

UNIVERSITY OF THESSALY

DIPLOMA THESIS

**ANALYSIS OF DATA PLACEMENT
TECHNIQUES IN HETEROGENEOUS
MEMORIES**

Author:

Dimitrios VOULGARIS

Supervisor:

Nikolaos BELLAS

Examiners:

Dimitrios KSATSAROS

Spyros LALIS

*A thesis submitted in fulfillment of the requirements for
the degree of Diploma*

Volos, October 2019



UNIVERSITY OF THESSALY

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

**ΑΝΑΛΥΣΗ ΤΕΧΝΙΚΩΝ ΤΟΠΟΘΕΤΗΣΗΣ
ΔΕΔΟΜΕΝΩΝ ΣΕ ΕΤΕΡΟΓΕΝΕΙΣ ΜΝΗΜΕΣ**

Συγγραφέας:

Δημήτριος ΒΟΥΛΓΑΡΗΣ

Επιβλέπων:

Νικόλαος ΜΠΕΛΛΑΣ

Εξεταστική επιτροπή:

Δημήτριος ΚΑΤΣΑΡΟΣ

Σπύρος ΛΑΛΗΣ

Βόλος, Οκτώβριος 2019

Acknowledgements

My greatest “Thank You” to my supervisor, Prof. Nikolaos Bellas. Once the experimental process was completed, writing of this thesis would be impossible without his assistance. His door was always open for valuable advice and continuous steering.

This work was partly completed in Barcelona Supercomputing Center (*BSC*) in the frames of “*PRACE SoHPC*” program. I am therefore grateful for sharing their tools and source code with me. In particular I consider myself greatly indebted to researchers Mr. Antonio Peña and Mr. Marc Jorda, as well as to all members of “*Accelerators and Communications for HPC*” team of the “*Programming Models*” group at the “*Computer Sciences*” department of Barcelona Supercomputing Center (*BSC*). They offered immense help in clarifying the experimentation process and demonstrating the tools to be used while tided me over every time a trouble was encountered.

Finally I couldn’t omit my beloved friends and family for their unconditional support and patience all these years. Their motivational words and encouragement was crucial.

Thank you all!

UNIVERSITY OF THESSALY

Abstract

Department of Electrical and Computer Engineering

Diploma

Analysis of data placement techniques in heterogeneous memories

By **Dimitrios VOULGARIS**

Heterogeneous memory systems have been recently introduced as a consequence of the continuously growing demand for fast accessible data. Simply assigning random data to the various memory subsystems of a memory system is not working as expected; on the contrary it might prove performance limiting. In this work we employ software profiling techniques using Valgrind (i.e. EVOP) in order to identify the memory access behavior of specific scientific applications and optimally distribute their data in a hybrid memory system with the ultimate scope of achieving a performance improvement. Doing so, we provide a detailed description of Valgrind's source code which reveals a potential bug but also acts in an explanatory way for future users and developers.

We also facilitated sampled memory access profiling by extending the basic tool's source code. In that aspect we provide a comprehensive report of our development stages as well as the available accumulated features. Profiling, in that case, enabled a comparative research between sampled and non-sampled results which eventually allowed us to draw conclusions that connect the application-characteristic memory access pattern and sampling periods that generate optimal performance speedup.

Περίληψη

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Διπλωματική Εργασία

Ανάλυση Τεχνικών Τοποθέτησης Δεδομένων σε Ετερογενείς Μνήμες

από τον Δημήτριο ΒΟΥΛΓΑΡΗ

Τα ετερογενή συστήματα μνήμης παρουσιάστηκαν ως απάντηση στην συνεχώς αυξανόμενη ανάγκη γρήγορης πρόσβασης σε δεδομένα. Μια απλή κατανομή των δεδομένων εφαρμογής στα διάφορα υποσυστήματα ενός συστήματος μνήμης, ωστόσο, δεν εγγυάται την βελτίωση της απόδοσης από άποψη χρόνου. Αντίθετα, μπορεί να την επιδεινώσει περαιτέρω. Σε αυτή την εργασία κάνουμε χρήση software profiling μεθόδων επιστρατεύοντας το εργαλείο Valgrind με σκοπό να καθορίσουμε την αλληλεπίδραση με την μνήμη που παρουσιάζουν συγκεκριμένες επιστημονικές εφαρμογές. Στην συνέχεια, κατανέμουμε τα δεδομένα αυτών στο υβριδικό σύστημα μνήμης με τελικό σκοπό την αύξηση της απόδοσης. Στα περιεχόμενα της εργασίας παρέχεται μια λεπτομερής περιγραφή του πηγαίου κώδικα (source code) του Valgrind η οποία αφενός φέρνει στην επιφάνεια ένα πιθανό bug του κώδικα και αφετέρου χρησιμεύει ως σημείο αναφοράς για δυνητικούς χρήστες και προγραμματιστές.

Επίσης, επεκτείνοντας το Valgrind καταστήσαμε δυνατή την δειγματοληπτική καταγραφή προσβάσεων στην μνήμη παραθέτοντας μια εκτενή καταγραφή των βημάτων που ακολουθήθηκαν αλλά και των επιλογών χρήστη που προστέθηκαν. Οι προσθήκες αυτές επέτρεψαν μια συγκριτική μελέτη μεταξύ αποτελεσμάτων που προήλθαν από δειγματοληψία και αυτών που λήφθηκαν από το κλασικό profiling. Τα αποτελέσματα συνδέουν το χαρακτηριστικό ανά εφαρμογή μοτίβο προσπέλασης μνήμης με την επιτρεπτή συχνότητα δειγματοληψίας που επιφέρει μέγιστη βελτίωση απόδοσης.

Copyright © 2019 by Dimitrios Voulgaris

“The copyright of this thesis rests with the authors. No quotations from it should be published without the authors’ prior written consent and information derived from it should be acknowledged”.

Contents

Acknowledgements.....	iii
Abstract.....	iv
1. Introduction	1
1.1 Background.....	1
1.1.1 General.....	1
1.1.2 Memory architectures.....	3
1.2 Problem Statement and Contributions.....	5
1.3 Thesis structure.....	6
2. Valgrind	7
2.1 Overview.....	7
2.2 Callgrind.....	8
2.2.1 Source code instrumentation.....	9
2.2.2 Callgrind options.....	10
2.3 Last level cache simulation.....	10
2.3.1 The matrix multiplication example.....	10
2.3.2 Source code inspection.....	12
3. Valgrind Tool Extension	14
3.1 Overview.....	14
3.2 Callgrind extension.....	15
3.2.1 API options.....	15
3.2.2 Sampling implementation.....	16
3.2.3	

4. Experimentation	19
4.1 Profiling.....	19
4.1.1 Sampling period.....	21
4.2 Analysis.....	21
4.3 Technical characteristics.....	22
4.3.1 Heterogeneous memory system.....	22
4.3.2 Test cases.....	22
5. Results	25
5.1 MiniMD.....	27
5.1.1 Complete memory access profiling.....	27
5.1.2 Sampled memory access profiling.....	28
5.1.3 Sampling period comparison – Conclusions	29
5.2 HPCCG.....	31
5.2.1 Complete memory access profiling.....	31
5.2.2 Sampled memory access profiling.....	31
5.2.3 Sampling period comparison – Conclusions.....	33
5.3 General conclusions.....	34
5.3.1 Performance.....	34
5.3.2 Sampling.....	34
6. Summary	36
6.1 Future research.....	37
Bibliography	39

List of FIGURES

1.1	Hierarchical versus equally managed memory view.....	4
2.1	Simplified high-level view of the interaction between Valgrind and its tools.	7
2.2	Warning informing about the LL cache configuration.	10
2.3	<code>cg_arch.c</code> function flow chart.	13
5.1	Object distribution for miniMD in Scenario 1 for zero sampling period. The objects in bold, placed in SP memory are discarded when performing sampled memory access profiling.	26
5.2	Object distribution for MiniMD – Scenario 1 – Sampling period: 52,177.....	27
5.3	Correlation between final speedup and sampling period for miniMD test case in Scenario 1.	29
5.4	Correlation between final speedup and sampling period for HPCCG test case in Scenario 1.	29
5.5	Object distribution for HPCCG in Scenario 1 for zero sampling period. The objects in bold, placed in SP memory are discarded when performing sampled memory access profiling.	30

List of Tables

2.1	Cache misses depending on the cache size.....	11
3.1	Flags for the extended Callgrind version.	16
3.2	Possible sampling cases.	16
4.1	Cache configuration for our experiments.	23
4.2	Memory configuration for our experiments.	23
5.1.a	Object distribution for MiniMD – Scenario 1 – Sampling period: none.....	26
5.1.b	Object distribution for MiniMD – Scenario 2 – Sampling period: none.	26
5.2.a	Object distribution for MiniMD – Scenario 1 – Sampling period: 52,177.	26
5.2.b	Object distribution for MiniMD – Scenario 2 – Sampling period: 52,177.	26
5.3.a	Object distribution for HPCCG – Scenario 1 – Sampling period: none.	30
5.3.b	Object distribution for HPCCG – Scenario 1 – Sampling period: none.	30
5.4.a	Object distribution for HPCCG – Scenario 1 – Sampling period: 37,012,243.	30
5.4.b	Object distribution for HPCCG – Scenario 1 – Sampling period: 37,012,243.	30

List of LISTINGS

2.1	Code-specific instrumentation and data collection macros.	9
2.2	Code-specific instrumentation and data collection example.	9
2.3	Instrumentation and data collection flags.	9
2.4	Matrix multiplication implementation.	11
3.1	Main object record algorithm.	18
3.2	Sampled object record algorithm.	18
5.1	The main computational part of HPCCG is enclosed in a for-loop. The source code is enhanced with macros that handle the interaction with Callgrind.	32

The complexity for minimum component costs has increased at a rate of roughly a factor of two per year [...]. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years.”

