# Modeling Cache Locality with Extra-P

**Modellieren von Cache-Lokalität mit Extra-P**
Bachelor-Thesis von Alexander Graf aus Mainz
Tag der Einreichung:

1. Gutachten: Prof. Dr. Felix Wolf
2. Gutachten: Alexandru Calotoiu

TECHNISCHE UNIVERSITÄT DARMSTADT

Department of Computer Science
Laboratory for
Parallel Programming

Modeling Cache Locality with Extra-P
Modellieren von Cache-Lokalität mit Extra-P

Vorgelegte Bachelor-Thesis von Alexander Graf aus Mainz

1. Gutachten: Prof. Dr. Felix Wolf
2. Gutachten: Alexandru Calotoiu

Tag der Einreichung:

Please cite this document as follows:
URN: urn:nbn:de:tuda-tuprints-65623
URL: http://tuprints.ulb.tu-darmstadt.de/6562

This work is licensed under a Creative Commons
Attribution – NonCommercial – NoDerivatives 4.0
International License.
https://creativecommons.org/licenses/by-nc-nd/4.0/

# Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 15. August 2017

______________________________

(Alexander Graf)

# Abstract

HPC applications usually run at a low fraction of the computer's peak performance. Empirical performance modeling is a helpful tool for automatically assessing the scaling behavior of applications, thereby finding bottlenecks and facilitating the process of improving an application's performance. Current tools for performance modeling neglect the cache behavior of applications, although it plays a significant role for overall performance due to the increasing gap between memory and processor speed. In this thesis, by creating an interface between ThreadSpotter, an open source memory sampler, and Extra-P, a tool for performance modeling, we present and evaluate a methodology to model how scaling affects an application's memory access locality, allowing the developer to easily find scalability bugs that impact the application's cache utilization and so helping to improve computing performance. For each instruction group, our performance models describe how parameters such as processor count or problem size influence the distribution of reuse distances and stack distances. Our novel toolset is evaluated on the HPC applications Kripke, LULESH, MILC, OpenFOAM and Relearn.

# Contents

**1 Introduction** 7

**2 Related Work** 9

**3 State of the Art** 11
3.1 Extra-P . . . . . . . . 11
3.2 ThreadSpotter . . . . . . . . 12

**4 Measuring Locality** 13
4.1 Memory Architecture . . . . . . . . 13
4.2 Locality Quantification . . . . . . . . 14
4.3 Example: Matrix-Matrix Multiplication . . . . . . . . 15

**5 Modeling Locality** 23
5.1 Data Generation . . . . . . . . 23
5.2 Statistics . . . . . . . . 24
5.3 Model Generation . . . . . . . . 25
5.4 Example: Matrix-Matrix Multiplication . . . . . . . . 26

**6 Case Studies** 29
6.1 Kripke . . . . . . . . 29
6.2 LULESH . . . . . . . . 30
6.3 MILC . . . . . . . . 30
6.4 OpenFOAM . . . . . . . . 32
6.5 Relearn . . . . . . . . 33

**7 Outlook** 35

**8 Conclusion** 37

**9 Bibliography** 39

# 1 Introduction

Instead of spending more energy on augmenting the per-core performance of processors, our rising demand for computing power is increasingly satisfied by incrementing the number of cores per processor, and in large-scale computers, by installing more processors per computer. Consequently, to unravel the performance of modern computers, we need to learn how to divide work efficiently.

A helpful tool for developing parallel applications is the generation of performance models. Performance models describe as a mathematical term how performance metrics change with variations of parameters such as the problem size or the processor count, thus aiding the developer to assess how the program scales. Typically modeled metrics are the number of floating point operations, the allocated memory, the amount of communication between processes or simply the execution time. These models can be generated at a fine granularity, such as for every method, or for every loop within a method, allowing to determine which parts of a program may act as a bottleneck when the application scales.

Extra-P is a tool developed by the Laboratory for Parallel Programming at TU Darmstadt to automatically generate performance models. It takes a set of performance profiles as input, each of them representing measurements for different parameters, and determines the best fitting model among a set of predefined functions using a statistical regression technique. Generating performance models in an automated fashion with a statistical approach makes it feasible to mass-generate models precisely extrapolating the scaling behavior of applications.

The access latencies of a computer's main memory evolved considerably slower than the performance of processors. To bridge the growing gap between processor performance and memory performance, modern processors employ caches, which are fast but small and expensive memories integrated into the die. Rather than fetching requested memory portions directly from the slow main memory, requests go through the caches, significantly reducing the access latency if the requested memory portion has already been present in the cache. Cache design exploits the principle of locality: Programs tend to access memory in a local fashion – both temporally and spatially – meaning that once a memory portion is accessed, it is likely that it is accessed again soon, and it is likely that nearby memory portions are accessed soon.

The widening gap between memory and processor speed makes it increasingly important to also model the cache utilization of applications, since it becomes a major factor for computing performance. Nevertheless, current tools for performance modeling do not take into account the effects of scaling on an application's memory access pattern.

We extend Extra-P by the capability of modeling metrics describing how scaling affects the utilization of cache, allowing developers to find cache-related scalability bugs. We define metrics describing an application's memory access behavior and extend ThreadSpotter, an open source tool that samples and analyzes memory access patterns of applications, to export these metrics after sampling an application. We define and implement an interface to enable Extra-P to predict how parameters such as process count and problem size influence the memory access behavior. To measure the locality, and thus assess the likelihood of cache misses, we use the *reuse distance* and *stack distance*, which already earned plenty of attention for studying cache behavior.

In this thesis we present a novel method for modeling the impact of scaling on the cache utilization of applications. More specifically, we make the following contributions:

- We propose modeling distributions of the *reuse distance* and *stack distance* for single addresses and cache lines, to determine how cache locality behaves when the application scales.
- We extend ThreadSpotter to export the distributions of the reuse distance and stack distance at each *instruction group*, i.e. for each set of instructions that access the same data. We create a software

that processes these values for usage with Extra-P to generate models describing the evolution of these distributions for varying configuration parameters.

- With our here-presented toolset, we analyze the locality-related scaling behavior of five MPI-parallelized applications which are considered representative high performance computer (HPC) applications, namely Kripke, LULESH, MILC, OpenFOAM and Relearn. Thereby we validate our approach and demonstrate the applicability and usefulness of both our methodology and our software toolset.

We structure this document as follows: In Chapter 2, we summarize the current state of related research and briefly describe how it differs to our approach. In Chapter 3, we present the open source tools Extra-P and ThreadSpotter, on which this work builds upon. Chapter 4 defines the metrics *reuse distance* and *stack distance* and explains how they are useful to assess an application's cache utilization. With a matrix multiplication program as a simple example, we discuss the relation of these metrics to the cache utilization and examine the locality's actual impact on performance. In Chapter 5, we explain how we process the values of these metrics to generate performance models that describe the change of their distribution when the application scales. Finally, we present case studies in Chapter 6.

# 2 Related Work

We propose a method for automatically quantifying how the memory access locality changes when an application scales to a different problem size or a different processor count. Our analysis is done on the level of instruction groups, i.e. instructions accessing the same data structure. This allows to easily determine in which parts of an application memory access patterns change such that the application's cache utilization is expected to drop when the processor count or the problem size varies. As metrics for modeling the memory access behavior, we choose statistics of samples of reuse distances and stack distances. We gather samples of these metrics with ThreadSpotter [18], modify it such that it outputs required information, process this data with a software we created for this purpose, and export it to Extra-P [8], which creates performance models.

In 1970, Mattson et al. presented evaluation techniques for storage hierarchies, discussing concepts for analytically modeling caches [15]. Having defined the term *stack distance*, the authors proposed a stack-based algorithm to measure the stack distances of memory accesses in an address trace. Since then, much effort has been spent on finding efficient techniques to study cache locality.

Berg and Hagersten presented StatCache [3, 4], an approach to efficiently and accurately analyze data locality. It was later extended to a multiprocessor cache model, analyzing the data locality of a multithreaded application [5]. Based on this work, Eklov and Hagersten developed StatStack [10], introducing a new statistical cache model that models a fully associative cache with an LRU replacement policy using reuse distance samples as input data. This technology was then commercialized as part of the ThreadSpotter tools [11] and later published as free software.

