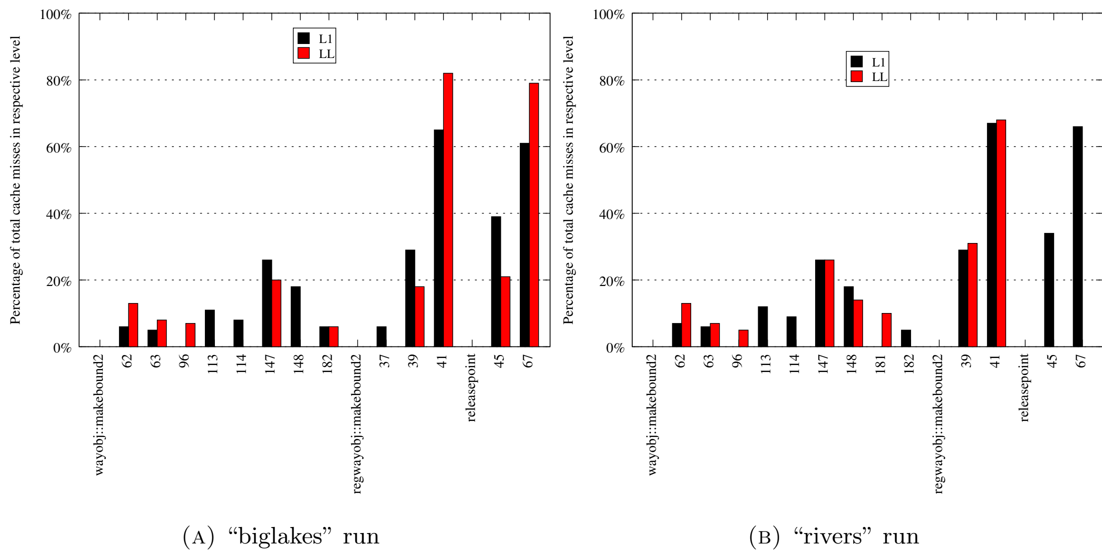
The cache profile, described as the contribution to overall cache misses per function (seen in Fig. 4.8a) shows that the majority of cache misses stem from just the one function, which also dominates program execution time. This is ideal for dynamic optimisation and this function is targeted, LBM_performStreamCollide. Its cache profile (seen in Fig. 4.8b) shows that within the function the misses are concentrated in just two lines, meaning that the effect of dynamic optimisation should be high. Both of these lines are prefetched.

4.3.7 astar



(A) “biglakes” run (B) “rivers” run

Figure 4.9: The L1 and last LL cache misses for the astar benchmark per function for both “biglakes” and “rivers” run, as a percentage of total misses.



(A) “biglakes” run (B) “rivers” run

Figure 4.10: The L1 and LL cache misses for the astar benchmark per code line, delimited by function, as a percentage of total misses within that function.

The astar benchmark is usually run two times with different inputs[29]. The cache profile for each of these runs, describing the distribution of cache misses per function (illustrated in Fig. 4.9), shows that the cache misses are fairly spread out over several functions. This means that the benchmark is not entirely suitable for a dynamic optimisation as there would be several access phases to dynamically compile. Because of

in-lining issues, it is difficult to determine which functions dominate the execution time of the program. However, by examining the post-optimisation intermediate representation of the code, I have determined that at least the functions wayobj::makebound2, regwayobj::makebound2 releasepoint are suitable targets. The cache profiles for these functions (Fig. 4.10) show that the L1 cache misses are fairly spread out within wayobj::makebound2 and fairly concentrated in regwayobj::makebound2 and releasepoint, meaning the effects of optimisation are expected to be high for the later two but less noticeable for the first.. The lines 147, 148, 39, 41, 45 and 67 are prefetched.

4.4 Granularity Testing

As explained in Chapter 1, the execution of a function that is optimised under DAE is divided into several "chunks", centred around a loop. The granularity represents the number of iterations of the loop executed in each chunk. Using the methods in Chapter 2, information was obtained regarding how much time each phase (access or execute) of the loop takes for varying granularity values. Together with the power model this information is also used to estimate how much energy is used in each phase of the loop. These tests are run using the TRAIN inputs of the benchmarks.

