

The Concept of Memory Stubs as a Specialization of Dynamic Performance Stubs to Simulate Memory Access Behavior*

Peter Trapp Christian Facchi Sebastian Bittl
University of Applied Sciences
Esplanade 10
85049 Ingolstadt
-Germany-
{peter.trapp, sebastian.bittl.eit, christian.facchi}@fh-ingolstadt.de

Abstract

Dynamic performance stubs provide a framework for the simulation of the performance behavior of software modules and functions. Stubs can be used for a cost-benefit analysis of the gain from performance optimization and therefore, for a gain oriented performance improvement, and can be also be used to identify “hidden” bottlenecks and the most relevant candidates for optimization. This paper evaluates memory stubs in detail as a special subset of dynamic performance stubs to optimize memory bound modules or functions. It describes and validates the possibility of simulating the memory and data cache access behavior of software modules and functions. Therefore, a new foundation for gain oriented optimization of memory behavior has been achieved.

Key words: performance improvements, memory access behavior, stubs, performance modeling

1 Introduction

Dynamic performance stubs have been introduced in [TRAP07]. They can be used for “hidden bottleneck” detection, and, by demonstrating the level of optimization potential, a cost-benefit analysis can be performed as well. This leads to more gain-oriented performance optimizations.

In the past, performance increases in many system architectures have been achieved through higher CPU speeds, and more recently, through using multiple cores. Yet the memory speed, and hence the memory access times, did not increase to the same order as the CPUs frequencies [SEAR00]. This has led to the fact that many current systems are heavily memory bound and consequently software performance optimization studies are often targeting the improvement of the memory usage. The methodology of *dynamic performance stubs* can be used to optimize these memory bound systems by *memory stubs*.

1.1 Dynamic Performance Stubs

The idea beyond *dynamic performance stubs* is a combination of performance improvements [JAIN91, FORT03, LILJ00, GUNT98] in already existing modules or functions and the stubbing mechanism from software testing [BERT05, SOMM01]. The performance behavior of the *component under study* (CUS) will be determined and replaced by a software stub. This stub can be used to simulate different performance behaviors which can be parameterized, and the optimization expert can use these to analyze the performance of the *system under test* (SUT). This procedure relates to stubbing a single software unit and hence it will be called “local”. Therefore a “local stub” has to be built. The *performance simulation functions* (PSF) can also be used to change the behavior of the complete system. A software module has to be created which interacts “globally” in the sense of influencing the whole system instead of only a single software component. This stub will be called a “global stub”.

Figure 1 sketches the design and the interaction between a real system on the left and the *dynamic per-*

*This paper has been presented at CMG 2009. <http://www.cmg.org/conference/cmg2009/>

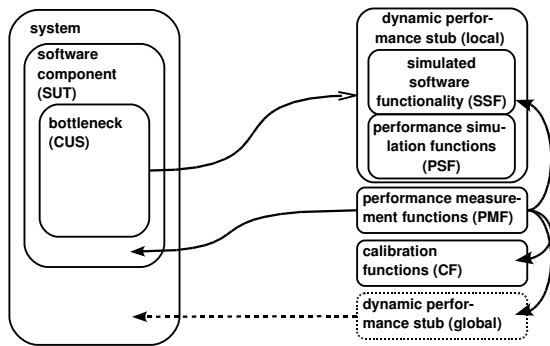


Figure 1: Interactions of “Dynamic Performance Stubs”

formance stubs on the right side. The lined arrows indicates replacements. Filled arrowheads describe the extension of a unit by this feature and the dashed block provides an additional functionality to the *dynamic performance stub* and will not really stub a software unit.

The framework of the *dynamic performance stub* consists of the following parts which are presented in Figure 1:

- Simulated Software Functionality (SSF)
The *simulated software functionality* is used to simulate the functional behavior of the CUS in order to provide proper system behavior.
- Performance Simulation Functions (PSF)
Performance simulation functions provide the ability to simulate the performance behavior of the replaced CUS, and are divided into four categories:
 - CPU
 - Memory
 - I/O
 - Network
- Performance Measurement Functions (PMF)
To provide a basic set of evaluation possibilities the *performance measurement functions* can be used. They are mainly glue/wrapper functions for the measurement functions already provided by the system.
- Calibration Functions (CF)
In order to provide trustworthy results, the stubs have to be adjusted to a dedicated system. This can be done using the *calibration functions*.

For more detailed information on *dynamic performance stubs* the reader is referred to [TRAP07].

1.2 CPU Stubs

CPU stubs have been targeting to handle CPU bound systems. Therefore, a general approach to parameterize the runtime behavior and CPU usage has been

achieved as well as a possible realization has been defined. In order to create these *CPU Stubs* a methodology for evaluating and creating these stubs is provided in [TRAP08].

1.3 Memory Stubs

Memory stubs are a special case of *dynamic performance stubs* regarding the memory usage. They can be used to simulate the memory performance behavior of a suspected software bottleneck. Hence, they combine the *simulated software functionality* and the *memory performance simulation functions*.

This paper describes the concept of *memory stubs* in more detail. It mainly discusses a possible implementation of the *performance simulation functions* and evaluates the impact on the data caching and memory behavior to the system. Furthermore, it explains a methodology to create cache misses and hits in a deterministic way at the desired level and granularity. Additionally, first steps towards a methodology are given. To conclude, a case study of simulating the performance behavior of a memory bottleneck is included.

The *memory stubs* will be used to improve the LTE¹ telecommunication software from Nokia Siemens Networks (NSN). The system under test consists of an Intel Pentium 4 CPU where the hyperthreading functionality has been disabled.