Alternative methods for collecting memory access data are using hardware performance counters, e.g. PAPI [16], or low-level analysis tools that require compile-time instrumentation of applications, such as Byfl [17]. ThreadSpotter samples natively during the application's runtime, produces high-quality data with low noise, supports MPI applications and does not require instrumentation or any other modification of the application at compile-time. It does a sophisticated analysis of the application's structure and calculates plenty of statistics about the memory access pattern. This makes it the ideal data collection software for our research.

Zhong et al. proposed techniques that predict how the locality of a program changes with its input by building parametrized models [22, 9]. They presented training-based whole-program locality analysis, consisting of algorithms for measuring reuse distance and prediction methods for modeling locality at a whole-program granularity of non-parallel programs.

Williams et al. presented the roofline model [21], a performance model graphically representing the attainable upper bound performance of a computer architecture. It shows floating point operations per second by operational intensity, which was defined as number of floating point operations normalized by the amount of data transferred between the processor's cache and the memory. These models show ceilings of expectable performance for different processors and thus allow to predict the performance of a kernel with given operational intensity. It was extended by Ilic et al. to be cache-aware [13], thus further improving the guidelines for application optimization by considering information about the cache architecture of a processor. However, these analysis were focused on modeling different machines rather than different applications and roughly estimating an unparametrized kernel's performance to find its place in the visual model by determining its operational intensity, not taking into account the effects of scaling an application, nor its access locality.

# 3 State of the Art

In this chapter, we briefly present the open source tools Extra-P and ThreadSpotter.

Extra-P is an application for empirical performance modeling, creating performance models showing the influence of parameters such as problem size or program count on metrics describing the performance of an application. ThreadSpotter is a tool analyzing an application's memory accesses in order to give optimization suggestions to the developer.

## 3.1 Extra-P

Extra-P [8] is a tool for automatic generation of empirical performance models. With a set of performance profiles as input, representing runs of an application with different configurations, it outputs a set of human-readable functions, one for each program and metric, describing the evolution of the metrics for varying configuration parameters.

The different configuration parameters usually represent different problem sizes or different processor counts, enabling to assess the scalability of an application. As a rule of thumb, Extra-P needs measurements for at least five different configurations of each considered parameter. Usually, the input performance profiles are provided by the Score-P profiler and are typically measurements of the computation or network communication of an application.

The resulting scaling models are assignments of the *performance model normal form* (PMNF), defined by the formula

$$f(p) = \sum_{k=1}^{n} c_k \cdot p^{i_k} \cdot \log^{j_k}(p).$$

As described in aforementioned paper, even though obviously not being exhaustive, it is assumed to work in most practical scenarios since it is a consequence of how most computer algorithms are designed. Also, neither the sets $I, J \subseteq \mathbb{Q}$, from which the exponents $i_k$ and $j_k$ are chosen, nor the number of terms $n$ have to be arbitrarily large to achieve a good fit. For all assignment options, a statistical approach is used to find coefficients $c_k$ with optimal fit, and the hypothesis with the best fit across all candidates is selected.

Further, the current version supports modeling multiple parameters simultaneously [6] with the performance model normal form being extended to multi-parameter models, allowing to model the effect of problem size and processor count simultaneously. For multiple parameters, the original performance model normal form is canonically expanded to include multiple parameters:

$$f(p_1, \dots, p_m) = \sum_{k=1}^{n} c_k \cdot \prod_{l=1}^{m} p_l^{i_{kl}} \cdot \log^{j_{kl}}(p_l).$$

The expanded performance model normal form allows a number $m$ of parameters to be combined in each of the $n$ terms that are summed up to form the model. Each term allows each parameter $p_l$ to be represented through a combination of monomials and logarithms. The sets $I, J \subseteq \mathbb{Q}$ from which the exponents $i_{kl}$ and $j_{kl}$ are chosen can be defined as in the one-parameter case. If multiple parameters are considered, performance experiments have to be conducted for all combinations of parameter values. Hence, for measuring the influence of both parameter count and problem size, prior rule of thumb expands to measure at least 25 different combinations of processor count and problem size. However, given the model's extrapolating capabilities, both problem sizes and processor counts can be kept small.

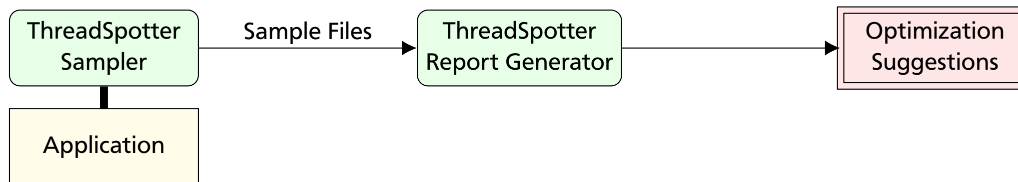



**Figure 3.1:** Illustration of ThreadSpotter's workflow. First, the application is sampled to gather its memory fingerprint, then the report generator is invoked to analyze the memory access patterns and produce a report listing optimization suggestions.

Extra-P does not yet have the ability to create models for an application's cache utilization. Though Score-P has access to hardware performance counters, the cache-related data delivered by these counters cannot be considered suitable for performance modeling.

For our purpose, we develop a method to provide Extra-P with input data characterizing an application's cache locality.

## 3.2 ThreadSpotter

ThreadSpotter [18] is an open source tool that gives developers insight into memory-related performance problems in an application. It analyzes an application and provides advice to the programmer on how to correct memory performance problems, and offers the programmer insight into where and why performance problems occur. We summarize the most important aspects of this tool mentioned in its documentation.

As illustrated in Figure 3.1, to utilize ThreadSpotter, an application is first sampled, capturing samples of its memory accesses and collecting information about its structure. Once the sampling is done, ThreadSpotter analyzes the captured data and emits a report listing the performance problems found in the code and maps them to the source code.

Rather than using hardware performance counters, it uses a sampling technology allowing the collected fingerprint to be richer in information than what can be obtained from the hardware performance counters. Since it is the behavior of the application and not the hardware that is sampled, the gathered data is independent of the hardware the sampling was done on. Also, by looking at the binary code of the application, it is programming language independent.

ThreadSpotter is able to analyze applications parallelized with MPI, creating a sample and report file for each process. This enables analysis of applications whose execution spans multiple processing nodes.

Its analysis specifically focuses on how the application's memory access patterns interact with the processor caches, aiding the developer decrease the cache miss ratio of the application and decrease the memory bandwidth requirement of the application. Both effects can be considered important bottlenecks for scaling application performance on multicore processors.

With its analysis, ThreadSpotter detects data layout problems, such as too large data types, alignment problems, structures that are only partially used or dynamically allocating many small structures. It detects problems regarding the memory access patterns itself, for example inefficiently nested loops, or access patterns that seem random, as it may be caused by using inappropriate techniques for data organization. These issues are reported to the developer along with optimization advices and statistics further guiding the developer in identifying and understanding the causes of memory-related performance problems.

For its own analysis, ThreadSpotter collects metrics describing the memory access patterns of a program. So it is an obvious solution to extract them and use them as input for Extra-P.

# 4 Measuring Locality

To build models describing the impact of parameters such as problem size or processor count on the cache behavior of an application, we define a few metrics characterizing the memory access patterns regarding their locality. In this chapter, after giving a brief introduction to the memory architecture of modern computers, we discuss the *reuse distance* and the *stack distance*.

## 4.1 Memory Architecture

Due to the high clock rates of modern microprocessors, usually within the range of a few gigahertz, along with their instruction-level parallelism, current CPU cores are capable of executing several instructions per nanosecond in typical conditions. In comparison, present-day memory modules have access latencies of about ten nanoseconds. As a consequence, between the time a memory access is issued and its completion, the processor could execute likely hundred instructions not involving memory accesses.

The growing gap between processor performance and memory performance, often referred to as the *memory wall*, led to the introduction of memory hierarchies in processors, featuring usually two or three levels of caches, each faster but smaller and more expensive the closer they are to the core. Rather than fetching requested memory portions directly from the slow main memory, requests go through the caches. If the portion is not found in a cache, it is fetched from the next-level cache, or from the main memory. So, the access latency of a memory access is determined by the latency of the fastest hierarchy level where the requested memory portion is present. A detailed discussion of how caches are organized is found in [12].

