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

September 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

List of Figures

List of Tables

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
SSA	Static Single Assignment

Symbols

C	(effective) capacitance	F	$(\mathrm{s}^4{\cdot}\mathrm{A}^2{\cdot}\mathrm{m}^{-2}{\cdot}\ \mathrm{kg}^{-1})$
f	frequency	Hz	(s^{-1})
$P_{dynamic}$	dynamic power	W	$(\mathrm{kg}{\cdot}\mathrm{m}^2{\cdot}\mathrm{s}^{-3})$
V	voltage	V	(V)

Chapter 1

Introduction

Energy is not free. This statement is true especially when considering the limited capacity of batteries in the multitude handheld devices and portable computers that has 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 and costs related to that process.

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 factor 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 with the formula

$$P_{dynamic} = CV^2 f \quad [1, \text{p. } 39]. \tag{1.1}$$

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 the memory is limiting factor of the overall speed of the computer, the frequency of the processor can be lowered to match that speed.

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 power usage [1, 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 it 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. [2] 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 in changing the (voltage and) frequency prevents this fine scale DVFS from working well.

Decoupled Access-Execute (DAE) as treated by Koukos et al. [3] and Jimborean et al. [4] can help remedy this problem. At compile time the program is split into a series of tasks. From each of these tasks two phases are created, one phase containing a subset of the memory operations in the task to be *prefetched*, called the *access* phase, and one phase containing the original task, the *execute* phase. The phases are then scheduled to run after each other, first the access phase then the execute phase. This scheme provides a larger memory bound piece of code, the access phase, suitable for 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 incur a temporal overhead, as it adds more 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, 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 determine the cost of calculating a memory address and what the benefits of prefetching it are. 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, only one access phase generation procedure 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*. Memory operations are not always independent but can sometimes depend on the result of other memory operations. This is what constitute 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 it may vary. The rules differentiate access versions by giving them a limit to how many indirections any prefetch of a memory address may have. E.g. if an access version is built with rule 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 phases is, as a final product, useless without a way to evaluate each version during run time and allowing the best one to be in use most of the time. While it is not the goal of this project to produce such a mechanism it presents a reason to investigate the rules: with each added version the cost of selecting the best one increases. Each version also adds to the code size which is a reason of concern if there are too many. As such, it is important to control the number of generated versions. Therefore, this thesis seeks to explore the space of access versions based on different levels of maximum number of indirections. Next, the versions are compared from the points of view of performance and energy consumption, to determine which rules yield better results. This opens up the possibility of estimating the optimal rule, with respect to the code characteristics. Finally, one can reason about combining certain rules to generate more efficient versions.

Chapter 2

Related Work

One of the first mentions of DAE was made by Smith [5] and involved the utilisation of two separate programs, one in charge of memory operations and one for processing of data. This technique does however require two programs to be constructed and a hardware architecture that allows the programs to interact.

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 phases are memory bound and are consequently run at a low frequency to reduce power consumption. Their role is bringing data to cache in order to speed up the execute phase. The access phase was created manually by Koukos et al. [3] and then an automatic compiler feature was developed by Jimborean et al. [4].

The concept of using two phases running in sequence has been used before. Both Salz et al. [6] and Chen et al. [7] used an inspector phase to analyse dependence patterns in order to provide information to an execution phase 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. [8]. 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 require 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 [9, 10].

The multi-versioning technique has been used [11] for parallelising loops. 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. Based on information collected during run time the most suitable version would be selected for a particular task.

Luo et al. [12] has also directed attention to multi-versioning. Then in an attempt to create a framework to select a representative subset from a large number of differently optimised code versions. This was achieved by using heuristics to select good versions for a wide range of data sets. The goal was to keep the run time selection process simple and the code size small.

Chapter 3

Methodology

The mechanism to generate multiple access versions is implemented as a compiler pass using LLVM [13]. This chapter describes the workings of the compiler pass from a general point of view, the reason being that the choice of LLVM is not essential to the implementation. However, some LLVM tools have been used to ease the design of the pass. In such situation the details 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 separation is done using components from the previous project [4].

