Static Multi-Versioning for Efficient Prefetching

Per Ekemark
Abstract

Static Multi-Versioning for Efficient Prefetching

Per Ekemark

Energy efficiency is one of the biggest challenges in modern computer architecture. Increased performance and improved energy efficiency is always in demand whether it is for battery longevity in mobile applications or thermal limits in high-performance computing. To reach the best result, hardware and software must compliment each other. Software Decoupled Access-Execute (DAE) is a technique where memory operations are rearranged and grouped together by the compiler. Larger sections of memory bound code allows more efficient use of hardware Dynamic Voltage and Frequency Scaling (DVFS), which can save energy without affecting performance. While previous work in automatically generated software DAE has used a one size fits all approach, this work adds a parametrisation aspect, making it possible to explore multiple versions of optimised code. Given a parameter, a heuristic can scale how aggressively DAE is applied in order to generate multiple optimised versions, increasing the chance of finding a better fit for a wider variety of programs. On targeted code in 7 programs from the SPEC CPU2006 benchmark suite, this technique yields an average energy delay product (EDP) improvement of 12 % (10 % energy and 2 % performance) over coupled execution (non-DAE), with a peak EDP improvement of 36 %. The multi-versioning aspect also proves useful as different programs, and even alternate workloads of the same program, reach peak performance with different optimisation versions.
# Contents

Abstract iii

Contents v

Abbreviations vi

1 Introduction 1
  1.1 Background .............................................. 1
  1.2 Approach: Multi-Versioning ............................ 4

2 Related Work 7

3 Methodology 9
  3.1 Access phase generation ............................... 11
    3.1.1 Control Flow Graph skeleton ........................ 11
    3.1.2 Prefetches ........................................... 14
    3.1.3 Removal of superfluous code ........................ 15
  3.2 Multi-versioning ........................................ 16
  3.3 Version selection ....................................... 18
    3.3.1 Compilation .......................................... 18
    3.3.2 Execution settings ................................... 19
    3.3.3 Results collection and processing .................. 20

4 Experiments 21
  4.1 Setup and environment ................................. 21
  4.2 Results .................................................. 23
  4.3 Discussion .............................................. 26

5 Conclusion & Future Work 31

References 35
Abbreviations

CAE  Coupled Access-Execute
  Program where computations and memory accesses are mixed. In other words, a program not optimised with DAE.

CFG  Control Flow Graph
  Structure of control flow within a function.

CPI  Cycles per Instruction
  Performance measurement: Average number of processor cycles required to complete an instruction in a program.

DAE  Decoupled Access-Execute
  Program where computations and memory accesses are separated.

DVFS  Dynamic Voltage and Frequency Scaling
  Technique to adjust voltage and frequency of the processor while running a program.

EDP  Energy Delay Product
  Performance measurement: A combined measurement of used energy and runtime (delay), where the components are multiplied.

IPC  Instructions per Cycle
  Performance measurement: Average number of instructions that can be completed every processor cycle in a program. The reciprocal of CPI.

IR  Intermediate Representation
  Computer code representation used between different stages of a compiler.

MLP  Memory Level Parallelism
  When a program can make use of parallelism in the memory system by having multiple memory operations in progress at once.

SSA  Static Single Assignment
  Type of IR code where variables are only assigned once.
Chapter 1

Introduction

Energy efficiency is key in any successful computer system of today. In mobile devices, such as phones or laptops etc., increased energy efficiency can help the ongoing struggle to make batteries smaller or last longer without taking away performance. In servers and large data centres, high performance is sought after but it is limited by available cooling capabilities. Increased energy efficiency allows for higher performance without additional cooling.

As a result, any energy savings with small or no negative effects on performance are welcome. This thesis builds on the continued work of Jimborean et al. [10] wherein a compiler component was built to statically separate memory access from computation in order to allow more efficient use of Dynamic Voltage and Frequency Scaling (DVFS) [16, 22]. The contribution of this thesis is to extend the previous compiler component to generate multiple versions of a program in order to increase the likelihood of finding the sweet spot between the degree of separation of memory accesses and the overhead caused by the separation.

1.1 Background

One source of inefficiency in computers is the fact that memory technology has not kept up with processors [8, pp. 288–289]. As a result, the processor must wait for the memory system when fetching data. To alleviate the problem, caches have become ubiquitous as they can hold and quickly serve commonly used data. Even with caches, the processor occasionally has to wait for memory, and when that happens the processor is wasting energy. Usually, the processor can enter a low power state to mitigate that effect but it only helps to a degree.
The power used by current CMOS based processors can be divided in two categories: static and dynamic power [12]. Static power is caused by leakage in the transistors of the processor, even when they are not switching. It is difficult to avoid loosing energy to static power and it is therefore desirable to keep execution time down such that components of the processor can be turned off sooner. Dynamic power, on the other hand, is directly related to switching activity, the charging and discharging of capacitances in the transistors, and can be described by the formula\(^1\)

\[
P_{\text{dynamic}} = CV^2 f \quad [17, \text{ p. 39}].
\]  

(1.1)

The capacitance factor can automatically be affected by the CPU through clock gating, where the clock signal can be cut to idle sections of the processor, decreasing the effective capacitance by preventing transistors from switching. The voltage and frequency factors can be changed explicitly through the means of DVFS. This has the potential to lower power use both when running, at the cost of reduced performance due to a lower frequency, and when idle, in parts of the processor that cannot be clock gated. In situations where the full performance of the processor cannot be used, the frequency of the processor can be lowered in order to decrease power consumption. A tempting use case of DVFS is to lower the parameters and save on power when computation stalls while waiting for memory. Unfortunately, adjusting DVFS parameters take significantly longer than the average memory stall.

Generally, the voltage can be lowered with the frequency, which can lead to significant savings in dynamic power due to the voltage’s quadratic participation in the power equation (1.1). However, over the processor generations, transistors have decreased in size and as a result voltages has been lowered; this is known as Dennard scaling [7]. Dennard scaling is now breaking down as current processors use voltages that are close to the threshold voltage, at which point the transistors stop working. As a result, voltage scaling is not as effective as it used to be and the available voltage range is expected to shrink in the future. This leaves frequency as the main knob when controlling the dynamic power.

While small reductions in frequency may not incur significant energy savings, due to frequency only being a linear factor in the power equation (1.1), larger adjustments might. The problem is that extensive reductions in frequency will hurt performance. Spiliopoulos et al. [22] proposed a way to solve this problem by performing DVFS on a finer scale. They would detect memory and compute bound areas in the code (where

---

\(^1\)Where \(P_{\text{dynamic}}\) is the dynamic power, \(C\) is the effective capacitance, \(V\) is the supply voltage, and \(f\) is the switching frequency.
Chapter 1. Introduction

Execution speed is primarily limited by the memory system or the processor, respectively) and adjust voltage and frequency accordingly. This way, full performance can be extracted when the processor is the limiting factor and energy can be saved without significant performance loss when execution speed is limited by memory operations. This works well provided there are long sequences of memory and compute bound code. If the memory bound parts get too tightly intertwined with the compute bound, the inherent latencies of changing the voltage and frequency prevents this fine scale DVFS from working efficiently.

