Uppsala University

Bachelor's Thesis

Static Multi-Versioning for Efficient Prefetching

Author:
Per Ekemark

Supervisor:
Alexandra Jimborean

Examiner:
Olle Gällmo

Reviewer:
David Black-Schaffer

A thesis submitted in fulfilment of the requirements for the degree of Bachelor of Computer Science

in the

Uppsala Architecture Research Team
Department of Information Technology

November 2014

UPPSALA UNIVERSITY

Abstract

Faculty for Science and Technology
Department of Information Technology

Bachelor of Computer Science

Static Multi-Versioning for Efficient Prefetching

by Per EKEMARK

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too…

Contents

Abbreviations

CAE	Coupled Access-Execute
CFG	Control Flow Graph
CPI	Cycles per Instruction
DAE	Decoupled Access-Execute
DVFS	Dynamic Voltage and Frequency Scaling
EDP	Energy Delay Product
IPC	Instructions per Cycle
IR	Intermediate Representation
MLP	Memory Level Parallelism
SSA	Static Single Assignment

Chapter 1

Introduction

Energy is not free. This statement gains significance especially when considering the limited capacity of batteries in the multitude of handheld devices and portable computers that have gained immense popularity in recent years. Energy is also an important factor when it comes to grid powered computers. Not only is there a monetary cost to energy but also an environmental one, as much energy is still produced by unsustainable means. Decreased energy usage also means a decreased requirement for cooling with associated costs.

As a result, any energy savings with small or no other negative effects are welcome. This thesis builds on previous work in the subject and aims to explore further possibilities to save energy by modifying the way programs are compiled and, in turn, how programs are run.

1.1 Background

One source of inefficiency in computers is the fact that memory technology has not kept up with processors. As a result, the processor must sometimes wait for memory in order to finish an operation. Much work has been done to decrease the waiting time, for example caches are a result of such work. Even with caches, the processor occasionally has to wait for memory, and when that happens the processor is doing nothing but wasting energy.

A large part of the power used by a processor is consumed when it is switching transistors. This is the dynamic power and is described by the formula[1]

$$P_{dynamic} = CV^2 f \quad [12, \text{p. } 39]. \tag{1.1}$$

The voltage and frequency factors can be changed through the means of Dynamic Voltage and Frequency Scaling (DVFS). In situations where the full potential of the processor is not required, e.g. when computation stalls while waiting for memory, the frequency of the processor can be lowered in order to decrease power consumption.

Generally the voltage can be lowered with the frequency, which can lead to significant saving in dynamic power due to the voltage's quadratic participation in the power equation (1.1). However, as voltages have decreased in newer processor generations, the transistors have begun to leak to the extent that the leakage represents a significant portion of the processor's power usage [12, p. 40]. Under these conditions, further scaling down voltage is no longer a viable option. This only leaves frequency as a means to control 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. [17] made an attempt to solve this problem by performing DVFS on a finer scale. The framework would detect memory bound areas in the code and adjust (voltage and) frequency accordingly. 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 treated by Koukos et al. [10] and Jimborean et al. [8] can help remedy this problem. At compile time repetitive tasks are selected and transformed. From each of these tasks two phases are created: one phase, called the *access* phase, containing a selection of memory operations, which are to be *prefetched*, extracted from the original task; 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, 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. The key here is that while the access phase will

[1] Where $P_{dynamic}$ is the dynamic power, C is the (effective) capacitance, V is the supply voltage and f is the frequency.

incur some temporal overhead, as it adds additional code to run, the time loss will be compensated in two ways: (i) relevant data will be brought into cache preventing the execute phase from having to wait for memory, making it compute bound, and (ii) compacted memory accesses allow for more Memory Level Parallelism (MLP), making total memory access time shorter than without DAE. The intention is to make the net temporal overhead small while decreasing the energy usage.

1.2 Approach

The aim of this thesis is to expand on the DAE variant described above in 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 impossible 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 becomes an educated guess at best.

The technique that is investigated in this thesis is called *multi-versioning*. The idea is to generate multiple versions of the access phase (different *access versions*) using different rules for each version. The usage of this technique is motivated by the fact that different programs may exhibit different characteristics in the trade-off between memory address computation cost and prefetch benefits. As such, a single type of access phase might not cover the needs of all programs that could benefit from DAE.