To simulate the effects of DVFS, each granularity is run twice, once under maximum CPU frequency and once under minimum CPU frequency. The time taken for each phase then uses the data for the run that had the frequency that would be used during that phase if DVFS was used, meaning that the access phase of the loop is timed under the minimum frequency and the execute phase of the loop is timed under the maximum frequency. The total time and energy is then calculated from the sum of each of the phases time and energy. For each run the dynamic energy is given by the formula: $E = T * C * F * V^2$ [1] T is measured by the library, F is set. C and V are fed to the power model[13].

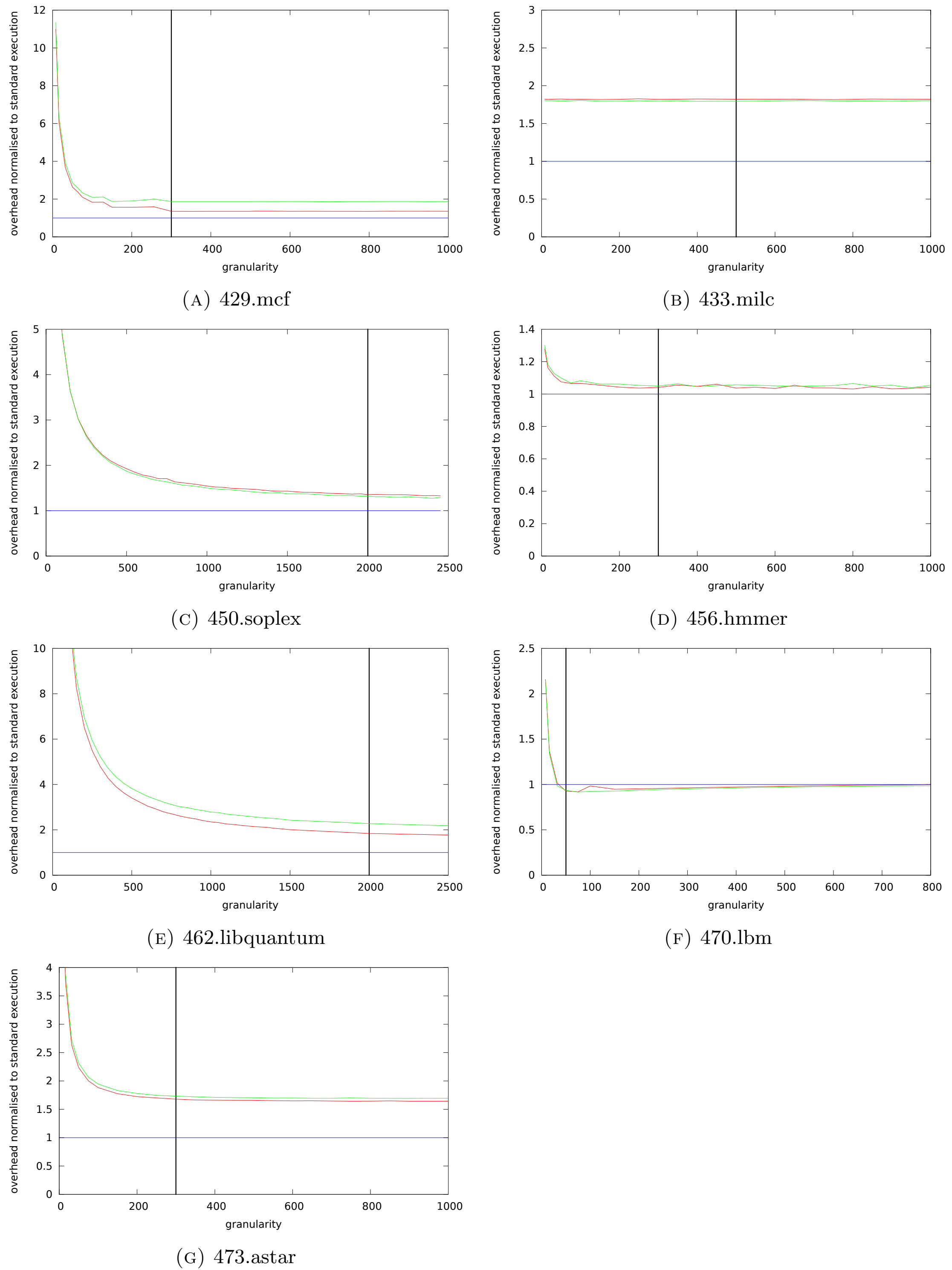
To better simulate a dynamic environment in which the benchmark would be one of many programs executing on the computer, the main bus of the computer must be made busy to mitigate the effects of the hardware prefetcher. This is done by running one instance of the program on each physical CPU core, which saturates the bus.