2 Caching Concepts

Most designers of architectures try to avoid an access of the main memory because of the time needed to load data from there to the CPU, e.g. about 240 cycles for an Intel Pentium M processor [DREP07]. Therefore, the developers of CPUs implement several strategies of using caches to decrease the need to access the main memory and to increase the speed of execution, e.g. instruction- and data-prefetching [vdPA02, SEAR00] or branch predictions.

2.1 Overview

Much research work in the area of caching has been done, and this section summarizes different research scopes. The overall caching structure has been explained in [GENU04, DREP07, HILL89, vdPA02]. In [BEYL01] a classification of different cache misses is described as the “Three Cs”: “compulsory”, “conflict” and “capacity”.

Compulsory Misses. Compulsory misses, also often referred as “cold” misses, can be seen if no data has been referenced to the specific cache line. Therefore, they occur only at the very first access to the cache line.

¹LTE: Long term evolution is the successor of UMTS.

These misses normally do not strongly influence the system behavior [LEBE95], except in the initial phase.

Conflict Misses. These misses occur when a still valid cache line will be replaced by a cache fill (see also [INTE06]) despite the the existence of empty or invalid data in the cache. These misses depend on the cache structure as discussed below.

Capacity Misses. The last category capacity misses can be seen if a valid cache line will be replaced because neither an empty nor a non-valid cache line can be replaced.

Cache Hierarchy To reduce the cost of accessing the memory, caches have been introduced in the hardware architecture. Additionally, to improve the access times, further different types of caches, e.g. data- and instruction-caches, as well as cache hierarchies are included in modern CPUs (see also [GENU04]).

Application Behavior Predictions Cache behavior predictions for applications have been done in [FERD99]. In [MANE09] a tool for evaluating cache architecture has been introduced. This tool is based on techniques as described in [MANE02].

Metrics To improve data locality and to evaluate memory reference, several metrics have been introduced. In [WU97] “unit strides” have been used to study the caching optimizations for scientific programs. The “stack distance” as proposed in [BEYL01] measures the distance between references to the same data location and has been applied to compiler optimization techniques. Another metric is called “reference distance” and is described in [PYO97, RIVA98]. This metric can be split further into “address distance” and “block distance”.

Loading Strategies In [SEAR00] several different aspects of caching have been described, e.g. cache line prefetching and cache alignments. Additionally, in [SEAR00] and [LEE99] “colored bits” are explained. These determine the cache set where the data will be stored, as well as enforcing that complete cache lines will always be fetched.

Replacement Policies The architecture uses replacement policies to determine the cache line which will be overwritten if a conflict exists. Different policies are explained in [AL-Z04, REIN06], while [INTE06] describes the policy used in some Intel Architectures.

Write Policies Different strategies exist to write modified data stored in cache back into the main memory. Commonly used write policies are “write-through” and “write back” [GENU04].

Compiler Optimization Last but not least, the caching behavior of applications can be improved during compile time. There are many optimization flags for the GCC (GNU Compiler Collection) available [BEYL01]. Some of the optimizations techniques are described in [RIVA98, PYO97, LEE99].

Cache Design Data consistency throughout the different cache levels can be realized by different cache designs. There are three basic design: “exclusive”, “inclusive” and “non-inclusive”. Please refer to [JACO07] for more information.

2.2 Related Work

In [MANE02] an approach for creating data cache hits and misses has been described. The algorithm constantly access data and evaluates the amount of cache hits and misses by evaluating the timing behavior of the application. It aims at measuring the caching architecture of a system and provides information about the cache levels, cache line size, associativity and access times. The tool determines information about data or unified caches as well as TLB caches.

The approach used in [MANE02] differs from our approach in the way that it constantly accesses data and the hit or miss rate is evaluated by timing measurements afterwards. In contrast to that, we want to simulate a desired amount of cache hits and misses, e.g. 1000 level two misses, 20 level two hits and 5000 level one hits, in a deterministic way.

3 Introduction of Memory Stubs

To achieve a deterministic cache behavior by a *memory stub*, methods for forcing cache misses have to be studied. There are many different approaches to changing the caching behavior of a system. Some global approaches to disable caching are:

- during start up:
Some operating systems provide the possibility of disabling the caching in the start up phase of the system. This can be used to simulate the worst case scenario which forces the access to the main memory.
- via processor registers:
Another possibility is to modify some CPU registers, e.g. the CR0 register of the Intel P4 CPU [INTE03]. This approach can be used to deactivate the cache, e.g. for test case runs. But it can also be used for periodically enabling and disabling the cache. Additionally it can be used to disable caching during the runtime of a dedicated software module or function.

These are more general approaches because the whole caching architecture will be deactivated which influences the complete system.

In order to simulate the memory behavior of modules or functions the *memory stubs* have to be able to describe the different memory access types in greater detail. In particular, the stubs should simulate cache references into each of the cache levels, e.g. level one data cache. The following list provides an overview of events which can occur: cache read and write hits and misses to specified cache levels. The events can be applied to non- and unified cache architectures. However, some of the items may not be applicable to every caching architecture, e.g. there are no write misses for the instruction cache. If a write miss to the instruction cache happens, e.g. using self-modifying code, the complete instruction cache will be flushed by the system [INTE08].

This paper elaborates a possibility to simulate data cache events for the different cache levels in detail. Therefore, an approach for creating data cache read/write hits and misses on the desired cache level is described and validated through a case study.

4 A Method for Creating Data Cache Events

This section describes a method for producing adjustable amounts of data cache events. The idea for building a *memory stub* for data cache misses is to access data, e.g. a variable inside a program that is not stored in the data cache on the target level and therefore has to be read from higher levels or main memory. This will cause a cache miss on the target level. A cache miss on a dedicated level will cause either a cache hit on the next level if the data is available on the next cache level or a cache miss if the data is not available on the next cache level. This is only applicable to inclusive caches.