The comparatively slow memory latency makes cache utilization a major factor for computing performance. The likelihood of a memory portion being present in a fast cache is not only influenced by the cache architecture itself, but also by the patterns of the application's memory accesses. To retain the most frequently accessed data, thus to efficiently utilize small caches, cache design exploits the principle of locality: Data is assumed to be accessed in a local fashion – both temporally and spatially – meaning that once a memory portion is accessed, it is likely that it is accessed again soon (*temporal locality*), and it is likely that nearby memory portions are accessed soon (*spatial locality*).

Instead of being organized byte-wise or word-wise, caches store equally sized chunks of data, called *cache lines*. This not only lowers the organizational overhead but also takes advantage of the spatial locality, as when one word is requested, it is likely that the neighbored words are needed soon. Most commonly, cache lines have a size of 64 bytes.

The exact cache design, i.e. the number of hierarchy levels and their respective sizes, the grade of associativity, their replacement algorithms as well as further optimizations a processor might implement, differs remarkably between processors. Processors in high-end computers, where the power consumption is merely pressured by optimizing the total cost of operation, usually employ three levels of caches, totaling several megabytes of size and utilizing sophisticated replacement algorithms. In contrary, on mobile phones, where the power consumption is strongly constrained by its influence on battery life and limited cooling possibilities, the size of processed data is usually small and thus a simpler cache design can be used without significant drawbacks.

However, since accessing the data in a local fashion, both temporally and spatially, is generally a major assumption for cache designs, we can assume that the cache works best, the more local the memory accesses are.

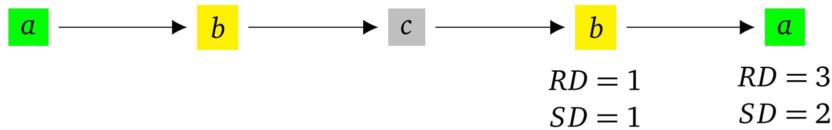



**Figure 4.1:** Example of reuse distances and stack distances in a well-defined access pattern. $a$, $b$, $c$ represent different memory locations. Accesses to these locations happen in the order indicated by the arrows. Reuse distances and stack distances are denoted with $RD$ and $SD$ respectively.

## 4.2 Locality Quantification

To model an application's cache behavior, we need to define a metric that characterizes the cache utilization. On the first glance, it would be obvious to choose the cache miss ratio or the ratio of cache misses of a certain type. These values could be easily obtained using hardware counters [16] and they would clearly show the cache utilization of an application. However, this methodology would have a major drawback: Naturally, these values do not only depend on the application itself, rather they are highly dependent on the cache architecture of the machine where the measurements are run, and in most cases it would be impossible to transform the results from one architecture to another.

By defining metrics describing the application itself rather than the processor architecture it is executed on, we avoid this issue. We propose metrics to describe the locality of an application's memory accesses. Measuring the locality itself allows us to assess the cache utilization of an application, as a better locality implies a better cache utilization.

With the temporal locality being defined as the property that accesses to the same object happen within a short time distance, it becomes evident that the time between two accesses to the same address would be a good measure for the temporal locality. Rather than measuring the actual time difference, we can measure the number of memory accesses that happen in between, as it is not only less noisy, but also machine-independent and meaningful due to resembling the activity on the memory system. Hence, as a metric for temporal locality, we define the *byte reuse distance* as the number of memory accesses between accesses to the same address.

With the spatial locality being defined as the property that once an address is accessed, it is likely that nearby addresses are accessed in the near future, we can measure the time distance between two accesses to the same cache line. Due to cache lines being blocks of memory consisting of multiple continuous addresses whose access can be requested individually, if memory is accessed in a spatially local fashion, memory accesses in the near future would likely be located on the same cache line, leading to yielding mostly low values for this metric. Hence, as a metric for spatial locality, we define the *line reuse distance* as the number of memory accesses between accesses to the same cache line.

Instead of only measuring the number of total memory accesses, we can also measure the number of memory accesses to unique locations, i.e. the number of distinct locations that have been accessed. We call this the *stack distance*, more precisely, we refer to the *byte stack distance* as the number of accessed addresses between accesses to the same address and likewise the *line stack distance* as the number of accessed cache lines between accesses to the same cache line.

Note the subtle but significant difference between the reuse distance and the stack distance: Whereas the reuse distance counts the total number of accesses between accesses to the same location, the stack distance only counts the number of distinct locations. Figure 4.1 shows a simple numerical example to demonstrate the difference of these terms.

To conclude, we define the

**Byte Reuse Distance** Number of accesses between accesses to the same address,

**Byte Stack Distance** Number of accessed addresses between accesses to the same address,

**Line Reuse Distance** Number of accesses between accesses to the same cache line,

**Line Stack Distance** Number of accessed cache lines between accesses to the same cache line.

The stack distance has first been mentioned by Mattson et al. in 1970 [15], where to study a cache architecture, a stack processing technique is used on an address trace, moving each accessed address to the top of the stack. The distance of an address to the top of the stack is then the stack distance, hence its name. Since then, it has earned plenty of attention as a measure to quantify and study locality. ThreadSpotter internally also measures reuse distances and estimates stack distances [2, 10, 11].

By counting how many different cache lines have been touched since the last access to the same line, the line stack distance gives a close relation to the eviction mechanism in a cache with a *least-recently-used* (LRU) replacement policy, where when a new cache line is stored in the cache, the least recently used cache line is replaced. In this case, a memory access leads to a capacity miss if and only if its line stack distance is larger than the capacity. This eviction mechanism is common for the first-level cache of a processor, and a similar variant, sometimes called *pseudo-LRU*, is often used in bigger, upper-level caches. Also, if the processor decides randomly which line to replace whenever a line is newly stored in the cache, as it is common for the upper-level caches in mobile processors [1], the stack line distance values would also indicate the chance of the cache line being present at the time of access. This close relation of the line stack distance not only to the locality itself, but also directly to the probability of a cache miss gives a further legitimation of this metric.

However, for modern processors we are not able to exactly predict the cache miss probability, because we cannot expect strong assumptions on the cache architecture to hold. Naturally, vendors seldomly publish detailed documentation about the internals of their processors. Nevertheless, since we can expect growing memory latencies for worsening locality, measuring the locality allows us to assess the memory access times.

For a fixed cache line size, the values of these metrics evaluate an algorithm, or more precisely an implementation of an algorithm, rather than the memory architecture or other hardware-dependent properties. Further, these values are deterministic, i.e. they yield the same values between different runs of the same program with the same data, given it is deterministic itself. Therefore we can expect the sampling of these metrics to be the only possible source of noise.

Our proposed metrics are meaningful because they well describe the locality of memory accesses, allowing to assess how well the cache is utilized. Further, they are machine-independent and deterministic, making them ideal candidates for use in performance modeling.

## 4.3 Example: Matrix-Matrix Multiplication

To further legitimate our chosen metrics, we discuss a more complex and practical example. For two $N \times N$ matrices $a$ and $b$ we calculate the product $c$, i.e. for $i$ denoting the row and $j$ denoting the column, we calculate the sums

$$c_{ij} = \sum_{k=0}^{N-1} a_{ik} b_{kj}.$$

We propose an implementation in Listing 1, naïvely iterating over $i$ and $j$ and translating above formula into C code. Each sum is calculated in a variable $v$ until it is stored in $c_{ij}$. We can safely assume the variable to be translated into a register, thus accesses to $v$ are not memory accesses. Also, we can safely assume $i$, $j$, $k$ and $N$ to be represented by registers. Matrices are stored row-major, i.e. such that the element in the $i$th row in the $j$th column has the array index $iN + j$.