Decoupled Access-Execute (DAE) as introduced by Koukos et al. [13] and Jimborean et al. [10] can help remedy this problem. They identify repetitive tasks, such as loops, on the hot path and transform them. From each of these tasks two phases are created: one phase, called the access phase, containing memory accesses (or loads) from the original task and only necessary computations to support the memory accesses, with the purpose of prefetching the data targeted by the memory accesses to the cache; and another phase, the execute phase, containing the original task in its entirety. The phases are then scheduled to run one after the other, first the access phase then the execute phase. This scheme exposes a larger memory bound piece of code, the access phase, suitable for a low frequency as the processor is idle much of the time. It is followed by a compute bound section, the execute phase, that would benefit from a high frequency, as data is primarily fetched from the cache which is much faster than main memory.

In short, DAE will replicate load operations, and necessary address computations, of a piece of code and run it before the original. The expected outcome is to see an improvement in the relationship between execution time and energy use. This can seem counter intuitive since DAE adds code, which of course introduces additional execution time. However, the access phase is added to prefetch data to the cache in order to eliminate time spent waiting for the memory system in the execute phase. This effectively means that waiting time is been moved from the execute phase to the access phase. Furthermore, collecting memory operations into one place may improve Memory Level Parallelism (MLP) as more memory operations could be issued simultaneously, saving even more time. On the energy side, this approach exposes larger sections of memory and compute bound code (the access and execute phases, respectively). This would allow more efficient use of DVFS: the access phase can run at a lower, more energy efficient frequency, and the execute phase at a high frequency but for a shorter time.

There are a couple of factors that affect how well DAE works. The one of most importance for this thesis is how much overhead there are in the access phase. A light (low overhead) access phase is one where the memory access require minimal address computations to complete and are independent from each other to allow them to be issued
at the same time. In contrast, a heavy (high overhead) access phase contains complex calculations, and, in particular, more costly memory accesses that depend on each other, meaning that they have to be issued and completed in sequence. A light access phase is preferred since it can get more done in a given amount of time. It is the original program that decides how light or heavy the access phase is. However, a heavy access phase can be made lighter by removing some of the memory accesses, instead shifting that work to the execute phase. The best access phase is a trade-off between the cost of including certain memory accesses (dependency on other memory accesses, address calculation complexity, cache behaviour, etc.) and the benefit they provide (decreased memory access delay).

1.2 Approach: Multi-Versioning

The aim of this thesis is to expand on continued work of Jimborean et al. [10] (the DAE variant outlined above) in the search for improved automatically generated access phases. The previous approach is limited by the information available at compile time, which is a general problem with static optimisations. As such, it is difficult to accurately weigh the cost of calculating a memory address against the benefit of prefetching the corresponding data from memory. The choice of which memory accesses to prefetch in the access phase then relies on various heuristics.

This thesis explores a technique called multi-versioning. The idea is to generate multiple versions of the access phase (different access versions), that can be chosen at run time, using different rules for each version. The rules determine which memory accesses to prefetch and can be anything that can statically be measured or estimated about memory accesses (e.g. degree of dependency on other operations, part of conditional control flow, likelihood of already being in the cache, etc.), however, in this thesis only a specific type of rules (discussed below) are considered. The usage of this technique is motivated by the fact that different programs, or even the same program under alternate workloads, may exhibit different characteristics in the trade-off between memory address computation cost and prefetch benefits. As such, a single recipe for creating an access phase, as used in previous work [10], might not cover the needs of all programs that could benefit from DAE.

In this project the set of rules explored are based on indirections (indirect memory accesses). An indirection is a memory access that has to complete in order to issue another memory access (i.e. the indirection is required as a part of the address calculation for the other memory access). Consider following examples:
Chapter 1. Introduction

| A := load X | A := load X |
| B := load A | B := load 4*A |
| C := load Y | C := load 4*A+2 |
| D := load B+C | D := load Y |
|                     | E := load D |
|                     | F := load B*C+E |

To the left, the loads to A and C have no indirections since the addresses X and Y are already defined. The load to B has one indirection, the load to A. Finally, the load to D has three indirections, two direct (loads of B and C) and one transitive (load of A). In the slightly more complex example to the right, the load to F has five indirections, three direct (B, C, and E), and two transitive (D, used to load E, and A, used to load both B and C).

The rules differentiate access versions by placing a limit on how many indirections each prefetch of a memory address may have. For example, if an access version is imposing rule number 3, only memory accesses depending on no more than three other memory accesses are prefetched. In the example above the load to F to the right would be excluded since it has five indirections, all other loads are allowed since they have three indirections or less. Thus, the rules provide a knob to adjust the trade-off: prefetching more memory accesses incurs a larger performance overhead in the access phase but may provide more benefits to the execute phase, while prefetching fewer accesses leads to a lighter, but potentially less efficient access phase.

The goal of this thesis is to customise a compiler to generate access phases for a selection of benchmarks from the SPEC CPU2006 benchmark suite [3] and evaluate the performance impact in terms execution time and spent energy. The outcome of main interest is whether multiple versions of the access is useful, i.e. whether the optimal rule varies between different programs.

To make multiple versions useful, it is necessary to be able to evaluate the different versions of the access phase and select which one to use at run time. Creating such a run time framework is not within the scope of the thesis, but it warrants a second point of interest: to find out if the optimal number of indirections follows some pattern. For example, if rules with a low number tend to result in the optimal access version it would be desirable to generate more versions with a low numbered rule than with a high numbered rule in order to minimise code size and time spent selecting the optimal version at run time.
Chapter 2

Related Work

One of the first mentions of Decoupled Access-Execute (DAE) was made by Smith [21] in 1982 and involved the utilisation of two separate programs, one in charge of memory operations and one for the processing of data. This technique requires two programs to be constructed and a hardware architecture that allows the programs to be run in parallel and interact with each other efficiently. Benefits from their architecture were double instruction issue as there was one processor for each program, a type of out-of-order execution since the memory operation program could fetch data ahead of the processing program, and hiding of memory access delays.

As mentioned in Section 1.1, this thesis builds upon a DAE technique where sections of code are split into access and execute phases in order to improve the trade-off between performance and energy use. The access phase is made memory bound and is run at a low frequency to reduce power consumption while waiting for the memory system. Its role is to bring data into cache in order to speed up the execute phase. The access phase was created manually by Koukos et al. [13] and then an automatic compiler feature was developed by Jimborean et al. [10].

The technique of using two phases running in sequence has been used before. Both Salz et al. [20] and Chen et al. [5] used an inspector phase to analyse dependence patterns in order to provide an execution phase with information about how to parallelise loops. This technique was used to overcome the fact that some dependence patterns cannot be explored statically beforehand.

Another way to perform prefetching of data is through the use of helper threads as proposed by Kamruzzaman et al. [11]. Alongside the normal execution thread, one or more helper threads are running on separate cores. The helper threads attempts to bring data into the cache and if the compute thread requires already prefetched data,
the compute thread is moved to the core where that data is already available in the cache. More research along similar paths has been done [19, 25]. While Kamruzzaman et al. claim to get good results, it is under the assumption that additional cores are available for helper threads when running a single thread program. In contrast, the work in this thesis aim to perform prefetching within the same thread as targeted single thread programs, leaving remaining cores for other tasks.