Gordon. E. Moore in [1]

CHAPTER 1

INTRODUCTION

1.1 Background

1.1.1 General

It was more than 50 years ago when Moore noticed this technological tendency of concentrating more “*complex electronic functions in limited space*”, and indeed for the last few decades the scientific and industrial community has been making a huge effort to live up to these expectations, elevating Moore’s saying to a globally accepted law. Computer components have gained complexity; have become compact and effective in order to cover the continuously increasing computational needs. So far, a CPU-oriented approach has been in the foreground: pipelined CPUs have targeted instruction throughput by assigning time intervals to different operations, whereas superscalar CPUs are exploiting instruction level parallelism. On the other hand, *multicore systems* combined computational power of several cores in order to achieve higher performance.

On the contrary, memory systems are still mainly based on DRAM technology which, although “*it maintains the best balance among capacity, bandwidth and cost*” [2], it presents a few drawbacks: various technological constraints as well as a linear cost relationship to overcome these hurdles have not permitted a large bandwidth increase, while the slow scale rate of interconnect capacitance of on- and off-chip has only worsen the situation. What is more, accessing main memory incurs high latency. Therefore, CPU improvements outpaced

the DRAM ones resulting in the so-called “*memory wall*” or “*memory gap*”. At a bandwidth-bound case, a significant part of the computational potential remains idle due to memory controller’s incapability of keeping up with its requests’ frequency.

Cache memories have been proposed to bridge the memory wall. Caches are highly sophisticated pieces of hardware usually arranged in increasing size and latency levels. An access to the first cache level (L1) presents unimportant CPU stall cycles; to the second cache level (L2) are required around 10 cycles and they keep increasing with the cache level [3]. Although cache access latency is negligible compared to the main memory, caches are transparent to compilers and programmers, i.e. they are explicitly hardware managed not allowing direct external control in data placement and replacement. Following the same algorithms to interchange their data, in extreme cases, cache behavior might affect software performance negatively. In the common case, we can say that software is restricted by the way cache memories handle their data, therefore demanding a full profiling process in order to avoid misuse. Cores working in parallel can only exacerbate the situation as they usually compete for accesses in shared caches. A program’s performance can be significantly degraded by contention such as wrongly evicted shared data: data is evicted from a shared cache due to capacity restrictions after having been utilized by core *P1* but not yet by core *P2*, creating the need to bring it back as soon as *P2* accesses it. As mentioned in [4] there are few, yet complex workarounds that permit indirect cache control: non-temporal instructions [5], cache partitioning based on page coloring [6] and memory footprint reduction via loop tiling [7].

With scarcely any optimization opportunities, a further optimization addressing the memory performance issue came from *Heterogeneous memory systems* (or *Hybrid memory systems - HM*). Such systems accommodate memories featuring different intrinsic characteristics. Namely, latency capacity, bandwidth, energy consumption or volatility can vary depending on the different architecture. Examples of commercially available HM systems are the following:

- A. KeyStone II [8] which is a server-class ARM and DSP heterogeneous architecture combining three HM levels: L2 scratchpad memory, L3 MSMC and DDR. It also provides the API to place the data in each layer.

- B. Intel® Xeon Phi™ [9] which contains both DDR and MCMDRAM, a 3D stacked memory. The API for data placement is also offered by the vendor, while 3 combinations of the memory subsystems can be found: cache mode, flat mode, hybrid mode.

1.1.2 Memory architectures

During this technological shift, many memory architectures have appeared both in scientific papers and in industry. What follows is a short review of the different commercially available memory system architecture. Some of them are considered key factors for this paper:

- A. *Scratchpad memory* [10] (or SPM) is an explicitly software managed, cache-like memory, placed as close to the ALU as a L1 cache. It offers rapid data retrieval and is ideal for storing small data objects. Its superiority lies in the high level of control which allows it to work without memory contention and faulty data eviction. On the other hand, the control in such small granularity that is required has restricted its integration solely to embedded systems not permitting general purpose processors to take advantage of it.
- B. *On-chip 3D-Stacked Memory* [11] is based on the traditional DRAM benefitting from the technology's advantages. What is different is the 3D organization which implies it is physically stacked among the layers of the processor die, therefore making better use of the die-to-die bandwidth and presenting lower access latencies in comparison to its external DRAM counterpart. According to [11] and [12] an overall 30% reduction in access latency can be achieved.
- C. *DRAM* (dynamic RAM) [13] which is to be found in the form of an integrated circuit uses capacitors to store one bit of information and presents the best balance between low-cost and high-capacity computer memory. The differentiating factor is the need of an external refresh circuit that recharges the capacitors (i.e. rewrites the data) even if it has not been accessed. This non-volatility intrinsic characteristic introduces a non-negligible energy consumption as well as additional access delay.
- D. *NVRAM* [14] (or non-volatile RAM) offers data storage without energy consumption. Data that is directly stored in NVRAM does not need be copied to the main memory

further decreasing memory traffic. On the contrary, this technology limits the number of data renewal (write-erase cycles) and favors loads versus stores by demanding more machine cycles for the latter rather than the prime. Moreover, NVRAM is solely block addressable thus may be obstructive for some access patterns. A workaround which provides with a byte-addressable view of the memory space can be found in libraries such as NVMalloc [15] however it may suffer from high overhead.

- E. *SRAM* (static RAM) uses a transistor rather than a capacitor to store each bit while during read and write operations another 2 access transistors are used to manage the availability to a memory cell. The term static refers to the fact that bits do not need be periodically refreshed. Cache memories, register file and other popular memory systems are based on this technology.

Although commercially available products deploy the aforementioned memory architectures only independently or in combinations, and despite we do not anticipate a future HM system that consists of a mosaic of the complete list, we will simulate such a system as an effort to depict the several types of memory existing at a cluster computing unit.

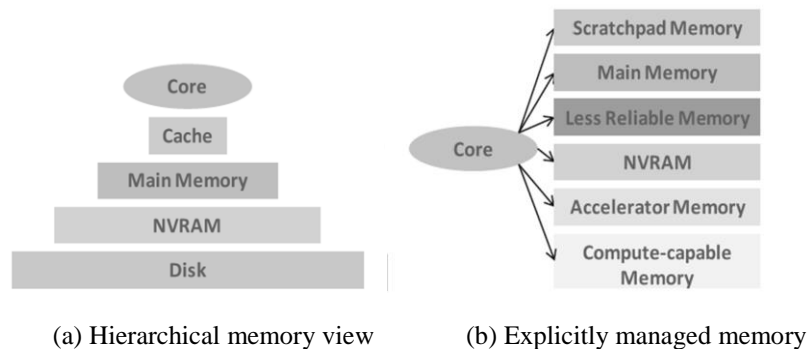


FIGURE 1.1: Hierarchical versus equally managed memory view.

What is important is the logical relationship of the memory subsystems. Their structure has to allow an equal treatment rather than following a hierarchical concept so as the benefits of each subsystem can be exposed and optimally utilized. In FIGURE 1.1 we juxtapose a hierarchical (a) against an equal (b) memory view. It can be easily concluded that the second solution permits an explicitly managed memory. That is, each data can be hosted by any subsystem

regardless of the order of access, instead depending on programmer-defined specifications such as its size, access frequency or any other characteristic considered of importance.

1.2 Problem Statement and Contributions

It is often the case that when it comes to application profiling a question has to be answered: *Hardware or software approach?* In order for the answer to be well-founded both approaches have to be based on similar concepts and respect equivalent principles. Since hardware modifications range from extremely difficult to impossible due to the lack of freedom of modifying silicon, we studied *Valgrind* in order to familiarize with its internal structure and consequently to extend it with additional functionalities. In particular, in the rest of this paper:

- There is a thorough explanation of *Valgrind*'s function as well as a short introduction to some of its available tools.
- An insight to the internal structure of the tool that simulates cache.
- Instructions on how you can efficiently use the tool depending on the final scope

Hardware counters supplied the means for a rather precise and quick method of profiling. The latter relies largely on a specified sampling frequency or on certain events, both of which trigger the HW counters increment. Although different architectures are equipped with different set of counters, monitoring the cache behavior is generally one of their most important objectives. However, performing a sampled event collection might give only the partial image of a program's memory interaction. In contrast, software methods exploiting the merits of a simulated cache system are allowed to account for every single memory access. In order to perform a comparison between the two aforementioned profiling techniques it is of essence to compel HW's limitation to SW approach. For this scope we:

- Extend *Valgrind*'s source code in order to perform sampled memory access detection

Finally, after having acquired a deeper understanding of the means to profile an application, we define two heterogeneous memory systems on which the data of our test cases are

optimally placed driving to a performance speedup. The detailed walkthrough of this thesis' contributions involve:

- Differentiate between total access sampling and access-type sampling
- Determine a performance speedup based on sampled application profiling of two distinct test cases
- Determine the optimal sampling period that permits a similar performance speedup
- Suggest future research

1.3 Thesis structure

CHAPTER 2 describes the functionality of Valgrind along with some of its tools and options. Moreover, an example as well as a thorough source code analysis reveals a bug that can cause doubts to potential users.

CHAPTER 3 presents EVOP, an already implemented tool based on Valgrind. It also serves the scope of analyzing the extensions that we added on this latter tool in order to support sampled profiling options.

CHAPTER 4 includes our experimentation method and analysis process. It introduces the sampling notion as well as defines the simulated memory system and the way we addressed it. It also offers a short description of the test cases used to realize our experiments.

CHAPTER 5 interprets the obtained results by comparing sampled and non-sampled profiling. It defines an optimal sampling rate for each test case and makes connections between memory access patterns and the aforementioned rate. General conclusions are presented in this chapter, too.

CHAPTER 6 makes a short summary of this paper's findings and proposes future work in two different directions.

CHAPTER 2

VALGRIND

2.1 Overview

“*Valgrind is an instrumentation framework for building dynamic analysis tools*” [17], It aids memory management and multi-threaded bug detection as well as offers profiling options. An entire set of tools which frame Valgrind by making use of its core functions enable the above and thus form its ecosystem.

