Uppsala University

Bachelor's Thesis

Static Multi-Versioning for Efficient Prefetching

DRAFT

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

August 2014

DRAFT

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

DRAFT

DRAFT

List of Figures

DRAFT

List of Tables

DRAFT

Abbreviations

CAE Coupled Access-Execute

CFG Control Flow Graph

CPI Cycles per Instruction

DAE Decoupled Access-Execute

DVFS Dynamic Voltage and Frequency Scaling

IPC Instructions per Cycle

DRAFT

Chapter 1

Introduction

DRAFT

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 as much energy is still produced by unsustainable means. Decreased energy usage also means a decreased requirement for cooling and costs relates to that process.

As a result every 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 the 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 a the processor sometimes have wait for memory to finish an operation. Much work has been done to decrease the need for waiting, caches are a result from that work. Even with caches the processor occasionally have to wait for memory, and in doing that 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 as

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

The voltage and frequency factors can be changed to match the current need through the means of Dynamic Voltage and Frequency Scaling (DVFS). As a result of the equation even small changes can bring a significant power decrease since the voltage is a quadratic factor. However, as voltages has decreased in newer processor generations the transistors have begun to leak to the extent that it makes up a significant portion of the processor power usage [1, p. 40]. In these conditions smaller changes does not have as much effect on the power usage as it used to. As such it may not be possible to control voltage and frequency on a fine grain level in the future.

While smaller reductions in voltage and frequency may not incur very big energy saving larger might. The problem is that extensive reductions in voltage and frequency will hurt performance. In [2] an attempt was made 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 good when 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.

DRAFT

Decoupled Access-Execute (DAE) as treated in [3] and [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 can be created, one phase containing a subset of the memory operations in the task, 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. Correctly done this will provide a somewhat long memory bound piece of code suitable for low voltage/frequency followed by a compute bound section that would benefit from a high voltage/frequency. The key here is that while the access phase will incur a temporal overhead it will bring relevant data into cache preventing the execute phase from having to wait for memory. 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 technique that is investigated is called *multi-versioning*. The idea is to generate multiple versions of the access phase (different *access versions*) using different rules for each one. The usage of this technique is motivated by the fact that different programs behave differently and may therefore have different needs. As such only one access phase generation procedure might not cover the need of all programs that could benefit from DAE.

The set of rules explored is based on indirections. Memory operations are not always independent but can sometimes depend on the result of other memory operations. Depending on what is already available in the cache the cost of waiting for an indirection to finish may vary between different tasks. However, knowing beforehand which indirections that are costly is not easy. The speculative generation of multiple access versions, each including only memory operations with less than a certain number of indirections, can in conjunction with a way to evaluate the different versions during run time provide the possibility to select the best version for each task.

To generate too many versions is however not god as there is a cost associated to testing each version at run time as well as including them into the binary. Therefore, this thesis seeks to explore the different versions with different maximum depths of indirections to see if it is a good protocol at all and whether or not certain versions is more useful than others. The later question opens up the possibility to remove or combine certain version if there is a way to tell which ones that is.

DRAFT

Chapter 2

Related Work

DRAFT

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 rather rely on a DAE technique where sections of code are split into access and execute phases. The phases are the memory bound access phase are then run at a low voltage/frequency to reduce power consumption while 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. In [6, 7] an inspector phase was uses to analyse dependence patterns and 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 it is moved to the core where that data is already available in the cache. More research along similar paths has been done [9, 10].

Multiple code version has been uses in a similar way as in this thesis [11] where multiple versions were generated for parallelising loops. Each version would then spawn a different number of threads in order to find the best solution for a particular task.

Luo et. al. [12] has also directed attention to muli-versioning. Then in an attempt to create a framework using heuristics to select a representative set of differently optimised code versions providing the best optimisations for a wide range of data sets while keeping the run time selection overhead and code size small.

DRAFT

Chapter 3

Methodology

DRAFT

This chapter describes the workings of the compiler pass developed for this thesis from a general point of view without mentioning the particular tools that were used. For more information about the implementation details see Appendix A.

The compiler component developed is only concerned with the creation of an access phase (Section 3.1) and the duplication of it into different versions (Section 3.2). However, in order to do this the code must already contain separated tasks to process. This separation is done using components from previous project [4].

The task creation component expects functions to be pointed out as suitable for DAE. It then extracts the code in loops into separate tasks and prepares them to be run in *chunks*, i.e. once the phases has been created the access phase is allowed to be run for a certain number of iterations followed by the compute phase running the same number of times. The size of a chunk is referred to as *granularity*.