There are different caching architectures, which aim to avoid cache misses as far as possible. Implemented in our testing system (see Section 6) is a n -way set associative cache where every location of the main memory can be loaded to n different locations in the cache [INTE08]. The physical address inside the main memory is used to determine into which cache set the data is stored. Therefore, it is possible to access data from memory which will be stored into a dedicated cache set (see also [BEYL01, RIVA98]).

These cache sets consist of n different cache lines per cache set ($ASSOC$). To permanently create conflict or capacity cache misses the minimum amount of sequential data ($SIZE$) can be calculated as described by Equation 1.

$$SIZE = CACHESIZE + \frac{CACHESIZE}{ASSOC} \quad (1)$$

Any value of $SIZE$ greater than the cache size ($CACHESIZE$) can be used to write into each cache line, so the cache has been completely filled. This applies to most of the cache replacement policies as will be discussed below. A $SIZE$ smaller or equal to cache size will only produce compulsory misses at the first access and every following accesses are a cache hits in an otherwise isolated environment. In order to reproducible overwrite a cache set, $ASSOC + 1$ accesses to different data referencing the same cache set has to be done. Therefore, the size of the array has to exceed the size of the cache by the amount of cache size divided by the associativity bytes. This can be achieved by adding $CACHESIZE/ASSOC$ bytes to the array. Hence, $ASSOC+1$ different data references for any cache set are possible.

$SIZE$ is the optimum size for constantly overwriting cache lines. Any smaller value will not lead to overwrite any cache line periodically. Values greater than $SIZE$ can add additional trashing in other cache levels because the data sub sequentially has to be loaded into the higher levels as well. The $SIZE$ value provides further advantages such as:

- A smaller amount of memory will be used.
- Runtime improvement of the application, because only data which has to be referenced has to be allocated or loaded into higher cache levels. Otherwise, it would lead to a reduction of performance.

However, in some cases an array size greater than $SIZE$ might be the best choice because of some side effects, e.g. to create different amounts of cache hits or misses in different cache levels at the same time, to reduce the applications overhead, or to improve the scalability of the amount of created cache events. This strongly depends on the needs of simulating the cache behavior for *memory stubs* and can be easily realized.

If less than n different data elements which are be stored to the same cache set are accessed in sequence, no more cache misses other than the first reference, which always generates a miss, will occur. Hence, it is necessary to access at least $n+1$ different data elements in sequence which are to be loaded into the same set of cache lines in order to consistently get cache misses during the runtime of the *memory stub*. The approach is intended to work for read misses as well as for write misses.

4.1 Realization

This behavior can be used to create a desired amount of cache misses or hits in the dedicated cache level.

```

1 #include <string.h>
2 #define BLOCKS      256
3 #define CACHELINE   64
4 #define ASSOC       8
5 #define ADD_ASSOC   1
6 #define USED_ASSOC  (ASSOC+ADD_ASSOC)
7 #define CACHESIZE   (BLOCKS*CACHELINE)
8 #define STRIDE      (CACHESIZE/ASSOC)
9 #define ARRAYSIZE   (CACHESIZE *
10                      USED_ASSOC/ASSOC)
11 #define SETS        1
12 #define ITERATIONS  10000
13
14 char myAr[ARRAYSIZE];
15
16 void createMisses ()
17 {
18     for (i=0; i<ITERATIONS; i++){
19         for (j = ARRAYSIZE-STRIDE; j >= 0;
20              j -= STRIDE){
21             for (k = (SETS-1)*CACHELINE;
22                  k >= 0; k -= CACHELINE){
23                 myAr[j + k] = a;
24             }
25         }
26     }
27     memset(myAr, 0, ARRAYSIZE/sizeof(int));

```

Listing 1: Algorithm to Specifically Access Different Cache Sets

Listing 1, called *dedicated memory access*, describes a possible realization of an algorithm to write into dedicated sets of the cache using the programming language C. The constants *BLOCKS*, *CACHELINE* and *ASSOC*² describe the cache architecture according to [MARI07]. The total cache size (*CACHESIZE*) can be calculated using the number of cache blocks (*BLOCKS*) multiplied by the cache line size in bytes (*CACHELINE*). The amount of different cache sets (*SETS*) is the number of cache blocks divided by *ASSOC*.

There are three more constants which have to be defined:

- *ADD_ASSOC*

Despite of the optimal array size as described in Equation 1, the evaluations done on the target hardware have shown that the size of the array often has to be increased to reduce further influences of the hardware, e.g. prefetching mechanisms. Therefore, the constant *ADD_ASSOC* has

²*ASSOC* is the order of associativity in a n-way set associative cache, e.g. *ASSOC* = 4 in a 4-way set associative cache.

been introduced. It specifies the amount of additional “cache size divided by assoc” bytes, which will be added to the cache size. The constant *USED_ASSOC* simply adds the values *ASSOC* and *ADD_ASSOC*. The total array size (*ARRAYSIZE*) can be calculated as shown in Listing 1 in Line 9.

- *SETS*

The integer constant *SETS* is used to denote the amount of different cache sets which will be used. The minimum value is set to “1”, which means that only a single cache set will be used. The maximum value in this algorithm for the *SETS* is the amount of blocks (*BLOCKS*) divided by the associativity (*ASSOC*). In this case the complete cache will be used for the evaluation.

- *ITERATIONS*

The *ITERATIONS* constant is defined to increase and to adjust the total number of cache access.

The constant *STRIDE* determines the minimum distance in bytes of two different memory locations which will be stored into the same cache set.

The realization of the *dedicated memory access* approach is done through accessing data inside an array. The array is initialized using the *memset* function as depicted in Line 26 of Listing 1. The access is performed inside two loops, providing the possibility to control the amount of cache misses through the amount of iterations.