Valgrind core can be considered as a virtual machine that takes control of the application-under-test before it starts executing. The application code is translated into a processor-agnostic intermediate representation that enables its execution on a synthetic CPU provided by the core. At the first code execution, control is handed to the selected tool which enhances it. Although each tool has its own implementation details, all of them share the basic principle of instrumenting the source code by inserting hooks (instrumentation directives) in order to perform different monitoring tasks at run time. The augmented code is then handed back to the core where is executed in superblocks (chunks consisting of single entry and multiple exit points) [17] [18]. The above process can be seen in FIGURE 2.1.

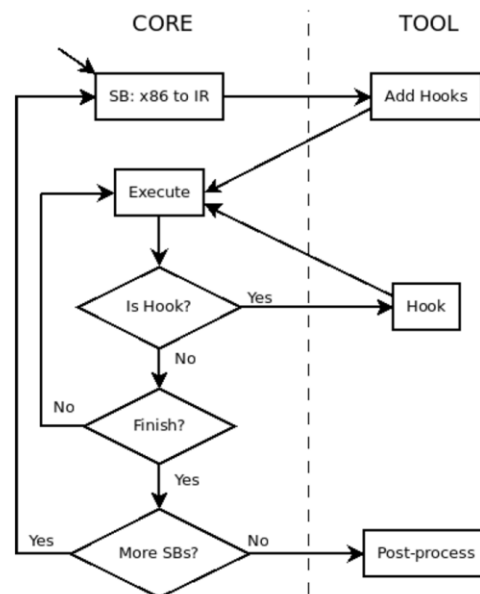


FIGURE 2.1: Simplified high-level view of the interaction between Valgrind and its tools [18].

The added instrumentation poses the main drawback of this process due to the overhead it brings. Every tool is more or less intense in terms of instrumentation; nevertheless the minimum incurred time delay caused just by enabling the Valgrind core is x4. Let it be emphasized that certain tools can make the execution 10 – 100 times slower.

Valgrind enriches its tools with an API as well as a client-request mechanism that enables the tools to interact with the core by getting debug information, manage stack memory traces and intercept different memory allocation calls so as to provide specific wrappers for them. Among its other uses, the client-request mechanism enables on-demand instrumentation initialization and ceasing in order to minimize the aforementioned extra time overhead.

In the following we provide a description of Callgrind, the tool used in our research as well as the details of how to perform code-specific profiling saving execution time. We expose a potential tricky point that accompanies LL cache simulation provided by Callgrind and finally we present a thorough walkthrough of our development process as well as explain the added features and how to deploy them.

2.2 Callgrind

Callgrind is a pure application profiler that combines its own features along with the functionality of Cachegrind, another tool of the ecosystem. Cachegrind simulates virtual I1, D1 and L2 (or LL) caches getting their characteristics either by default from the native CPU or explicitly by the user. It offers per instruction, per function or entire-program information regarding the number of cache misses (per cache level), memory references and issued instructions. Although it limits the available cache levels to two, causing the measurements to differentiate themselves from other profiling methods, it serves the crucial scope of providing memory insight. Regardless the exact cache configuration it helps in identifying performance limiting memory patterns and fix unfriendly memory accesses.

On top of these features, Callgrind collects the number of instructions and functions calls, extracts the caller-callee relationship among them, relates them to the source code and finally gives the opportunity of a branch predictor simulation. This data is then processed by other tools (such as KCachegrind, a graphic visualizer) in order to result in a call-graph.

```

1. CALLGRIND_START_INSTRUMENTATION
2. CALLGRIND_STOP_INSTRUMENTATION
3. CALLGRIND_TOGGLE_COLLECT
4. CALLGRIND_ZERO_STATS
5. CALLGRIND_DUMP_STATS

```

LISTING 2.1: Code-specific instrumentation and data collection macros.

```

1. for(n = 0; n < ntimes; n++) {
2.     CALLGRIND_START_INSTRUMENTATION;
3.     CALLGRIND_TOGGLE_COLLECT;
4.     ....
5.     ....
6.     CALLGRIND_TOGGLE_COLLECT;
7.     if(n == 0)
8.     CALLGRIND_ZERO_STATS;
9. } //end of ntimes for-loop
10. CALLGRIND_STOP_INSTRUMENTATION;

```

LISTING 2.2: Code-specific instrumentation and data collection example.

```

1. --instr-atstart=<yes|no>
2. --collect-atstart=<yes|no>

```

LISTING 2.3: Instrumentation and data collection flags.

2.2.1 Source code instrumentation

Valgrind (and thus Callgrind) uses internal event counters in order to account for the actual number data was accessed. Under normal circumstances these counters are initialized at the beginning of the execution and are printed at the end of it or upon request. However, it is feasible to focus profiling only to a specific part of the code by disabling event aggregation for the uninteresting part and allow Callgrind to progress at much higher speed.

For this aim, the tool offers the set of macros provided in LISTING 2.1. The first two are self-descriptive since they commence and terminate simulation and profiling. The third macro signals the beginning and the end of a region of

interest i.e. it acts interchangeably as a start / end point enabling or disabling the event collection, respectively. It can be used to monitor nonconsecutive regions of source code or to terminate, dump and zero the counters multiple times in a run. Event counters can be set to zero using the fourth flag or, alternatively, can be printed and set to zero using the fifth one. In LISTING 2.2 we present a loop in which we wish to start the cache simulation at iteration 1, while account for the access statistics starting from iteration 2. For this scope, both instrumentation and event collection are initialized along with the loop in lines 2 and 3, yet the counters are zeroed just before the second iteration, in line 6. They are printed by default as soon as the loop completes its execution, in line 8.

2.2.2 Callgrind options

In order for the previous macros to be effective, Callgrind has to be initialized with the appropriate flags. The tool API provides the flags seen in LISTING 2.3 in order to handle instrumentation, profiling and collection regions. When the first flag is set to “no”, simulation and profiling should be programmatically enabled by the directive, as shown in LISTING 2.2. The un-instrumented code region results in a slowdown to solely 4 times. When the second flag is deactivated, the counters shall remain zero until it is instructed otherwise by the source code.

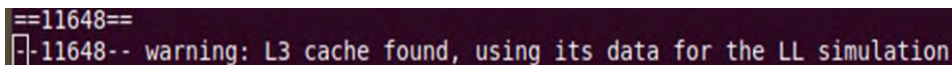
A screenshot of a terminal window with a dark background and light-colored text. The text consists of two lines: the first line is "=="11648==" and the second line is "[11648-- warning: L3 cache found, using its data for the LL simulation." The second line is preceded by a small square icon containing a minus sign.

FIGURE 2.2: Warning informing about the LL cache configuration.

2.3 Last level cache simulation

In order to account for the validity of the experiments and subsequently interpret the results it is crucial to have an exact image of the simulated caching system. Nevertheless, Callgrind has a gray point when it comes to simulating the last level cache: a warning is printed informing that the user-specified LL cache shall be disregarded since a native one has been detected. In our understanding, the particular warning, which is to be seen in FIGURE 2.2, is wrongly printed.

2.3.1 The matrix multiplication example

In an initial effort of examining its validity we set a trivial example. We created the code shown in LISTING 2.4 which is a plain implementation of the matrix multiplication algorithm. We used 3 dynamically allocated matrices of size 1024 x 1024 elements which, under the system’s architecture translate to 4 MB each. This choice serves the scope of not allowing the data to completely reside in the cache memory of any level. The experiments that followed are similar to one another with the sole difference of the last level cache size. Specifically, in the first experiment LL cache was not specified so that Valgrind take the default hardware specification from the detected cache (4 MB of L3 cache) while in the second one, LL cache was manually set to 128 KB. The size of the other caches was identical in both cases.

```

1. #define N 1024
2. void matmul (int **a, int **b, int **c) {
3.     int i, j, k;
4.     for(i = 0; i < N; i++) {
5.         for (j = 0; j < N; j++) {
6.             c[i][j] = 0;
7.             for (k = 0; k < N; k++) {
8.                 c[i][j] += a[i][k] * b[k][j];
9.             }
10.        }
11.    }
12. }

13. int main (){
14.     int **a, **b, **c;
15.     int total[N][N];
16.     int i, j;

// omitted code: dynamically allocate a, b,
// c and initialize a, b

17.     matmul (a, b, c);
18.     return (0);
19. }

```

LISTING 2.4: Matrix multiplication implementation.

Experiment	1	2
LL cache size	4 MB	128 KB
I1 misses	992	992
D1 misses	1,277,495,238	1,277,495,232
LL misses	21,419,460	575,742,112

TABLE 2.1: Cache misses depending on the cache size

imposed by the algorithm, this effect is to be reflected in the total number of accesses that have to access the main memory as they cannot be served in cache.

The above observation however is solely an indication that suggests a probable explanation and a faulty warning. Further investigation is needed in order to obtain a concrete argument and verify the hypothesis. For this scope we examined the source code of Valgrind and its tools in order to locate the lines responsible for printing the warning as well as to extract an overview of the cache memory internal implementation and the mechanisms that regard this particular feature. We consider this short research useful for future developers and researchers that wish to deploy or extend Valgrind and its tools in this direction. Therefore we proceed in

Assuming the warning is correct, if we observe the LL cache misses we anticipate no divergence in the final results. On the contrary, as shown in TABLE 2.1, the number of memory accesses that missed LL cache in the second experiment is significantly larger than the respective number in the first one while the number of memory accesses that missed the other cache levels is exactly the same. This fact can be justified if we disregard the printed warning and thus consider that the LL cache size actually follows the user specification. Using a cache with smaller capacity directly affects the frequency of data substitution by increasing it since the amount of data that can be stored is less. Given the repeatability of data usage that is

a walkthrough of the internal function interconnection that handle the parameters of the virtual cache.

2.3.2 Source code inspection

To begin with, since the cache simulator is provided by Cachegrind and shared with Callgrind, as noted in section 2.2, we have to focus on the prime. Also, in order to ease the description we avoid the details of the specific function naming and representation. For further information refer to [16].