3.1 Access phase generation

When generating the access phase there are three goals to keep in mind: inserting prefetches at suitable locations; preserving the Control Flow Graph (CFG), including all instructions it depends on; and removing all side effects. The task to be converted into DAE is first copied to create separate access and execute phases that can be run sequentially. Once this has been done the instructions that are required to be in the access phase are identified and prefetches are inserted. Other instructions are removed.

3.1.1 Control Flow Graph skeleton

To find out which instructions that must be kept in order to leave the CFG in a working state following steps are taken in order to create the set K of instructions to keep:

Algorithm 3.1. Finding instructions forming the CFG skeleton.

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.
3. If I is a load instruction, find all store instructions that may produce the value that I later reads and add them to K. (See below, in Algorithm 3.2, for 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.

DRAFT

When searching for stores causing the value of 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:

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

1. If I has already been visited, stop.
2. If I is a store instruction with address B, perform following:
   (a) If A and B alias with some certainty, add I to S.
   (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 starting with their last instruction as I. Once the algorithm has stopped for each of the predecessors, stop also this instance. (Observe that all instances share the set S and the list of visited instructions.)
4. Let the instruction that occurs immediately before I be denoted I instead and start over from step 1.

Observe that steps 2a and 2b require an alias analysis (not described here) to be performed. The wording "alias with some certainty" describes a minimum which can mean anything from "cannot be proven not to alias" to "alias without doubt". Depending on the strictness of the interpretation different results may follow.

At this point all instruction that must be kept are known. If any of these instructions may cause side effects on data used outside of the access phase the task must be disqualified from DAE in order not to affect the results of the program in any way. At this point disqualification may occur for two reasons. If there are any function calls not marked as preserving the global state among K the access phase is discarded. Alternatively, if there are any store instructions touching memory not guaranteed to be local (details on how this is detected in this implementation are available in Appendix A (specify section)) the access phase must be abandoned as well.

DRAFT

3.1.2 Prefetches

The purpose of the prefetches, and the access phase in general, is to bring data into 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. Furthermore, if a load is present in the access phase (because it is required to compute a control flow decision or the address of a prefetch) it is not prefetched to avoid the overhead of accessing the same data twice, once with the prefetch and once with the load.

When a prefetch is inserted it is also necessary to make sure that all instructions required to compute the address A in question is present in the access version. To find the set D of such dependency instructions following algorithm is used:

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 D, are not added to K in order to keep the correctness of the program.

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 Multiversioning

When applying multiversioning more than one version of the access phase are created. In each access version the instructions making up the CFG skeleton are identified as described in Section 3.1.1. The difference between the versions is the policy used when selecting which loads that are going to be prefetched.

The rules used in this paper, to decide if a prefetch is inserted in the access version, are based upon the number of indirections required to calculate the address of where to prefetch from. A threshold is set for each of the access versions. If the number of indirections required for a prefetch exceed the threshold (or otherwise violates the rules mentioned in Section 3.1.2) the prefetch is not allowed to be in that particular access version.

DRAFT

3.2.1 Indirection measure

Indirections are a measure of how many intermediate loads that are required to compute an address. There are several conceivable ways to define the level of indirection for a load or prefetch. Different variant may use different rules to define what intermediate loads that are counted toward the indirection measure, in what way they are counted, and whether or not indirection counts are independent between different loads and prefetches.

The definition of the indirection measure used here follows Algorithm 3.4 to find the number of indirections required to compute an arbitrary address A (used by a load or prefetch):

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.

With this approach to measure the indirection level both situations when an address has to be loaded and when data, such as an index, is required from memory are covered.

(Pointer dereferences and array accesses in C are examples illustrating these circumstances). Another consequence of counting all loads among the dependency instructions of an address without discrimination is that every level of indirection does not have to be represented among the dependencies.

Figure 3.1 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 an access version with a threshold of four indirections may be exactly the same as the one with a threshold of three (and two in this particular example) while the version with a maximum of five indirections is different.

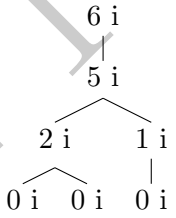


FIGURE 3.1: Tree of indirections illustrating that not all indirection levels have to be present. (Each node represents a load instruction with the number denoting how many indirections i depends on.)

DRAFT

3.3 Evaluation

Just like the rest of this chapter this section aims to describe the evaluation process from a general point of view. The details of the experiment are available in Chapter 4.

3.3.1 Compilation

When evaluating, this compilation pass is combined with other components providing the necessary transformations of a program to make it workable for the multi-versioning step.