The rules determine which memory addresses should be prefetched. In this project the set of rules explored are based on *indirections* (indirect memory accesses). Memory operations are not always independent but can sometimes depend on the result of other memory operations; this is what constitutes an indirection. Depending on what is already available in the cache, the cost of waiting for an indirection to finish and the benefit of prefetching something requiring it may vary. The rules differentiate access versions by placing a limit on how many indirections each prefetch of a memory address may have. E.g. if an access version is imposing rule number 3, only memory accesses depending on no more than 3 other memory accesses are prefetched. Thus, the rules provide a knob to adjust the trade-off: prefetching more memory accesses incurs larger temporal overhead but provides more benefits, while prefetching less accesses leads to a lighter, but potentially less efficient access phase.

This speculative generation of access versions is, as a final product, not useful without a way to evaluate each version during run time and allowing the best to be in use most of the time. While it is not the goal of this thesis to produce such a mechanism, there are still reasons to study the rules used to generate access versions. First of all, the question of the benefit of multi-versioning must be raised; will multi-versioning provide an edge over what is possible to do with a single access phase? Providing the use of multiple access versions is useful, there is still a desire to keep the number of versions limited. This is because each version inevitably adds to the code size and incurs an additional overhead in the selection mechanism. For this reason it would be beneficial to investigate if there is a way to approximate a set of optimal rules that would gain priority should a limit be imposed on the number of allowed access versions.

Chapter 2

Related Work

One of the first mentions of DAE was made by Smith [16] and involved the utilisation of two separate programs, one in charge of memory operations and one for 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.

As mentioned in Section 1.1, this thesis relies on a DAE technique where sections of code are split into access and execute phases. The access phase is memory bound and is consequently run at a low frequency to reduce power consumption. 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. [10] and then an automatic compiler feature was developed by Jimborean et al. [8].

The technique of using two phases running in sequence has been used before. Both Salz et al. [15] 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. [9]. 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 [14, 18].

The multi-versioning technique has been used for parallelising loops [6]. Each version would be given a number. Based on that number a particular version would spawn that number of threads to run loop iterations in parallel, exploiting the fact that loop iterations sometimes do not depend on each other. Based on information collected during run time the most suitable version would be selected for a particular task.

Luo et al. [11] 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 the overhead and code size small.

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 the previous project [8].

During compilation, functions on the critical path are (currently manually) pointed out 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; next, the DAE run of the following chunk is invoked, until the entire loop completes its execution. Figure 3.1 provides a visual 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.

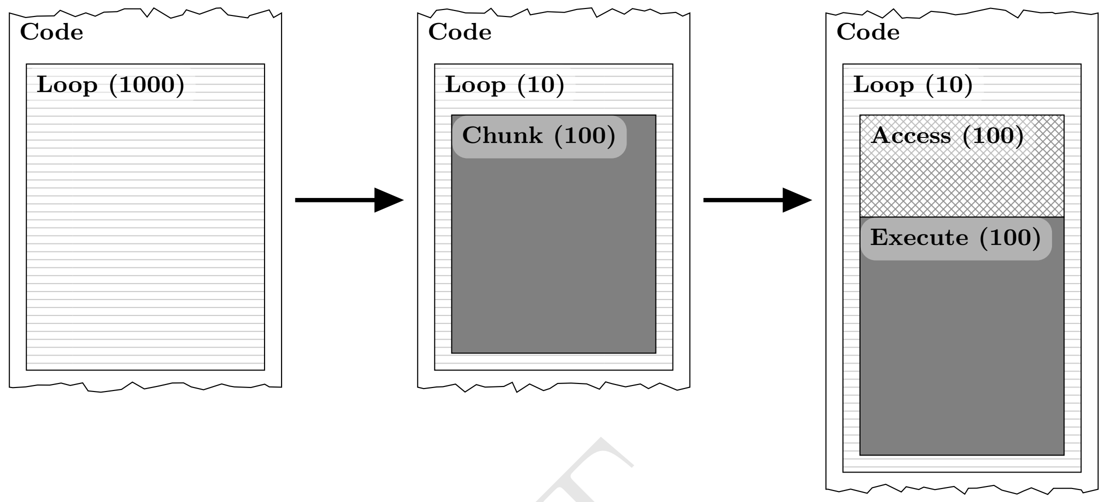


FIGURE 3.1: Overview of transformation from original code to DAE.
To the left, the code contains a loop to be transformed. In this example the loop has 1000 iterations. In the middle, the loop has been chunked with a granularity of 100. The outer loop will perform 10 iterations while the inner loop, 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.

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 fulfil 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, everything not contributing to that goal is not allowed in the access phase and is thus removed. The structure of the access phase, the Control Flow Graph (CFG), is necessary and therefore all instructions defining it are left in the access phase. 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.