In Cachegrind's source code directory resides `cg_arch.c` folder which is of particular interest for our purpose. As pictured in FIGURE 2.3, a first Boolean function detects the existence or not of the appropriate cache-describing flags. In case of success the flags are parsed by an additional function and their values are stored in special variables. In the meantime, a validity check is implemented to ascertain that the given values are acceptable. A third function is defined with aim of:

- Detecting the caches existing in the hardware of the native machine,
- Checking the compliance with the tool's standards and
- Setting their values to the virtual cache.

In these lines of code a warning informing about the superiority of auto-detected caches over the user-specified ones is printed. What follows is a comparison between these two cache types. In case the user-specified ones are valid they actually override the default (hardware) ones. This is done by an independent function.

Analyzing the tool's source code we have shown that user specifications always prevail the default configuration regardless the cache level, therefore we can now verify our hypothesis and prove the warning false. It can be taken for granted that when a cache level configuration is specified by user, then the actual characteristics are simulated by the tool.

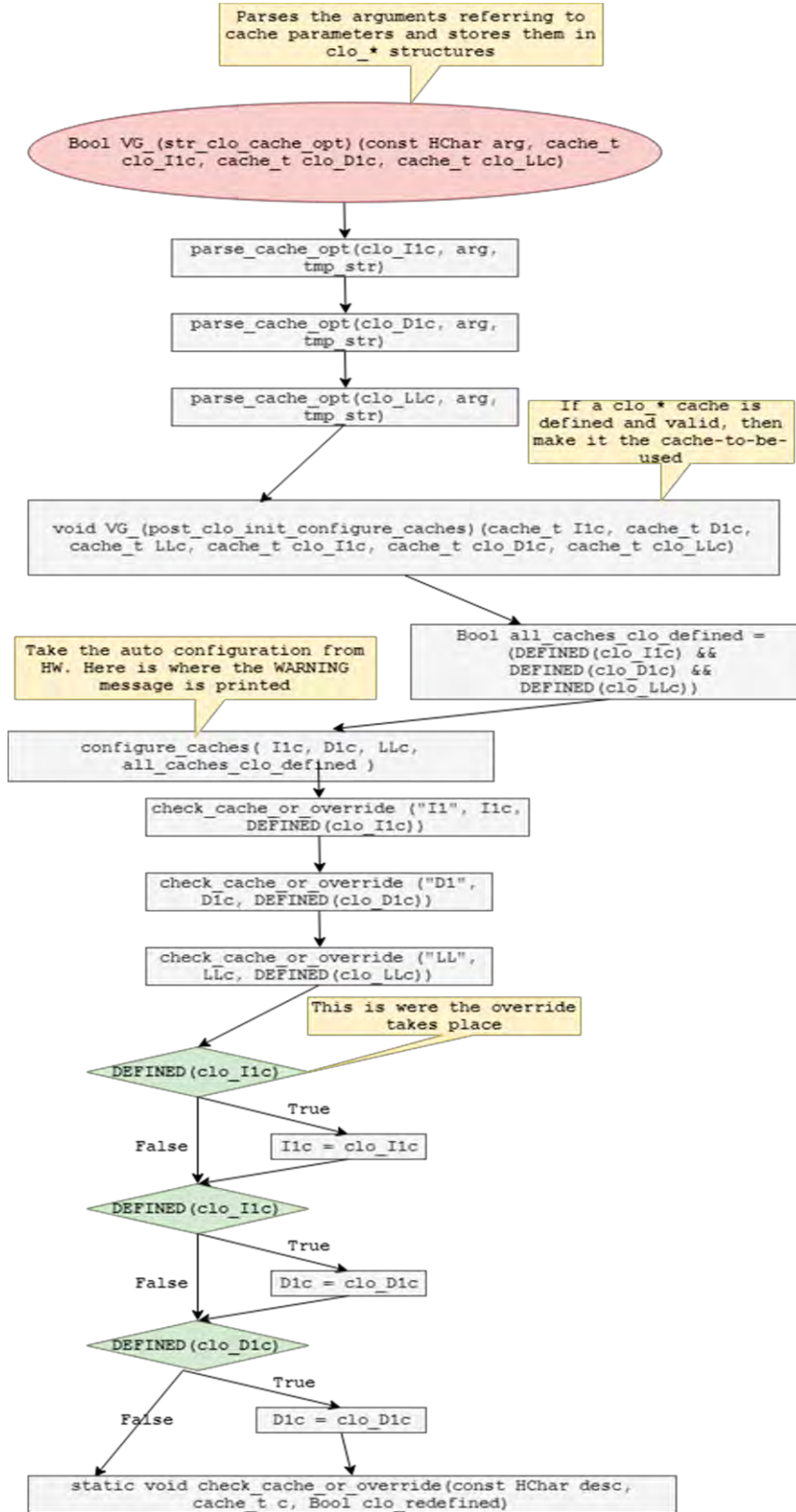


FIGURE 2.3: cg_arch.c function flow chart.

CHAPTER 3

VALGRIND TOOL EXTENSION

After a closer look to the means with which cache is represented and simulated by Callgrind, we proceeded with implementing some additional features that enable sampling memory accesses. We decided to begin this chapter by introducing some important prior information regarding the granularity of memory accesses.

3.1 Overview

Our extensions are based on the already extended Valgrind profiler known by the name EVOP [18] [19] [20] which is an enhanced version of the development branch of Valgrind 3.10.0. For our purpose we consider critical to clarify the notion of “*memory object*” that has been introduced by [19].

Based on [18] and [19] a “*memory object*” can be defined as memory data the semantics of which allow them be referenced as an entity. In other words, as “*memory object*” is referred every memory entity that can be seen as such from the code level. Examples include structs, arrays etc. that can be either statically or dynamically allocated. Depending on the latter, Valgrind employs different interception and system call wrapping mechanisms by using debug information to track their characteristics. Address, size and trace are saved in a sorted structure that offers efficient (logarithmic) searches.

From now on, this research makes extensive reference to “*memory objects*” or “*memory items*” interchangeably, characterizing them as the main interaction unit between the profiled

application and the memory system. In this chapter we apprise of a feature that we added in the tool functionality. Our modifications are focused on making sampled data collection feasible by integrating extra functionality to keep track of the memory access number. We distinguish between two diverse tastes of sampling: one that refers to the total number of memory accesses and one that discriminates between loads and stores and subsequently performs sampling.

3.2 Callgrind extension

We start our explanatory description by introducing the extra functionalities provided in our tool version.

3.2.1 API options

In order to launch Callgrind, except for the standard edition flags that are described in the official manual and those added in the EVOP version, we have accumulated another set of three flags. The options presented in TABLE 3.1 have the following functionalities:

- `--sampling-period=<integer>`: by setting this value equal to an integer value we instruct the tool to perform a sampled collection of memory accesses. As sampled value is considered the total amount of memory accesses, regardless if they write or read data. If the integer value is set to 0 then the execution proceeds by default without performing sampling.
- `--sample-loads=no|yes`: this flag indicates the sampled value by setting it to “loads”. In combination with the previous one it indicates that only one out of `<integer>` loads will be accounted for. On the contrary, all store accesses will be monitored.
- `--sample-stores=no|yes`: in analogy to the previous flag, this one changes the sampled variable to “stores”. Again, all loads will be monitored.

While the first flag is optional on its own, as long as one of the two following flags is active then it becomes mandatory. Also, the default behavior of the two last flags is to be deactivated. This means that, for example, in case of sampling in loads, there is no need to

1.	<code>--sampling-period=<integer></code>
2.	<code>--sample-loads=no yes</code>
3.	<code>--sample-stores=no yes</code>

TABLE 3.1: Flags for the extended Callgrind version.

Case	A	B	C
<code>--sampling-period</code>	integer	integer	integer
<code>--sample-loads=</code>	-	yes	no
<code>--sample-stores=</code>	-	no	yes

TABLE 3.2: Possible sampling cases.

explicitly deactivate the store sampling. The case where both flags are active is supported; it results yet in the same behavior as if both were deactivated, i.e. total memory access sampling.

3.2.2 Sampling implementation

Having explained the user-options we can now proceed to the source code modifications that made these features possible.

Our code alteration is generally focused on `sim.c` file which is under Callgrind directory in Valgrind's source code package. Some minor interventions are to be noticed in files `global.h` and `clo.c`, nonetheless they are trivial since they serve in complying with the tools existing variable declaration hierarchy and in providing with user information, respectively. No mention of these changes shall be made.

In LISTING 3.1 we present the function from `sim.c` which is responsible for tracking and recording an access to cache memory, determining if it is a cache hit or cache miss. By default, the function correlates the access to a referenced object, determines if the object is statically or dynamically allocated and, depending on that, follows different paths in order to store the details of the access.

Our intervention is to be located mainly in the beginning of the function as presented in LISTING 3.2. In case a sampling period has been set by the user, the value is stored in an internal variable. An internal counter is used in order to control which memory accesses are to

be monitored. In particular, in case “A”, as pictured in TABLE 3.2, the counter is incremented every time memory is accessed. All these accesses that result in a counter value smaller than the sampling period are disregarded while the one that equates the two variables is the one which will be accounted for.

Cases “B” and “C” are complementary. In case “B” the user has defined loads as the sampled access type. For every memory access we have to identify its type and increase the counter only when a load is encountered. All loads that keep the counter’s value smaller than the period are disregarded while the one that results in equalizing them is monitored. On the contrary, every store is accounted for since it does not interact with the sampling mechanism. An analog concept is followed for case “C” in which loads are replaced by stores and vice versa.

```
1. record_access(){
2.     found = false;
3.     if (trace_dynamic) {
4.         found = record_mem_access (addr, size, metadata_dyn)
           // keep track of additional details
5.     }
6.     if (trace_static) {
7.         found = record_var_access (addr, size, metadata_stat)
           // keep track of additional details
8.     }
9. }
```

LISTING 3.1: Main object record algorithm.

```
1. period = SAMPLE;
2. record_access(){
3.     if ((sample_loads and access == load) or
         (sample_stores and access == store) or
         (sample_total)) {
4.         counter++;
5.     }
6.     if (period > 0) {
7.         if (counter != period) {
8.             return;
9.         }
10.        counter = 0;
11.    }
12.    found = false;
13.    if (trace_dynamic) {
14.        found = record_mem_access (addr, size, metadata_dyn)
           // keep track of additional details
15.    }
16.    if (trace_static) {
17.        found = record_var_access (addr, size, metadata_stat)
           // keep track of additional details
18.    }
```

LISTING 3.2: Sampled object record algorithm.