The “inner loop” (Listing 1 Line 19) is used to write into more than a single cache set. The “outer loop” (Listing 1 Line 18) shifts the position in the array by *cache size* divided by the associativity bytes. This has been coded as a loop to provide the possibility to access the array with a distance of *STRIDE* bytes. Therefore, the next access to the same cache set uses data which have not been since *USED_ASSOC* accesses. The array has the size of *ARRAYSIZE* as described above and can therefore be used to repeatedly access a complete cache level. The value for *ARRAYSIZE* will always be an integer because the value of *CACHESIZE* is always a decimal multiple of *ASSOC*. The algorithm walks backwards through the array to prevent the application from prefetching functions of the CPU [MANE02]. A value will be written into the array (*myAr*), therefore a write miss will be created. In order to create read events, the line “*myAr[j+k]=a;*” can be replaced by “*a = myAr[j+k];*”.

When using the described method for data cache misses on a cache level, it produces cache hits in the upper level automatically as long as the amount of constantly referenced data can be contained in the upper level. Otherwise the upper level will also be overwritten. The upper level cache typically has more capacity than the dedicated level, so after the unavoidable first

time misses for each accessed data element, the dedicated data cache misses are automatically hits for the higher level data cache.

It is not possible to get hits on the cache level without getting misses on lower cache levels because data found in the upper cache level is brought into the lower cache levels. This is the normal behavior for any inclusive cache architecture.

The approach described here can additionally be used to create level one data cache hits, but it has to be modified slightly. Every access to an element of the array has to be inside a single cache line ($ARRAYSIZE = CACHELINE$), and the number of sets ($SETS$) has to be set to one. Additionally, the loop of Listing 1 in Line 19 has to be adjusted to access only bytes within the same cache line. Due to the access to different locations within the same cache line the data cannot be stored in a register. Therefore it accesses the level one cache where the data is already available. Referencing a simple variable in a loop will not be usable since this variable will be stored in the registers of the CPU due to computer optimization.

To summarize the possibilities of the *dedicated memory access* approach. It is able to simulate, in a deterministic way, the memory performance behavior of a software component and therefore can be used by adjusting the parameters for *memory stubs*. Particularly, it can simulate the following data cache behavior:

- Level one hits
- Level one misses / level two hits
- Level two misses / memory hits

The approach as described above creates write misses. It can be easily changed to create read misses and, as well, to simulate further caching levels, e.g. level three.

The validation has been done in a simulated environment to receive detailed information about single cache hits and misses. Additionally, to ensure the usability of the approach the test has been evaluated on the target hardware. The results are discussed in the Section 6.

4.2 Discussion

This section discusses the amount of cache hits or misses occurring during the execution of the *dedicated memory access* approach more in detail.

$$ICSR(a_1, a_2) := \exists n \in \mathbb{N}^0 : |a_2 - a_1| = n \times \frac{CACHESIZE}{ASSOC} \quad (2)$$

We define A as a set of main memory addresses, e.g. a_1, a_2 , of memory references. Equation 2 can be used to determine if two memory accesses referencing the

same cache set (*identical cache set references, ICSR*). Two data elements will be stored in the same cache set if their main memory addresses (a_1, a_2) differ by the amount of $\frac{CACHESIZE}{ASSOC}$. Therefore, all multiples $n \in \mathbb{N}^0$ of the distance of two references will be stored to the same cache set.

$$CSR = |\{a \in A | \exists a_2 \in A : a \neq a_2 \wedge ICSR(a, a_2)\}| \quad (3)$$

We define the number of *cache set references (CSR)* as the number of independent memory accesses in the inner loop body to the same cache set. In Equation 3 a formal definition is given. CSR^3 is the number of elements of the set A with different addresses (a, a_2) satisfying the constraints of Equation 2.

To describe the possible range of cache events created by the *dedicated memory access* approach with arbitrary replacement policies⁴ different types of misses according to [BEYL01] have to be considered: compulsory ($MISS^{comp}$) and conflict/capacity misses ($MISS^{con}$). The cache will be treated as empty at the start of the application. Therefore, compulsory misses will occur if the application has not used the cache line before. Every first access of data will always create a cache miss. This happens during the first iteration of the inner loop described in Listing 1. If more than a single cache set has been referenced the values given in the equations from this section have to be multiplied by the number of used cache sets. Due to the fact that the outer loop will call the inner loop it has to be called at least one time ($ITERATIONS \geq 1$). The following cases have to be distinguished:

- $1 \leq CSR \leq ASSOC$:

$$MISS^{comp} = CSR \quad (4)$$

$$MISS^{con} = 0 \quad (5)$$

$$HITS = ITERATIONS \times CSR - CSR \quad (6)$$

- $CSR > ASSOC$:

$$MISS^{comp} = ASSOC \quad (7)$$

$$MISS^{con} = ITERATIONS \times CSR - ASSOC \quad (8)$$

$$HITS = 0 \quad (9)$$

For $CSR \leq ASSOC$ the independent cache references in the first iteration of the *dedicated memory access* loop will produce compulsory cache misses (so Equation 4 holds). After this iteration the data which will be referenced are stored inside the cache because

³We use CSR as counter for different elements of A . We neglect the parameter for convenience.

⁴Except of the random replacement strategy as discussed in Section 4.3

there are enough different cache lines (*ASSOC*) available in the same cache set. All other occurring accesses, therefore, will result in a cache hit on the desired level (so Equations 5 and 6 hold). This is only partly true for the level one cache. Here the independent cache set references can also be served from registers, depending on the available registers in the CPU and on the amount of *CSR*. A slightly modified *dedicated memory access* approach is additionally described at the end of Section 4.1 for level one cache hits. In this case the amount of compulsory misses is 1^5 and the amount of hits is the total amount of references minus the compulsory miss ($CACHELINE * ITERATIONS - 1$).