While prefetching as much as possible is generally desirable, this may lead to a heavy access phase, which is in conflict with the goal of keeping a low overhead. Therefore, it could sometimes be more beneficial in the long run to skip some of the prefetching in favour of a lighter access version. 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. Pick an instruction I from K and find all instructions that produce values that I requires in order to execute (instructions I depends on). Add these instructions to K.
3. 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.)
4. Repeat steps 2 and 3 until no choice of I will cause further instructions to be added to K.

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, the following algorithm is used to produce the set S of potential sources:

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.

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 including 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. If any of the instructions in K could cause side-effects on data used outside of the access phase, the function must be disqualified from DAE in order not to affect the results of the program in any way. Disqualification may occur for two reasons: if there are any function calls not marked as preserving the global state (e.g. read-only), the access phase is discarded; alternatively, 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) the access phase must be abandoned as well.

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 are 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 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 a subset of loads are selected is explained in Section 3.2. This 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 makes sure that data is loaded to cache without returning it for later use in the program. Sometimes the loaded data is required (to compute a control flow decision or the address of another prefetch) and therefore the original load must remain. In such cases it is not desirable to insert a duplicate prefetch of the 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. The 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. Follow steps 2-4 from Algorithm 3.1 using D in place of K.

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 depend 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 (remember Figure 3.1 on page 8), 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. This may however not be true for a couple of 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.

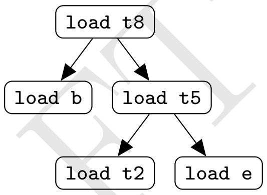
The rules used in this project, to decide if a prefetch is inserted in an access version, are based upon the number of indirections required to calculate the address to be prefetched. A threshold is set for each of the access versions. If the number of indirections required

```
x = a[*b+c->d[*e]];
```

(A) C statement performing a load. (Note that the single statement could have been broken up into several smaller statements without affecting the rest of this example.)

```
t1 = load b
t2 = gep c, 0, index_of_d
t3 = load t2
t4 = load e
t5 = gep t3, 0, t4
t6 = load t5
t7 = add t1, t6
t8 = gep a, 0, t7
x  = load t8
```

(B) The same C statement represented in an SSA instruction form similar to LLVM IR. ("gep" is the GetElementPointerInst indexing instruction from LLVM, it calculates an address but does not access memory.)



(C) A tree representation of the dependencies between the loads. The indirection measure of the root load is the size (number of nodes) of the tree excluding the root.

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

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.

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.

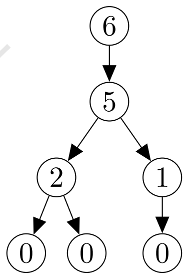


Figure 3.3: Tree of indirections illustrating that not all indirection levels have to be present. Each node represents a load instruction with the number denoting how many load instructions it depends on. There are no loads with exactly 3 or 4 indirections while there are nodes with 5 and 6.

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 are 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 and studied after execution.

3.3.1 Compilation

The following steps are performed to prepare the code for the DAE transformation:

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.

In order to get a point of reference one binary is generated without an access phase at all, relying solely on Coupled Access-Execute (CAE). This acts as a baseline for the other versions.

3.3.2 Execution settings

During evaluation, multiple identical instances of the same program are started, one per physical core of target machine. This is to emulate a busy machine in order to disallow allotment of resources the machine could not afford to provide in a scenario where all cores are occupied with a similar workload.

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

$$EDP = (Energy_{Access} + Energy_{Execute}) \cdot (Time_{Access} + Time_{Execute}). \tag{3.1}$$

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 are used to normalise the results of the multiple DAE versions.

Chapter 4

Experiments

4.1 Setup and environment

The experiments were run on a machine with an Intel® Core$^{\text{TM}}$ 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 benchmarks that could possibly benefit from DAE, but also some that might not benefit as much. The selection of benchmarks has been guided by Prakash et al. [13] and Jaleel [7] where high L1d cache misses (misses per 1000 instructions) and high Cycles per Instruction (CPI) have been interpreted as indicators of memory bound applications. 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 *astar*, which are less memory bound; and finally *hmmer* 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.

SPEC CPU2006 provide inputs for all benchmarks; the reference versions have been used for the experiments. Some of the benchmarks (*soplex*, *hmmer* 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

Benchmark	File	Function
mcf	pbeampp.c	primal_bea_mpp *
milc	m_mat_na.c	mult_su3_na
soplex	ssvector.cc spxsteeppr.cc ssvector.cc	SSVector::assign2product4setup SPxSteepPR::entered4X SSVector::setup
hmmer	fast_algorithms.c	P7Viterbi
libquantum	gates.c	quantum_toffoli
lbm	lbm.c	LBM_performStreamCollide
astar	Way2_.cpp Way_.cpp RegWay_.cpp	way2obj::releasebound wayobj::makebound2 regwayobj::makebound2 **

Table 4.1: These are the functions targeted for DAE.
* This function in *mcf* has a large working set in the outer loop. As such, the child loops were targeted instead.
** While this function in *astar* was targeted 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.)