CHAPTER 4

EXPERIMENTATION

The fundamental experimentation step is to discover these objects that present the biggest number of LL cache misses, i.e. to define the objects the access to which demands accessing the main memory rather than the cache. This is done by profiling the application-under-test and monitoring the memory objects it leverages. LL cache misses are of interest since they are essentially the ones that access main memory and thus cause excessive time overhead. By explicitly choosing the memory subsystem from which a LL cache miss shall be served we can minimize the extra access latency. Next, we appraise their optimal distribution to the subsystems of the suggested heterogeneous memory system with the aim of minimizing the total stall CPU stall cycles caused by accessing each of them.

In this chapter we firstly describe the profiling methodology that has been followed; we illustrate the analysis procedure that was chosen; we specify the emulated heterogeneous memory system on which our experiments run and finally we lay a summary of the applications used as test-cases. Our work let be characterized as a revision of the experiments reported in [20] augmented by the sampling mechanism that was implemented by us.

4.1 Profiling

In order to associate accesses that missed LL cache to the memory objects that they refer we deployed the Valgrind framework and specifically its tool Callgrind. The way cache misses are correlated to a memory object is an automated method of the tool and generally relies on the debug information provided with the executable (usually controlled by `-g` option when

compiling) as well as on the allocation type of the particular memory object. For further information please refer to [20].

Running the application under Callgrind tool while simulating a virtual cache adds a significant timing overhead that reduces profiling performance. Hence, we opted for the optimization techniques described in section 2.2.1 in order to start and terminate Callgrind's instrumentation under user request. In particular, LISTING 2.2, that has already been presented, depicts the main computational part of one of our test cases which is controlled by the specific Callgrind macros. On top of that, we tried to achieve more accurate cache miss statistics by performing a warm-up round. This is feasible by commencing instrumentation at the beginning of the computational part of the code and zeroing all counters after the end of the first loop. This way cold misses are not accounted for in the final results while caches are already filled with contents and the profiling can proceed. Counters will start fresh to count the memory traffic at the beginning of the second loop until the end of the application.

Initially, we performed the classic profiling process of monitoring every single memory access. The obtained results refer to the exact number of LL cache misses per memory object. Objects with big LL cache miss rate were the performance limiting ones, so they were optimally distributed into the subsystems and an initial speedup was computed. This speedup was considered the maximum achievable one given the subsystem architecture and the application.

Regarding the sampled memory access profiling, we made use of the functionality that we developed in order to monitor memory accesses based on different sampling periods. Sampling refers to the total memory access number (case A as described in TABLE 3.2). The output is the sampled per-object number of cache misses and the aim is to define the optimal sampling period which leads to identical (or similar) speedup as the non-sampled results. Note that in case of large sampling periods a big number of accesses are discarded. Should a particular object be referenced solely by these accesses, this memory object will never be identified and thus remain hidden from the final results. As a consequence it is estimated that the final distribution has fewer objects to choose from, hence diverting the final performance optimization.

4.1.1 Sampling period

One of the most crucial points is defining the optimal sampling period. To that end, we developed automated processes written in `bash` language in order to choose from a set of numbers, perform profiling, data distribution and speedup calculation. Depending on the latter, a smaller or bigger period was selected in order to advance the simulation until the optimal one was discovered.

While a first glance would approve any number to be adopted as sampling period, a more meticulous inspection of the facts would advocate against it. The memory behavior of an application which is under question here is a function of the application's source code. This source code can consist of loops, indirect references to the same basic memory objects or other access patterns. Selecting to monitor a *random* memory access out of a set of memory accesses is susceptible of discovering an unwilling access pattern. Therefore, in case of a loop for example, the result would be to monitor the same access (or accesses) in every code iteration. The outcome of such a profiling process is considered biased since it was generated by a problematic sampling rate, accounts for a limited number of objects and eventually leads to a non-optimal performance improvement.

We foresee that a case similar to the one outlined before is rather rare to encounter. Nevertheless, in order to eliminate every possibility, in our experiments we used exclusively prime numbers as sampling periods. Prime numbers are characterized by all these properties that disallow a memory pattern discovery. Especially, by setting a prime number as sampling period we can guarantee that in every iteration (if the examined source code is iterative) different memory accesses are monitored. In general, we anticipate a better distribution of intercepted memory accesses.

4.2 Analysis

The profiled data analysis is conducted with the ultimate target of producing an object distribution among the memory subsystems. To address this issue we use EVOP's `dmem_advisor`. As reported in [19] and [20] this tool implements a relaxation of the classical textbook 0/1 knapsack problem [21]: Different knapsacks are the various memory subsystems,

their capacity is the memory size while the items to pack are the memory objects with their size to represent their weight. Every knapsack modifies the value of its items by multiplying it by the CPU stall cycles (each knapsack, i.e. each memory subsystem demonstrates different load latency). This situation is addressed effectively by targeting each subsystem independently in a latency-ascending order and by placing objects with more cache misses to the “fastest” memories, provided they fulfill the subsystem’s size requirements.

To perform data distribution based on the sampled data we modified the `dmem_advisor` source code in order to calculate appropriately the CPU saved stall cycles.

It is important to note that memory objects are prioritized depending on their load cache misses. We consider zero stall cycles when a store is issued by assuming a buffered write-through cache with infinite buffer bandwidth. In practice we expect this simplification not alter the final results since the CPU stall cycles caused by loads are notably more than the ones caused by stores.

4.3 Technical characteristics

In the following subsection we define the simulated heterogeneous memory system that was used as the basis of our experiments, as well as the test cases which produced the final results.

4.3.1 Heterogeneous memory system

Our memory system can be divided into two distinct subsystem entities: (a) a baseline standard one which consists of caches and (b) a modifiable one which is a hodgepodge of various memory types. The simulated cache characteristics for every cache level such as size, associativity and line size are presented in TABLE 4.1. Regarding cache latency, we consider zero stall cycles when L1 cache is accessed and 20 CPU stall cycles when L2 (or LL) cache is accessed. L2 latency is set to a relatively large number in order to compensate for the absence of an L3 level cache which typically has higher latency, yet it cannot be simulated by Callgrind.

We then distinguish among a Baseline scenario, Scenario 1 and Scenario 2. The first one is our point of reference: it consists solely of a DRAM memory (this system represents a trivial

DESCRIPTION	SIZE	ASSOCIATIVITY	LINE SIZE
L1 INSTRUCTION	32 KB	8	64 B
L1 DATA	32 KB	8	64 B
L2 UNIFIED	8 MB	16	64 B

TABLE 4.1: Cache configuration for our experiments.

MEMORY		SCENARIO		
DESCRIPTION	LATENCY	BASELINE	SCENARIO 1	SCENARIO 2
L1	0 c	32 KB + 32 KB		
L2	20 c	8 MB		
SP	20 c	0 B	8 MB	8 MB
3D	135 c	0 B	8 GB	1 GB
DRAM	200 c	32 GB	32 GB	4 GB
NVRAM	20,000 c	0 B	0 B	32 GB

TABLE 4.2: Memory configuration for our experiments.

computing system). In Scenario 1 we maintain the same DRAM memory while adding a Scratchpad and a 3D stacked memory. Scenario 2, which manifests a more energy-friendly approach, consists of the same mosaic of memory subsystems as the previous one, plus a NVRAM component. In TABLE 4.2 we list the characteristics of each memory subsystem in function to the Scenario they belong.

Let it be emphasized that we do not actually simulate the memory subsystems; on the contrary we deploy average latency estimations.

4.3.2 Test cases

In our research we have opted for two miniapplications from the Mantevo application performance project [22]: MiniMD and HPCCG. They combine some or all of the dominant numerical kernels contained in an actual stand-alone application. Thus, they can model the behavior of more complex applications.

- MiniMD is a parallel molecular dynamics (MD) simulation package following many of the LAMMPS MD simulator [23]. It simulates the movement of atoms in a box area according to Newtonian laws. Users are given the freedom to control a variety of simulation parameters such as the simulated size, atom density, timestep size, number of timesteps etc. Velocities, positions and forces of the atoms are computed iteratively

on every timestep while every n timesteps a re-neighboring is performed so that each atom's new neighbors are computed. In our experiments we use MiniMD version 1.2 that leverages a Lennard-Jones (LJ) interaction among $2.9 \cdot 10^6$ atoms. The flags used are `(-t 8 -n 2 -s 90)`.

- HPCCG is a simple conjugate gradient benchmark mimicking the behavior of applications deploying this method as their main computational kernel. The problem results in the solution of a sparse symmetric matrix that requires high memory interaction. We use HPCCG reference version 1.0 to run a $400 \cdot 400 \cdot 400$ problem which uses roughly 23 GB of memory. Due to the long simulation time we have reduced the upper limit of the allowed iterations therefore amplifying the achieved accuracy.

CHAPTER 5

RESULTS

In this chapter we analyze the experimental results. First we profile the miniapplications monitoring all memory accesses. From the obtained data we distribute the memory objects optimally in our emulated memory subsystems and compute the performance speedup. This is defined as the maximum possible speedup.

We repeat the same process imposing sampled memory access profiling and compare the achieved performance gain among the various sampling periods. This results in determining an optimal sampling period for each test-case. We define as optimal sampling period the biggest prime number that when used as sampling period produces similar speedup (or in general execution performance) as in the non-sampled profiling case. We also interpret the correlation between sampling period, access patterns and discovered memory objects.