Figure 4.2 is a plot of the memory access pattern of the implementation presented in Listing 1, showing a point for each memory access. Their position indicates its location on the vertical axis, and its time

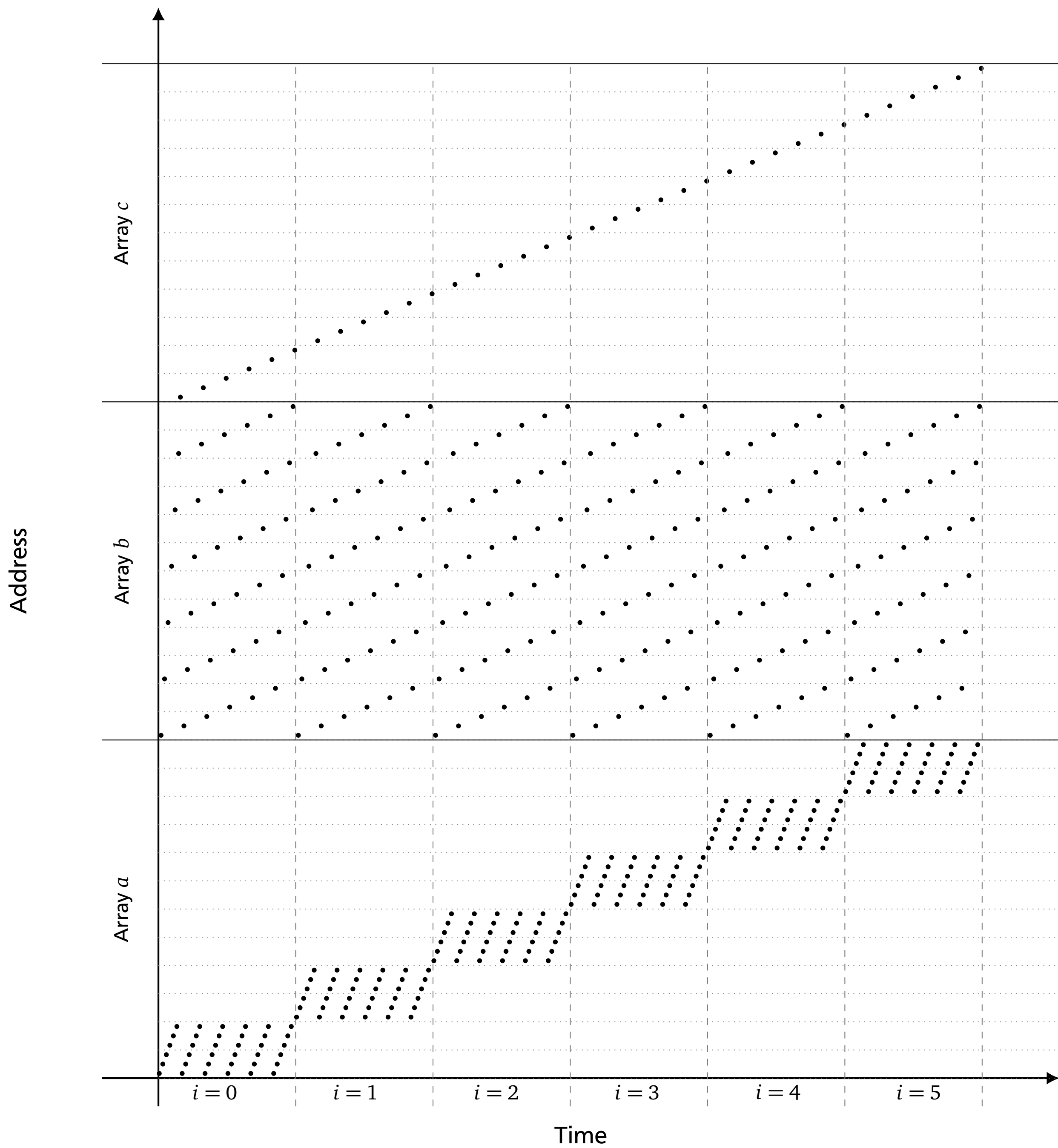



**Figure 4.2:** Access pattern of Listing 1 for multiplying $6 \times 6$ matrices. Points represent memory accesses, their position encode the time and the location. The boundaries of the cache lines with a hypothetical size of $\mathscr{L} = 3$ entries are represented by horizontal dotted lines. Thus, each matrix row occupies two cache lines. Iterations of the $i$-loop are indicated by vertical dashed lines.

**Listing 1:** Naïve matrix-matrix multiplication $c = ab$.

```
void mmm(const float *a, const float *b, float *c, int N) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            float v = 0.0f;
            for (int k = 0; k < N; k++)
                v += a[i * N + k] * b[k * N + j];
            c[i * N + j] = v;
        }
    }
}
```

(i.e. number of previously executed memory accesses) on the horizontal axis. When plotting the memory access addresses over time, the spatial locality roughly equivales the vertical distance of the points and the temporal locality equivales the horizontal distance of the points at the same address.

Accesses to array $a$ happen with a good spatial locality, due to it being accessed in a consecutive fashion inside the $k$-loop. However, the accesses to $b$ are barely local: Consecutive iterations of the inner loop access elements of $b$ with a gap of $N$ entries, even crossing cache line boundaries if $N$ reaches or exceeds the cache line length, for which a matrix size of $16 \times 16$ would suffice. The accesses to $c$ don't have a good spatial locality either, since even though the array is accessed consecutively, many accesses to $a$ and $b$ happen between each access to $c$.

Similarly behaves the temporal locality of this implementation. Whereas between accesses to the same element in $a$ only one row of $a$ and one column of $b$ is iterated, considerably more memory accesses happen between accessing the same element in $b$: Accesses to a certain element of $b$ repeat when the outermost loop iterates, meaning that $N^2$ accesses to each $a$ and $b$ happen in between.

When returning to the innermost loop after an iteration of the $j$-loop, the spatial locality of accesses to $a$ is not optimal either, as after accessing the last item of a row, the first item is accessed again, instead of consecutive accesses. This is inevitable in nested loops whose inner loop consecutively iterates across fixed elements in an array: During the iteration of the inner loop, accesses happen to neighbored elements, but when leaving and returning to the inner loop, the first element is accessed again, which is far from the one accessed in the last step of the previous iteration of the outer loop.

**Table 4.1:** Distribution of reuse distance and stack distance values of implementation in Listing 1. For the sake of simplicity, we assume $N$ to be a multiple of the cache line size $\mathscr{L}$, and we assume $a$, $b$ and $c$ to be aligned on cache line boundaries.

| | **a** | **b** | **c** |
|---|---|---|---|
| **Byte Reuse Distance** | $2N$ | $2N^2 + N - 1$ | - |
| **Byte Stack Distance** | $2N$ | $N^2 + 2N - 1$ | - |
| **Line Reuse Distance** | $\begin{cases} 1 & \text{in } \frac{\mathscr{L}-1}{\mathscr{L}} \text{ of cases,} \\ 2N & \text{else.} \end{cases}$ | $\begin{cases} 2N & \text{in } \frac{\mathscr{L}-1}{\mathscr{L}} \text{ of cases,} \\ 2N^2 + N - 1 & \text{else.} \end{cases}$ | $2N$ |
| **Line Stack Distance** | $\begin{cases} 1 & \text{in } \frac{\mathscr{L}-1}{\mathscr{L}} \text{ of cases,} \\ N + \frac{N}{\mathscr{L}} & \text{else.} \end{cases}$ | $\begin{cases} N + \frac{N}{\mathscr{L}} & \text{in } \frac{\mathscr{L}-1}{\mathscr{L}} \text{ of cases,} \\ N^2 + \frac{2N}{\mathscr{L}} - 1 & \text{else.} \end{cases}$ | $N + \frac{N}{\mathscr{L}}$ |

**Listing 2:** Blocked, locality-optimized matrix-matrix multiplication $c = ab$. $c$ is assumed to be zero-initialized.

```
void mmm(const float *a, const float *b, float *c, int N, int B) {
    for (int jj = 0; jj < N; jj += B) {
        for (int kk = 0; kk < N; kk += B) {
            for (int i = 0; i < N; i++) {
                for (int j = jj; j < min(jj + B, N); j++) {
                    float v = 0.0f;
                    for (int k = kk; k < min(kk + B, N); k++)
                        v += a[i * N + k] * b[k * N + j];
                    c[i * N + j] += v;
                }
            }
        }
    }
}
```

Table 4.1 shows the reuse and stack distances, and how they depend on the problem size parameter $N$. With some of the distances involving quadratic polynomials of $N$, our chosen metrics clearly show that the locality strongly worsens when incrementing the problem size. Thus, we would expect a bad cache hit rate and a bad performance at large matrix sizes and a worsening performance for incrementing matrix sizes.

The bad locality of the implementation in Listing 1 can be fixed. We present a locality-optimized algorithm for multiplying matrices, quickly discuss its access pattern and compare the run times of these implementations on real computers to see the impact locality has on the computing performance.

The main cause of the bad locality of the naïve implementation is iterating $b$ column-wise although it is stored row-wise. However, we cannot simply interchange the loops, because $a$ is iterated row-wise and is iterated in the same loop body as $b$.