Multi-versioning has been used by Chen et al. [6] for parallelising loops. In their work, multiple versions of a program would be generated, each based on a number from a reasonable pre-determined selection. Based on the number a particular version would spawn that many threads to run loop iterations in parallel (using OpenMP), exploiting the fact that loop iterations sometimes do not depend on each other. Based on comparison between information collected during run time and profiling results about each version, the most suitable variant of the parallelised loop would be selected for use. Much like Chen et al. generates versions based on a single parameter, so does the version generator in this thesis.

Luo et al. [15] have also directed attention to multi-versioning as an optimisation technique. Their framework used heuristics to select representative versions from a set of differently optimised code. The goal was to enable run time optimisations through version selection while keeping a low overhead by using machine learning to map optimised versions to data set characteristics. While their work is a tangent to this in the multi-versioning aspect, they use different techniques to generate their optimised versions.

The work for this thesis began in 2014 and most of it, including experiments, was finished the same year. In the time it has taken to complete this thesis report a few closely related works to this has been published. One of them is a sister project, carried out by Jonatan Waern [23], and set out to explore the application of just-in-time compilation to DAE. Another is a follow-up work to this, by Georgios Zacharopoulos [24], that uses hardware transactional memory to tackle a problem discussed in more depth toward the end of Chapter 5. Finally, a paper by Koukos et al. [14] reexamines the work in this thesis, with some additions (including hardware transactional memory), and performs a more thorough set of experiments.
Chapter 3

Methodology

The mechanism to generate multiple access versions is implemented as a compiler pass using LLVM [1]. This chapter describes the workings of the compiler pass from a general point of view, the reason being that the choice of LLVM for the implementation is not required. However, some LLVM tools have been used to ease the design of the pass. In such situations the details of those particular features are mentioned, as something equivalent must be found or implemented if using another compiler framework.

The compiler component developed is only concerned with the creation of an access phase and the duplication of this phase into different versions. However, in order to generate the DAE versions, the code must already contain delineated code regions of interest to process. This process is performed using components from a previous project [10].

During compilation, functions on the critical path are (currently manually) identified as interesting targets for DAE. The compiler then automatically prepares the code of the outermost loop bodies contained in these functions to be run in chunks. A chunk represents a subset of consecutive iterations of the outermost loop; its size, the number of iterations in the chunk, is referred to as granularity. The loop is split into chunks and the granularity is adjusted experimentally, such that the working set of each chunk fits in the private cache of each core. Next, the access and execute phases are generated per chunk.

The execution model is the following: the access phase of the first loop chunk is run, followed by the execute phase of the same chunk; the same procedure is repeated for the following chunks until the entire loop completes its execution. Observe that it is important that the working set of a chunk is not too large as DAE relies on the cache as an intermediate store between the access and execute phases. Should data fetched
Figure 3.1: Overview of transformation from original code to DAE.

The boxes represent sections of code. The height roughly corresponds to code size while width signifies nesting level (narrower means more nested).

To the left, the code contains a loop to be transformed. In this example, the loop has 1000 iterations. In the middle, the loop has been chunked with a granularity of 100. The outer loop will perform 10 iterations while the inner loop, the chunk, will run for 100 iterations each time it is invoked. The chunk contains the body of the original loop.

To the right, the final DAE transformation has been applied. The execute phase is an exact copy of the chunk while the access phase is a reduced clone of the chunk that only retains certain memory operations and necessary address computations. The access phase is always inserted before the execute phase and they operate as two individual inner loops, the former running all of its iterations before the latter. When both access and execute phases exist in a pair, the pair may for convenience be referred to as a chunk.

by the access phase be evicted from the cache before it is used in the execute phase, the benefit of the access phase will diminish while leaving the overhead. Figure 3.1 provides an overview of the transformation process and the structure of the resulting code. Normally, the outermost loop is chunked but if the working set of the resulting chunk cannot be adjusted to fit in the private cache, even with a low granularity, smaller segments of code, such as child loops, must be targeted for chunking instead.

The execute phase is an exact copy of the chunk created by the first transformation step seen in Figure 3.1. The generation of the access phase begins with a clone of the chunk but it is later reduced to fulfill requirements placed on it. The goal of the access phase is to load as many global values as possible into cache for the execute phase, while still keeping a low overhead. With this in mind, instructions not contributing to that goal are not allowed in the access phase and are thus removed. The structure of the access phase, the Control Flow Graph (CFG), is necessary to preserve the original access pattern (in particular looping within the chunk) and therefore all instructions defining the CFG are left in the access phase. (Should any parts of the CFG turn out to be superfluous, they are removed by dead code elimination in a later stage.) Next, all loads of global values, i.e. data stored in memory not local to the access phase, are identified. All global
loads that are not required for any computation in the access phase are replaced with a prefetch instruction, bringing the data to the cache without using the value. Details on how the access phase is generated are available in Section 3.1. The access phase creation process presented here is a new implementation that heavily draws ideas from the access phase creation process presented in the predecessor project [10].

While prefetching as much as possible is generally desirable, this may lead to a heavy access phase, where too much effort is spent on prefetching in relation the benefit gained, which is in conflict with the goal of keeping a low overhead. Therefore, it could sometimes be more beneficial in the long run to favour a lighter access version, where some of the prefetching is skipped in an attempt to improve the effort to benefit ratio. It is, however, hard to know which prefetches to skip in order to find the best balance between prefetching and overhead. This is what the multi-versioning aspect of this thesis explores. Multiple versions of the access phase are created with different restrictions on which prefetches to keep or not. Details on the selection process are found in Section 3.2.

3.1 Access phase generation

There are three major parts to the access phase generation. First, the CFG skeleton (control flow decisions and instructions they depend on) must be identified in order to preserve it; this is described in Section 3.1.1. Second, the loads of global values need to be identified and prefetches inserted. How to find loads and insert prefetches is described in Section 3.1.2. The method of choosing prefetches to create different access versions is treated in Section 3.2. Third, when every instruction required for the access phase has been identified, the rest are removed; the removal process is explained in Section 3.1.3.

Both when identifying the CFG skeleton and when inserting prefetches it is important to remember that the access phase is not allowed to have any side-effects affecting the result of the program. Sections 3.1.1 and 3.1.2 present strategies for how to deal with side-effects when constructing the CFG skeleton and inserting prefetches, respectively.

3.1.1 Control Flow Graph skeleton

To identify which instructions must be kept (collected in set $K$) in order to preserve the CFG skeleton, the following steps are taken:
Algorithm 3.1. Find instructions that comprise the CFG skeleton, add them to set $K$.

1. Find all instructions that directly define the CFG, i.e. branch instructions, and add them to $K$.

2. For each instruction $I$ in $K$ (including instructions added to $K$ during the loop).
   
   (a) Find all instructions that produce values that $I$ requires in order to execute (instructions $I$ depends on). Add these instructions to $K$.

   (b) If $I$ is a load instruction, find all store instructions that may store the value that $I$ reads and add them to $K$. (See below, in Algorithm 3.2, the details on how to find these stores.)

