Annales Mathematicae et Informaticae
44 (2015) pp. 59–69
http://ami.ektf.hu

Lightweight simulation of programmable memory hierarchies*

Gergely Dévai

Eötvös Loránd University, Fac. of Informatics
deva@elte.hu

Submitted September 15, 2014 — Accepted May 13, 2015

Abstract

In most performance critical applications the bottleneck is data access. This problem is mitigated by applying memory hierarchies. There are application domains where the traditional, hard-wired cache control mechanisms are not satisfactory and the programmer is given full control where to place and when to move data. A known optimization technique on these kind of architectures is to do block-transfer of data instead of element-by-element access.

This contribution presents a lightweight library, written in and for C, to support experiments with algorithm performance on simulated programmable memory hierarchies. The library provides functions and macros to define memory layers, available block-transfer operations and data layout. The algorithm then can be run on a simple desktop computer and the library combines its real runtime with simulated memory access penalties.

Keywords: Memory hierarchy, block transfer, simulation

MSC: 68U20

1. Introduction

Memory access is the performance bottleneck of many computer architectures, because memory is an order slower than processors. This is mitigated by applying a memory hierarchy ranging from very fast but small to slow but large memory layers. If data is loaded to the faster layers by an automatic policy hidden from

*Supported by EITKIC 12-1-2012-0001.

the programmer we speak about cache layers. In some application areas, however, these policies do not perform well enough. In these cases the programmer is given control over the faster memory layers: She or he is in charge of which data to store in which layer and when to move data between the layers.

Platforms with programmable memory hierarchy usually provide special block copy operations: These can move given amount of data (typically 8, 16 or 32 bytes) from one layer to another. The cost of these operations in terms of waiting for the memory is similar to a single byte memory access. These operations make the so called bulk access optimization possible: A large array is stored in slow memory that needs to be processed elementwise. This task can be done by copying chunks of the array using block copy operations one by one to fast memory and process the chunks there. If the processing involves modification of the array then the processed chunk is copied back to its original location by a second block copy.

The goal of the created C library is to provide means of experimentation with bulk access and related optimizations without actually having a hardware platform with programmable memory hierarchy. Detailed requirements:

- The library has to provide means to define the memory hierarchy, together with the access costs.
- It has to provide means to define which object is placed in which layer.
- To be able to use the library to measure memory-related performance of an existing C program should only require minimal changes to the program.
- The library does not have to provide cycle accurate results of a given platform, instead it has to enable estimations for any platform with a memory hierarchy configured by the user of the library.

In the next section, related work is presented. Then, section 3 describes how to use our C library, followed by the implementation details in section 4. Section 5 presents measurement results of various experiments using the simulation library.

2. Related work

The theoretical background of architectures featuring storage elements with different access costs and providing block copy like instructions has been established decades ago [1]. Regarding the technical aspects of such architectures, paper [4] describes the details of efficiently using the memory hierarchies. However, this paper mainly concentrates on traditional cache systems, as opposed to programmable memory hierarchies, which is the main focus of our work.

Software controlled memory hierarchies generated a large amount of research about related optimization problems. Let us mention a few of these: [9] discusses heuristics to find good data layout, [10] concentrates on the optimization of energy consumption and also deals with allocation instructions. Paper [6] shows a dynamic memory management solution.

Complementary to the optimization aspect, there have been experiments to increase the abstraction level of programming software controlled memory hierarchies. Several extensions to existing programming languages [2, 3] or newly proposed languages [5] emerged. However, in practice, C is still the most widely used language to program these performance oriented platforms. For this reason C is selected as the base language of the simulation solution of this paper.

Simulation of program behavior on special hardware is important because it enables software evaluation without actual deployment. Another use case is the pre-evaluation of not yet existing hardware. There are simulators targeting specific hardware platforms accurately or aiming at the simulation of many different aspects of software behavior, eg. [8]. The XMSIM system [7] uses code transformations and C++ to evaluate memory hierarchies. It is, in many aspects, similar to the work presented in this paper. The main differences are that our solution uses C, and it is much simpler and lighter weight.