A technique for improving locality is *blocking*: Instead of iterating over entire rows and columns of the matrix, we partition the matrix into submatrices of size $B \times B$ and calculate the product of these submatrices, as shown in Listing 2. The difference to the unblocked variant is that the loops are surrounded with two further loops defining the iteration boundaries of the $j$ and $k$ loops.

Figure 4.3 shows the memory access pattern of this implementation for a small matrix. To allow the presentation of the pattern, we once again chose a hypothetically small cache line size of $\mathscr{L} = 3$. Note that in reality, cache line sizes of 32 or 64 bytes, i.e. $\mathscr{L} = 8$ or $\mathscr{L} = 16$ are common, and the small cache line size in the figure is used for illustration purposes only, even exaggerating the effect of adding memory accesses to $c$.

Even tough the array $c$ is accessed more frequently, we see the accesses to $b$ now happen in a significantly local fashion, both spatially and temporally. Since requested data is more likely to be available in the cache, we expect this implementation to be faster than the one we presented before.

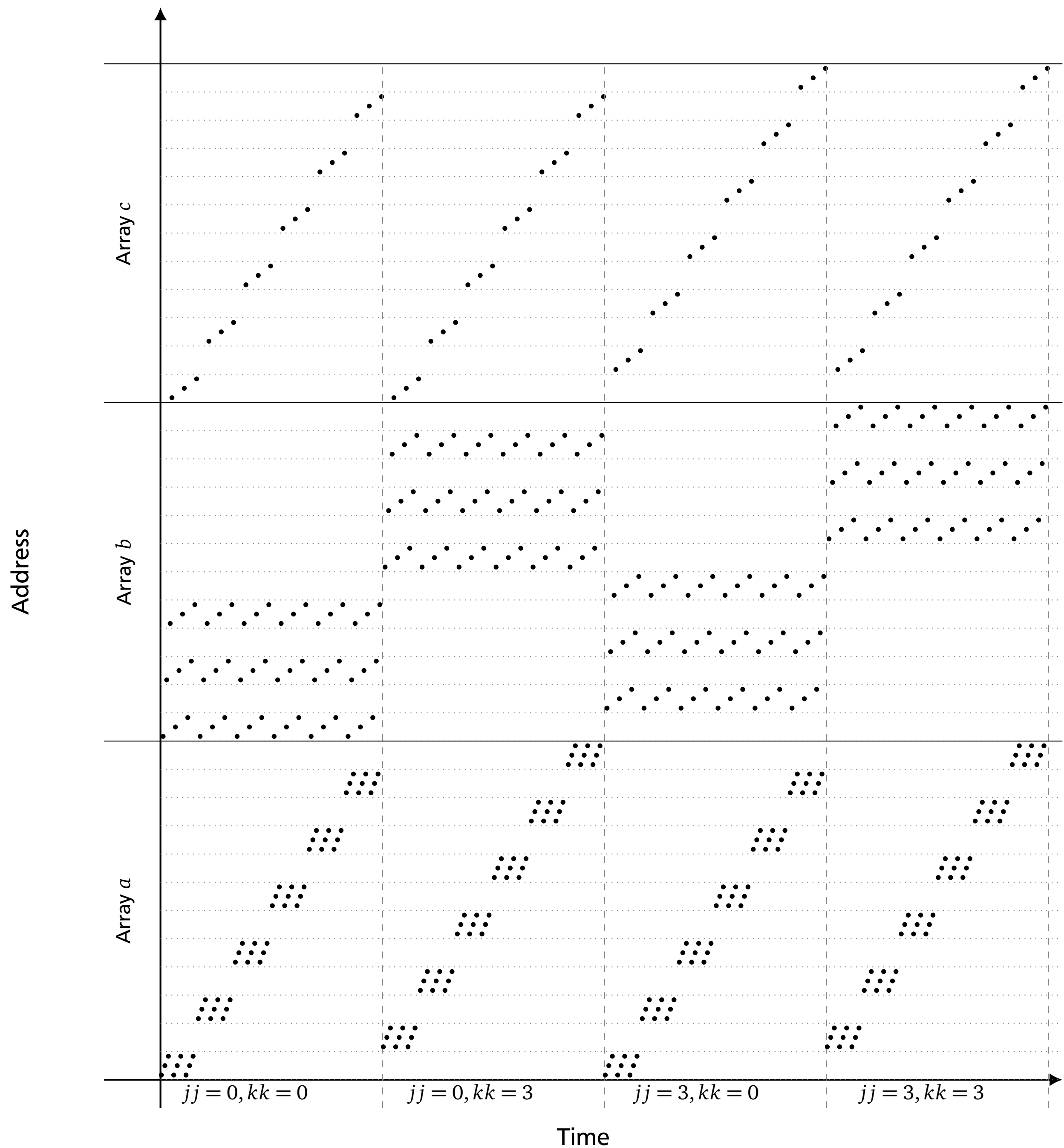



**Figure 4.3:** Access pattern of blocked implementation (Listing 2) when multiplying $6 \times 6$ matrices blocked into $3 \times 3$ submatrices. Like in Figure 4.2, the boundaries of the cache lines with a hypothetical size of $\mathscr{L} = 3$ entries are represented by horizontal dotted lines. Thus, each matrix row occupies two cache lines. Iterations of the $kk$-loop are indicated by vertical dashed lines. The accesses to $c$ are plotted as one access even tough they load, increment and store each.

Figure 4.4 shows the difference of performance measured as floating point operations per second between the naïve, unblocked implementation of Listing 1 and the implementation with optimized locality of Listing 2. We ran the matrix-matrix multiplication several times with an increasing matrix size, leading to increasing memory requirements and worsening locality on the naïve implementation. A block size of $16 \times 16$ has been chosen, causing one row of a matrix to exactly fit into a cache line of the processor.

To show that memory locality heavily impacts performance regardless of the exact cache architecture, we conducted these measurements on mobile processors with a peak per-core power consumption of less than 4 watt, such as the Samsung Exynos 5422 and the Intel Core m3-6Y30, and more complex processors of the AMD Opteron or Intel Xeon series, typically employed in servers or compute clusters where they process large data sets.

The Samsung Exynos 5422 is a representative of state-of-the-art mobile processors, designed for ultra-low power consumption and small die size, while the Intel Xeon E5-2680 v3, itself employed in one of the compute clusters we use for our research, is a representative of the Haswell architecture, which is installed in more than 40 % of the Top 500 Supercomputer Sites as of June 2017 [20]. As one might expect due to their different field of application, there are differences in their cache architecture: Whereas the Samsung Exynos 5422 processor, of which we tested one of its ARM Cortex-A15 cores, has a 1 MB second-level cache with a random replacement policy [1], the Intel Xeon E5-2680 v3 has a 256 kB per-core second-level cache as well as a 30 MB third-level cache. However, all tested cores have a 32 kB first-level data cache with a least-recently-used replacement policy and a cache line size of 64 bytes.

We see how reality matches our expectations, in that whereas the exact impact of the difference in locality differs from machine to machine, the performance significantly drops the worse the locality is, regardless of the exact architecture – whether on a mobile phone or on a huge supercomputer.

We presented two algorithms solving the same problem, only differing in the way the memory is accessed, and showed that they hugely differ in performance. Neither the compiler, nor the sophisticated algorithms in a supercomputer CPU effectively mitigated this issue – instead, it was a slight code modification that led to a performance gain of at least factor three for big matrix sizes, allowing us to call it a bug, or more precisely, a *scalability bug*.

When we measure the locality of the involved data accesses and model it using a statistical regression technique, we would see that the locality of the naïve implementation strongly varies with the problem size. Thus, we are presenting a novel methodology automatically finding this type of scalability bug, aiding the software engineer improve efficiency of the code.

In this chapter, we have given a brief overview of the memory architecture in modern computer systems and how it affects performance, proposed to measure and model the locality in order to predict in a machine-oblivious way how well an application utilizes the cache and showed an example to demonstrate the actual performance impact of bad locality. Whereas the matrix-matrix multiplication is a well-chosen example due to its low complexity, it is undoubtedly very simple compared to the problems a supercomputer typically solves. After presenting in Chapter 5 how we implemented our methodology, we show case studies applying our toolset on real-world applications in Chapter 6.