The functions on the critical path are pointed out as an interesting target 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, and its size is referred to as *granularity*. The loop is split into chunks and the granularity is adjusted experimentally, such that the workload 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. Note that, in most cases the chunking is performed on the outermost loop, however, in case the workload of only one iteration of the loop overflows the private cache, the child loop is chunked instead.

Insert image

The generation of the access phase begins with a clone of the execute phase (which is what is left by the chunking process). The goal with the access phase is to load as many global values as possible into cache for the execute phase, while keeping a low overhead. With this in mind everything not contributing to that goal is not allowed in the access phase. The structure of the access phase, the Control Flow Graph (CFG), is necessary and thus all instructions defining it is left in the access phase. Next, all loads of global values, i.e. data stored in memory not local to the access version, are identified. All loads of global values 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. This 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 two major parts to the access phase generation. First, the CFG skeleton must be identified in order to preserve it. How to do it is described in Section 3.1.1. Second, the loads of global values need to be identified and prefetches inserted. The general part of finding loads and inserting prefetches is described in Section 3.1.2. (The method to select among the prefetches to create different access versions is treated in Section 3.2.) When every instruction required for the access phase have been identified the rest are removed.

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. The respective sections (3.1.1 and 3.1.2) presents strategies of what to do to avoid side effects.

3.1.1 Control Flow Graph skeleton

To identify which instructions must be kept (collected in set K) in order to preserve the CFG, the following steps are taken:

Algorithm 3.1. Finding instructions forming the CFG skeleton, adding them to set K.

1. Find all instructions that directly defines the CFG, i.e. branch instructions, and add them to K.
2. Pick an instruction I from K and find all instructions that produces a value that I requires in order to execute. 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 I, which uses the address A, the following algorithm is used to produce the set S of potential sources. The algorithm make use of a set V to keep track of already visited instructions in order to provide an exit condition for the algorithm.

Algorithm 3.2. Finding potential sources for a load instruction I.

1. If I is in V, stop.
2. If I is a store instruction with address B, perform the following:
   (a) If A and B alias with some certainty, add I to S, and proceed to step 2b.
   (b) If A and B alias without doubt, stop.
3. If I is the first instruction in its basic block, apply this algorithm to each predecessor basic block in the reversed CFG, starting with their last instruction as I. Once the algorithm has stopped for each of the predecessors, stop also this instance of the algorithm. (Note that all instances of the algorithm share the sets S and V.)
4. Let the instruction that occurs immediately before I be denoted I instead and start over from step 1.

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 2a and 2b require an alias analysis to be performed. LLVM provide tools to perform that 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, this is what “alias without doubt” in step 2b refers to; (ii) *PartialAlias* meaning that the memory areas each address refer 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; (iv) *NoAlias* referring to the situation where the addresses cannot point to the same memory. The somewhat fussy formulation “alias with some certainty” in step 2a refers to PartialAlias (or stronger, i.e. MustAlias) in the implementation used in this project; it is, however, possible to use MustAlias or MayAlias instead but being too strict could leave out essential stores while being too lax may include unwanted stores causing other problems.

Once all instruction 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 may 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 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 it is used by a memory operation, most notably an indexing operation (*GetElementPtrInst*) and various types of casts (*CastInst*). If the input pointer to any of these types of instructions are local, the output pointer is considered local as well. Finally, there are 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, or external to the function.

3.1.2 Prefetches

The purpose of the access phase is to prefetch data to the cache to be 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 are detailed in Section 3.2, this section focuses on the necessary actions 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 delivering usable data. Sometimes the loaded data is required (to compute a control flow decision or the address of a prefetch) and thus 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.

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 below algorithm is used. Observe that 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. Finding dependencies of an address A.

1. If A is produced by an instruction, add it to D, otherwise stop.
2. Follow steps 2-4 from Algorithm 3.1 using D in place of K.

Upon inserting a prefetch, itself and the content of D should be added to K.

As with the CFG skeleton it is possible to run into situations where computing the address for a prefetch require 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 pointer address is available. This is in order to enable more aggressive dead code elimination.