Our results include an estimation of the saved CPU cycles per application due to the effective data placement given the latency parameters introduced in 4.3.1. MiniMD results are followed by HPCCG ones while at the end we draw some overall conclusions

NAME	OBJECTS	OCCUPANCY	SAVED CYCLES	EXECUTION	NAME	OBJECTS	OCCUPANCY	SAVED CYCLES	EXECUTION
SP	19	53%	-8 M	-0.06%	SP	19	53%	-48 M	-0.06%
3D	9	39%	-1.2 G	-10.15%	3D	8	65%	-623 M	-5.08%
DRAM	0	0%	-	-	DRAM	1	61%	-	-
					NVRAM	0	0%	0	0.00%

(a)

(b)

TABLE 5.1(a): Object distribution for MiniMD – Scenario 1 – Sampling period: none

TABLE 5.1(b): Object distribution for MiniMD – Scenario 2 – Sampling period: none

NAME	OBJECTS	OCCUPANCY	SAVED CYCLES	EXECUTION	NAME	OBJECTS	OCCUPANCY	SAVED CYCLES	EXECUTION
SP	1	28%	0 M	0.00%	SP	1	28%	0 M	0.00%
3D	7	37%	-1245 M	-10.15%	3D	6	55%	-623 M	-5.08%
DRAM	0	0%	-	-	DRAM	1	61%	-	-
					NVRAM	0	0%	0	0.00%

(a)

(b)

TABLE 5.2(a): Object distribution for MiniMD – Scenario 1 – Sampling period: 52177

TABLE 5.2(b): Object distribution for MiniMD – Scenario 2 – Sampling period: 52177

```

-- SP - 8388608 bytes --
48292 - 28 loads - 2349600 bytes
48624 - 44208 loads - 2125604 bytes
48572 - 4 loads - 24 bytes
48620 - 4 loads - 48 bytes
48304 - 3 loads - 120 bytes
48300 - 3 loads - 40 bytes
48556 - 2 loads - 24 bytes
48612 - 2 loads - 24 bytes
48592 - 2 loads - 24 bytes
48568 - 2 loads - 24 bytes
48564 - 2 loads - 24 bytes
48560 - 2 loads - 24 bytes
48588 - 2 loads - 24 bytes
check_safeexchange [integrate.cpp:86] - 1 loads - 4 bytes
comm [ljs.cpp:259] - 6 loads - 336 bytes
thermo [ljs.cpp:258] - 2 loads - 128 bytes
neighbor [ljs.cpp:256] - 9 loads - 248 bytes
atom [ljs.cpp:255] - 5 loads - 200 bytes
_ZTV7ForceLJ [???;0] - 1 loads - 56 bytes
--
19 objects; 4476576 bytes (53.364944458%); 7971840 saved

-- 3D - 8589934592 bytes --
48760 - 9561769 loads - 2615680936 bytes
48740 - 4037613 loads - 156976544 bytes
48724 - 2979497 loads - 156976544 bytes
48732 - 2187000 loads - 156976544 bytes
48756 - 182250 loads - 13078540 bytes
48720 - 99687 loads - 52320144 bytes
48736 - 99384 loads - 52320144 bytes
48728 - 2 loads - 52320144 bytes
48744 - 1 loads - 52320144 bytes
--
9 objects; 3308969684 bytes (38.5214770678%); 1244568195
saved

-- DRAM - 34359738368 bytes --
--
0 objects; 0 bytes (0.0%); 0 saved

```

FIGURE 5.1: Object distribution for MiniMD – Scenario 1 – Sampling period: none

```

-- SP - 8388608 bytes --
48292 - 0 loads - 2349600 bytes
--
1 objects; 2349600 bytes (28.0094146729%); 0 saved

-- 3D - 8589934592 bytes --
48760 - 180 loads - 2615680936 bytes
48740 - 79 loads - 156976544 bytes
48724 - 58 loads - 156976544 bytes
48732 - 43 loads - 156976544 bytes
48756 - 4 loads - 13078540 bytes
48720 - 2 loads - 52320144 bytes
48736 - 1 loads - 52320144 bytes
--
7 objects; 3204329396 bytes (37.3033037875%); 1244568000 saved

-- DRAM - 34359738368 bytes --
--
0 objects; 0 bytes (0.0%); 0 saved

```

FIGURE 5.2: Object distribution for MiniMD – Scenario 1 – Sampling period: 52,177.

5.1 MiniMD

Our results for MiniMD initially include 28 identified objects that cause LL cache misses. These objects tend to decrease in number as the sampling period increases.

5.1.1 Complete memory access profiling

SCENARIO 1: TABLE 5.1(a) summarizes our results for MiniMD in Scenario 1. Only 19 out of the 28 memory objects are small enough in order to leverage of the low-latency SP memory leaving 47% unoccupied space. The rest of them are stored in 3D memory by occupying less than the half of its capacity. The performance improvement is 10.21% comparing to the baseline execution and it is interestingly due to the objects placed in 3D memory. These objects, although fewer than those placed in SP memory system are responsible for more data accesses. Indeed, in FIGURE 5.1 we present the partial output of `dmem_advisor` in which objects are related to their memory accesses and their size. SP memory hosts more than the 3D memory’s objects, yet CPU does not interact significantly with them.

SCENARIO 2: TABLE 5.1(b) presents the results obtained by simulating the architecture of Scenario 2. The object distribution is identical to the last one with the exception that one object is moved from 3D memory to DRAM. Although this might seem as a minor change, in fact it influences the final speed up greatly. The number of memory accesses to this object

proves to be the main contributing factor to the Scenario 1 speedup in a manner that its transition to DRAM causes the performance improvement to lessen to one third of the initial.

5.1.2 Sampled memory access profiling

After having defined the maximum possible speedup, which is obtained by accounting for every memory access we initiated the sampling experimentation. The sampling period spectrum was set from 199 to 4,037,729. Both values however are in the extremes resulting in non-optimal results: the first one has identical behavior to the non-sampled profiling in terms of data discovery; the only difference lays on the fact that objects with insignificant number of read accesses, such as the first objects of FIGURE 5.1, are omitted from the final distribution as they are not discovered. The second sampling period value, on the other hand, has exactly the opposite outcome. Sparse sampling results in the discovery of only one memory object which does not improve the speedup significantly.

We continued following a relaxed binary search among the sampling period values and concluded that the maximum acceptable period is 52,177. In that case only 8 objects are discovered. In FIGURE 5.2 we present the partial output of `dmem_advisor` for this dataset. The per-object memory references are rather fewer than their counterparts in FIGURE 5.1 since they refer to the sampled memory accesses, yet in both scenarios the final performance is very close to the initial one.

SCENARIO 1: TABLE 5.2(a) summarizes our results for MiniMD in Scenario 1 when memory accesses are intercepted every 52,177. Only one object fits in the fast SP memory covering $\frac{1}{4}$ of its available storage capacity; however its accesses are not enough to contribute in the final speedup. The rest of the discovered objects are stored in 3D memory and offer a performance improvement of 10.15%.

SCENARIO 2: TABLE 5.2(b) summarizes our results for MiniMD in Scenario 2. Only 1 object is small enough to fit in SP memory but similarly to the previous outcome it does not offer additional speedup. This is also the case for the object that is stored in DRAM memory. On the contrary, 3D memory hosts 6 objects which are entirely responsible for the total performance speedup that rises to 5.08%

5.1.3 Sampling period comparison – Conclusions

In our experimentation we aimed in modifying the sampling period in order to define the one that preserves the highest possible speedup. In FIGURE 5.3 we have depicted graphically the correlation between sampling period and the final speedup for Scenario 1. We can observe that for the chosen prime numbers the speedup remains generally stable until the threshold of 52,177. After that it drops significantly and remains low, oscillating between two uninteresting values.

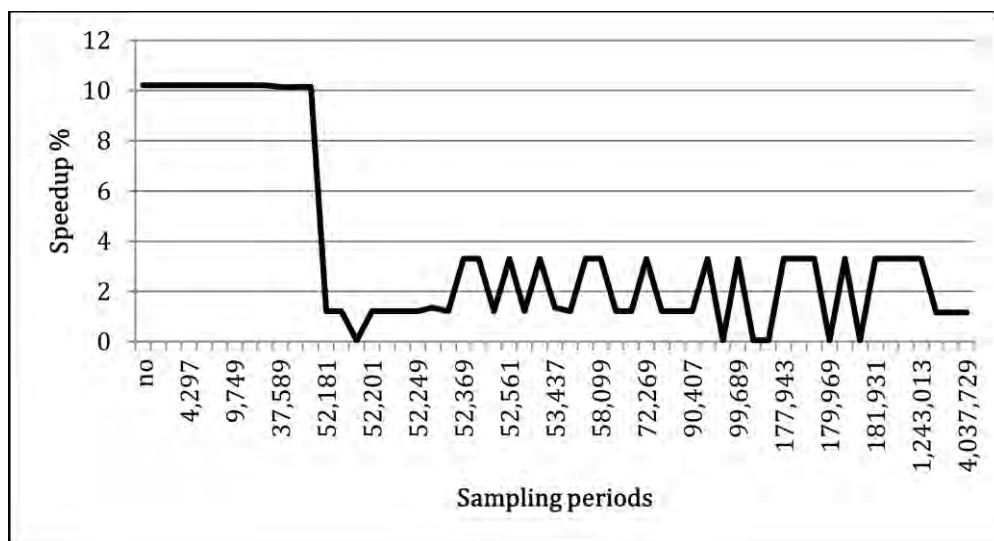


FIGURE 5.3: Correlation between final speedup and sampling period for miniMD test case in Scenario 1.