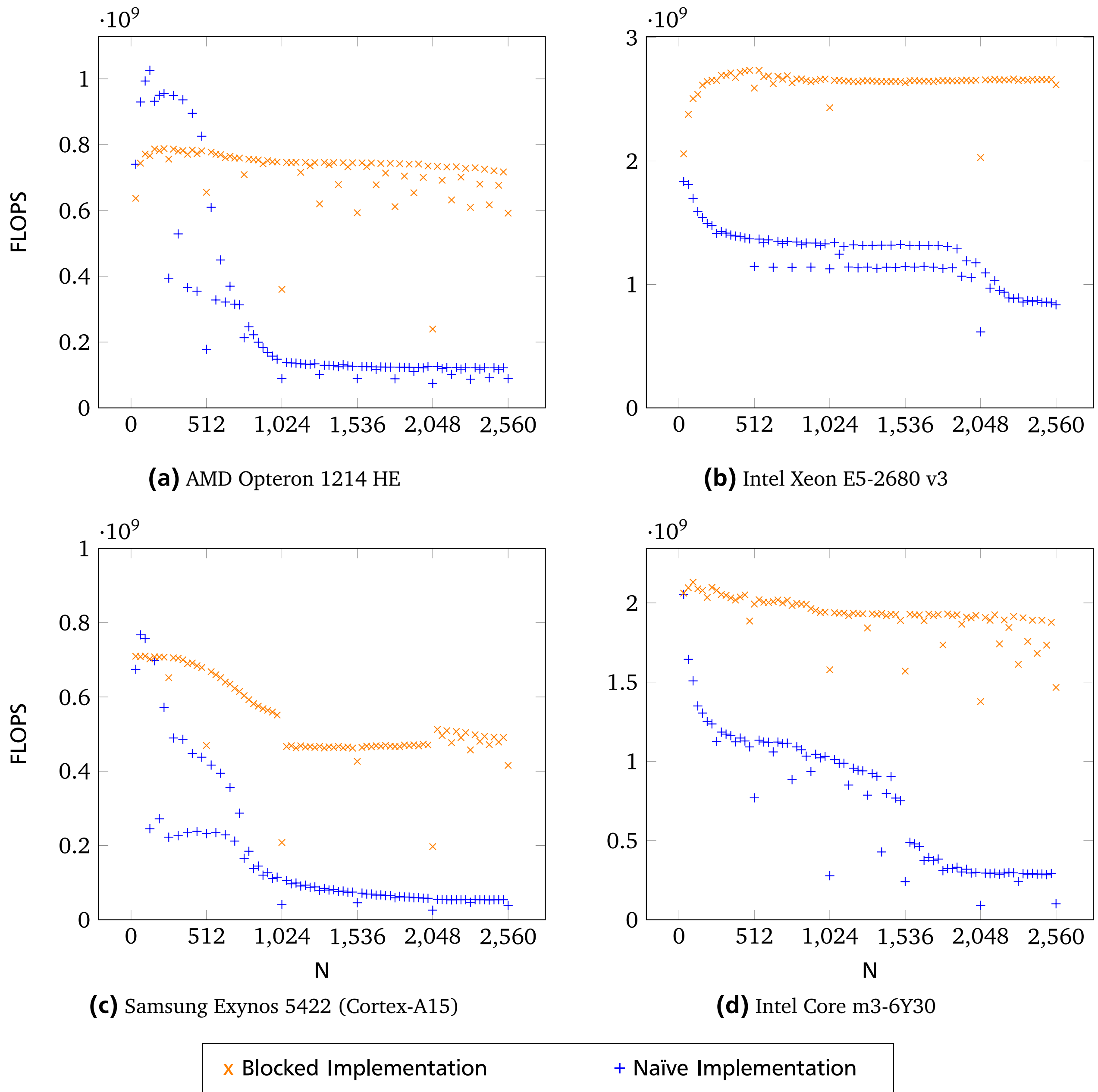



**Figure 4.4:** Performance comparison of naïve (Listing 1) and blocked (Listing 2) matrix-matrix multiplication on different processors, block size of $16 \times 16$, compiled with GCC 4.9, -O2. Regardless of the processor, performance drops significantly for worsening locality.

# 5 Modeling Locality

Having defined a metric describing how local the memory is accessed, we can measure the memory accesses of an application for different configuration parameters, i.e. different problem sizes or different processor counts. Then we can create models describing as a simple, human-understandable mathematical term how the locality is influenced by the respective parameters.

It comes handy that ThreadSpotter already collects our four metrics at the granularity of *instruction groups*, which are sets of instructions within a loop that access the same data. For each measured configuration, we collect samples of reuse distances and stack distances for each instruction group, calculate statistics of these samples well describing the most meaningful properties of their distributions, and then use Extra-P to create models of these statistics by running a regression technique. Before running our approach on complex real-world applications, we validate our method on the matrix-matrix multiplication discussed in the previous chapter.

## 5.1 Data Generation

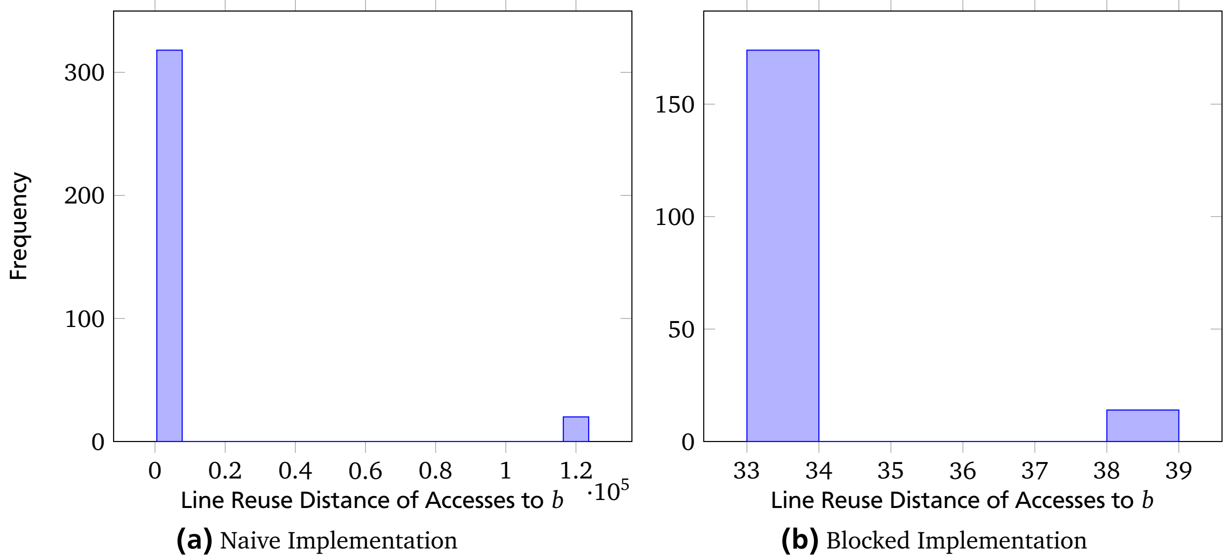
Normally, i.e. when ThreadSpotter is utilized for its original purpose, ThreadSpotter is used as follows: As a first step, ThreadSpotter is invoked to sample the memory accesses of an application, recording the accesses to a sample file, or to a sample file for each involved process when running on a parallel application. Then ThreadSpotter's report generator is invoked with a sample file as input. It analyzes the program structure, identifying loops and *instruction groups*, collects information about these, and then emits a report which is used to create a document containing optimization advice.

An instruction group is defined as a set of instructions within a loop accessing the same data structure. In our matrix-matrix multiplication example, whose C code is shown in Listings 1 and 2, ThreadSpotter would identify three instruction groups, each consisting of the instruction accessing $a$, $b$ and $c$ respectively. Instruction groups do not cross loops: If the same data structure is accessed by instructions in separate loops, ThreadSpotter puts these instructions in separate groups.

For each instruction group, ThreadSpotter internally determines statistics such as the median or certain quantiles of the byte and line reuse distances and analyzes these statistics in order to find optimization opportunities. Additionally, ThreadSpotter estimates distributions of line stack distances to approximate the probability of whether a capacity cache miss would occur on an LRU cache. The employed method of estimating stack distances shows a high accuracy and allows sampling efficiently [10].

We modified ThreadSpotter's report generator such that for each instruction group it outputs the raw values of the byte and line reuse distances of the sampled memory accesses, as well as the byte and line stack distances.