3.2 Multi-versioning

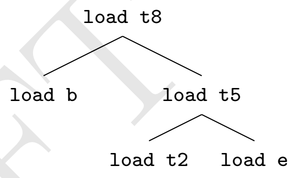
Multi-versioning is used to adjust the balance between cost and efficiency of the access phase. Multiple access versions are created, each employing its own policy for selecting which loads are going to be prefetched. Each access version is built with the same

```
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.)



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

Figure 3.1: An example of a load statement and different representations of it, ultimately arriving at the indirection measure for the load.

CFG skeleton described in Section 3.1.1. Prefetches are then inserted as detailed 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 exceed the threshold the prefetch is not allowed to be in that particular access version.

3.2.1 Indirection measure

Indirections are a measure of how many intermediate loads are required to compute an address. Figure 3.1 shows an example of how the indirection value of a load can be determined. In 3.1a there is a C statement where a value is ultimately loaded to the variable `x`. In 3.1b the C statement has been converted into a Static Single Assignment (SSA) form roughly representing 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 load depends on. 3.1c shows this tree with all non-load instructions removed. The indirection measure is then calculated by taking the size of the tree (in 3.1c), 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 3.1 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 3.2 illustrates an example where the origin load depends on six other loads. There are however no intermediate load with exactly three or four indirections as some loads depends 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 may be exactly the same as each other, while the version with a maximum of five indirections is different.

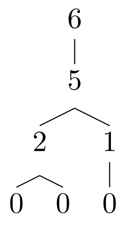


FIGURE 3.2: Tree of indirections illustrating that not all indirection levels have to be present. (Each node represents a load instruction with the number denoting how many load instructions it depends on.)

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 one chunk. Meanwhile, the time and energy performance of each access version is measured. The best performing version is selected to run the remaining chunks of the loop.