In the case of $CSR > ASSOC$, more independent data references will be used than can be stored in a cache set. Therefore, every access to the data element will result in a cache miss (so Equation 9 holds). The number of compulsory misses is *ASSOC* since every cache line in the set will be written (so Equation 7 holds). Hence all other references will be conflict/capacity misses (so Equation 8 holds).

The amount of cache hits as described in this approach can be less than the adjusted value in real environments. This is expected as other processes running on the CPU will influence the amount of data in the cache. As this is an expected behavior it does not influence the usability of the *dedicated memory access* approach for the *memory stubs*. Moreover, the behavior can be used to spot memory problems of the application (see Section 7). The evaluations done in this sections cover a single cache set. In order to get the total amount of cache references the results from the equations have to be multiplied by the number of referenced cache sets.

4.3 Replacement Policies

The deterministic behavior of the approach depends on the replacement policy of the caching algorithm. The following replacement policies will be discussed (see also [REIN06, AL-Z04]):

- First in first out (FIFO), least recently used (LRU), least frequently used (LFU)
- Pseudo least recently used (PLRU)
- Random Replacement

The amount of cache hits or misses will only be discussed in the following subsections, if they differ from the values described in Section 4.2.

4.3.1 FIFO, LRU, LFU

The replacement policy FIFO always overwrites the cache line which has the longest stay in the cache.

⁵Assuming that the cache line is aligned.

Due to the nature of the algorithm there will always be cache misses when data elements which are not in the cache are accessed when there are no empty cache lines available. Therefore, the algorithm in Listing 1 can be used to deterministically create cache misses with every access because it references more cache lines per cache set than can be stored in the cache set.

The LRU always overwrites the cache line which has not been used for the longest time. If every cache line is sequentially used every cache line will be accessed by the amount of references. Therefore, there is no difference between the FIFO and the LRU policy in Listing 1. The *dedicated memory access* approach shows the same behavior for the LRU as for the FIFO strategy and can easily be used for *memory stubs*.

The same behavior as described for LRU can be applied to LFU because of equidistant data access. There will be no cache line which has more often been referenced by the application.

4.3.2 PRLU

The tree-based Pseudo-LRU (PLRU) as described in [REIN06] approximates the least recently used data. This is realized using tree-bits to point to the (approximated) oldest data stored in the cache set. On sequential accesses to independent data (*CSR*) this approximation gives the same results as the LRU. After *ASSOC* sequential accesses, the used tree bits point to the cache line which will be replaced next exactly as the LRU does. Therefore, the *dedicated memory access* approach can be used with the PLRU.

4.3.3 Random Replacement

Random replacement strategies, e.g. implemented via a linear feedback shift register (LFSR) [AL-Z04] will influence the amount of produced cache misses. Due to the fact that a random replacement policy is in place, no exact value can be given.

5 Towards a Methodology

So far only the content of a toolbox has been presented, but now the first steps towards a methodology on using memory stubs for performance improvement will be given. So this chapter can be seen as first version of a how to use *memory stubs* for optimizing memory bound systems.

1. Determine CUS: As a first step of any optimization the timing behavior of the system in total has to be examined. Then one component of the system (CUS) has to be chosen, which seems to be a bottleneck [TRAP07, JAIN91]. Several performance measurements have to be done until the results

seem to be deterministic within a given confidence interval. The following values, as discussed at the end of Section 4.1, shall be deterministic:

t^{CUS} the time which will be used in the CUS

t^{SUT} a performance indicator of the module or function. This value will be measured to validate the systems performance in total. Often it is the time used for the completion of a service.

$L1_{readhit}^{CUS}$ a counter where the number of L1 cache read hits caused by the CUS has been measured.

$L1_{writehit}^{CUS}$ a counter where the number of L1 cache write hits caused by the CUS has been measured.

$L1_{readmiss}^{CUS}$ a counter where the number of L1 cache read misses caused by the CUS has been measured.

$L1_{writemiss}^{CUS}$ a counter where the number of L1 cache write misses caused by the CUS has been measured.

$L2_{readmiss}^{CUS}$ a counter where the number of L2 cache read misses caused by the CUS has been measured.

$L2_{writemiss}^{CUS}$ a counter where the number of L2 cache write misses caused by the CUS has been measured.

Please note that the way to determine the values of L1 and L2 parameters strongly depends on the available tool set. As an example the OProfile tool set can be chosen. Additionally, further cache events, e.g. level three read hits, can be used if applicable. This step has to be repeated a couple of times in order to get some statistical distribution of the required values. If there is only a small variation we can go further.

2. Create Memory Stub. After creation of the functional stub, which covers the functional behavior, a performance stubs has to be created, according to Section 4.1.

So the above listed values will be simulated by the component. Please note that some values can only be approximated because of the construction of the cache, e.g. the cache line size. As a last step, the runtime behavior of the CUS has to be simulated by an loop of NOPs to simulate the execution time t^{CUS} . For further details see [TRAP08].

3. The measurement results have to be validated. The value of t^{SUT} with or without memory stub has to be equal. Whenever there is a significant difference, further analysis of the memory stub has to be done. E.g. it might be possible that the CUS is too big and has to be decomposed.

4. Now, a first estimation of the CUSs optimization has to be validated. Normally, the developer of this component has rough estimations of the performance improvements which will be used for further studies. If not, he will simply reduce the value of cache usage by e.g. 10% and start a measurement. Therefore, he simply reduces the number of L2 accesses and furthermore the number of L1 accesses. The determined time will be called $t_{improved}^{SUT}$. This measurement is an approximation for the improved system behavior after optimization. It should be repeated with further reductions of the number of access operations.