Figure 5.1 shows histograms of the reuse distances of the accesses to $b$ of our matrix-matrix multiplication (Listing 1 and Listing 2), clearly demonstrating the quality of ThreadSpotter's data generation and how precise the measured data matches our expectations. When multiplying a $256 \times 256$ matrix, the naïve implementation of Listing 1 shows line reuse distances matching our derivations in Table 4.1, where the majority of values is 512 and about $\frac{1}{16}$ of the samples have value $123,632$, whereas the blocked, locality-optimized variant of Listing 2 shows significantly lower line reuse distance values of 33 and 39.




**(a)** Naive Implementation

**(b)** Blocked Implementation

**Figure 5.1:** Histogram of line reuse distance of accesses to $b$ of naïve (Listing 1) and locality-optimized (Listing 2) implementation of matrix-matrix multiplication when multiplying a $256 \times 256$ matrix.

## 5.2 Statistics

Being able to access raw values of reuse distances and stack distances for each instruction group, we can repeat the sampling process for different configuration parameters, such as different problem sizes or different tuples of problem size and processor count, create statistics of the reuse distances and stack distances for each of these measurements, and feed this data into Extra-P, which then runs a statistical regression to output human-readable models showing how the locality behaves when the parameters vary.

Both the spatial locality and the temporal locality are likely to have a high variance. This becomes obvious when considering a loop that is executed multiple times during the program's runtime. In the loop itself the values for stack distances and reuse distances are low if it has good locality. However, many memory accesses can happen between different executions of the loop, leading to considerably higher values when returning to the loop [7]. Also, in a loop consecutively iterating over an array spanning multiple cache lines, where each cache line contains $\mathscr{L}$ elements, each cache line would be accessed $\mathscr{L}$ times until the next cache line is accessed, yielding $\mathscr{L} - 1$ zero-valued line reuse distances and line stack distances. But each time a cache line is accessed first by a particular execution of this loop, the line reuse/stack distance values can be much higher. This effect can be seen in our matrix-matrix multiplication example and explains the case differentiation in Section 4.3, as well as the two peaks in the histograms in Figure 5.1.

As these cases of higher reuse/stack distance values would influence the application's cache behavior, we should also model their farness from the typical values, rather than considering them outliers. Consequently, as statistics to model, we choose the median and 95 % interquantile range, i.e. the difference between the 97.5 % percentile and the 2.5 % percentile. Whereas the median shows the typical access locality, the 95 % interquantile range is a statistic indicating the variance.

To be qualified as input data for model generation, these statistics itself must show at most very little variation, i.e. between measurements of the same application with the same configuration, the median and the 95 % interquantile range of the reuse and stack distances of each instruction group should be equal or close-to-equal. We disregard instruction groups having less than 100 samples, as except for the median, our chosen statistics cannot be considered reliable for a low sample count. The impact

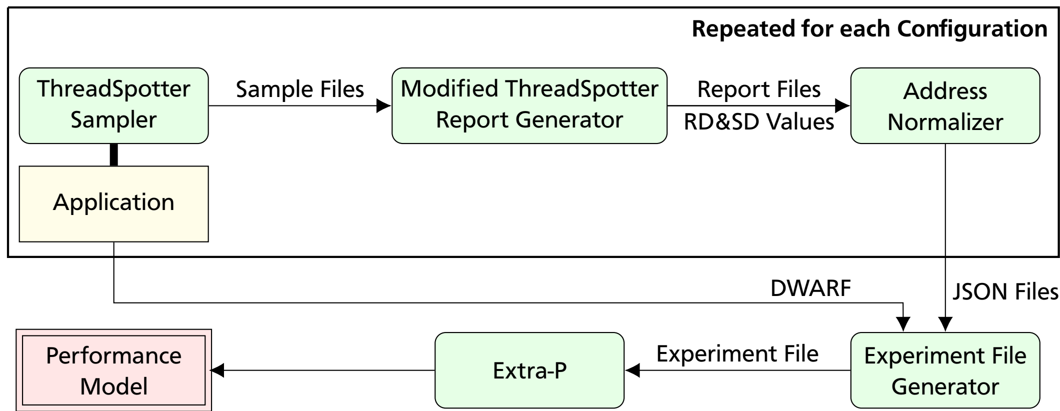



**Figure 5.2:** Illustration of our toolset's workflow. For each configuration, the application is sampled, the modified report generator is invoked and its output is further processed by our software, storing the locality data for each measured configuration in JSON files. Once all measurements are done, these files are processed into an Extra-P experiment file and the locality model is generated.

of these instruction groups are low, thus this does not worsen the quality of our models. For its own analysis, ThreadSpotter aims for a sample count of at least $100,000$. Considering that most memory accesses happen in few instruction groups, it is likely that for the most important instruction groups ThreadSpotter outputs a number of samples which is sufficient for our analysis.

## 5.3 Model Generation

Having defined which statistics to model, we can run and sample an application several times with different parameters, calculate the proposed statistics for each instruction group, and export this information to Extra-P for modeling. In this section, we discuss the detailed workflow.

First, the user chooses which configurations of the application should be measured, where it is recommended by Extra-P to choose at least five different values for each parameter, i.e. five different problem sizes if we want to model the influence of the problem size, or the cross product of five different problem sizes and five different processor counts if we want to create multiparameter models showing how both problem size and processor count influence the application's locality.

After different configurations to measure the application are chosen, our toolset, consisting of a modified version of ThreadSpotter, our sophisticated raw data processing software and Extra-P, are invoked as illustrated in Figure 5.2: After for each chosen configuration a sample file is created for each process, the modified report generator creates a report file as well as a file containing the sampled reuse distance and stack distance values.

These files are then further processed by our software: For each memory accessing instruction, its address within the virtual address space of the program is looked up in the report. As a part of the address space layout randomization, a security feature of modern operating systems, the dynamic linker randomizes the locations of relocatable code when a program starts, causing shared object code linked into the program to be at different virtual address space locations between runs of the program and between each process belonging to the program. To counteract this, we normalize instruction addresses as tuples consisting of the name of their executable file and their offset to the beginning of the `.text` section[1] of respective file. The reuse distance and stack distance values, augmented with normalized instruction addresses are then output as JSON files, a data format being both human-readable and machine-processable. To generate an Extra-P experiment file, the instruction groups of the differently

[1] The section within an ELF file that contains the executable code

**Table 5.1:** Asymptotic bounds of locality models of matrix-matrix multiplication, naïve implementation (Listing 1) and locality-optimized implementation (Listing 2).

| | **Naïve Implementation** | | **Blocked Implementation** | |
|---|---|---|---|---|
| | Median | Range | Median | Range |
| **Accesses to** $a$ | | | | |
| Line Reuse Distance | const. | $N$ | const. | const. |
| Line Stack Distance | const. | $N$ | const. | const. |
| Byte Reuse Distance | $N$ | const. | const. | $N^2$ |
| Byte Stack Distance | $N$ | const. | const. | $N^2$ |
| **Accesses to** $b$ | | | | |
| Line Reuse Distance | $N$ | $N^2$ | const. | const. |
| Line Stack Distance | $N$ | $N^2$ | const. | const. |
| Byte Reuse Distance | $N^2$ | const. | const. | const. |
| Byte Stack Distance | $N^2$ | const. | const. | const. |

configured program executions and their respectively involved processes are then matched by the set of normalized instruction addresses, where an instruction group is considered if it occurs in *all* different configurations in *any* involved process. The latter is important to allow modeling of applications where different code regions are executed by different processes, such as in a master-slave application, where one process coordinates worker processes. For each configuration and instruction group, the distributions of reuse distance and stack distance values among the processes are merged into one and the median and 95 % interquantile range statistics for each instruction group are exported to Extra-P, where the instruction groups are identified by the normalized addresses of each instruction. If DWARF debugging information is available in the program's Executable and Linking Format (ELF) file, the source file and line number is also exported for each instruction address.

Extra-P then generates models of the median and 95 % interquantile range of the byte reuse distance, line reuse distance, byte stack distance and line stack distance of each instruction group. The sample count of each instruction group relative to the total sample count is also exported, making it easy for the user to assess the importance of the respective instruction group.

As mentioned in Section 3.1, Extra-P [8] models the data by choosing a fit to a function of the *performance model normal form*, which are terms of the form

$$\sum_{k=1}^{n} c_k \cdot p^{i_k} \cdot \log^{j_k}(p).$$

Thus, it not only shows asymptotic bounds, but human-readable terms closely matching the observed data.