3. Users' perspective

This section presents the API provided by our library. Figures 1 and 2 show a possible architecture definition and a program using it. The API elements demonstrated by these examples are explained in the following subsections in detail.

```
#ifndef __EXAMPLE_ARCHITECTURE_H
#define __EXAMPLE_ARCHITECTURE_H

enum memory { scratch = 10, ram = 100 };

enum copy_size { _8bytes = 8,
                 _16bytes = 16,
                 _32bytes = 32
               };

#endif
```

Figure 1: Example memory architecture definition (`example_architecture.h`)

3.1. Architecture definition

The first step of using the library is to describe the basic properties of the memory hierarchy to be simulated in a header file, see figure 1. Types `memory` and `copy_size` have to be defined as enumeration types, like in the following example.

```
enum memory { scratch = 10, ram = 100 };
```

```
#define ARCHITECTURE "example_architecture.h"
#include "memory.h"
#include <stdio.h>

void calibration_function() {
    static int i, data[128];
    for(i=0; i<128; ++i)
        data[i] = i;
}

int main() {
    calibrate(&calibration_function, 4*128);
    int i;
    char sum;
    char _a[8];
    #define a access(_a,scratch)
    char _b[256];
    #define b access(_b,ram)
    for( i=0; i<256; ++i)
        b[i] = i;
    start();
    sum = 0;
    for( i=0; i<256; ++i)
        sum += b[i];
    long long elapsed = stop();
    printf("Simple solution: %lld microseconds.\n",
            elapsed );
    start();
    sum = 0;
    for( i=0; i<256; i+=_8bytes ) {
        block_copy(_a,scratch,&(_b[i]),ram,_8bytes);
        sum += a[0] + a[1] + a[2] + a[3]
             + a[4] + a[5] + a[6] + a[7];
    }
    elapsed = stop();
    printf("Block copy solution: %lld microseconds.\n",
            elapsed );
    return 0;
}
```

Figure 2: Summing an array with and without block copying

```
enum copy_size { _8bytes = 8, _16bytes = 16, _32bytes = 32 };
```

Here, two memory layers are defined: scratch is a relatively fast memory, its access costs ten cycles, while accessing ram takes 100 clock cycles. The example also defines three different block copy widths: 8, 16 and 32 bytes respectively.

3.2. Importing the architecture

The next step is to implement the algorithms to be measured in some C source file(s). Let us assume that the header file `example_architecture.h` defines the simulated architecture. One can refer to it from the source files as seen in lines 1-2 of figure 2:

```
#define ARCHITECTURE "example_architecture.h"
#include "memory.h"
```

The `ARCHITECTURE` symbol defines the name of the header file describing the architecture, and `memory.h` is the header file providing the block copy operations and utility functions to measure simulated execution time.

3.3. Variable definition and access

Lines 15-18 of the example uses special notation to declare the variables that we want to store in one of the simulated memory layers.

```
char _a[256];
#define a access(_a,scratch)
```

The previous two lines define a character array of length 256 stored on the memory layer called `scratch`. The real name of this object, seen by the C compiler is `_a`. Using a macro definition, we create a synonym (`a`) for this variable and specify a memory layer (`scratch`) for it. Whenever the algorithm accesses the data of this object, the synonym has to be used. This enables the simulation library to incorporate the configured memory access cost in the execution time results. On the other hand, if the object is used without data access, for example, as an argument of the `sizeof` or the `&` operators, the real variable name (`_a`) has to be used.

3.4. Block copy operations

Block copy operations are simulated by calling the following function:

```
void* block_copy( void* destination, enum memory dest_mem,
                  void* source, enum memory src_mem,
                  enum copy_size size );
```

Destination and source are two addresses to copy to and from respectively. Parameters `dest_mem` and `src_mem` are memory layers of the destination and the source: these can be values of the memory enumeration type as given in the architecture description. The last parameter, `size`, is one of the values of the `copy_size` enumeration type as given in the architecture description: It defines the number of bytes to copy.

Line 31 of figure 2 demonstrates a call to this function. It copies 8 bytes from array `b` in RAM to array `a` stored on the scratchpad.

3.5. Performance measurement

Before starting the first measurement cycle it is important to call the

```
double calibrate( void (*f)(), long long cycles );
```

function. The parameters are a calibration function and the number of clock cycles the function would take on the simulated hardware to execute. The `calibrate` function calculates the execution time of simulated clock cycle on the current machine. See line 12 of figure 2, where this calibration takes place.

In order to get the simulated execution time of a piece of code, it needs to be surrounded by calls to the `start()` and `stop()` functions. The latter one returns the simulated execution time in clock cycles. Figure 2 performs two such measurements, between lines 21-25 and 28-35.

4. Implementation

The C library is implemented in a single header file in terms of macros and inline functions. It first includes the header file indicated by the `ARCHITECTURE` symbol to get access to the configuration of the simulated platform.

The `calibrate` function executes the calibration function in its first parameter, measures its execution time, which is then divided by the number of clock cycles in the second parameter. The result is the amount of time that a clock cycle of the simulated machine takes on the simulator machine. This is saved for later use.

The `start` function initializes `timeval` structures from the `sys/time.h` standard header to measure execution time up to the execution of the `stop` function. This amount of time is converted to simulated machine clock cycles according to the ratio given by the calibration. This is one component of the end result.