Based on the measurement results, an ideal target value for memory optimization can be determined and a cost estimation of the improvement function can be done. Now, a detailed cost-benefit analysis is possible to determine the ideal optimization factor. This will lead to a solid basis for a performance optimization.

5. Based on the measurement results of Step 4 the optimization of the SW module can be started. The proper working can be validated by measuring t^{SUT} afterwards.

If the values are equal, then the optimization has had the desired effect and the next optimization can be done (Step 6). If not than either the optimization environment or the *memory stub* does not behave correctly and has to be modified (Step 2).

6. After the optimization, determine whether the application will reach the desired performance targets. If the applications has to be optimized further, determine a new CUS and go to Step 1.

In some HW environment the determination of some values of Step 1 is not possible. In this case a different approach can be chosen by simply adding additional memory needs. Instead of replacing the CUS by a SSF the original code will be kept. Only a PSF will be included. With this setup measurements can be achieved, where any additional need of memory can be checked. Whenever t^{SUT} afterwards increases, then the system is at least memory bound. So increasing the memory or optimization of memory needs will be required. However, with that approach no cost benefit analysis or a hidden bottleneck detection can be achieved, which is the case if the before proposed method can be chosen.

6 Case Study

As a test environment, the equipment of the Nokia Siemens Networks ngRNC (next generation Radio Net-

work Controller) has been used to validate the measurements. It hosts a 2.8 GHz Intel Pentium 4 central processing unit with hyperthreading disabled. The operating system is a standard Linux running on a 2.6.22 kernel. The system has separated data and trace (similar to an instruction cache) caches on level one and an unified cache at level two. The caching architecture of the processor is described according to [MARIO7], the values have been derived from [INTE00, INTE06] and verified on the target:

```
MemoryArchitecture =
L1D [ 256, 64, 8, *, L2, 4],
L2U [ 16384, 64, 8, *, Mem, 18];
```

Listing 2: Caching Architecture

In Listing 2 the architecture of the target is described. Each line starts with the name and type of the available caches, e.g. L1D means a level one data cache and L2U describes the unified level two cache. Each level of the cache hierarchy is described using the following items:

1. The amount of different available cache lines (*BLOCKS*).
2. Cache line size (*CACHELINE*).
3. Associativity of the cache (*ASSOC*).
4. Bandwidth between the current and the lower level on a miss (bytes/cycles). This value has not been used (*).
5. The next level in the hierarchy.
6. The cycles needed to access this caching level. The times are described for integer operations.

The values described in this list match the constants as defined in Listing 1. All evaluations of this section have been done using either the configuration for the level one cache (see line “L1D” in Listing 2) or level two cache (see line “L2U” in Listing 2). Only the additional required parameters as described in section 4.1 will be described in the following section.

To determine the necessary values as described in Section 5 the hardware counters of the CPU have been used and accessed through OProfile⁶. Some values could not have been derived by OProfile. Callgrind⁷ measurements as well as binary analysis were applied in this case.

We ran the tests with executables generated by the GCC of version 4.2.1. In order to avoid unwanted optimizations by the compiler, optimization flags were not used for compiling the stub.

⁶OProfile - A System Profiler; <http://oprofile.sourceforge.net/news/>

⁷Callgrind is part of Valgrind. See also: <http://valgrind.org/info/tools.html>

Original Function The application, which has been used for the case study, is based on matrix multiplication as described in [SKIE97]. Three two dimensional arrays have been used for storing two input matrixes and one output matrix. The size of a row as well as for a column was set to a value of 1024. So each matrix has a size of one mega byte which is the size of the second level cache on the used testing system. The calculation inside of the algorithm has not been optimized, i.e. no array transformations has been done. The determined values for different characteristic parameters of the original CUS are given in Table 1.

t^{CUS}	28.28s
$L1^{CUS}_{readhit}$	1147703472
$L1^{CUS}_{writehit}$	1073741824
$L1^{CUS}_{readmiss}$	2073486000 ⁸
$L1^{CUS}_{writemiss}$	0
$L2^{CUS}_{readmiss}$	36000 ⁹
$L2^{CUS}_{writemiss}$	0

Table 1: Measured Memory Performance Behavior of the Original Function

As the used application includes only elements that will be stubbed the times t^{SUT} and t^{CUS} are equal. Therefore, the t^{SUT} time will not be mentioned separately. The Intel Pentium 4 CPU does not provide a write miss counter for the level one cache. Hence, these events cannot be measured with OProfile. An analysis of the binary of the CUS has shown that the used algorithm performs a read access to all data just before a write access to these data happens. As read accesses bring data to the first level cache all write accesses result in first level cache hits. Neither $L1^{CUS}_{writemiss}$ nor $L2^{CUS}_{writemiss}$ happens in the system. Callgrind has been used to determine the level one read and write hits as the architecture does not provide appropriate hardware counters. The remaining values, $L1^{CUS}_{readmiss}$ and $L2^{CUS}_{readmiss}$, of Table 1 has been measured with OProfile. As it is a sampling based profiling application, the measurements has been done ten times in order to evaluate the statistical behavior of the application using the squared coefficient of variation (SCV).

Dynamic Performance Stub In order to create the stub, the algorithm of the approach has to be used multiple times to create the different cache events. To distinguish between the parameters a suffix has been used, e.g. *ITERATIONS.L1RH* denotes the *ITERATIONS* constant for simulating level one read hits (“L1RH”).

According to the values of Table 1, the parameters for the stub have been adjusted. As the matrix multipli-

⁸The squared coefficient of variation is 0.00000353.

⁹The squared coefficient of variation is 0.00617284

cation uses the three two-dimensional arrays with one mega byte each the complete cache for level one and two will be used. Therefore, the stub will also be set to use the complete cache by setting its *SETS* value to the maximum value of the cache level.