When searching for stores writing to the memory address targeted by a load, the reversed Control Flow Graph is used in order to easily find predecessors of basic blocks. Starting with the load instruction $L$, which uses the address $A$, Algorithm 3.2 is used to produce the set $S$ of potential sources. In words, the algorithm can be understood as searching backwards from $L$, along all control flow paths, for the most recent stores that has an address that is a MustAlias or PartialAlias with $A$.

Algorithm 3.2. Find potential sources for load instruction $L$ with address $A$, add them to set $S$.

1. Starting from $L$, search backwards through the instruction list of the basic block $L$ resides in.

2. For each instruction $I$ found in the basic block, check (in order of observation)
   
   (a) If $I$ has been encountered before, cancel the search of the current basic block and go to step 4.

   (b) If $I$ is a store instruction, touching address $B$, check
      
      i. If $A$ and $B$ alias exactly (MustAlias, see below), add $I$ to $S$, cancel the search of the current basic block and go to step 4.

      ii. If $A$ and $B$ alias partially (PartialAlias, see below), add $I$ to $S$ and proceed.

   (c) If $I$ is the first instruction in the basic block, go to step 3.

   (d) Otherwise, continue step 2.

3. Enqueue the basic blocks preceding the current basic block for subsequent search.

4. If there are no queued basic blocks, stop. Otherwise, pick the first queued basic block and go to step 2, searching the basic block backwards from its last instruction.
Chapter 3. *Methodology*

There are a few notes to be made on the implementation of this algorithm. First, while the reversed CFG is conceptually required, all that is needed is a way to find the predecessors of any basic block. The LLVM framework provides this functionality, thus no reversed CFG has to be created manually in this case. Second, the steps 2(b)i and 2(b)ii require an alias analysis to be performed. LLVM provides tools to perform this alias analysis. Once the analysis has been made, the relationship between any two addresses can be described in one of four different ways: (i) *MustAlias* denoting a spot on match; (ii) *PartialAlias* meaning that the memory areas each address refers to overlap, such as a field in a larger data structure; (iii) *MayAlias* where there are no proof that the addresses do alias nor that they do not, this is the default description; and (iv) *NoAlias* referring to the situation where the addresses cannot point to the same memory. The implementation uses, as suggested, *MustAlias* in step 2(b)i after which it is possible to stop the search in the current basic block as the effect of any previous store touching the same memory would have been overwritten. Step 2(b)ii allows more stores to be marked as possible sources; in the implementation, stores with an alias of *PartialAlias* are considered as their addresses somehow overlap with the load’s address yet it is not safe to say that those stores actually touch the same memory and as such search must continue.

It should be understood that the choice of only including *MustAlias* and *PartialAlias* stores is a compromise. Ideally, *MayAlias* stores should also be included when encountered. The compromise is made because including too many stores increase the risk of unnecessarily including a store with external side-effects. Limiting the number of stores could, on the other hand, cause a load instruction to load an incorrect value, which can affect computations and control flow decisions in a harmful way. In a benign scenario this could lead to an inefficient access phase but in a more severe case it could lead to an erroneous computation leading to a program crash or halt.

Once all instructions that must be included in the access phase are identified, the compiler verifies that the access phase is indeed side-effect free. Side-effects refer to modifications to externally visible memory, i.e. changes to memory not allocated locally in the access phase. Side-effects in the access phase are unacceptable as the program behaviour may be altered when the execute phase performs the same operation a second time. If any of the instructions in $K$ could cause side-effects the function must be disqualified from DAE. Disqualification may occur for two reasons: (i) if there are any function calls not marked as preserving the global state (e.g. read-only); and (ii) if there are any store instructions touching memory not guaranteed to be local (see below for details of how to determine the locality of a pointer).
In the LLVM Intermediate Representation (IR), memory used locally by a function is allocated by a special allocation instruction (*AllocaInst*) immediately when the function is entered. These instructions are the primary source of local pointers. The pointers can pass through a number of instructions before they are used by a memory operation, most notably an address calculation instruction (*GetElementPtrInst*) and various types of casts (*CastInst*). If the input pointer to any of these types of instructions is local, the output pointer is considered local as well. Finally, there is the question of whether or not loaded pointers (pointers loaded directly from memory) are to be considered local.

The policy used in the compiler pass is based on the assumption that loaded pointers are local if they are loaded from locally allocated memory. All pointers loaded from other sources (e.g., global variables, function arguments) are considered global (external to the function).

### 3.1.2 Prefetches

The purpose of the access phase is to prefetch data to the cache to make it ready for use once the execute phase takes over. As the phases do not share local data, it is only useful to prefetch global variables. While it is possible to try and prefetch all global variables, doing so may lead to a very heavy access phase. Instead, only a subset of all loads are prefetched. How different subsets of loads are selected is explained in Section 3.2. The current section focuses on the actions necessary when adding a prefetch to the access phase.

In the access phase, loads are generally replaced with prefetches. A prefetch instruction provides a hint for the CPU to load data to the cache. Prefetch instructions are useful in the access phase since some loads no longer provide data to any instruction, due to the access phase being a stripped version of the execute phase, and are therefore up for automatic removal in a later stage of the compilation. Sometimes, however, data from loads are required (to compute a control flow decision or the address of another prefetch) and therefore the original loads must remain. In such cases it is not desirable to include a both a prefetch and a load instruction as this may cause two requests to load the same data from memory, incurring an unnecessary overhead. Therefore, any prefetch duplicating a required load is removed from the access phase. There are, however, an exception to this rule, which is discussed further in Section 3.1.3.

When a prefetch is inserted it is also necessary to make sure that all instructions required to compute the address, $A$, being prefetched are present in the access version. To find the set $D$ of such dependency instructions the following algorithm is used. Observe that
the mechanism to search for dependencies is exactly the same as the one used to find the instructions forming the CFG skeleton in Algorithm 3.1.

Algorithm 3.3. Find dependencies of an address $A$.

1. If $A$ is produced by an instruction, add the instruction to $D$, otherwise stop.

2. For each instruction $I$ in $D$ (including instructions added to $D$ during the loop).
   
   (a) Find all instructions that produce values that $I$ requires in order to execute (instructions $I$ depends on). Add these instructions to $D$.

   (b) If $I$ is a load instruction, find all store instructions that may store the value that $I$ reads and add them to $D$. (See above, in Algorithm 3.2, the details on how to find these stores.)

Upon inserting a prefetch, the prefetch itself and the contents of $D$ should be added to $K$ (the same $K$ used to mark the CFG skeleton in Section 3.1.1).

As with the CFG skeleton it is possible to run into situations where computing the address for a prefetch requires modification to externally visible memory. If this is the case, the prefetch, and its corresponding $D$, are not added to $K$ in order to keep the correctness of the program.

Finally, when a prefetch instruction is inserted, it should not be added at the location of the load it replaces but rather as soon as the address is available. This is in order to enable more aggressive dead code elimination.