In this project the selection mechanism has been simplified. 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:

Consider adding an image

1. The source code are compiled into LLVM IR.
2. The LLVM code are prepared for chunking.
3. The relevant loops are identified and transformed to have it structured to run in chunks.
4. The pieces of code constituting the chunks are extracted to separate functions. These are the compute phases.
5. The newly separated functions are processed by the DAE pass to create access phases. The DAE pass are run several times, each time with different settings for the maximum number of allowed indirections, in order to create multiple versions of the access phases.
6. The compute and access phases are delimited by instrumentation code designed to measure performance (time and energy). PAPI [14] is used to make actual measurements.
7. Finalising optimisations are applied.
8. Binaries are created from the transformed LLVM code, one for each access version created.

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 testing.

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 for each 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 give in a scenario where all cores are occupied with a similar workload.

In order to achieve the desired energy savings the access phase should be run with a low frequency and the execute phase on a high frequency. 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 different 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 one. The low frequency run collects data regarding the execution of the access phases while the high frequency run provides information concerning the execute phases.

3.3.3 Results collection and processing

The measurement code included in the binaries provide a time measurement and an energy estimate for the execute and access phase separately. The data on the execute phase from the high frequency runs are brought together with the corresponding results on the access phase in 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 version as provides a standard way to weigh time and energy usage against each other.