The different parameters are determined as follows:

L1 Hit Access To create level one cache events, a single cache line, as described, will be used. The amount of iterations can be determined by dividing the number of events by the amount of bytes of the level cache line size. The determined value has to be rounded. The parameters from Line “L1D” of Listing 2 has been used to determine the constants *BLOCKS*, *CACHELINE* and *ASSOC*.

The following settings have been used:

- $ARRAYSIZE_L1H = CACHELINE_L1$
- $SETS_L1 = 32$
- L1 read hit: $ITERATIONS_L1RH = 17932867$
- L1 write hit: $ITERATIONS_L1WH = 16777216$

L1 Read Miss / L2 Read Hit Access The parameters from Line “L1D” of Listing 2 has been used to determine the constants *BLOCKS*, *CACHELINE* and *ASSOC*. ADD_ASSOC_L1M is set to the minimum value in order to use as few memory as possible for generation of the cache misses. As the used data size is bigger than the first level cache and every location is accessed at least once, as the number of *SETS* was set to its maximum. Measurements have shown that, by using the *dedicated memory approach* to create level two misses, only one miss out of four accesses happens in the level two cache. The remaining three accesses will be hits in level two. This is because of the out-of-order execution possibility and the L2 hardware prefetcher. It has to be taken into account by determining the amount of iterations for creating level one miss accesses, which are the same as level two hits. The following equation can be used to determine the level two cache read hits, which have additionally to be simulated.

$$ITERATIONS_L1RM = \frac{L1^{CUS}_{readmiss} - 3 * L2^{CUS}_{readmiss}}{USED_ASSOC_L1RM \times SETS_L1}$$

The total amount of level two read hits created by the approach for creating level two read misses has to be subtracted from the total amount of level two read hits. The total number of iterations can be calculated by dividing the result by the number of events created in the inner loops (Listing 1 Line 18 and 19). The result for the iterations has to be rounded to the appropriate value.

The different parameters are determined as follows:

- $ADD_ASSOC_L1RM = 1$
- $SETS_L1 = 32$
- $ITERATIONS_L1RM = 7199230$

L2 Read Miss Access The parameters from Line “L2U” of Listing 2 has been used to determine the constants *BLOCKS*, *CACHELINE* and *ASSOC*. The number of accessed cache lines ($SETS_L2RM$) was set to its maximum as explained above. The value of ADD_ASSOC_L2 was chosen in a way so that the algorithm given in Section 4.1 generates the maximum percentage of second level cache misses per access but also the fewest possible amount of memory is used by the stub.

As more cache events are created inside of the inner loops (Listing 1 Lines 18 and 19) the value for $ITERATIONS_L2$ was set to one as the number of generated events are sufficient. The success rate of generation of ongoing second level read misses is only 0.25% due to the capability of the Pentium 4 processor to continue execution while up to 4 read misses are outstanding at the same time (out-of-order execution) [HINT01].

The different parameters are determined as follows:

- $ADD_ASSOC_L2RM = 56$
- $SETS_L2RM = 2048$
- $ITERATIONS_L2RM = 1$

L1 and L2 Write Miss These events will not be simulated as these events do not occur in the original function.

Timing Behavior The timing behavior of the stub has been adjusted according to [TRAP08]. The time measured in the stub is 15.51 seconds. Therefore, the remaining 12.77 seconds ($t^{CUS'} = 12.77s$) has been simulated. The values are reasonably close as the original function is memory bound but also has to do some CPU intensive work.

Validation The stub has been measured in the same way as the original function. The results are provided in Table 2.

The values have been measured in the same way as in the original function (see Table 1). There are two cache events which are slightly different: level one read hit and level one read miss. The differences of these values are rather small, therefore it does not limit the approach. The values of the stub can be improved by

¹⁰The squared coefficient of variation SCV is 0.00003809.

¹¹The squared coefficient of variation SCV is 0.01234568.

t^{STUB}	28.28s
$L1^{STUB}_{readhit}$	1147702488
$L1^{STUB}_{writehit}$	1073741824
$L1^{STUB}_{readmiss}$	2072232000 ¹⁰
$L1^{STUB}_{writemiss}$	0
$L2^{STUB}_{readmiss}$	36000 ¹¹
$L2^{STUB}_{writemiss}$	0

Table 2: Measured Memory Performance Behavior of the *Memory Stub*

changing the number of iterations in the inner loop (Listing 1 Line 19) in the last iteration. This can be easily realized but is denoted as future work.

As can be seen from the comparison of the measured values for the original function (Table 1) and the memory stub (Table 2) the characteristic parameters of both are quite similar. Small discrepancies that exist can be easily improved.

7 Conclusion and Future Work

It has been proven that our *dedicated memory access* approach can be used for simulating the memory access behavior of applications in a deterministic way. Therefore, the approach can be used for *memory stubs* as a special subset of *dynamic performance stubs*. Especially, it simulates cache read and write hits and misses throughout all caching levels. Unfortunately, there are some issues with the out-of-order execution. This behavior will be studied further but does not limit the approach.

The following items will be evaluated in the future:

1. Improve *dedicated memory access* approach to reduce out-of-order effects.
2. Provide a methodology for calculating the suggested amount of cache references depending on the array size, the associativity and the caching level.
3. Simulate the memory behavior of applications, e.g. heap and stack usage.

This paper evaluates the data caching behavior for *memory stubs*. The algorithm and a case study as well as steps towards a methodology is provided in this paper. Further approaches to simulate the memory access behavior, e.g. heap and stack and to simulate instructions- or trace caches remains to be done in the future.