[1] Where E is energy, T is time, C is Capacitance, F is CPU frequency and V is CPU voltage

The results of the tests can be seen in Fig. 4.11 and Fig. 4.12, in these figures the data for three versions of each program are shown:

- red: The pruned version of the program
- green: The statically compiled DAE version of the program
- blue: The coupled access-execute version of the program, meaning the regular execution

In the graphs, the data for the pruned and statically compiled version are normalised to the execution of the regular program. The vertical line marks the granularity that was used in the later benchmark tests, where the performance of the JIT is tested.



(A) 429.mcf

(B) 433.milc

(C) 450.soplex

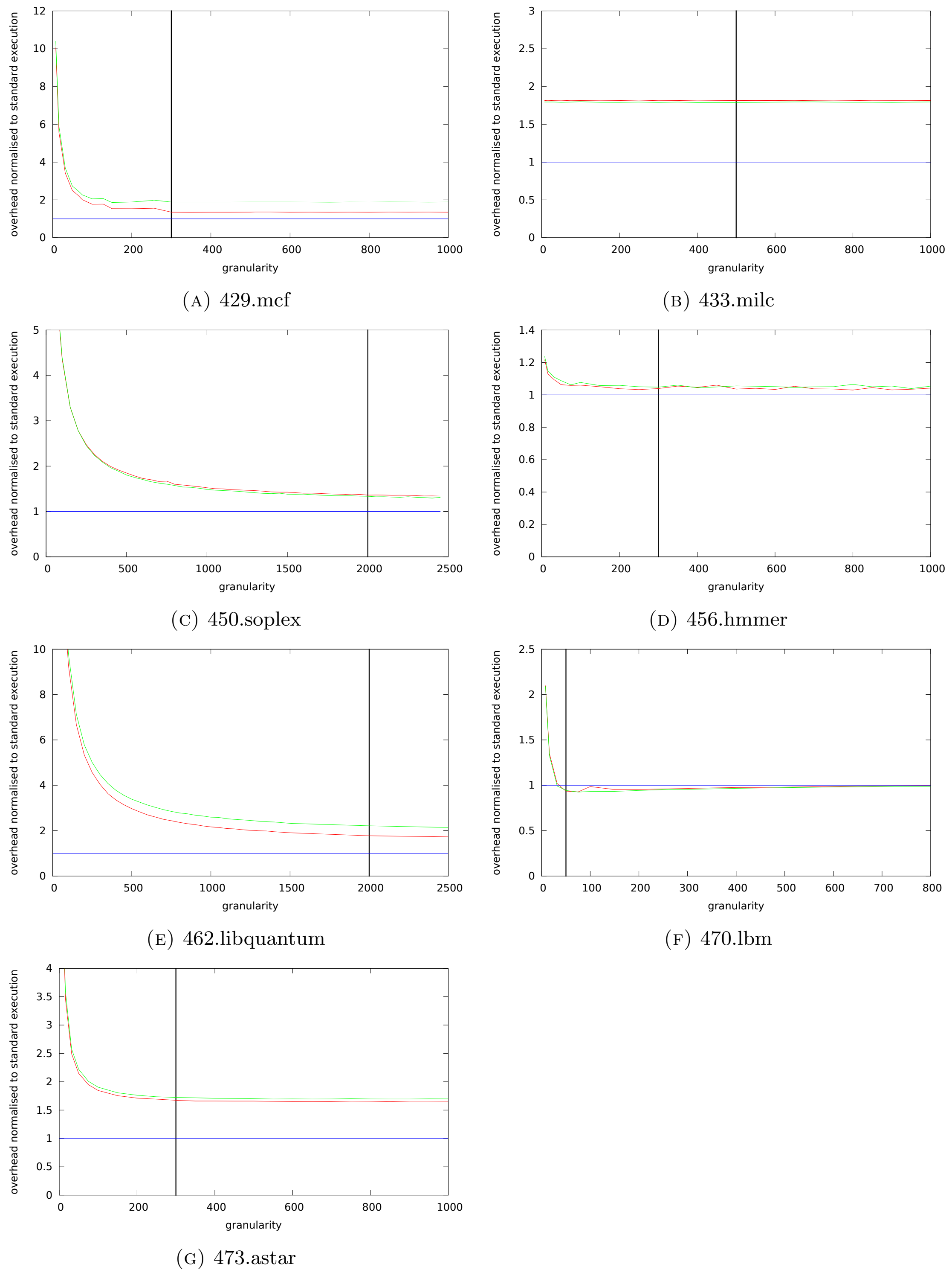
(D) 456.hmmer

(E) 462.libquantum

(F) 470.lbm

(G) 473.astar

FIGURE 4.11: The total time taken for the loops using different granularity, sorted by benchmark.



(A) 429.mcf

(B) 433.milc

(C) 450.soplex

(D) 456.hmmer

(E) 462.libquantum

(F) 470.lbm

(G) 473.astar

FIGURE 4.12: The total energy used by the loops using varying granularity of chunks.

4.5 Evaluation

Based on the results in Fig. 4.11 and Fig. 4.12 the optimal granularity for the loops was determined, such that the workload of each chunk fits in the L1 cache. To examine the performance of the dynamic optimisation and compilation the program is run twice using different frequency, similarly to 4.4. For programs with multiple inputs, the result is the sum of the results of the inputs. The JIT overhead is calculated from the time it took under the maximum frequency run.

Fig. 4.13 shows the time and energy taken during loop the execution for three versions of the program:

1. DAE with JIT;
2. DAE compiled statically;
3. original code (coupled execution).

The first two executions *(1)* and *(2)* show the overhead of JIT-ing, as well as the benefit of dynamic optimisation as seen by a reduction in time taken by the access phase. The execution of the original code compared to the optimised version shows the overall, benefits (or overhead) of the dynamically optimised decoupled execution.