Benchmark	Granularity
mcf	1000
milc	1650
soplex	300
hmmer	75
libquantum	1450
lbm	150
astar	300

Table 4.2: These are the granularities used while compiling each benchmark.

benchmarks. (TODO: refer to Jonatan somehow) In this test train inputs were used with an earlier version of the DAE framework. 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.

The energy estimates presented in the results section are obtained using the power equation (1.1) and an experimentally determined estimation of the effective capacitance, C, based on Instructions per Cycle (IPC):

$$C = 0.19 \cdot \text{IPC} + 1.64 \tag{4.1}$$

This estimation was done in [10] and is tailored for Sandy Bridge processors and SPEC benchmarks. Note that the energy estimates only include dynamic power consumption.

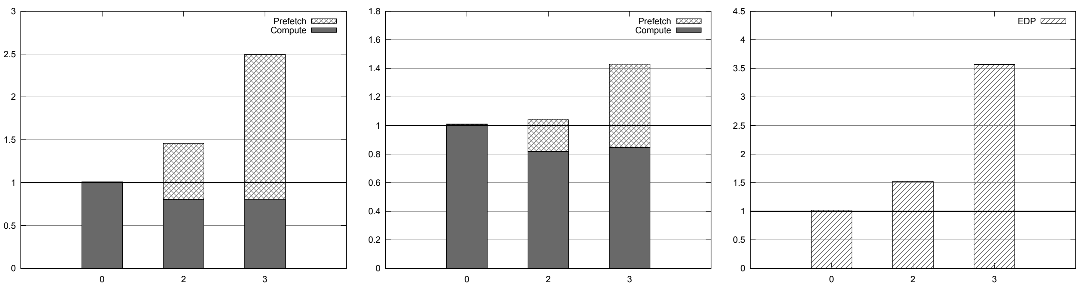
Benchmark	**Chunk count**
mcf	21854886
milc	460800000
soplex - ref	11262052
soplex - pds	22823811
hmmer - nph3	1272953
hmmer - retro	4944811
libquantum	94107092
lbm	26001000
astar - lakes	3148366
astar - rivers	21028729

Table 4.3: The number of chunks that were run during each benchmark.

4.2 Results

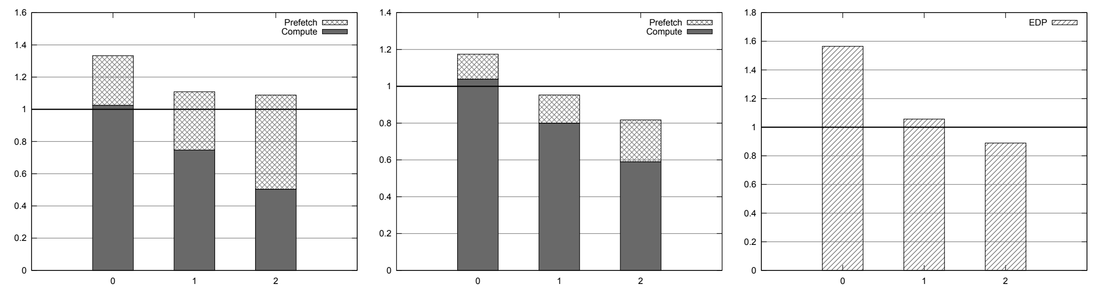
Table 4.3 shows the number of chunks that were run during each benchmark. Each chunk consists of an access phase and an execute phase.

The 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. 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 is only 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 are missing values along the horizontal axis. This is because these versions are code-wise the same as the previous, more restrictive versions. (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.)