3.1.3 Removal of superfluous code

The first step in removing unnecessary code in the access phase is to ensure that as few loads and prefetches of the same data as possible coexist. If a prefetch $P$ depends on a load instruction $L$, the prefetch $Q$ duplicating $L$ can be removed as $P$ will ensure that $L$ stays in the access phase. If, however, $L$ were to be in the access phase only because the CFG skeleton depend on it, not a prefetch, $Q$ should not be removed as $L$ might be deleted by dead code elimination. The set $K$ (introduced in Section 3.1.1) should be modified accordingly.

At this point the access phase, which started out as a clone of the corresponding chunk (Figure 3.1 on page 10), still contains all original instructions as well as a number of added prefetch instructions. As the access should be as lightweight as possible all unnecessary instructions should be removed. In order to do this the set $K$ acts a guide
to determine which instructions should be spared from deletion. A pass is made over the access phase and all instructions that do not belong to $K$ are deleted.

Even with the deletion of all instructions not in $K$ there may still be unnecessary instructions left. These instructions are left as a result of considering every instruction in the CFG skeleton as necessary. The entire CFG skeleton may however not be necessary for at least three reasons: (i) some basic blocks may never have included anything of interest for prefetching; (ii) some load instructions may not have been selected for prefetching, leaving previously meaningful basic blocks now unused; (iii) some basic blocks may have been vacated as prefetches are not necessarily placed in the same location as the corresponding load but instead as soon as the target address is available, potentially outside the basic block hosting the original load. The removal of this remaining unnecessary code is left to the compiler’s existing features for performing dead code elimination.

### 3.2 Multi-versioning

Multi-versioning is used to adjust the balance between cost and efficiency of the access phase. Multiple access versions are created, each employing its own policy for selecting which loads are going to be prefetched. Each access version is built with the same CFG skeleton described in Section 3.1.1. Prefetches are then inserted as outlined in Section 3.1.2, however, only a subset are selected for inclusion in each version.

The rules used in this project, to decide if a prefetch is inserted in an access version, are based upon the number of indirections required to calculate the address to be prefetched. A threshold is set for each of the access versions. If the number of indirections required for a prefetch exceeds the threshold the prefetch is not selected to be included in that particular access version.

Indirections are a measure of how many intermediate loads are required to compute an address. Figure 3.2 shows an example of how the indirection value of a load can be determined. In 3.2a there is a C statement where a value is ultimately loaded to the variable $x$. In 3.2b the C statement has been converted into a Static Single Assignment (SSA) form resembling the LLVM IR. From the last load (to the $x$ variable) it is possible to backtrack and build a tree (or a graph in the general case) of the instructions that the load depends on; 3.2c shows this tree with all non-load instructions removed. The indirection measure is then calculated by taking the size of the tree (in 3.2c), excluding the root node. In this instance the indirection measure for the load to $x$ is 4.
Chapter 3. Methodology

17

x = a\[\ast b + c \rightarrow d\[\ast e]\];

Figure 3.2: An example of a load statement and different representations of it, ultimately arriving at the indirection measure for the load.

From the example it is possible to construct a general algorithm to find the indirection measure for an arbitrary address \(A\) (used by a load or prefetch). (Relating to the example in Figure 3.2, \(A\) would be \(t8\).)

**Algorithm 3.4.** Finding the number of indirections for an address \(A\).

1. Find the set \(D\) of dependencies for \(A\) using Algorithm 3.3.
2. Count all load instructions present in \(D\). The result of the count is the level of indirection.

As a consequence of this definition of the indirection measure, every level of indirection does not have to be represented among the dependencies of \(A\). Figure 3.3 illustrates an example where the root load depends on six other loads. There are however no intermediate loads with exactly three or four indirections as some loads depend on more than one other load. This phenomenon may lead to situations where, in this example, the access versions with thresholds of two, three and four indirections are identical, while the version with a maximum of five indirections is different.
3.3 Version selection

When running a program where multi-versioning has been applied, the idea is to select the most suitable access version at run time. Upon entering a loop (targeted by DAE) each access version is allowed to run in a limited number of chunks. Meanwhile, the time and energy performance of each access version are measured. The best-performing version is selected to run in the remaining chunks of the loop.

For this thesis there is no selection mechanism. Instead, each access version is allowed to be in use for the entire duration of a program run. Measurement results are collected in order to manually analyse execution.

3.3.1 Compilation

The following steps are performed to prepare the code for the DAE transformation. Note that step 5 alone incorporates the techniques presented in Sections 3.1 and 3.2 while the other steps uses features already available from LLVM (steps 1, 2, 7 and 8) or components implemented by the group continuing the work from [10] (steps 3, 4, 6 and 7).

1. The source code is compiled into LLVM IR.
2. The LLVM IR code is prepared for chunking.
3. The relevant loops are identified and transformed into a structure that allows execution in chunks.
4. The piece of code constituting each chunk is extracted to a separate function. This is the execute phase.
5. The newly separated function is duplicated and processed by the DAE pass to create the access phase. The DAE pass is run several times, each time with a different setting for the maximum number of allowed indirections, in order to create multiple versions of the access phase.

6. The access and execute phases are delimited by instrumentation code designed to measure performance (time and energy). PAPI [2] is used to make actual measurements.

7. Finalising optimisations are applied (including dead code elimination).

8. Binaries are created from the transformed LLVM IR code, one binary for each access version.

Note that this setup inlines the compute phase during step 7. The access phase could be inlined as well but inspection of the resulting code has shown that this sometimes cause common elements of the two phases to combine and thus alter the statistics per phase. For this reason the access phase has been prevented from inlining while running experiments.

For reference, one binary is generated without an access phase at all (which more or less means skipping step 5 above), relying solely on Coupled Access-Execute (CAE). This acts as a baseline for the other versions as it is the version that closest resemble a standard binary of the benchmark, yet allows the placement of measurement code in the proper place.

3.3.2 Execution settings

Since the benchmarks are single thread applications and the test system has a multi-core processor (see Section 4.1), multiple instances, one per physical core, of the benchmarks are run simultaneously during evaluation. This mimics a system where each instance can only use its fair share of the system resources, e.g. bus bandwidth, shared cache, etc. All instances have their affinities set to different cores to avoid excessive context switching. The results from all instances are averaged to produce a single result.

In order to achieve the desired energy savings the frequency should be dynamically adjusted to run the access phase with a low frequency and the execute phase with a high frequency. However, the machine used for evaluation does not support frequency control for individual cores, which becomes a problem as the access and execute phases are not expected to sync up between the multiple instances of a program. Instead, programs are run twice with the same configuration (access version and input), once
with the processor set to a low frequency and once with a high frequency. The low frequency run collects data regarding the execution of the access phase while the high frequency run provides information concerning the execute phase.

### 3.3.3 Results collection and processing

The measurement code included in the binaries provide a time measurement and an energy estimate for the access and execute phases separately. The data on the execute phase from the high frequency runs are brought together with the corresponding results on the access phase from the low frequency run to produce one set of results for each version.

From the collected results of time usage and energy consumption a composite measurement of efficiency, Energy Delay Product (EDP), can be calculated using the formula

\[
EDP = (Energy_{Access} + Energy_{Execute}) \cdot (Time_{Access} + Time_{Execute}).
\]  

The EDP makes it easier to compare different access versions, as it provides a standard way to weigh time and energy usage against each other.