The results from the CAE version is 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 i7-2600K processor and 16 GB RAM. The processor have 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 the experimental setup selected benchmarks from the SPEC CPU2006 [15] suit has 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. [16] and [17] has been guiding in selection where high L1d cache misses and high Cycles per Instruction (CPI) (or low Instructions per Cycle (IPC)) has been interpreted as an indication of memory bound applications. The benchmarks that has been selected are *mcf*, *milc*, *soplex*, *lbm* and *astar* as well as *hmmer* and *libquantum* where the former group is suspected to be most memory bound.

In addition to selecting benchmarks specific functions has to be targeted for DAE. In general the functions that the most time is spent in has been targeted. As a reference the function profile [18] of the SPEC CPU2006 benchmarks has been used. The targeted functions are displayed in Table 4.1.

SPEC CPU2006 provide inputs for all benchmarks and the reference versions has been used for the experiments. Some of the benchmarks (*soplex*, *hmmer* and *astar*) have multiple selections of inputs and have been run once with each.

The compilation procedure require granulates to be set and those selected (see Table 4.2) have been chosen following an inspection of the performance of the benchmarks while running on train inputs with an earlier version of the DAE framework.

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.
* While this function was targeted the compiler pass rejected it due to a function call that could not be determined to be safe.

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.

The energy estimate presented in the results are estimated using the power equation (1.1) and an experimentally determined estimation of the effective capacitance, C, based on IPC:

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

This estimation was done in [3] and is tailored for Sandybrige processors and SPEC benchmarks. Note that the energy levels only include dynamic power consumption.

4.2 Results

Table 4.3 displays the number of tasks, or rather chunks, that were run during each benchmark. Each task consist of an access phase and an execute phase each run for a number of iterations decided by the granularity.