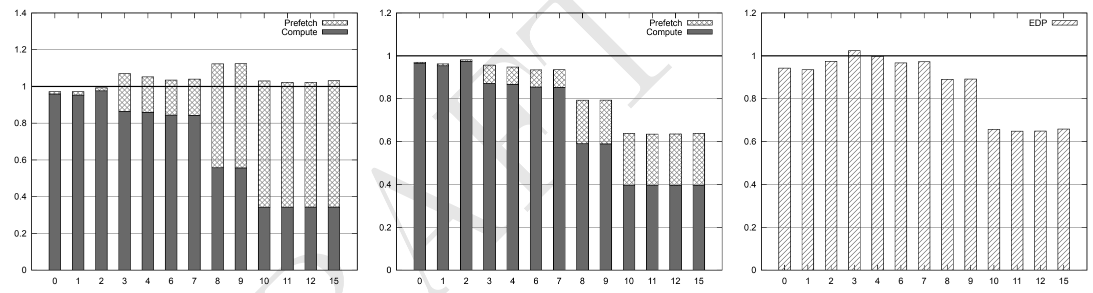
(A) Time (B) Energy (C) EDP

Figure 4.1: 429.mcf



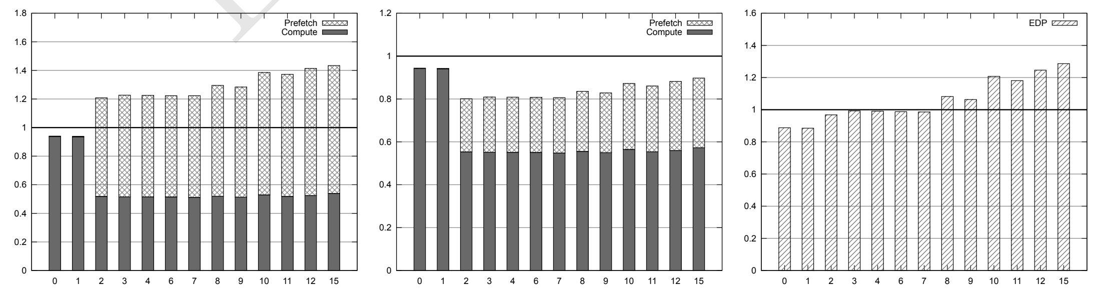
(A) Time (B) Energy (C) EDP

Figure 4.2: 433.milc



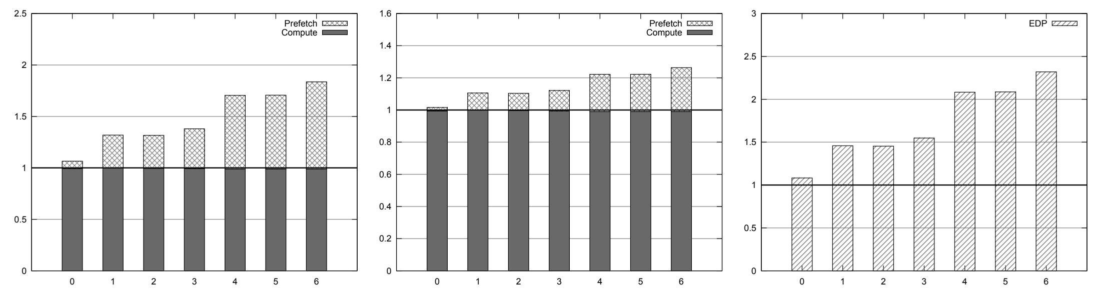
(A) Time (B) Energy (C) EDP

Figure 4.3: 450.soplex - pds



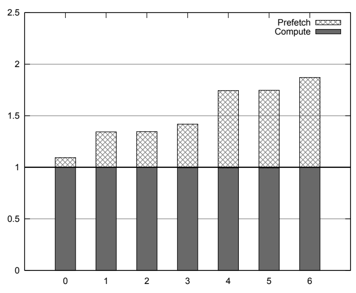
(A) Time (B) Energy (C) EDP

Figure 4.4: 450.soplex - ref

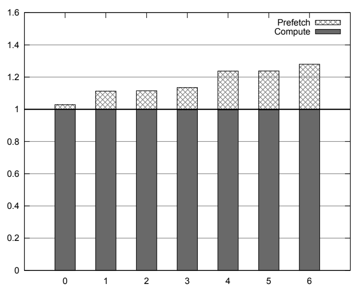


(A) Time (B) Energy (C) EDP

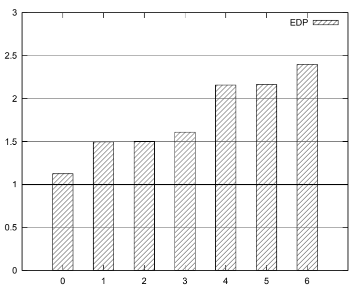
Figure 4.5: 456.hmmer - nph3



(A) Time

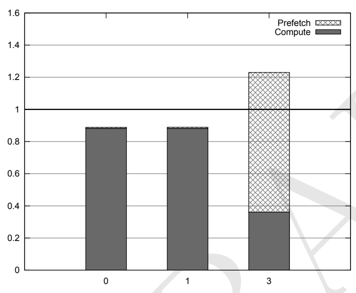


(B) Energy

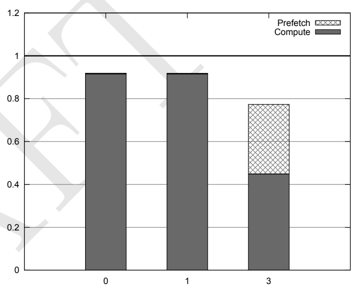


(C) EDP

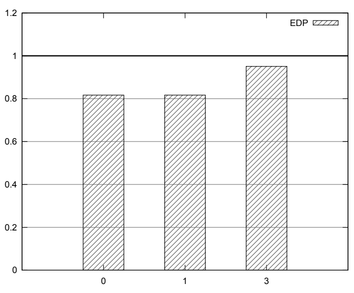
FIGURE 4.6: 456.hmmer - retro



(A) Time

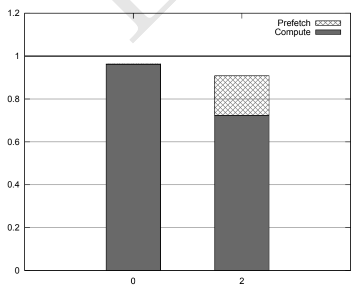