The results from the CAE version, which are expected to reflect the performance of an unmodified benchmark, are used to normalise the results of the multiple DAE versions. Since the CAE version does not have any access phase the access components of both energy and time in Equation 3.1 are set to 0. The normalisation allows the comparison of different benchmarks against each other.
Chapter 4

Experiments

4.1 Setup and environment

The experiments were run on a machine with an Intel® Core™ i7-2600K processor and 16 GB of DDR3 memory. The processor has 4 physical cores, a frequency range of 1600 MHz–3400 MHz and cache sizes L1d: 32 K, L1i: 32 K, L2: 256 K, L3: 8192 K.

In this experimental setup, selected benchmarks from the SPEC CPU2006 [3] suit have been used. In the selection of benchmarks care has been taken to select both benchmarks that could possibly benefit from DAE, and also some that might not benefit as much. The selection of benchmarks has been guided by Prakash et al. [18] and Jaleel [9] where high L1d cache misses (misses per 1000 instructions or MPKI) and high Cycles per Instruction (CPI) have been interpreted as indicators of memory bound applications. Both many cache misses and high CPI are symptoms that an application has to wait for the memory system to deliver data. It is precisely this situation that DAE is meant to target by grouping memory accesses in order to have them consume less power by lowering the frequency and consume less time by increasing MLP.

The benchmarks that have been selected (sorted in groups of how memory bound they are predicted to be) are: mcf, milc, soplex and lbm, the most memory bound applications investigated; libquantum and astart, which are less memory bound; and finally hmer which is compute bound.

In addition to selecting benchmarks, specific functions have to be targeted for DAE. In general the functions that the most time is spent in have been targeted. As a reference, a function profile [4] of the SPEC CPU2006 benchmarks has been used. The targeted functions are listed in Table 4.1.
Table 4.1: Functions targeted for DAE, including the cumulative percentage of total time spent in the selected functions according to the function profile [4].

* This function in mcf has a large working set in the outer loop. As such, the child loops were targeted instead.

** Initially, a third function was targeted for astar, however, the compiler pass rejected it due to a function call that could not be determined to be safe. (bzip2 was also considered for the experiments but it was excluded altogether as the compiler pass rejected it due to too many unsafe stores in the CFG skeleton.)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>File</th>
<th>Function</th>
<th>% Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcf</td>
<td>pbeampp.c</td>
<td>primal_bea_mpp *</td>
<td>38.29</td>
</tr>
<tr>
<td>milc</td>
<td>m_mat_na.c</td>
<td>mult_su3_na</td>
<td>15.10</td>
</tr>
<tr>
<td>soplex</td>
<td>ssvector.cc</td>
<td>SSVector::assign2product4setup</td>
<td>44.65</td>
</tr>
<tr>
<td></td>
<td>spxsteeppr.cc</td>
<td>SPxSteepPR::entered4X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ssvector.cc</td>
<td>SSVector::setup</td>
<td></td>
</tr>
<tr>
<td>hmmr</td>
<td>fast_algorithms.c</td>
<td>P7Viterbi</td>
<td>87.39</td>
</tr>
<tr>
<td>libquantum</td>
<td>gates.c</td>
<td>quantum_toffi</td>
<td>61.91</td>
</tr>
<tr>
<td>lbm</td>
<td>lbm.c</td>
<td>LBM_performStreamCollide</td>
<td>99.79</td>
</tr>
<tr>
<td>astar **</td>
<td>Way2.cpp</td>
<td>way2obj::releasebound</td>
<td>57.31</td>
</tr>
<tr>
<td></td>
<td>Way.cpp</td>
<td>wayobj::makebound2</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Granularities used while compiling each benchmark.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Granularity</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcf</td>
<td>1000</td>
</tr>
<tr>
<td>milc</td>
<td>1650</td>
</tr>
<tr>
<td>soplex</td>
<td>300</td>
</tr>
<tr>
<td>hmmr</td>
<td>75</td>
</tr>
<tr>
<td>libquantum</td>
<td>1450</td>
</tr>
<tr>
<td>lbm</td>
<td>150</td>
</tr>
<tr>
<td>astar **</td>
<td>300</td>
</tr>
</tbody>
</table>

SPEC CPU2006 provide inputs for all benchmarks; the reference versions have been used for the experiments. Some of the benchmarks (soplex, hmmr and astar) have multiple reference inputs; in those cases all inputs have been used in the experiments.

The compilation procedure requires granularities to be set manually. Those selected (see Table 4.2) have been chosen following an inspection of the performance of the benchmarks \(^1\). In this test train inputs were used with an earlier version of the DAE framework based on the continued work of [10], which uses a different implementation of the access phase creation process. The tests were repeated with increasing granularities in order to find the best-performing option, presumably the one where the L1 cache is used to its fullest.

\(^1\)The selection of the granularities is based on preliminary performance analysis data for Jonatan Waern’s thesis project [23] which was conducted in parallel to this.
Table 4.3: The number of chunks that were run during each benchmark and the percentage of total time spent in chunks for the reference CAE version.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Chunk count</th>
<th>% Time CAE</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcf</td>
<td>21854886</td>
<td>33.06</td>
</tr>
<tr>
<td>milc</td>
<td>460800000</td>
<td>5.81</td>
</tr>
<tr>
<td>soplex - pds</td>
<td>22823811</td>
<td>26.53</td>
</tr>
<tr>
<td>soplex - ref</td>
<td>11262052</td>
<td>64.60</td>
</tr>
<tr>
<td>hmmer - nph3</td>
<td>1272953</td>
<td>97.32</td>
</tr>
<tr>
<td>hmmer - retro</td>
<td>4944811</td>
<td>93.29</td>
</tr>
<tr>
<td>libquantum</td>
<td>94107092</td>
<td>52.16</td>
</tr>
<tr>
<td>lbm</td>
<td>26001000</td>
<td>93.23</td>
</tr>
<tr>
<td>astar - lakes</td>
<td>3148366</td>
<td>17.00</td>
</tr>
<tr>
<td>astar - rivers</td>
<td>21028729</td>
<td>41.29</td>
</tr>
</tbody>
</table>

The energy estimations presented in the results are based on a power model stemming from [13]. An energy estimate is obtained by multiplying execution time with the estimated power, calculated using the power equation (1.1). The effective capacitance, $C$, is estimated by

$$C = 0.19 \cdot \text{IPC} + 1.64$$

(4.1)

where the Instructions per Cycle (IPC) is based on performance counters sampled during execution. The voltage, $V$, is based on the formula

$$V = 0.10 \cdot f \cdot 10^{-9} + 0.810$$

(4.2)

Note that the energy estimates only include dynamic power consumption.

4.2 Results

Table 4.3 shows the number of chunks that were run during each benchmark. Each chunk consists of an access phase and an execute phase. The table also shows the percentage of time spent chunks for the reference CAE version (where there are no access phase).