As a fist step, once the code has been converted to an intermediate representation, certain functions, indicated beforehand, are analysed and loops are extracted into separate task. These functions will run a limited number of iterations (granularity) before they return and have to be called anew to continue the computation. This is to prevent the cache from filling up with the results of too many prefetches while running the compiled program.

The next step is the generation of an access phase. Each task are treated by the algorithms discussed in Section 3.1 and Section 3.2. The multiple versions of the access phase are created by allowing the access phase generation to run multiple times using different settings for the maximum number of allowed indirections.

The original task (the compute phase) and the access phase are enclosed with code making measurement to allow later analysis. Other general optimisations are applied as well including inlining of the compute phase. The compute phase could be inlined as well but inspection of the resulting code has shown that this sometimes cause common elements of the both phases to combine and thus only count toward the statistics for one of them. For this reason the access phase has been prevented from inlining while testing.

There are currently no feature to weave all these versions into the same binary but the characteristics of each version can still be tested separately by generating one binary for each level of indirection.

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.

DRAFT

3.3.2 Running

When running a program to assess the effect of the different access versions the program is run twice, once with the processor set to a low frequency and once with a high one. This is because there are no mechanism in place to change frequency between the execution of the access and execute phases. The low frequency run provide data about the access version while the high frequency run is for the execute phase.

While running a program with a certain configuration multiple instances of that same configuration 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 busy with a similar workload.

3.3.3 Result collection

The measurement code included in the binaries provide a time measurement and an energy estimate (based on the power equation (1.1) and the estimate of the effective capacity used in [3]) 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 measurement of efficiency, EDP (more on why this is useful) , can be calculated using the formula

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

The results from the CAE version is used to normalise the results of the multiple DAE versions.

DRAFT

Chapter 4

Experiments

DRAFT

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 [13] 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. [14] and [15] 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.

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 (reference)) 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.

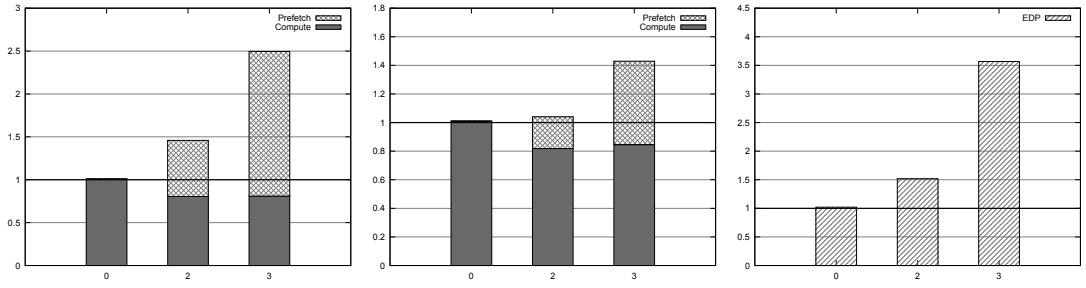
- Granularities
- Power estimation (Ceff)

4.2 Results

- Task count

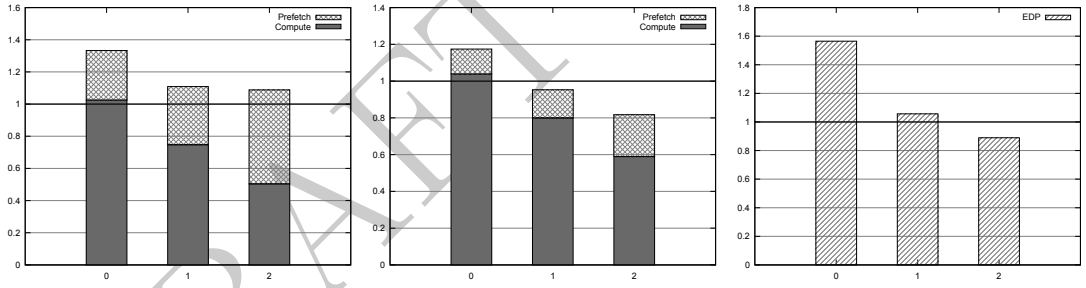
Following graphs summarise the results obtained from running the various benchmarks. Each group of three graphs represent the characteristics of a specific combination of benchmark and input. Common for all graphs is that the y-axis represent the relative performance of the DAE versions compared to the 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.)