The expected results for compute bound benchmarks were that there would be a small overhead in time taken and no benefit in energy consumption. For memory bound benchmarks, the expectation was that there would be a small overhead in time taken and some benefit in energy consumption. In general the time overhead is small for all benchmarks, but the energy benefits for memory bound benchmarks is not always as expected.

For the milc benchmark there is an enormous JIT-ing overhead, which overshadows an otherwise mostly breakeven performance of DAE. When comparing the initial non-optimised version of the DAE code to the original execution there is a small energy benefit and a small time overhead, as expected. The lack of change to this behaviour after optimisation, coupled with the strangely large JIT overhead, indicates that the benchmark is suitable for DAE optimisation but unsuitable for dynamic optimisation.

In the case of soplex there is a slight overhead both in time and energy when comparing the optimised execution to the regular. However when comparing the initial version with the regular version there is a break-even in energy and a slight time overhead. This, coupled with that there is no discernable improvement in the access phase behaviour from the non-optimised to the optimised version, leads me to believe that the soplex benchmark should be suitable for dynamic optimisation with better heuristics for how to optimise the acces phase. The JIT overhead is for this benchmark small and should thus be of little concern.

LBM behaviour is excellent, with an extremely lightweight access phase the energy cost of the program is not only reduced, but the program is also sped up. The JIT overhead for this benchmark is large, so improving JIT performance would further increase the gains of optimised DAE on this benchmark.

Interestingly, while there are no energy improvements measured for the compute bound benchmark in this work, mcf and libquantum show improvements when comparing the optimised DAE version to the original execution when the overhead of the JIT is ignored. Meaning that they would show improvements if the JIT overhead was reduced. This indicates that optimised DAE may be suitable for compute bound benchmarks *if there are loops within the benchmark that are sufficiently memory bound*, this is however not further investigated in this paper.