The other component is the simulated memory access cost. Each object synonyme (see section 3.3) in the program is expanded to a call to the `access` macro. Its definition is as follows:

```
#define access(var,mem) (*(stall(mem),&var))
```

This is an expression composed by the comma (sequencing) operator of C. First, the `stall` function is called with the actual memory layer. Recall that each layer is an element in an enumeration type, and its integer value is the number of clock cycles the memory access takes on the simulated platform. The `stall` function simply accumulates these simulated stall times, which is the second component of the simulation's result. The left hand side of the comma operator defines the value of the expression, which is the address of the accessed variable. The dereferencing operator (`*`) takes this address and returns the variable itself (`var`).

The reason for using the combination of the operators `&` and `*` instead of simply writing `(stall(mem),var)` is that the latter expression is not a left-value and would not allow the programmer to use the object synonyms on the left-hand side of assignment operations, for example.

The `block_copy` function computes the maximum stall of the destination and source memory layers and uses this maximum as the parameter to the `stall` function. The functionality of the operation is implemented by a standard `memcpy` call.

5. Experiments

The measurements were carried out on an Acer TravelMate 8572TG laptop with Intel Core i5-460M processor (2.53 GHz, 3MB L3 cache) and 4 GB DDR3 memory. The following diagrams show the simulated clock cycles translated back to microseconds according to the capabilities of this machine.

5.1. Proof-of-concept

In the first, basic experiment we have implemented the summation of a 1024 byte long unsigned character array two different ways. The array is configured to be stored in a memory layer causing 100 cycles average stall time at each access.

The first solution processes the array element-by-element. The second one copies array chunks to faster memory and processes data there. The buffer used to store the chunks has 10 cycles average stall time and the block-copy operations used are 32 bytes wide.

In this setup, the block copy solution results in more than 4 times speedup, according to our simulation. That is, block copying array chunks to faster memory and processing data there is faster than element-wise processing of the original array in slow memory.

5.2. Slowdown caused by increasing memory access penalties

This experiment examines the effect of increasing stall time to the run time of summing the elements of a 1024 element character array. The computation is done element-by-element, without block copy operations. The stall time of the memory layer storing the array was increased from 2 cycles up to 256.

The diamonds on the graph show linear slowdown for reference. The runtime results (squares) show that slowdown is smaller than linear in case of low stall times and it converges to linear slowdown as stall time increases. This is because, in the first case, the stall is not overwhelming the overall runtime of the algorithm. As stalls get longer, they become the most significant factor. This observation

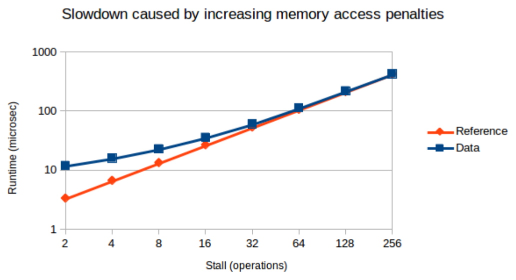


Figure 3: Slowdown caused by increasing memory access penalties

suggests the next experiment that examines the stall time versus the computation time of algorithms.

5.3. Stall – computation ratio

This experiment examines the effect of increasing the number of arithmetic instructions while keeping the stall time constant. The processed array is 3000 bytes long with unsigned character elements. Each arithmetic instruction is multiplication, and no block copying is used. Two memory layers were tested, with stall time of 2 and 256 cycles respectively.

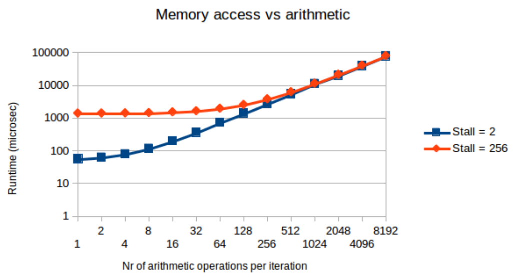
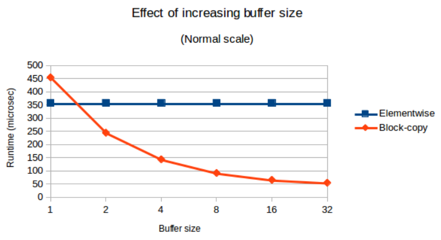


Figure 4: Memory access vs. computation

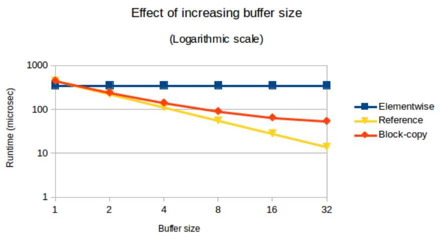
In case of small number of arithmetic instructions the stall time is the most significant factor of runtime, therefore there is considerable difference between the fast and slow memory layers. Increasing the number of arithmetic instructions causes sub-linear slowdown in this case. As more and more computation overwhelms stall time, slowdown tends to be linear and the difference between memory layers disappears.