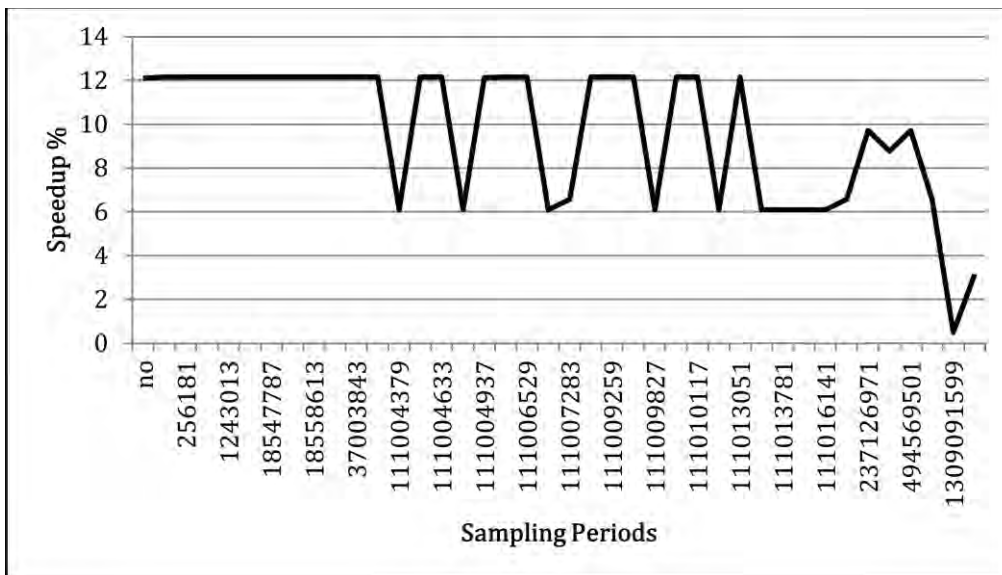


FIGURE 5.4: Correlation between final speedup and sampling period for HPCCG test case in Scenario 1

NAME	OBJECTS	OCCUPANCY	SAVED CYCLES	EXECUTION	NAME	OBJECTS	OCCUPANCY	SAVED CYCLES	EXECUTION
SP	10	0%	0M	0.00%	SP	1	0%	0M	0.00%
3D	4	98%	-59.6 G	-12.11%	3D	6	95%	-22.6 G	-4.60%
DRAM	5	45%	-	-	DRAM	1	54%	-	-
					NVRAM	2	60%	+29.4 T	+5,970.32%

(a) (b)

TABLE 5.3(a): Object distribution for HPCCG – Scenario 1 – Sampling period: none
TABLE 5.3(b): Object distribution for HPCCG – Scenario 1 – Sampling period: none

NAME	OBJECTS	OCCUPANCY	SAVED CYCLES	EXECUTION	NAME	OBJECTS	OCCUPANCY	SAVED CYCLES	EXECUTION
SP	0	0%	0M	0.00%	SP	0	0%	0M	0.00%
3D	4	98%	-59.6 G	-12.16%	3D	2	95%	-22.9 G	-4.62%
DRAM	3	43%	-	-	DRAM	3	36%	-	-
					NVRAM	2	60%	+29.4 T	+5,970.85%

(a) (b)

TABLE 5.4(a): Object distribution for HPCCG – Scenario 1 – Sampling period: 37,012,243

TABLE 5.4(b): Object distribution for HPCCG – Scenario 1 – Sampling period: 37,012,243

```

-- SP - 8388608 bytes --
  8 - 37 loads - 96 bytes
rtrans [HPCCG.cpp:94] - 37 loads - 8 bytes
  p [HPCCG.cpp:90] - 37 loads - 8 bytes
  r [HPCCG.cpp:89] - 37 loads - 8 bytes
  t4 [HPCCG.cpp:82] - 74 loads - 8 bytes
  t2 [HPCCG.cpp:82] - 37 loads - 8 bytes
  t0 [HPCCG.cpp:82] - 74 loads - 8 bytes
max_iter [HPCCG.cpp:73] - 65 loads - 4 bytes
  A [HPCCG.cpp:73] - 37 loads - 8 bytes
  normr [main.cpp:170] - 36 loads - 8 bytes
--
10 objects; 164 bytes (0.00195503234863%); 84780 saved

-- 3D - 8589934592 bytes --
44 - 494569966 loads - 6912000000 bytes
52 - 237127308 loads - 512000000 bytes
56 - 74004337 loads - 512000000 bytes
48 - 111005063 loads - 512000000 bytes
--
4 objects; 844800000 bytes (98.3476638794%); 59585933810 saved

-- DRAM - 34359738368 bytes --
12 - 18549196 loads - 256000000 bytes
28 - 37002331 loads - 512000000 bytes
20 - 37045783 loads - 512000000 bytes
16 - 37045795 loads - 512000000 bytes
40 - 989080399 loads - 13824000000 bytes
--
5 objects; 1561600000 bytes (45.4485416412%); 0 saved

```

FIGURE 5.5: Object distribution for HPCCG in Scenario 1 for zero sampling period. The objects in bold, placed in SP memory are discarded when performing sampled memory access profiling.

This behavior can be explained by gaining an insight to the source code and the memory access pattern it imposes. Firstly we remind that the main computational code which is profiled is run iteratively for every timestep, nevertheless we only execute two timesteps, from which only the second one is taken into consideration (as stated before, the first one acts as a cache warm-up stage). Therefore, the access pattern can be considered serial in the sense that memory objects are accessed (and thus have the chance to be discovered) only during this one iteration. Should the sampling period be too wide in comparison to the total memory accesses of an object, this particular object has more possibilities to be discarded, compromising, this way, the ultimate speedup.

5.2 HPCCG

Our results for MiniMD initially include 19 identified objects that cause LL cache misses. These objects tend to decrease in number as the sampling period increases

5.2.1 Complete memory access profiling

SCENARIO 1: In TABLE 5.3(a) we present the object distribution for the Scenario 1 architecture. 10 objects are small enough in order to fit in SP memory, though leaving it almost empty. 3D memory utilizes almost 100% of its capacity by storing 4 objects; these are the unique factor that results in performance a speedup of 12.11%. Finally, 5 objects are too big to be stored anywhere but in DRAM.

SCENARIO 2: TABLE 5.3(b) shows the memory object distribution in Scenario 2. This case diverges from the previous ones since we obtain a final slowdown rather than a speedup: although 10 objects can be stored in SP memory (as in the previous case), 2 in 3D memory and 5 in DRAM, this time NVRAM is being utilized by 2 memory objects. The long latency of this memory architecture in combination to the high memory access frequency of the particular objects results in a tremendous slowdown of ~5990%.

5.2.2 Sampled memory access profiling

Similarly as before, we set two prime numbers as lower and upper limit for the examined sampling periods. Our experiments range from 20,441 to 1,909,095,829. Profiling using the

first period behaves exactly as the non-sampled case discovering the same objects apart from the “obviously unimportant” ones that are shown in the beginning of FIGURE 5.5 and that do not play any role in the final speedup. The second sampling period identifies only one memory object that is placed in 3D memory thus achieving a very limited speedup. After the relaxed binary search the optimal sampling period for HPCCG was defined as 37,012,243. Up until this sampling period it is guaranteed that the final calculated speedup is close to the maximum possible one.

SCENARIO 1: In TABLE 5.4(a) we lay data distribution for Scenario 1 when profiling monitored one out of 37,012,243 memory accesses. There are no small enough objects to fit in SP memory, therefore it remains empty. On the contrary, 3D memory is almost fully occupied by the same objects as in the non-sampled profiling. Once again speedup rises up to 12.16% and is exclusively achieved because of these memory objects. Due to size limitations 3 memory objects are obliged to be stored in DRAM memory.

SCENARIO 2: TABLE 5.4(b) presents the data distribution for Scenario 2 when using the same sampling period. No objects are stored in SP memory while 3D and NVRAM are occupied by the same objects as before. DRAM in this case is the hosting memory of only 3 objects, 2 fewer than before. As it was anticipated memory architecture in combination with the memory object distribution result in a slowdown of ~5990%, instead of a speedup. NVRAM is again the prime reason for this compromise.

```

1.   for (int k=1; k<max_iter && normr > tolerance; k++ ) {
2.       if(k == 2) {
3.           CALLGRIND_START_INSTRUMENTATION; // start cache simulation
4.           CALLGRIND_TOGGLE_COLLECT; // start collecting
5.       }
6.       else {
7.           CALLGRIND_TOGGLE_COLLECT; // start collecting
8.       }
9.       // Computations
10.      if(k == 2) {
11.          CALLGRIND_TOGGLE_COLLECT; // stop collecting for iter 2
12.          CALLGRIND_ZERO_STATS; // zero counters
13.      }
14.  } // end of for-loop
15.  CALLGRIND_TOGGLE_COLLECT; // stop collecting
16.  CALLGRIND_STOP_INSTRUMENTATION; // terminate simulation
    // Finalize results
    return(0);

```

LISTING 5.1: The main computational part of HPCCG is enclosed in a for-loop. The source code is enhanced with macros that handle the interaction with Callgrind.

5.2.3 Sampling period comparison – Conclusions

The correlation between sampling periods and final speedup is slightly different from the previous case. The difference can be seen initially from the sampling period limits that were chosen for profiling. Intuitively, the absolute memory access number is to account for that, i.e. the more the accesses to memory, the wider sampling period can be. There is a proportional relationship between these two values that allows us to generalize.

On the other hand, there is an additional intrinsic characteristic of the application that allows such big sampling periods be effective: its memory access pattern. HPCCG, being a purely iterative method, places its main computational code in a loop. As seen in LISTING 5.1 this loop executes until the algorithm converges and the predicted lower limit is reached (in line 1). Computations refer to the same variables, to the same structures and eventually to the same memory locations, therefore obliging a repetitive memory access scheme. This can be interpreted as follows: memory objects that represent computation variables are accessed in every iteration; sampled profiling forces some memory accesses be discarded; prime numbers that are used as sampling periods exclude discovery of memory access patterns; sampling period can widen in reasonable frames (compared to the absolute access number) and still produce effective speedup. Indeed, the special way that memory interaction is formed enables lower sampling frequencies be implemented.

In FIGURE 5.4 we present the experimentation process for various sampling periods. We can pinpoint the period threshold that was defined above; nonetheless there is another interesting characteristic worth mentioning. Although we set 37,012,243 as the optimal sampling period, in fact there are even bigger periods that result in the same maximum speedup. For instance, periods 111,005,099, 111,008,371 and 111,013,051 all provide the same optimal performance improvement; yet in between these numbers can be found other periods that have a compromising behavior. In our understanding, there is a correlation between (a) the source code iteration number, and subsequently how many times each object access is repeated, and (b) the sampling period. In other words, when sampling period approaches the order of magnitude of the absolute access number per object, then small variations at the first might

result in discovering or omitting objects that play a definitive role in the final performance. In Figure 5.6 we emphasize that fact.

