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

Research Group Name
Department of Information Technology

August 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

CFG Control Flow Graph

CAE Coupled Access-Execute

DAE Decoupled Access-Execute

Physical Constants

Speed of Light $c = 2.997\ 924\ 58 \times 10^8\ \mathrm{ms}^{-\mathrm{S}}$ (exact)

Symbols

a	distance	m
P	power	W (Js^{-1})
ω	angular frequency	rads^{-1}

Chapter 1

Introduction

General introduction...

1.1 Background

- Decreased power consumption through DVFS
  - Power equation
  - Breakdown of Dennard scaling
- Decoupled access/execute
  - Finding tasks suitable for optimisation
  - Breaking them down in small enough chunks
  - Creating the access version

1.2 Approach

- Multiple access versions
  The idea is to generate more than one access version. These are to be generated from different rule sets. The generated access versions are evaluated in a runtime system in order to determine which version that produces the best results for a specific task. That version is selected to be used for that task while others may use different access versions if they prove to be better.

Chapter 2

Related Work

- Fix the code [1]
- Towards More Efficient Execution [2]
- Polyhedral model and affine code
  (Not strictly required for this thesis.)

Chapter 3

Methodology

A few introductory words...

- Mention what a task is.
- Assume tasks to be available.

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 Decoupled Access-Execute (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.

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 (How can this be known?) the access phase must be abandoned as well.

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. (Again, how to know if it is global?) 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.

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.

3.3 Evaluation

This section could possibly be moved to the experiments chapter.

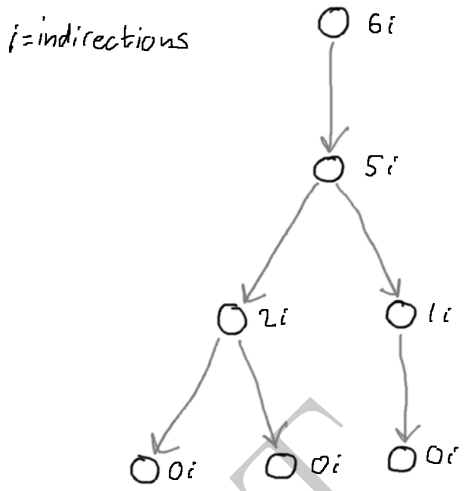

FIGURE 3.1: Tree of indirections illustrating that not all indirection levels have to be present. (A crude placeholder.)

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 multiversioning 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 functions or task as they are being called here. 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.

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 a energy estimate (more detail on this) 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

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

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

- Measurement

  - Energy estimation
  - EDP

TODO:

- Evaluation model
  - Power model
  - Measurement method
- Environment setup
  - LLVM 3.4
  - SPEC CPU2006

Chapter 4

Experiments

... or Evaluation

Chapter 5

Conclusion

Appendix A

Appendix Title Here

Write your Appendix content here.

Bibliography

[1] 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.

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