Recall that the expected effect of DAE and dynamic optimisation on benchmarks were initially based on whether the benchmark was memory bound or not, these expectations were further refined by examining the CPI, the L1 cache miss amounts and the number of instructions within critical functions that dominate the L1 cache misses. There are however some other factors that affect the effectiveness of dynamically optimised DAE; the code size of targeted functions, the reuse of memory locations, the number of targets and the reuse of targets.

The code size of a targeted program impacts the overhead of the JIT, a larger code means more code to scan for prefetches to remove when optimising, more code to analyse when running DCE and more code to compile before the function is returned. [2]

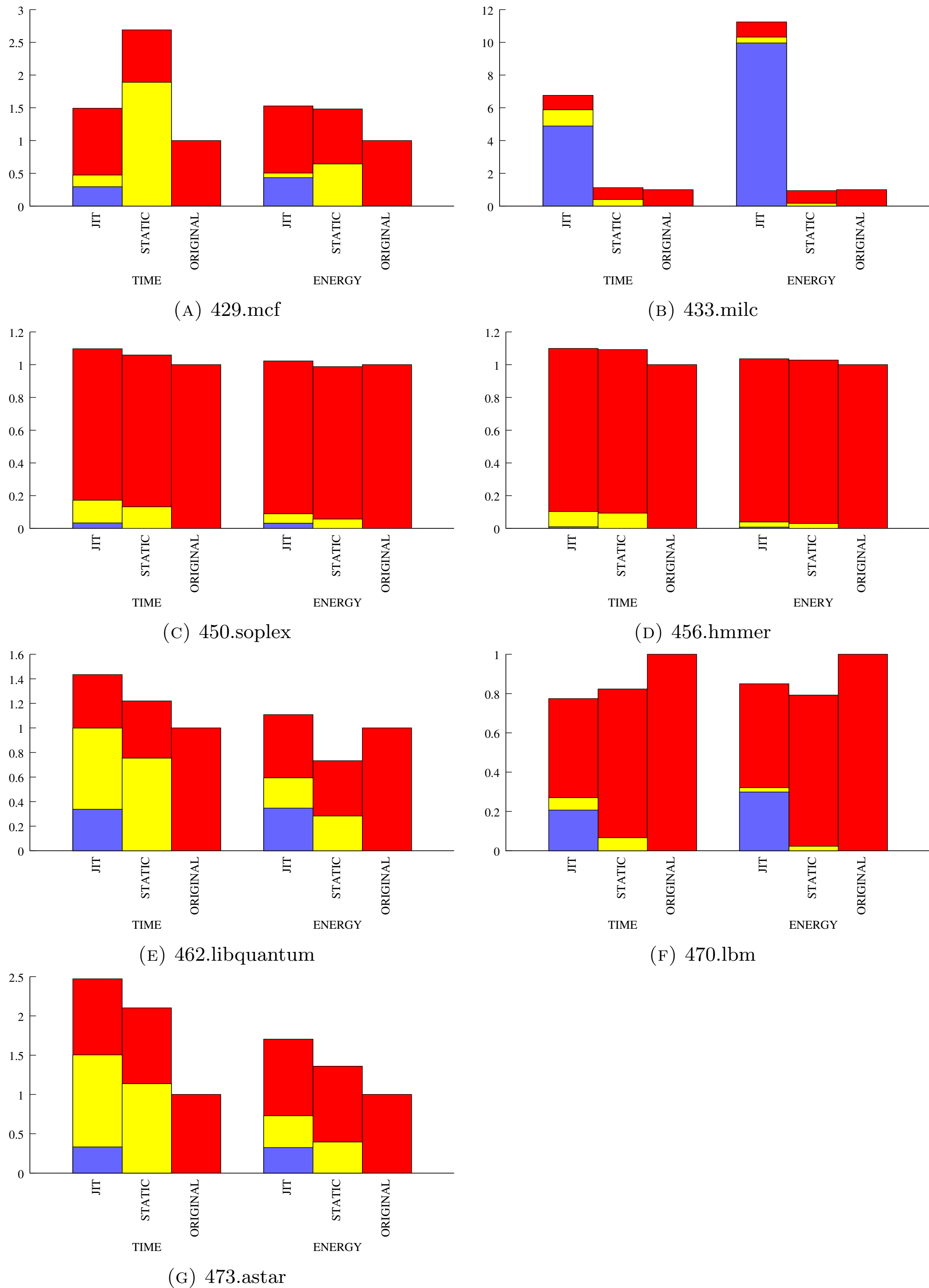
[2]This does not explain the exceedingly large overhead of JIT-ing in the milc benchmark, the target code in the milc benchmark is not larger by any significant margin than the other benchmarks and it does not contain an above average amount of targets