The graphs in Figures 4.1–4.10 present the performance results obtained from running the various benchmarks. Each group of three graphs represents the characteristics of a specific combination of benchmark and input, denoted in the captions. Common for all graphs is that the vertical axis represents the relative performance of the DAE versions compared to the CAE version. (Observe that performance has only been measured on the optimised parts of the benchmarks, that is, the access and execute phases.) The horizontal axis gives the maximum allowed number of indirections used in a particular version. Most graphs do not account for all levels of indirections, this is because these
versions are code-wise indistinguishable from the previous, more restrictive, version. (E.g. if version 4 is missing it would be the same as version 3, or even version 2 if version 3 were to be missing as well, etc.) This also means that they would have the same performance.

Please note that the results are discussed in Section 4.3.

Figure 4.1: 429.mcf

Figure 4.2: 433.milc

Figure 4.3: 450.soplex - pds
Chapter 4. Experiments

Figure 4.4: 450.soplex - ref

Figure 4.5: 456.hmmer - nph3

Figure 4.6: 456.hmmer - retro

Figure 4.7: 462.libquantum
4.3 Discussion

Starting with the compute bound benchmark, *hmmer* (Figures 4.5 and 4.6), it is clear that no version benefits from the addition of an access phase. This is expected as the targeted dataset fits in the L1 cache, which diminishes the benefit of prefetching. The performance of the execute phase remains the same while the access phase only adds superfluous overhead. In this kind of situations, when the application appears to be compute bound, it is better not to use any access phase at all in order to limit the overhead that it incurs. This exemplifies a side-benefit of multi-versioning with DAE:
even if the target application is not susceptible to DAE optimisations, overhead can be avoided by selecting to skip the access phase at run time.

The benchmarks milc, soplex, libquantum and lbm (Figures 4.2, 4.3, 4.4, 4.7 and 4.8) all exhibit performance characteristics that are expected from memory bound applications (this goes well with the prediction in Section 4.1). Each of these benchmarks displays at least one version with improved EDP. This is in part because a portion of the wait for memory is moved to the access version where energy is saved by running at a lower frequency. The total temporal performance is, for milc and soplex, unchanged or slightly worsened, which is natural as the memory waiting times are not shortened and some computations are duplicated. Yet libquantum, some versions of soplex and specifically lbm manages to improve execution time. This phenomenon can be attributed to increased MLP as described by Koukos et al. [13].

The results for mcf and astar (Figures 4.1, 4.9 and 4.10) are somewhat unexpected (especially for mcf as it was predicted to be memory bound). While the performance of the execute phase has been improved somewhat, that did not compensate for the overhead added by the access phase (in higher versions), causing significant increase in EDP. The reason for the combined performance degradation is that the access pattern is too complex to capture in a memory bound fashion. This means that in order to get any significant performance improvements in the execute phase, too many instructions for address computation and too much of the CFG structure have to be duplicated in the access phase. That the access phase can take up to twice as much time to complete, while running at roughly half the clock speed, compared to the CAE version is evidence that most of the execute phase has been duplicated in the access phase. Ideally, only a small set of instructions should be copied to create a lightweight access phase such that duplicate computations running at an inopportune frequency is kept to a minimum. Fortunately, in the cases at hand it is thanks to multi-versioning possible to fall back on a lower access version or even skip the access phase altogether.

The benchmark results are only presented for the part of the code that was targeted for DAE. For the benchmarks that did not see any or only little improvement, this does not matter much. For the benchmarks that did better, this is more important. For soplex with pds input this is particularly interesting. While that benchmark shows the best results for the targeted code, that part of the program only covers about a quarter of the execution time in the reference version. This means that when considering the whole program, the EDP drops quite dramatically to around 9 % for the best version. The new champion in this regard becomes lbm, with an already good, best case EDP improvement of over 20 % on targeted code, that makes up over 90 % of the execution time in the reference. An outlier in the other direction is milc. The execution time of
the targeted code in the reference is only 5%, making the total EDP change rather negligible. However, there are a few more functions in milc with a decent share of the execution time. It may be interesting to attempt to target a larger part of the benchmark in future trials.

In many of the benchmarks, versions 0 and sometimes 1 have an access phase that is hardly or not visible in the graphs. The access phases are there but they are run for a very short amount of time. Normally, the access phase is a loop since it performs a chunk of iterations determined by the granularity. However, at a low level of indirection some loads are independent of the loop (e.g. loading base pointers). In these cases, prefetches replacing the loads can be placed before the loop of the access phase. When this applies to all prefetches in an access version the entire loop of the phase can be removed as dead code. Indeed, this happens for all benchmarks, except hammer, in version 0. In version 1 only libquantum and astar still exhibits this phenomenon. The result is an access phase that is only run for one iteration for every 100 or 1000 iterations (according to granularity) of the execute phase, explaining the small footprint. Why these small access phases often give a benefit to the execute phase is harder to answer since low indirection loads can be expected to be held in the cache for the duration of the outer loop. A plausible explanation for this is that the optimised functions, containing the targeted outer loops, are called many times. As such, the access phase, even without a loop, will have many opportunities to bring cold data to the cache, assuming that the data is evicted from the cache between calls. Further investigations would have to be made to confirm this.

The only benchmark with two input sets that have dramatically different results is soplex. With the pds input (Figure 4.3) the exchange from the access phase becomes greater the more prefetches (with more indirections) that are made. With the ref input (Figure 4.4) it appears to be more beneficial to select a version with less prefetching. Jaleel [9] notes that the relative performance between the two data sets is related to cache size, suggesting that the data sets causes different behaviour when fed to soplex. The difference in DAE performance may be another symptom of the differing characteristics of the data sets.

The differences between the two input sets for soplex is one of the main motivations for multi-versioning. As the exact input workload is often unknown at compile time it is not possible to select the best optimisation option for DAE statically. With multi-versioning it is possible to be prepared for different situations where it is preferable to prefetch more or less.
A drawback of multi-versioning is the need to house and manage more code versions. Each access version is essentially a copy of the execute phase, though somewhat condensed. If many functions in a program are targeted for multi-versioning a non-negligible part of the code will be copied, possibly many times over. In order to limit this code explosion it is important to target only hot functions and to limit the number of versions generated. In this thesis, DAE has been focused on hot functions and only a moderate number of versions have been generated, resulting in an increase in code size of only a few percent at most.

Another reason to limit the number of versions is to minimise the selection overhead at run time. Even though no mechanism for run time selections has been made for this thesis, an estimate of the overhead can be useful. All of the benchmark executes at least 1 000 000 chunks, and most an order of magnitude more. Running just a single chunk per version may be enough to tell which one is best, but to be on the safe side assume 10. No benchmark have more than 15 versions to test, use that as a worst case. Furthermore, each benchmark can have multiple targeted loops, in the order of 10, which may benefit from separate evaluation. As such, up to 1 500 chunks out of 1 000 000 may run with a suboptimal version in order to find the best, that is less that a fifth of a percent. Even with a high estimate, the number of versions used in this thesis does not appear to generate significant overhead in a selection mechanism.

Even though it is not necessary to limit the number of versions in this thesis, there is a pattern in the result that seem to indicate that some versions are more useful than others. As the indirection level goes up the EDP, as a trend, either increases or decreases (there are no prominent local extremes). This indicates that the access version that provides the best EDP is likely to be either one with a low level of indirection or one with a level of indirection close to the maximum for the application, not a version in the middle of the range.