5.4. The effect of changing block-copy size

This experiment examines the effect of increasing block-copy width. The algorithm used is scalar product working on two unsigned character arrays of 1024 elements length each. These are stored in a memory layer with 128 cycles long stall, while the buffer used is stored in memory with only 8 cycles long stall time. The block-copy width was increased from 1 byte to 32 bytes.



(a) Normal scale



(b) Logarithmic scale

Figure 5: Effect of increasing buffer size

The results show that block-copying only single bytes chunks actually degrades performance compared to the element-wise solution. This is expected and shows the pure cost of the complication of the algorithm caused by block copying. Copying 2 array elements in each iteration already worth it. Figure 5b shows the speedup on logarithmic scale with the triangles depicting linear speedup for reference. Speedup is close to linear in the beginning, but using more and more powerful block-copy instructions result in sub-linear speedups, but still decreasing runtime considerably. The reason for sub-linearity is that the constant non-stall related cost of the algorithm gets more and more significant as stall time is successfully reduced by more and more effective block-copy instructions.

5.5. Array-of-structs vs. struct-of-arrays

Block-copy related optimizations often require data layout changes. A typical example is the array-of-structs to struct-of-arrays transformation. This experiment uses an algorithm that counts the RGB-coloured pixels with red component stronger than a threshold. The first data layout is array-of-structs: An array with 1024 structs, each storing the RGB values of a pixel. The array is stored in a memory layer with 100 instructions stall time. The first implementation processes the array element-wise, the second one uses 32 bytes wide block-copy operations. This is expected to result in considerable speedup, however, it copies the green and blue components of the RGB structs superfluously. In order to improve the solution further, a data layout change is needed: The second layout uses a struct of three arrays, storing the 1024 red, green and blue components respectively. In this case the red components are adjacent in memory, leading to much effective block-copying.

The simulation data showed almost 3.5 times speedup due to the introduction of block copying on the first data layout. The layout change results in more than 1.5 times speedup between the two block-copying implementations. At this point, one could expect more than 1.5 speedup, considering that 2/3 of data copying is spared. In applications using structs with more fields this speedup could certainly be increased.

6. Summary

This paper presented a light-weight library, written in C, that allows programmers to simulate programmable memory hierarchies. The memory hierarchy, its stall times and the data layout of the program can be freely configured. Only minor changes are required in existing C code to make the simulation possible.

The paper discussed the API of the library, the implementation details and presented five different experiments carried out by using the library. On one hand, the results of the experiments are interesting because they reveal behavioral features of memory hierarchies in detail. On the other hand, the results can be understood and explained, which increases confidence in the simulation library itself.

References

[1] Alok Aggarwal, Ashok K Chandra, and Marc Snir. Hierarchical memory with block transfer. In *28th Annual Symposium on Foundations of Computer Science*, pages 204–216. IEEE, 1987.

[2] Alex Aiken, Phil Colella, David Gay, Susan Graham, Paul Hilfinger, Arvind Krishnamurthy, Ben Liblit, Carleton Miyamoto, Geoff Pike, Luigi Semenzato, et al. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 10:11–13, 1998.

[3] David E Culler, Andrea Dusseau, Seth C Goldstein, Arvind Krishnamurthy, Steven Lumetta, Steve Luna, Thorsten von Eicken, and Katherine Yelick. Introduction to Split-C. 1995.

[4] Ulrich Drepper. What every programmer should know about memory. *Red Hat, Inc*, 11, 2007.

[5] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J Dally, et al. Sequoia: programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83. ACM, 2006.

[6] Mahmut Kandemir, J Ramanujam, J Irwin, Narayanan Vijaykrishnan, Ismail Kadayif, and Amisha Parikh. Dynamic management of scratch-pad memory space. In *Proceedings of the 38th annual Design Automation Conference*, pages 690–695. ACM, 2001.

[7] Theodoros Lioris, Grigoris Dimitroulakos, and Kostas Masselos. Xmsim: Extensible memory simulator for early memory hierarchy evaluation. In *VLSI 2010 Annual Symposium*, pages 199–216. Springer, 2011.

[8] Peter Magnusson and Bengt Werner. Efficient memory simulation in simics. In *Simulation Symposium, 1995., Proceedings of the 28th Annual*, pages 62–73. IEEE, 1995.

[9] Preeti Ranjan Panda, Nikil D Dutt, and Alexandru Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *Proceedings of the 1997 European conference on Design and Test*, page 7. IEEE Computer Society, 1997.

[10] Stefan Steinke, Lars Wehmeyer, Bo-Sik Lee, and Peter Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 409–415. IEEE, 2002.