The reuse of memory locations impacts the overhead of the prefetch phase. If the targeted function has a large reuse of memory accesses that are selected for prefetching it will be sufficient to prefetch these locations once in the prefetch phase, meaning one prefetch in the prefetch phase can correspond to several loads in the execute phase.

Similarly, the number of targeted loops increases the number of JIT callbacks. Each JIT callback necessitates the use of two more optimisation passes and one more compilation.

As with memory locations, reusing a loop target reduces the overhead of JIT-ing compared to the benefits. Because the JIT stores the compiled access phase for future use, the more times the targeted loop is executed, and the more chunks there are in that loop, the better the ratio of benefits per compile time.

It is not determined in this work how much each of these factors impact the benefits or overhead of DAE or JIT-ing.



(A) 429.mcf

(B) 433.milc

(C) 450.soplex

(D) 456.hmmer

(E) 462.libquantum

(F) 470.lbm

(G) 473.astar

FIGURE 4.13: The total Time and Energy taken by the chunks in each benchmark, when run either through the JITing process (JIT), the static DAE process (STATIC) or the original execution (ORIGINAL). Results for time and Energy are divided up into sections, based on what phase of the chunk incurred that part. Data for energy is normalised to the energy taken by the original execution, and similarly the time taken is normalised to the time taken by the original execution.

- red: Regular Computation (Execute phase)
- yellow: Prefetch Phase (Access phase)
- blue: JIT callbacks (Optimisation and Compilation phase)

Chapter 5

Conclusions

5.1 Conclusions & Future Work

For memory bound benchmarks, there was an improvement in performance when disregarding the JIT overhead. The only benchmark that showed improvements when including the JIT overhead was LBM. For many benchmarks there was no improvement for access phase performance when comparing the optimised access phase to the static access phase. This likely means that the addresses for the prefetches that were kept in the access phase were too expensive to calculate. Some heuristic of calculation time versus cache misses could potentially show more accurately what prefetches would be worth keeping, compared to the method used in this work which only eliminated the prefetches that were expected to not benefit the execute phase. Depending on the L1 cache miss distribution of the benchmarks, there may be an optimisation of the access phase that would result in a more efficient access phase while retaining most of the benefits in the execute phase.

The overhead of DAE on compute-bound benchmarks is in general quite small, and it seems possible to obtain benefits for compute-bound benchmarks given that the targeted loops inside the benchmark are sufficiently memory bound.

Overhead of the JIT is for most benchmarks rather high, which in the cases of the more memory bound out-weights the benefits of optimising the access phase in some cases. Meaning that reducing the overhead of the JIT would be desirable. To do this one could, for example, merge the two optimisation passes (See Chapter 2.3.1) into one. The DCE

pass could then use the knowledge about which prefetching instructions were removed to more quickly decide which other instructions should be removed as a result. Currently the JIT is using the LLVM MCJIT engine, which is known to be slow. Improving the compilation and optimisation pass speed of the JIT would reduce the overhead further.

To further extend on the work presented in this report, it would be desirable to move the profiling of the application into the online environment and actually obtain the dynamic information dynamically. A way to do this would be to, similarly to this work, chunk the loops statically, but change their behaviour slightly. Rather than call-backing the JIT immediately for the access phase in the first chunk, the first chunk or the first few chunks would be profiling runs that determine what loads in the loop iterations miss the most. This information is then fed into the JIT which optimises and compiles the access phases for the remaining chunks.