8 Acknowledgment

This research is granted by Nokia Siemens Networks (NSN). The authors would like to thank the LTE group,

especially Oliver Korpilla, Jörg Monschau, Helmut Voggenauer and Jochen Wessel as representatives for the excellent support and contributions to this research project. For careful reading and providing valuable comments on draft versions of this paper we would like to thank Ray Williams.

We would also like to thank the Software Technology Research Laboratory (STRL) from the De Montfort University, especially Francois Siewe and Hussein Zedan, for providing the appropriate environment for research.

References

- [AL-Z04] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*. ACM Press, 2004.
- [BERT05] A. Bertolino and E. Marchetti. *Software Engineering: The Development Process - A Brief Essay on Software Testing*, volume 1, chapter 7, pages 393–411. John Wiley & Sons, Inc., 3 edition, 2005.
- [BEYL01] K. Beyls and E. H. D'Hollander. Reuse distance as a metric for cache behavior. In *In Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*, pages 617–662, 2001.
- [DREP07] U. Drepper. What Every Programmer Should Know About Memory. Technical report, Red Hat, Inc., November 2007.
- [FERD99] C. Ferdinand. Efficient and Precise Cache Behavior Prediction for Real-Time Systems. *Real-time systems*, pages 131–181, 1999.
- [FORT03] P. J. Fortier and H. E. Michel. *Computer Systems Performance Evaluation and Prediction*, volume 1. Digital Press, Burlington, 2003.
- [GENU04] P. Genua. A Cache Primer. Technical report, Freescale Semiconductor, Inc., Austin, TX, USA, 2004. AN2663, Rev.1.
- [GUNT98] N. H. Gunther. *The practical performance analyst*. McGraw-Hill Education, 1998.
- [HILL89] M. D. Hill and A. J. Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, 12(38):1612–1630, 1989.

- [HINT01] G. Hinton, D. Sager, M. U. D. Boggs, D. Carmean, A. Kyker, and P. Rousel. The Microarchitecture of the Pentium®4 Processor. Technical report, Intel Corporation, February 2001. online: <http://www.intel.com/technology/itj/archive/2001.htm>.
- [INTE00] Intel Corporation. A Detailed Look Inside the Intel®NetBurst™Micro-Architecture of the Intel Pentium®4 Processor, November 2000. online: http://people.virginia.edu/~z14j/CS854/netburst_detail.pdf.
- [INTE03] Intel Corporation. *IA-32 Intel® Architecture Software Developer's Manual - System Programming Guide*, June 2003.
- [INTE06] Intel Corporation. *IA-32 Intel® Architecture Software Developer's Manual - System Programming Guide, Part I*, June 2006.
- [INTE08] Intel Corporation. *Intel®64 and IA-32 Architectures Software Developer's Manuals Volume 3A*, November 2008. online: <http://developer.intel.com/products/processor/manuals/index.htm>.
- [JACO07] B. Jacob, S. W. Ng, and D. T. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufman Publ Inc, 2007.
- [JAIN91] R. Jain. *The art of computer systems performance analysis*. Wiley and sons, Inc., 1991.
- [LEBE95] A. R. Lebeck. *Tools and Techniques for Memory System Design and Analysis*. PhD thesis, University of Wisconsin - Madison, 1995.
- [LEE99] M. E. Lee. Optimization of Computer Programs in C. online: http://www.prism.uvsq.fr/~cedb/local_copies/lee.html, 1999. April 25, 2007.
- [LILJ00] D. J. Lilja. *Measuring Computer Performance: a practioner's guide*. Cambridge University Press, New York, 2000.
- [MANE02] S. Manegold. *Understanding, Modeling, and Improving Main-Memory Database Performance*. Ph.d. thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, December 2002.
- [MANE09] S. Manegold and P. Boncz. Cache-Memory and TLB Calibration Tool. online: <http://homepages.cwi.nl/~manegold/Calibrator>, 2009. [February 11, 2009].
- [MARI07] G. Marin. *Application Insight Through Performance Modeling*. PhD thesis, Rice University, 2007.
- [PYO97] Changwoo Pyo, Kyung-Woo Lee, Hye-Kyung Han, and Gyungho Lee. Reference distance as a metric for data locality. In *HPC-ASIA '97: Proceedings of the High-Performance Computing on the Information Superhighway, HPC-Asia '97*, page 151, Washington, DC, USA, 1997. IEEE Computer Society.
- [REIN06] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Predictability of Cache Replacement Policies. Technical report, Universität des Saarlandes, Saarbrücken, September 2006.
- [RIVA98] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 38–49, New York, NY, USA, 1998. ACM.
- [SEAR00] C. B. Sears. The elements of cache programming style. In *ALS'00: Proceedings of the 4th annual Linux Showcase & Conference*, pages 18–18, Berkeley, CA, USA, 2000. USENIX Association.
- [SKIE97] S. S. Skiena. *The Algorithm Design Manual*. Springer, Berlin, November 1997.
- [SOMM01] I. Sommerville. *Software Engineering*. Addison-Wesley, 6 edition, 2001. german redaction.
- [TRAP07] P. Trapp and C. Facchi. Performance Improvement Using Dynamic Performance Stubs. Technical Report 14, Fachhochschule Ingolstadt, August 2007.
- [TRAP08] P. Trapp and C. Facchi. How to Handle CPU Bound Systems: A Specialization of Dynamic Performance Stubs to CPU Stubs. In *CMG 08: International Conference Proceedings*, pages 343 – 353, 2008.
- [vdPA02] R. van der Pas. Memory Hierarchy in Cache-Based Systems. Technical report, Sun Microsystems, Inc., November 2002.
- [WU97] PEI-CHI WU and KUO-CHAN HUANG. Case Studies on Cache Performance and Optimization of Programs with Unit. In *Software - Practise and Experience*, pages 167–172. John Wiley & Sons, Ltd., 1997.