DRAFT



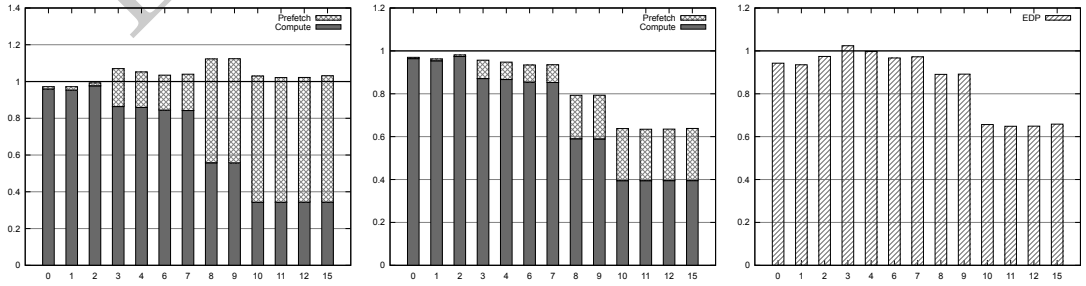
(A) Time (B) Energy (C) EDP

FIGURE 4.1: 429.mcf



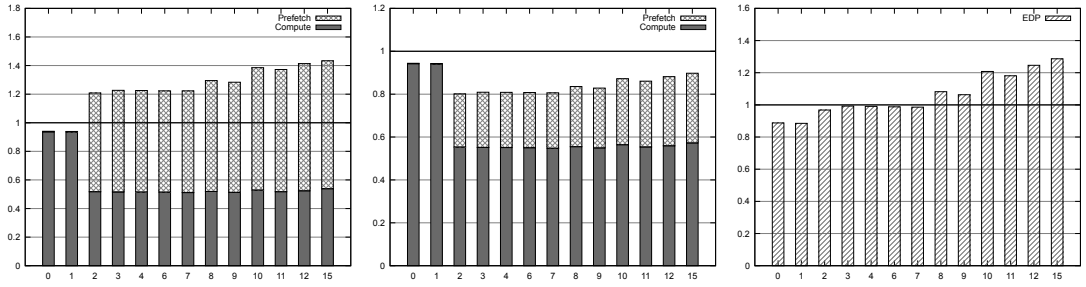
(A) Time (B) Energy (C) EDP

FIGURE 4.2: 433.milc



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

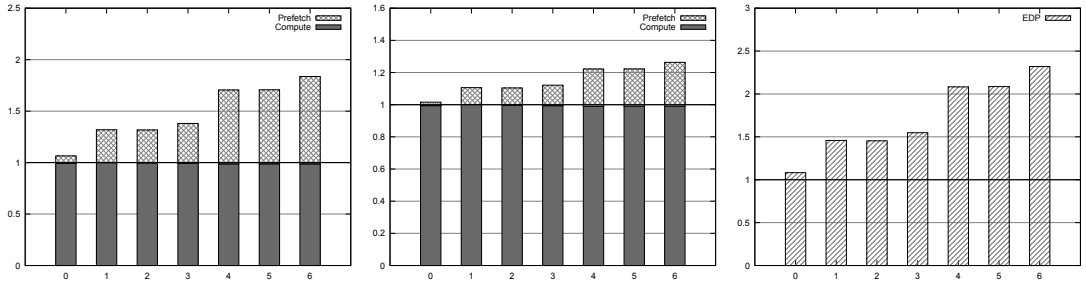
FIGURE 4.3: 450.soplex - pds



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

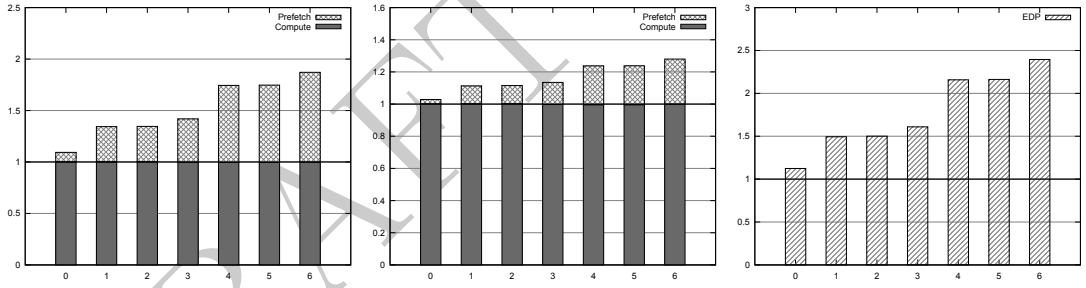
FIGURE 4.4: 450.soplex - ref

DRAFT



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

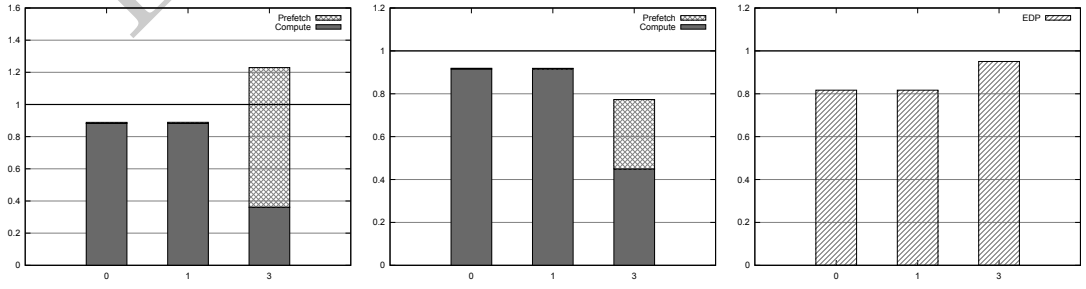
FIGURE 4.5: 456.hmmer - nph3



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

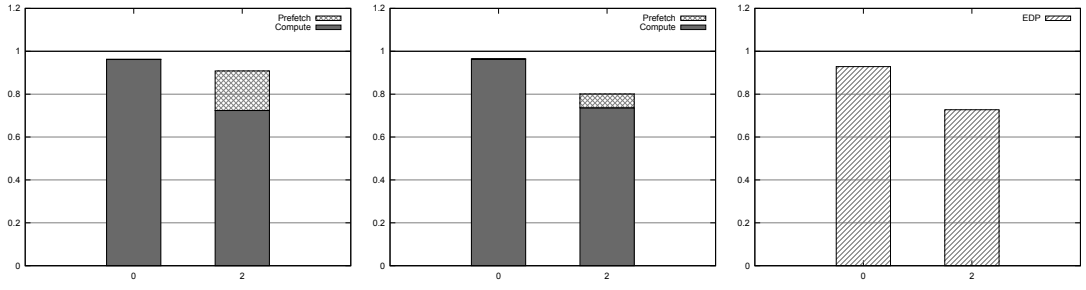
FIGURE 4.6: 456.hmmer - retro

DRAFT



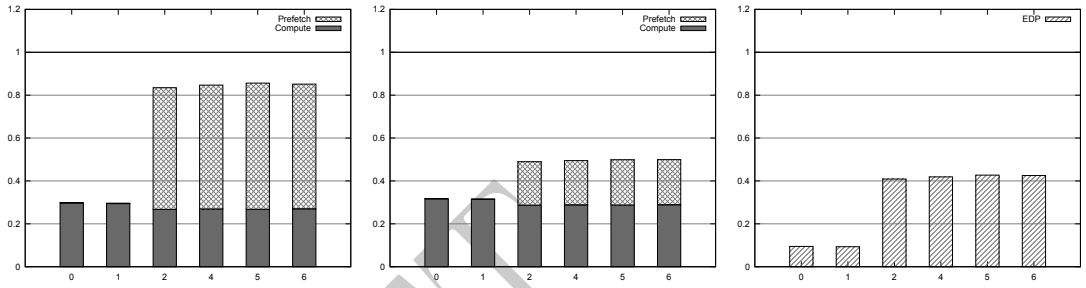