Following graphs summarise the results obtained from running the various benchmarks. Each group of three graphs represent the characteristics of a specific combination of

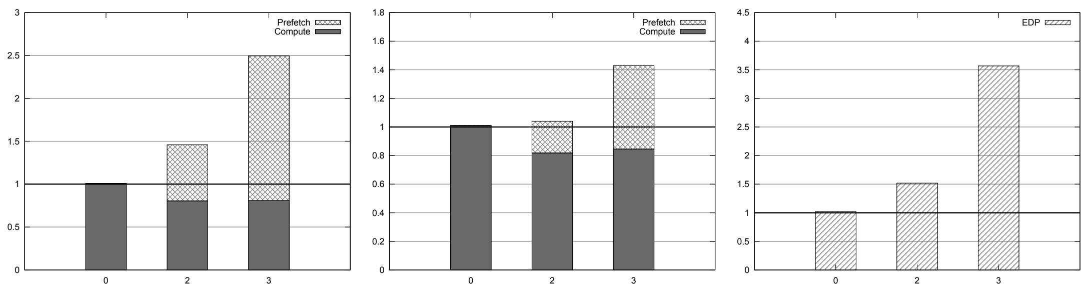
Benchmark	Task count
mcf	21854886
milc	460800000
soplex - ref	11262052
soplex - pds	22823811
hmmer - nph3	1272953
hmmer - retro	4944811
libquantum	94107092
lbm	26001000
astar - lakes	3148366
astar - rivers	21028729

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

		Benchmarks						
		mcf	**milc**	**soplex**	**hmmer**	**libquantum**	**lbm**	**astar**
Max Indirections	**0**	9	24	61	48	10	6	22
	1	–	42	92	91	12	–	64
	2	18	63	137	103	–	65	105
	3	20		152	140	21		–
	4			148	150			114
	5			–	155			121
	6			146	158			123
	7			145				
	8			148				
	9			148				
	10			149				
	11			148				
	12			150				
	15			149				

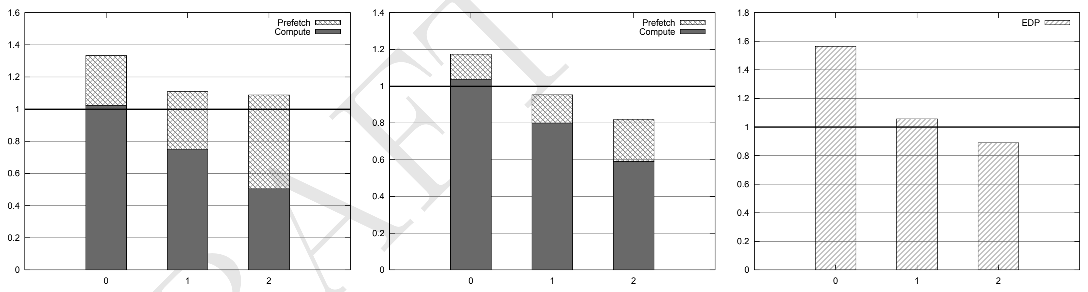
Table 4.4: The number of loads and prefetches in the access phases of each benchmark after optimisation.

benchmark and input. Common for all graphs is that the y-axis represent the relative performance of the DAE versions compared to the CAE version. The x-axis gives the maximum allowed number of indirections used in that particular version. Most graphs present a hole in the x-axis, this is because these versions are code wise the same as the previous more restrictive version. (E.g. if version 4 is not represented it is the same as version 3, or version 2 if version 3 is not represented, etc.)



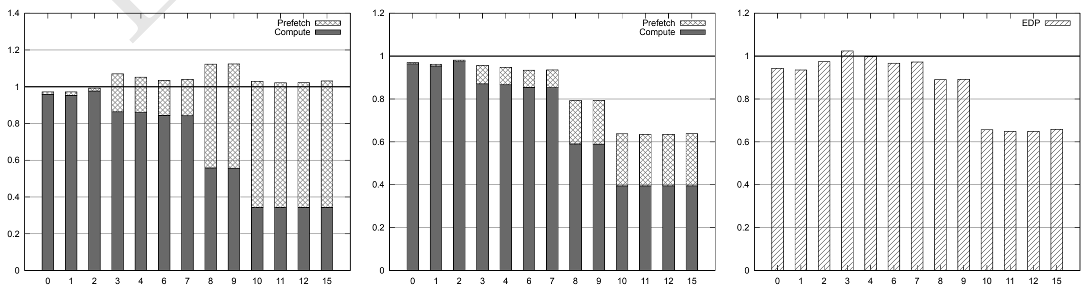
(A) Time (B) Energy (C) EDP

Figure 4.1: 429.mcf



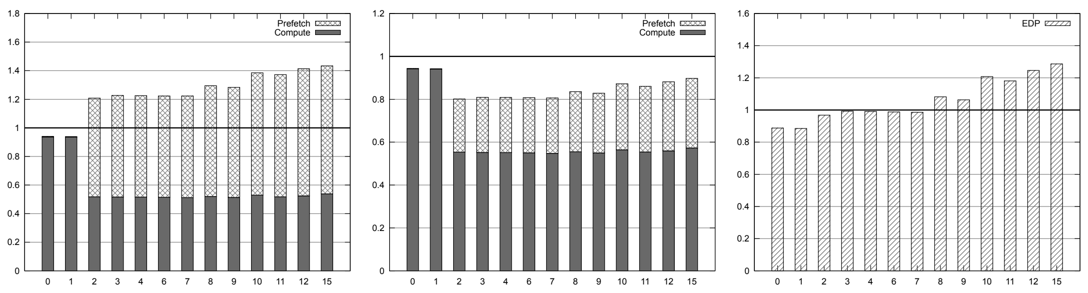
(A) Time (B) Energy (C) EDP

Figure 4.2: 433.milc



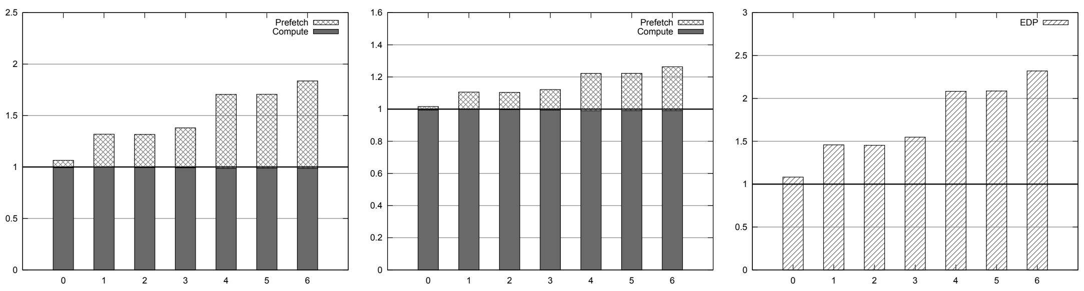
(A) Time (B) Energy (C) EDP

Figure 4.3: 450.soplex - pds



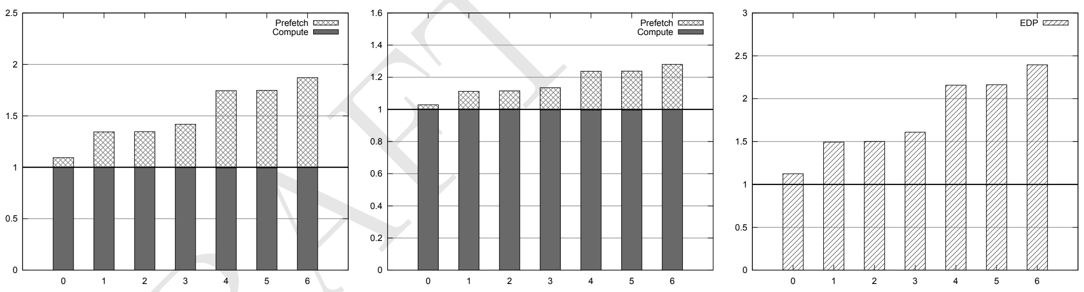
(A) Time (B) Energy (C) EDP

Figure 4.4: 450.soplex - ref



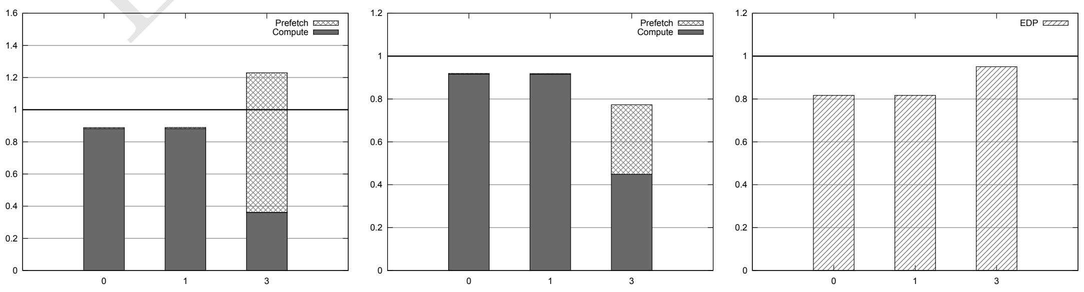
(A) Time (B) Energy (C) EDP

Figure 4.5: 456.hmmer - nph3



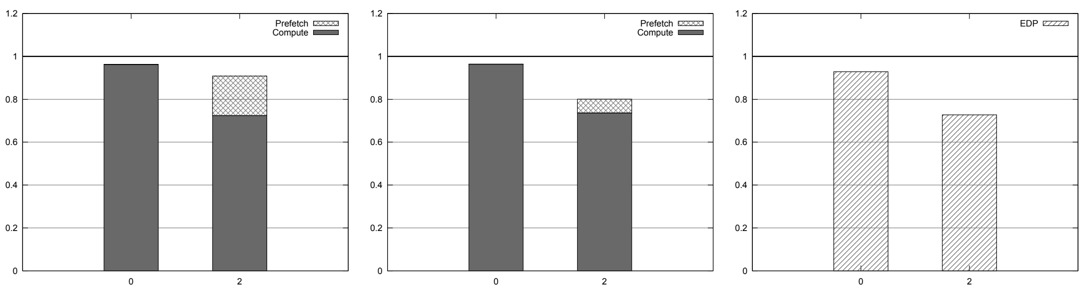
(A) Time (B) Energy (C) EDP

Figure 4.6: 456.hmmer - retro



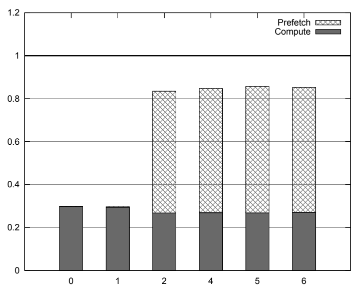
(A) Time (B) Energy (C) EDP

Figure 4.7: 462.libquantum

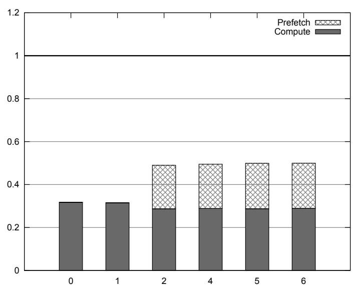


(A) Time (B) Energy (C) EDP

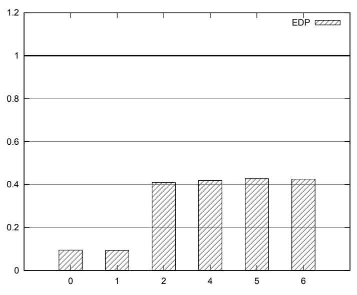
Figure 4.8: 470.lbm



(A) Time

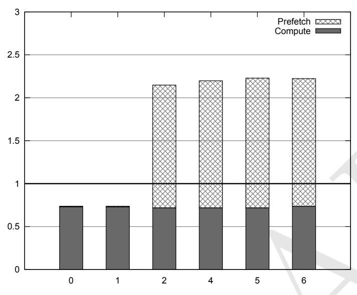


(B) Energy