This approach would grant the following additional benefits:

- Adapt to any program input or architecture change automatically.
- Possibly perform program work while profiling, meaning the overhead would only be the measurement, the profiling would not cause repeated calculations.
- Possibility to adapt granularity of chunks on the fly, by measuring performance of chunks during runtime
- Un-DAE. If the measured performance is too bad or the profiling chunks indicate it, it would be possible to execute the remaining chunks as regular coupled execution. Thus it would be possible to skip the DAE-ing of individual loops for inputs or environments where they would be inefficient. Some overhead would remain in such cases however, as profiling or in some cases even optimisation and compiling may still have been done.

The parts of the approach that would remain static would be:

- Deciding what loops in the program to chunk. This is likely to be the same irregardless of input or environment, and deciding these dynamically would incur a large unnecessary overhead.

- Generating the initial chunked version. The initial DAE transformations would be the same for each loop regardless of dynamic factors, so there is no reason to generate them dynamically.

- Preparing the code for the JIT. Similarly to the previous point, there is no reason to do this dynamically since it will always be done in the same way.

Bibliography

[1] Qiang Wu, Margaret Martonosi, Douglas W. Clark, V. J. Reddi, Dan Connors, Youfeng Wu, Jin Lee, and David Brooks. A dynamic compilation framework for controlling microprocessor energy and performance. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 271–282, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2440-0. doi: 10.1109/MICRO.2005.7. URL `http://dx.doi.org/10.1109/MICRO.2005.7`.

[2] Qiang Wu, Margaret Martonosi, Douglas W Clark, Vijay Janapa Reddi, Dan Connors, Youfeng Wu, Jin Lee, and David Brooks. Dynamic-compiler-driven control for microprocessor energy and performance. *IEEE Micro*, 26(1):119–129, 2006.

[3] Chung-Hsing Hsu and Ulrich Kremer. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. *SIGPLAN Not.*, 38(5):38–48, May 2003. ISSN 0362-1340. doi: 10.1145/780822.781137. URL `http://doi.acm.org/10.1145/780822.781137`.

[4] Jacob R Lorch, Alan Jay Smith, et al. Improving dynamic voltage scaling algorithms with PACE. In *SIGMETRICS/Performance*, pages 50–61. Citeseer, 2001.

[5] Tajana Simunic, Luca Benini, Andrea Acquaviva, Peter Glynn, and Giovanni De Micheli. Dynamic voltage scaling and power management for portable systems. In *Proceedings of the 38th Annual Design Automation Conference*, DAC ’01, pages 524–529, New York, NY, USA, 2001. ACM. ISBN 1-58113-297-2. doi: 10.1145/378239.379016. URL `http://doi.acm.org/10.1145/378239.379016`.

[6] V. Spiliopoulos, S. Kaxiras, and G. Keramidas. Green governors: A framework for continuously adaptive DVFS. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1–8, July 2011. doi: 10.1109/IGCC.2011.6008552.

[7] Konstantinos Koukos, David. Black-Schaffer, V. Spiliopoulos, and Stefanos. Kaxiras. Towards more efficient execution: a decoupled access-execute approach. In *Proc. 27th ACM International Conferemce on Supercomputing*, 2013.

[8] Alexandra Jimborean, Konstantinos Koukos, Vasileios Spiliopoulos, David Black-Schaffer, and Stefanos Kaxiras. Fix the code. don't tweak the hardware: A new compiler approach to voltage-frequency scaling. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 262:262–262:272, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2670-4. doi: 10.1145/2544137.2544161. URL `http://doi.acm.org/10.1145/2544137.2544161`.

[9] Cachegrind: a cache and branch-prediction profiler. http://valgrind.org/docs/manual/cg-manual.html, 2014.

[10] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007. ISSN 0362-1340. doi: 10.1145/1273442.1250746. URL `http://doi.acm.org/10.1145/1273442.1250746`.

[11] Function profile. `http://hpc.cs.tsinghua.edu.cn/research/cluster/SPEC2006Characterization/fprof.html`, 2013. Accessed 13/5 2014.

[12] Josef Weidendorfer. KCachegrind. http://kcachegrind.sourceforge.net/html/Home.html, 2012.