(B) Energy

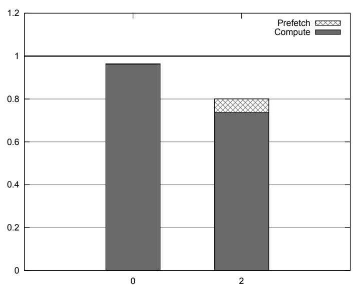


(C) EDP

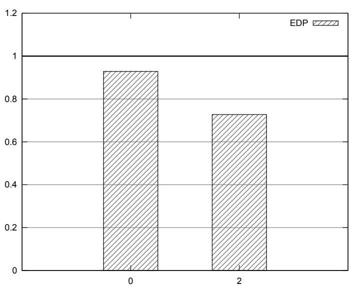
FIGURE 4.7: 462.libquantum



(A) Time

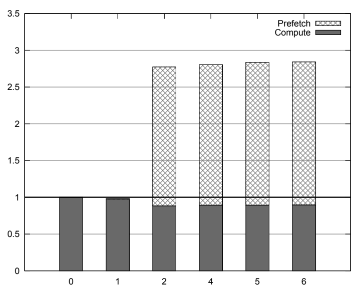


(B) Energy

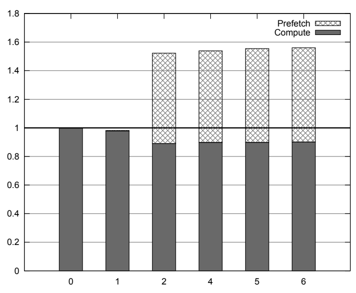


(C) EDP

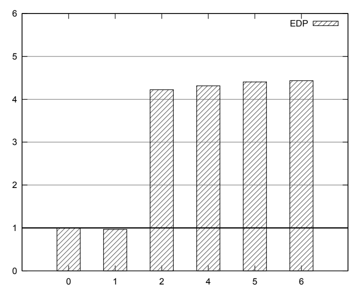
FIGURE 4.8: 470.lbm



(A) Time

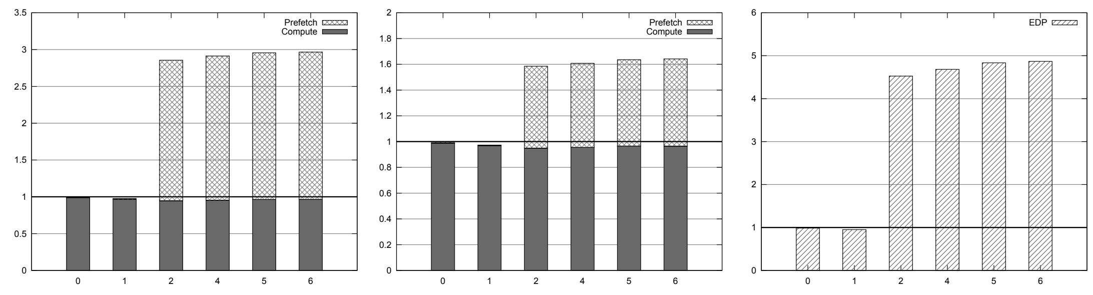


(B) Energy



(C) EDP

FIGURE 4.9: 473.astar - lakes



(A) Time (B) Energy (C) EDP

FIGURE 4.10: 473.astar - rivers

4.3 Discussion

Starting with the compute bound benchmark, *hmmer* (Figures 4.5 and 4.6), it is clear that no version benefit 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 it is better not to use any access phase at all in order to limit the damage.