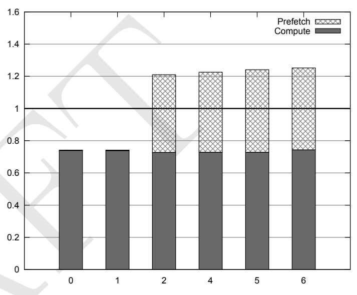


(C) EDP

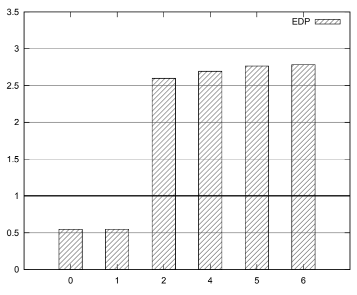
Figure 4.9: 473.astar - lakes



(A) Time



(B) Energy



(C) EDP

Figure 4.10: 473.astar - rivers

4.3 Discussion

Chapter 5

Conclusion

Appendix A

Implementation

Write your Appendix content here.

Bibliography

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

[2] V. Spiliopoulos, S. Kaxiras, and G. Keramidas. Green governors: A framework for continuously adaptive dvfs. In *IGCC'11*, July 2011.

[3] K. Koukos, D. Black-Schaffer, V. Spiliopoulos, and S. Kaxiras. Towards more efficient execution: A decoupled access-execute approach. In *ICS'13*, June 2013.

[4] 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 *GCO'14*, February 2014.

[5] J. E. Smith. Decoupled access/execute computer architectures. In *ACM Transactions on Computer Systems*, pages 289–308, November 1984.

[6] J. H. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. In *IEEE Transactions on Computers*, pages 603–612, May 1991.

[7] D. K. Chen, J. Torrellas, and P. C. Yew. An efficient algorithm for the run-time parallelization of doacross loops. In *ACM/IEEE Conference on Supercomputing*, pages 518–527, 1994.

[8] M. Kamruzzaman, S. Swanson, and D. M. Tullsen. Inter-core prefetching for multicore processors using migrating helper threads. In *ASPLOS'11*, pages 393–404, March 2011.

[9] 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 *PLDI'05*, pages 269–279, June 2005.

[10] W. Zhang, D. M. Tullsen, and B. Calder. Accelerating and adapting precomputation threads for efficient prefetching. In *HPCA'07*, pages 85–95, February 2007.

[11] X. Chen and S. Long. Adaptive multi-versioning for openmp parallelization via machine learning. In *ICPADS'09*, December 2009.

[12] L. Luo, Y. Chen, C. Wu, S. Long, and G. Fursin. Finding representative sets of optimizations for adaptive multiversioning applications. In *IWSMART'09*, 2009.

[13] Llvm. `http://llvm.org/`.

[14] Papi. `http://icl.cs.utk.edu/papi/`.

[15] Spec cpu2006. `http://www.spec.org/cpu2006/`.

[16] 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, 2007.

[17] 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.

[18] Spec cpu2006 function profile. `http://hpc.cs.tsinghua.edu.cn/research/cluster/SPEC2006Characterization/fprof.html`. Accessed: 2014-08-17.