[13] Vasileios Spiliopoulos, Andreas Sembrant, and Stefanos Kaxiras. Power-sleuth: A tool for investigating your program's power behaviour. *2012 IEEE 20th International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, 0:241–250, 2012.

[14] Kim Wonyoung, M.S Gupta, Gu-Teon Wei, and D Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *High Performance*

Computer Architecture, 2008. HPCA 2008. IEEE 14 International Symposium on, 2008.

[15] Liu Yongpan, Yang Huazhong, R.P Dick, and Hui Wang. Thermal vs energy optimisation for DVFS-enabled processors in embedded systems. In *Quality Electronic Design, 2007. ISQED '07. 8th International Symposium on*, 2007.

[16] Wang Lizhe, G. von Laszeqski, J Dayal, and Fugal Wang. Towards energy aware scheduling for precedence constrained parallel tasks in a cluster with dvfs. In *Cluster, Cloud and Grid Computing (CCGRID), 2010 10th IEEE/ACM International Conference on*, 2010.

[17] M Bao, A Andrei, P Eles, and Z Peng. Temperature-aware task mapping for energy optimization with dynamic voltage scaling. In *Design and Diagnostics of Electronic Circuits and Systems, 2008. DDECS 2008. 11th IEEE Workshop on*, 2008.

[18] Ozgun Erdogan and Pei Cao. Hash-AV fast virus signature scanning by cache-resident filters. *Int. J. Secur. Netw.*, 2(1/2):50–59, March 2007. ISSN 1747-8405. doi: 10.1504/IJSN.2007.012824. URL `http://dx.doi.org/10.1504/IJSN.2007.012824`.

[19] Melanie Kambadur, Kui Tang, and Martha A. Kim. Harmony: Collection and analysis of parallel block vectors. *SIGARCH Comput. Archit. News*, 40(3): 452–463, June 2012. ISSN 0163-5964. doi: 10.1145/2366231.2337211. URL `http://doi.acm.org/10.1145/2366231.2337211`.

[20] Abu Asaduzzaman and Imad Mahgoub. Cache modeling and optimization for portable devices running mpeg-4 video decoder. *Multimedia Tools Appl.*, 28(1): 239–256, January 2006. ISSN 1380-7501. doi: 10.1007/s11042-006-6145-y. URL `http://dx.doi.org/10.1007/s11042-006-6145-y`.

[21] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A dynamic optimization framework for a java just-in-time compiler. *SIGPLAN Not.*, 36(11):180–195, October 2001. ISSN 0362-1340. doi: 10.1145/504311.504296. URL `http://doi.acm.org/10.1145/504311.504296`.

[22] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the jalapeño jvm. *SIGPLAN Not.*, 35(10):

47–65, October 2000. ISSN 0362-1340. doi: 10.1145/354222.353175. URL
http://doi.acm.org/10.1145/354222.353175.

[23] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. *SIGPLAN Not.*, 35(5):1–12, May 2000. ISSN 0362-1340. doi: 10.1145/358438.349303. URL
http://doi.acm.org/10.1145/358438.349303.

[24] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006. ISSN 0163-5964. doi: 10.1145/1186736.1186737. URL
http://doi.acm.org/10.1145/1186736.1186737.

[25] AA Nair and L.K. John. Simulation points for SPEC CPU 2006. In *Computer Design, 2008. ICCD 2008. IEEE International Conference on*, pages 397–403, Oct 2008. doi: 10.1109/ICCD.2008.4751891.

[26] Tribuvan Kumar Prakash and Lu Peng. Performance characterization of SPEC CPU2006 benchmarks on intel core 2 duo processor. *ISAST Trans. Comput. Softw. Eng*, 2(1):36–41, 2008.

[27] Aamer Jaleel. Memory characterization of workloads using instrumentation-driven simulation. *Web Copy: http://www. glue. umd. edu/ajaleel/workload*, 2010.

[28] A.R Lebeck and D.A. Wood. Cache profiling and the SPEC benchmarks: a case study. *Computer*, 27:15–26, October 1994.

[29] Kenneth Hoste. SPEC CPU2006 command lines.
http://kejo.be/ELIS/spec_cpu2006/spec_cpu2006_command_lines.html, 2010. Accessed 6/5 2014.