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. [10].

The results for *mcf* and *astar* (Figures 4.1, 4.9 and 4.10) are somewhat disappointing (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 major increase in EDP. The reason for the performance degradation is that the access pattern is too complex to capture in a memory bound fashion. As a result, computations that benefit from a higher processor frequency are duplicated into the access phase.

It should be noted that most of the access versions have a number of prefetches that do not depend on much other data (no or a low number of indirections); some of these prefetches can be placed outside the inner loop. The consequence is that access versions with a low indirection limit can sometimes lack loops entirely as they are optimised away as dead code. In fact, only *hmmer* has a version 0 that does contain a loop in the access phase. For version 1 only *libquantum* and *astar* are free of loops in the access phases. In versions with higher indirection limits, each benchmark has a loop in at least one access phase. When there is no loop the access phase it only runs one iteration during a chunk while the execute phase is running as many iterations as the granularity dictates. This quirk can be observed in the results as a seemingly non-existent access phase.

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 [7] notes that the performance difference is related to cache size.

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 increase in code size. 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. The usefulness of the later strategy is most apparent in *soplex*, and to some extent *astar*, where there are a large number of versions, some of which perform very similarly. It would be preferable to only generate one of these versions. Unfortunately, no clear indicators, such as a large difference in the number of prefetches, hinting at whether a version will perform similar to another, have been detected.

The results from each individual benchmark may look somewhat discouraging as almost all benchmarks have some versions that significantly hurt performance. This is however not by any means bad as the purpose of multi-versioning is to speculatively generate different access versions at compile time and to evaluate and select the best one (or none at all) at run time, which will run in the vast majority of all chunks. Figure 4.11 summarises the performance of the best version (or no version (CAE) if that is better) from each benchmark. As a mean over all benchmarks EDP has dropped by 12 % and energy use is down by 10 % while time consumption is mostly unchanged (down by 2 %).

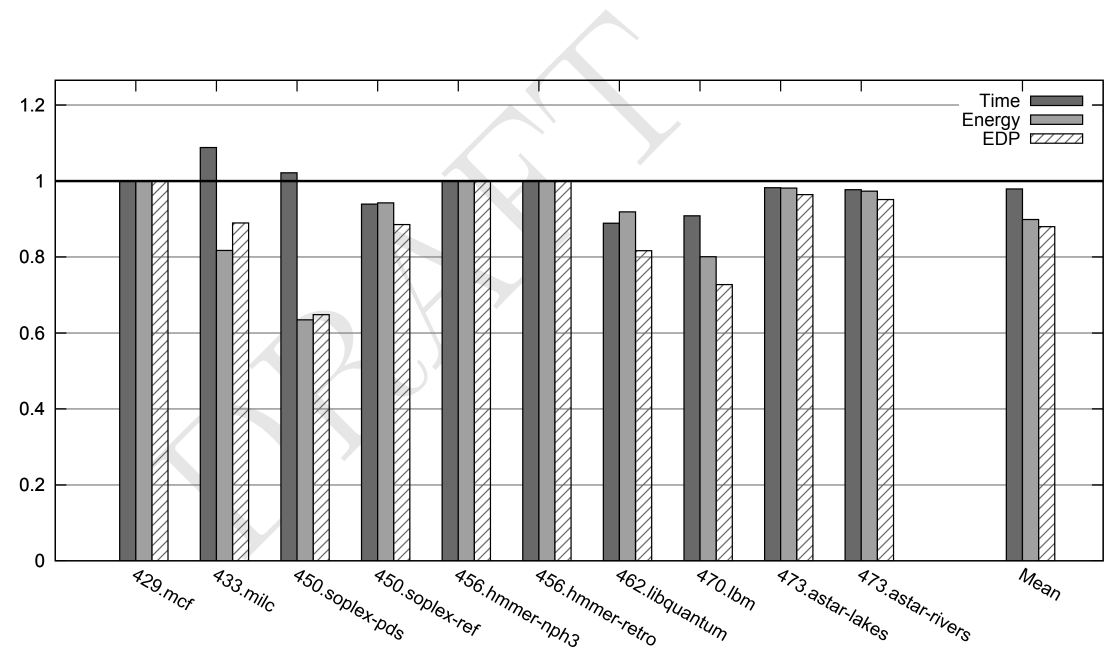


FIGURE 4.11: This plot summarise the results of all benchmarks used in the experiments. For each benchmark, the best version has been selected by an imagined selection mechanism. The choice of version is based on EDP, where lower is better. The version selection shown in the plot is *mcf*: CAE, *milc*: 2, *soplex-pds*: 11, *soplex-ref*: 1, *hmmer-nph3*: CAE, *hmmer-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.