5.3 General conclusions

5.3.1 Performance

Regarding data distribution, in all cases SP memory presents either low or medium occupancy rates, thus not contributing to the final speedup. This is due to the low LL cache miss rate of the small objects that are placed in this memory subsystem. We observe an idle NVRAM in MiniMD case, meaning that the optimal performance can be achieved without its usage. In HPCCG case, two objects are placed in NVRAM because of size limitations causing a tremendous slowdown. Special attention should be paid in the combination between memory subsystem capacity and memory object size since performance undermining is possible. 3D memory with its specific characteristics is the architecture that proves beneficial in all cases. Despite the performance drop in one case, in general the varying memory latencies and sizes that are provided by the different subsystems of the heterogeneous memory system are proven beneficial and confirm that many applications can merit from a cautious data distribution.

5.3.2 Sampling

Sampled profiling that aims in detecting memory objects and distributing them can offer both advantages if used correctly and present a memory image that diverges greatly from the real one, if used improperly. In both cases, sampling periods up to the one that was defined as the optimal had similar effect on the detected objects: they mimicked a filtering function of discarding objects that have low access rate and kept the ones that are accessed frequently. The low access rate objects are allowed to be discarded since they cannot be beneficial in performance improvement while the others must be taken into consideration for the final distribution. Depending on the sampling period, the same or slightly different objects are discovered, however this should pose no threat as long as the latter is set carefully. In case of HPCCG, when the sampling period is set too large in Scenario 2 we foresee a potential speedup which is far from what is actually happening. This should emphasize on the importance of setting an application-specific (or application type-specific) sampling period.

We also discovered a close relationship between the memory access behavior and the rate in which accesses should be monitored which, in our understanding, can be applied on every application that exhibits similar access pattern.

CHAPTER 6

SUMMARY

This work commenced by introducing the notion of heterogeneous memory systems as an answer to the saturated CPU performance. Systems that deploy such memory organization have already established their place both in academic / scientific field and in market. Every component of such system presents distinct characteristics, thus, in order for an application to benefit the most from it, it is mandatory that the data it leverages be carefully distributed in the various subsystems.

To address this issue we proposed application profiling techniques distinguishing between software and hardware approach. For software, we chose Valgrind and in particular EVOP, an extension of it. Our interaction with the tool drove us in clarifying a potentially misleading point regarding the cache simulator employed by the tool. In the process of doing so we introduced Callgrind, a complete software profiler, and analyzed various parts of Valgrind's (i.e. EVOP's) source code that is used in cache simulation, easing future developers and technology enthusiasts to gain a more user-friendly insight to the tool's internal organization. In the same spirit, we presented a full set of choices provided by Callgrind in order to control the profiled execution and achieve more time efficient profiling execution.

On the other hand, we presented some hardware profiling principles such as the need for sampling. To address this problem we implemented a sampling mechanism by extending EVOP's source code and integrating our changes to the tool's API. Our experiments advanced by defining two similar, yet diverse heterogeneous memory architectures, profiling two test

cases with characteristic memory behavior in order to monitor LL cache misses, extracting the memory objects they interact with and distributing them to the different memory subsystems. Profiling was performed both by intercepting every single memory access and in a sampled manner in order to define the optimal sampling period that allows maximum performance speedup.

Our results can be divided in two categories. The first one regards the maximum speedup achieved by the combination of hybrid memory system and cautious data distribution; in both test cases we observe that applications merit from it. The second one reflects how different sampling periods are related to diverse memory access patterns. Our conclusions can be used in order to eliminate the sampling rate margins when opting for hardware profiling.

6.1 Future research

In this subsection we propose future research activities that can be triggered by our research. They can be divided in two groups, those aiming in assessing the application-specific sampling periods in hardware profiling mechanisms and those defining the optimal memory subsystem mosaic.

ASSESS SAMPLING ON HARDWARE

Software profiling, as the one described in this work, although accurate, poses the threat of extreme timing overhead that can be a forbidding factor when the application size is too big. On the contrary, hardware profiling eliminates this drawback by intercepting actual CPU events. These events however have to be sampled, and depending on the sampling period, a different sort of timing overhead can be added. If sampling frequency is too high, then profiling performance is damaged, while in case of too low sampling frequency, the effectiveness and the accuracy of the results might be doubted.

The sampling period thresholds that we defined are effective when used in software. We also foresee that they can be applied in hardware-aided profiling, in order to save execution time; however this needs to be examined and verified. In case the results are similar, and given that the tested miniapplications (test cases) have characteristic access patterns of more complex

applications, generalizations and assumptions about the sampling rates of other applications are safe to be made in order to maximize profiling performance.

MEMORY SUBSYSTEM ARCHITECTURE

In this work we assess the optimal distribution of application data among different memory subsystems. The methodology that is followed is based on a form of data oriented profiling, emphasizes the importance of cautious data placement while implying the influence of the subsystem combination and object sizes to the final performance. In two cases, a particular subsystem either remains unutilized or diminishes performance. Following our analysis we consider feasible to determine prototype heterogeneous memory system architectures for different applications depending on the size of the problem they are addressing to the end of achieving improved performance.

Bibliography

- [1] Moore, Gordon E., “*Cramming more components onto integrated circuits*”. *Electronics*, Volume 38, Number 8, April 19, 1965
- [2] Wen-mei W. Hwu, Izzat El Hajj, Simon Garcia de Gonzalo, Carl Pearson, Nam Sung Kim, Deming Chen, Jinjun Xiong, Zehra Sura, “*Rebooting the Data Access Hierarchy of Computing Systems*”, *Rebooting Computing (ICRC) 2017 IEEE International Conference on*, pp. 1-4, 2017
- [3] D. Levinthal, “*Performance Analysis Guide for Intel Core™ i7 Processor and Intel® Xeon™ 5500 processors*”, Intel® Corporation, 2009. http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf
- [4] Shen, Du & Liu, Xu & Lin, Felix. (2016). “*Characterizing emerging heterogeneous memory*”. 13-23. 10.1145/2926697.2926702
- [5] A. Sandberg, D. Eklov, and E. Hagersten. “*Reducing cache pollution through detection and elimination of non-temporal memory accesses*”. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10*, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7559-9
- [6] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. “*Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems*”. In *Proceedings of IEEE 14th International Symposium on High Performance Computer Architecture*, pages 367–378, Feb 2008. doi: 10.1109/HPCA.2008.4658653
- [7] B. Bao and C. Ding. “*Defensive loop tiling for shared cache*”. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), CGO '13*, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society
- [8] Texas Instruments. DSP products website. <http://www.ti.com/lstds/ti/dsp/overview>. page. Last accessed: Dec. 08, 2014
- [9] A. Sodani, “*Knights Landing (KNL): 2nd Generation Intel® Xeon Phi™ Processor*”. <https://www.alcf.anl.gov/files/HC27.25.710-Knights-Landing-Sodani-Intel.pdf>. Last accessed: Sep. 10, 2019

-
- [10] Banakar, Rajeshwari & Steinke, Stefan & Lee, Bo-sik & Balakrishnan, M. & Marwedel, Peter. (2002). “*Scratchpad Memory: A Design Alternative for Cache On-chip memory in Embedded Systems*”. 10.1145/774789.774805
- [11] 3D stack memory: G. H. Loh. “*3d-stacked memory architectures for multi-core processors*”. In Proceedings of the 35th Annual International Symposium on Computer Architecture, ISCA '08, pages 453–464, 2008
- [12] D. H. Woo, N. H. Seong, D. L. Lewis, and H.-H. Lee. “*An optimized 3D-stacked memory architecture by exploiting excessive, high-density TSV bandwidth*”. In IEEE 16th International Symposium on High Performance Computer Architecture (HPCA). IEEE, Jan. 2010.
- [13] “*Dynamic random-access memory*”. https://en.wikipedia.org/wiki/Dynamic_random-access_memory. Last accessed: Sep. 10, 2019
- [14] M. Baker, S. Asami, E. Deprit, J. Ousetterhout, and M. Seltzer. “*Nonvolatile memory for fast, reliable file systems*”. In Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS V, pages 10–22, 1992
- [15] C. Wang, S. S. Vazhkudai, X. Ma, F. Meng, Y. Kim, and C. Engelmann. “*NVMalloc: Exposing an aggregate SSD store as a memory partition in extreme-scale machines*”. In 26th International Parallel & Distributed Processing Symposium (IPDPS). IEEE, 2012, pp. 957–968
- [16] Valgrind repository. <http://www.valgrind.org/downloads/repository.html>. Last accessed: Sep. 14, 2019
- [17] Valgrind official webpage. <http://www.valgrind.org>. Last accessed: Sep.15, 2019
- [18] A. J. Peña and P. Balaji, “*A framework for tracking memory accesses in scientific applications*”. In 43rd International Conference on Parallel Processing Workshops (ICPP Workshops), Minneapolis, MN, 2014.
- [19] Servat, Harald & Peña, Antonio & Llort, Germán & Mercadal, Estanislao & Hoppe, Hans-Christian & Labarta, Jesús. (2017). “*Automating the Application Data Placement in Hybrid Memory Systems*”. 126-136. 10.1109/CLUSTER.2017.50.

-
- [20] A. J. Peña and P. Balaji. “*Toward the efficient use of multiple explicitly managed memory subsystems*”. In IEEE International Conference on Cluster Computing (CLUSTER), 2014, pp. 123–131.
- [21] G. Mathews, “*On the partition of numbers*”. Proceedings of the London Mathematical Society, vol. 1, no. 1, pp. 486–490, 1896.
- [22] Mantevo project official webpage. <https://mantevo.github.io/index.html>. Last accessed: Sep.18, 2019
- [23] Sandia National Laboratories, “*LAMMPS molecular dynamics simulator*.” Official webpage: <http://lammps.sandia.gov>. Last accessed: Sep.18, 2019