(A) Time (B) Energy (C) EDP

FIGURE 4.7: 462.libquantum



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

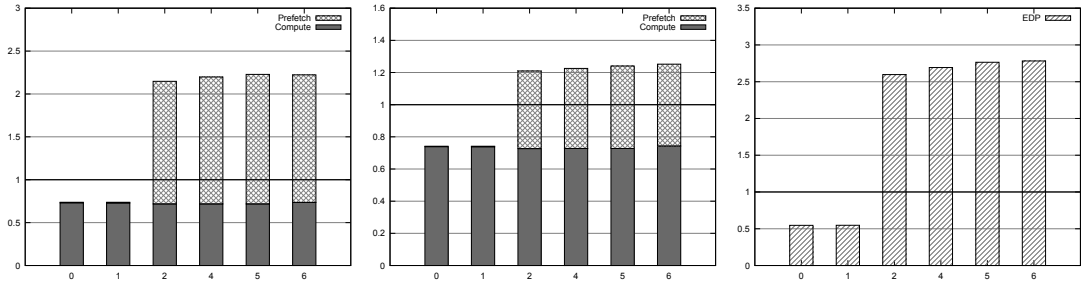
FIGURE 4.8: 470.lbm



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

FIGURE 4.9: 473.astar - lakes

DRAFT



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

FIGURE 4.10: 473.astar - rivers

4.3 Discussion

DRAFT

Chapter 5

Conclusion

DRAFT

Appendix A

Implementation

Write your Appendix content here.

DRAFT

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.

DRAFT

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

[14] 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.

[15] 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.

DRAFT