Chapter 5

Conclusion

5.1 Summary

5.2 Future Work

This work has been an exploration of multi-versioning in conjunction with DAE. While the results are encouraging, showing that there is potential for multi-versioned DAE for some types of code, there are still parts missing to make it usable in practice.

In this thesis focus has been put on the creation of access versions. This calls for an 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.

Limiting the number of versions is important for two reasons: (i) to make sure that the code size does not grow out of proportions and (ii) to minimise the overhead incurred by run time selection algorithms. A naive approach would be to select or remove versions from the range of the available ones. A more intricate solution could involve an analysis that attempts to find versions that can safely be assumed to be worse or equally good as other versions and prioritise the former for omission.

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 14) 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 of how hot individual loads are.

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 correct program behaviour all stores that write values later read by loads must be preserved in the access phase. However, to achieve this it is necessary to include all stores that cannot be proven to be unrelated to the loads in question, on the account that alias analysis is not perfect. This volume of stores increases the risk of global side-effects (something *bzip2* was particularly prone to). Ideally, the side-effects should be dealt with by other means than removing the underlying stores. An idea would be to track and revert all global side-effects (stores) before the access phase finishes.

Bibliography

[1] LLVM. `http://llvm.org/`.

[2] PAPI. `http://icl.cs.utk.edu/papi/`.

[3] SPEC CPU2006. `http://www.spec.org/cpu2006/`.

[4] SPEC CPU2006 function profile. `http://hpc.cs.tsinghua.edu.cn/research/cluster/SPEC2006Characterization/fprof.html`. Accessed: 2014-08-17.

[5] D. K. Chen, J. Torrellas, and P. C. Yew. An efficient algorithm for the run-time parallelization of DOACROSS loops. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, Supercomputing '94, pages 518–527, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[6] X. Chen and S. Long. Adaptive multi-versioning for OpenMP parallelization via machine learning. In *2009 15th International Conference on Parallel and Distributed Systems (ICPADS)*, ICPADS '09, pages 907–912, Dec 2009.

[7] A. Jaleel. Memory characterization of workloads using instrumentation-driven simulation. `http://http://www.glue.umd.edu/~ajaleel/workload/`, 2007. Secondary source: `http://www.jaleels.org/ajaleel/workload/SPECanalysis.pdf`, Accessed: 2014-08-17.

[8] A. Jimborean, K. Koukos, V. Spiliopoulos, D. Black-Schaffer, and S. 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.

[9] M. Kamruzzaman, S. Swanson, and D. M. Tullsen. Inter-core prefetching for multicore processors using migrating helper threads. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 393–404, New York, NY, USA, 2011. ACM.

[10] K. Koukos, D. Black-Schaffer, V. Spiliopoulos, and S. Kaxiras. Towards more efficient execution: A decoupled access-execute approach. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 253–262, New York, NY, USA, 2013. ACM.

[11] L. Luo, Y. Chen, C. Wu, S. Long, and G. Fursin. Finding representative sets of optimizations for adaptive multiversioning applications. In *International Workshop on Statistical and Machine learning approaches to ARchitectures and compilaTion*, Paphos, Cyprus, Jan. 2009.

[12] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design.* Morgan Kaufmann, revised fourth edition, 2012.

[13] T. K. Prakash and L. Peng. Performance characterization of SPEC CPU2006 benchmarks on Intel Core 2 Duo processor. *ISAST Trans. Comput. Softw. Eng.*, 2(1):36–41, 2008.

[14] C. G. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 269–279, New York, NY, USA, 2005. ACM.

[15] J. H. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *Computers, IEEE Transactions on*, 40(5):603–612, May 1991.

[16] J. E. Smith. Decoupled access/execute computer architectures. *SIGARCH Comput. Archit. News*, 10(3):112–119, Apr. 1982.

[17] V. Spiliopoulos, S. Kaxiras, and G. Keramidas. Green governors: A framework for continuously adaptive DVFS. In *Proceedings of the 2011 International Green Computing Conference and Workshops*, IGCC '11, pages 1–8, Washington, DC, USA, 2011. IEEE Computer Society.

[18] W. Zhang, D. M. Tullsen, and B. Calder. Accelerating and adapting precomputation threads for effcient prefetching. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 85–95, Washington, DC, USA, 2007. IEEE Computer Society.