The overall results for all benchmarks are very promising. In most benchmarks there are at least one access version that improves EDP and in more cases than not this means a lower energy use and a lower execution time. In the few benchmarks where no access version improves EDP it is still possible to fall back on the CAE version (without access phase) to avoid performance degradations. Figure 4.11 summarises the performance of the best version from each benchmark. As a mean over all benchmarks EDP has dropped by 12 % (with a peak of 36 % for soplex with the pds input) on code targeted for DAE, and energy use is down by 10 % while time consumption is largely unchanged (down by 2 %).

The closest relative to this thesis among the related work is the paper by Jimborean et al. [10]. In a naive comparison, Jimborean et al. get better results with an average
Figure 4.11: This plot summarises the results of all benchmarks used in the experiments. For each benchmark, the best version has been selected to simulate the run time evaluate and select mechanism. The choice of version is based on EDP, where lower is better. The version selection shown in the plot is mcf: CAE, milc: 2, soplex-pds: 11, soplex-ref: 1, hmmr-nph3: CAE, hmmr-retro: CAE, libquantum: 0, lbm: 2, astar-lakes: 1, astar-rivers: 1. The final column is a geometric mean over the best selection for all benchmarks.

EDP improvement of 25 % over coupled execution (compared to 12 % for this thesis). The results are, however, not directly comparable. First, the evaluations do not cover the exact same benchmarks, in fact, only libquantum and lbm overlap. In that regard Jimborean et al. achieved a much better result for libquantum with about a 50 % EDP improvement and a slightly better result for lbm with about a 30-35 % EDP improvement. For the later there was actually a performance degradation, though compensated with a good energy improvement. Second, the core DAE approach (not just the multiversioning part) of this thesis is directly applicable to a wider range of applications. In the case of Jimborean et al. they targeted affine loops through polyhedral analysis (does not apply to libquantum and lbm), and non-affine task-based applications through an analysis similar to the one in this thesis. To make libquantum and lbm amenable to the later strategy they had to manually transform the applications to become task-based. The strategy used in this thesis, however, is directly applicable to the unmodified versions of all presented benchmarks (including libquantum and lbm).
Chapter 5

Conclusion & Future Work

This thesis has explored the addition of multi-versioning to Decoupled Access-Execute. The results are very encouraging with a peak EDP improvement of 36% and an average of 12% on code targeted for DAE. As expected it is generally the memory bound applications that is most susceptible to improvements from DAE, especially when it is possible generate an access phase that can perform its prefetching task using only a small part of the original code of the corresponding chunk, making it a lightweight access phase. The most exiting result, however, is that there is a motivation for the use of multi-versioning. Different benchmarks benefit from different versions of the access phase and, in the case of soplex, even different datasets affect which access version gives the best result. Especially the latter case would be hard to cover with only a single version of the access phase. Finally, having alternatives for the access phase also opens up for optional execution altogether. When DAE can benefit a program the best access version can be used but when DAE cannot offer improvement it is possible to skip the access phase, falling back on the standard coupled execution. This makes it less important to avoid applying DAE to applications that would otherwise see performance degradations.

While the results are good there are still opportunities for improvement and ways to make multi-versioned DAE easier to use in practice. The rest of this chapter will discuss some possible avenues of improvement.

In this thesis focus has been put on the creation of access versions. This calls for the addition of a component that will evaluate and select among the versions at run time. A compiler component has to be implemented to pack multiple versions in a single binary in such a way that it is easy to switch between access versions. A run time component then needs to test all versions and allow the best to run the majority of the time. This requires a heuristic to control how often the decision is to be reevaluated. Suggestions
of such heuristics include: only once, at each invocation of the outer loop, after a certain number of chunks have been run, and if the performance of the current version diminishes. Another challenge in this endeavour would be to decide on the thoroughness of the evaluation in order to weigh overhead and probability of finding the best version against each other.

The experiments in this thesis have examined benchmarks using certain indirection limits throughout the entire application. As multiple loops may be targeted there can be several access phases for different code regions in an application. It is not certain that all of these access phases will benefit from the same indirection level. Further observations of access phases being allowed to run a suitable version independent of other access phases in the same application may reveal the possibility even greater energy savings.

In order to apply this DAE technique to any application, the current requirement of specifying functions to target has to be removed. As processing the entire application would be excessively expensive the challenge is to evaluate (preferably statically) how hot functions (or even individual loops) are and tell the DAE framework which code regions to optimise.

The version differentiation technique (counting indirections) proposed in this thesis has shown potential. There may however be room for improvement by making alternate cost evaluations for individual loads. For example, a load with a shallow dependence tree (compare to Figure 3.2c on page 17) may be cheaper to prefetch than another load with a same-size but deeper dependence tree as more of the dependency loads could be performed in parallel. Another idea would be to look for indicators that reveal the critically of loads.

In Section 3.1.1 there was a discussion on how to handle stores that might write values that loads later depend on. To ensure desired program behaviour all stores that write values later read by loads, i.e. stores that alias with loads, must be preserved in the access phase. Should relevant stores not be included in the access phase, there is a possibility that that a load will fetch data from the wrong address, which will decrease the effectiveness of the access phase. As long as memory locations that are not being updated (due to omitted stores) contain a valid memory address to begin with, this should not lead to bad behaviour. However, if any of the included stores are to global memory, it, and the instructions relying on it, must be removed since the access phase has to be side-effect free. In the case control-flow instructions depend on a global store the entire targeted function is rejected for DAE, and in case a prefetch depend on a global store that prefetch is skipped. In this thesis only MustAlias and PartialAlias relations were considered when including aliasing stores as also including MayAlias stores led to too many targeted functions being rejected. In practice, no abnormal behaviour was
detected when ignoring MayAlias stores, however, in order be certain that the access phase behaves as desired all aliasing stores should be included in it. The side-effects arising from stores to global memory then have to be dealt with another way. (As an added benefit this would allow DAE to be applied to more applications, such as bzip2 from the SPEC suite which was rejected entirely even when only considering MustAlias and PartialAlias.) An idea to handle the side-effects would be to track and revert all global stores before the access phase finishes.\footnote{This route has been explored by Giorgios Zacharopoulos in a follow-up thesis \cite{Zacharopoulos2014} by using hardware transactional memory to capture stores in the access phase and revert them.}

The evaluation of the benchmarks for this thesis only includes time and energy measurements. Another property that would have been interesting to collect is cache statistics for hits and misses in the access and execute phases. This could in part be used to confirm the results from the already performed experiments. The expectation would be that when an execute phase performs better in time and energy, compared to coupled execution, it would also have a better cache hit ratio than the reference. The cache analysis could also provide insight into how well access phases work with corresponding execute phases. For an access version with the highest level of indirection, the execute phase would be expected to always hit in cache for the targeted memory accesses. If this is not case it would be an indicator that there is a cache collision issue or that the granularity is set inappropriately. Cache analysis could also provide insight into the internal efficiency of the access phase. While the access phase can be expected to have some cache hits, misses are more useful. If an access phase have excessively many cache hits it might be an indication that the same cache line is being hit by prefetches multiple time, in which case it would be of interest to find a heuristic that can prevent this.
References


References