## 5.4 Example: Matrix-Matrix Multiplication

We applied our toolset on the $N \times N$ matrix-matrix multiplication, both the code of Listing 1 and Listing 2. As configurations to measure, we chose $N = 256, 512, 768, 1024, 1280$. These values are big enough to produce a sufficient sample count, yielding hundreds of samples with ThreadSpotter's default settings even at the smallest problem size and small enough to have a short runtime; they are equally distant, which is likely to be advantageous for the statistical regression, and they are multiples of 16, allowing each row of the matrices to exactly fit on cache line boundaries, allowing to compare the results with the analytical derivations of Table 4.1, where this was an assumption.

Figure 5.3 shows the measurement results along with the models generated by Extra-P of the median and the 95 % range of the line reuse distance of accesses to $b$ of both tested implementations. The results coincide with the theoretical observations in Chapter 4. The models clearly show that in the naïve

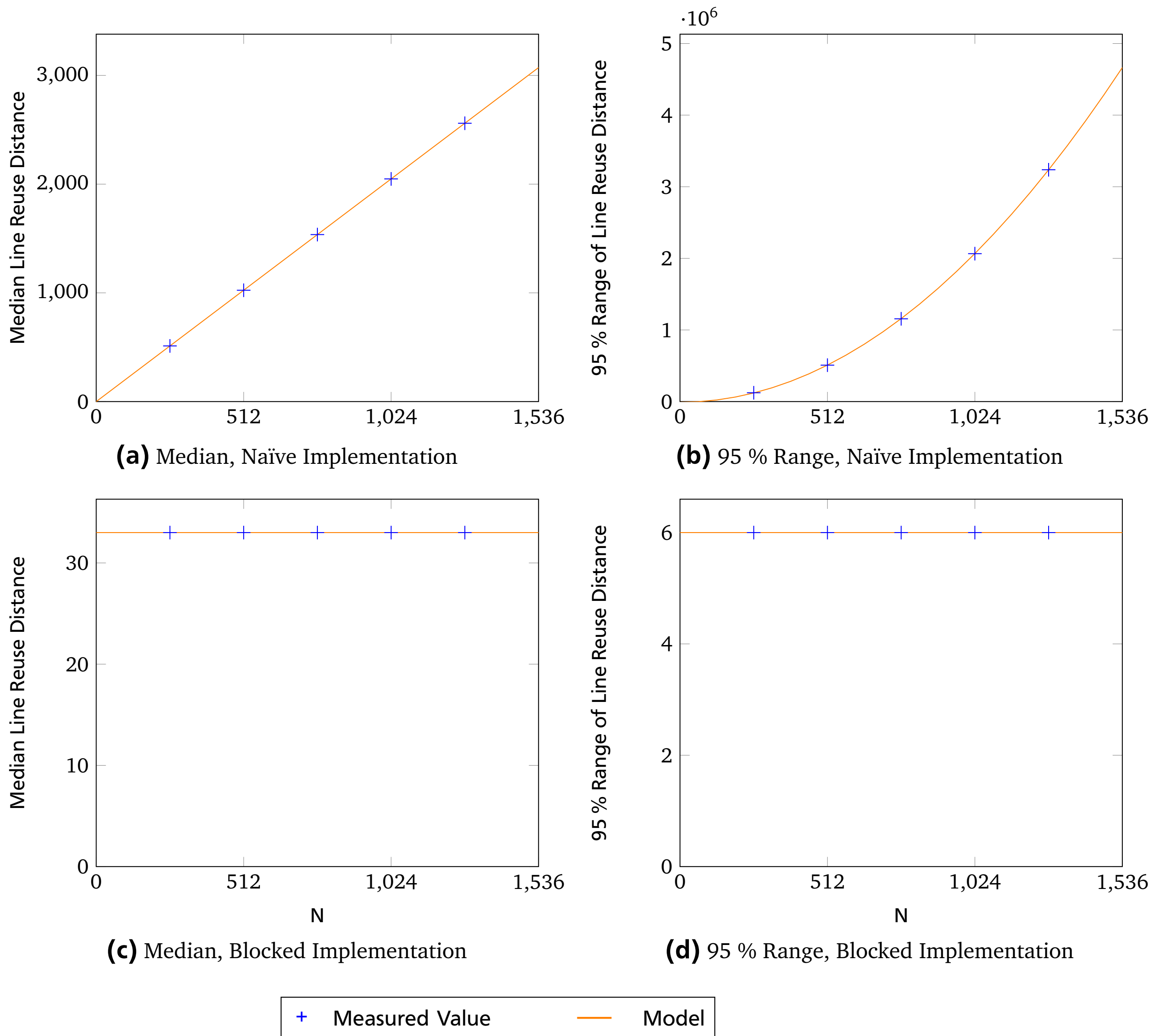



**Figure 5.3:** Measurement values and models of the median line reuse distance and the 95 % interquantile range of the line reuse distance of accesses to $b$ of naïve (Listing 1) and locality-optimized (Listing 2) implementation of matrix-matrix multiplication.

implementation the median line reuse distance grows linearly, and its variance grows quadratically in terms of $N$, accordingly to Table 4.1. Furthermore, the models demonstrate that the blocked variant of Listing 2 completely mitigated this scalability bug, yielding constantly-bounded line reuse distances.

The asymptotic bounds of the models generated by our toolset are shown in Table 5.1, demonstrating our methodology's ability to detect the influence of the problem size parameter $N$ on the locality of the implementation's memory accesses, enabling the software engineer to easily find locations of the code where scaling worsens memory performance.

We discussed the memory access locality's influence on application performance and defined metrics to measure it. Then we discussed a methodology to generate human-readable models for the locality of an application, and validated our approach on a simple, yet meaningful, application. In the next chapter we present the results of a few applications similar to what would typically be run on real supercomputers.

# 6 Case Studies

Having presented our approach and having it validated on a simple example, we can now go over to more complex applications. The findings are also presented in [7].

With our toolset presented in Chapter 5, we sampled Kripke, LULESH, MILC, OpenFOAM and Relearn with ThreadSpotter, measuring the stack distances and reuse distances for cache lines and memory addresses and modeled their median and 95 % interquantile range with Extra-P for each instruction group. We consider the line stack distance and line reuse distance as metrics for spatial locality and the byte stack distance and byte reuse distance as metrics for temporal locality. Why this is reasonable is detailed in Chapter 4.

The measurements were gathered at the Lichtenberg High Performance Computer, a cluster at TU Darmstadt containing multiple sections with different hardware. We ran our experiments on an MPI section that consists of 706 nodes with two 8-core Intel Xeon E5-2670 processors each.

As described in Section 3.1, we used Extra-P to create multiparameter models, describing the evolution of the metrics as the number of processes and the problem size per process are changed. Therefore, for every tested software, we gathered at least 25 measurements, testing at least five different values for problem size and at least five different values for processor count.

Our methodology can be used to assess how scaling an application on larger machines or larger problems impacts the data locality, and thus its cache utilization and therefore its overall computing performance. Thanks to modeling the locality with a fine granularity of instructions or few instructions, it helps developers find exact spots in the code where the locality is influenced by scaling. If debugging symbols are available, the instruction addresses are mapped to source code lines, making it easy to fix the scalability bugs discovered by our toolset. With its automated approach it is easy and feasible to integrate into the development workflow of HPC applications.

In this chapter, we prove the usability of our toolset and show that it is effective to find cache-related scalability bugs. We see that we are able to detect cache-related scalability bugs in three of the five tested applications. Further, the here-presented findings legitimate our approach.

## 6.1 Kripke

Kripke is a 3D Sn particle transport code. It is written in C++ and implements an asynchronous MPI-based parallel sweep algorithm. A major goal of Kripke is the evaluation of programming models, data layouts, and sweep algorithms in terms of their performance impact. In our test runs, we varied the simulated volume per process and the number of processes.

For every analyzed instruction group, our toolset produces two tables as in Table 6.1 for each of the metrics line reuse distance, line stack distance, byte reuse distance and byte stack distance. These tables show the distribution of the locality metrics for every configuration of processor count and problem size per process. Extra-P is then invoked to create statistical models fitting the values in these tables, allowing to extrapolate the locality distribution when further scaling to a larger problem or to a larger computer.

Whereas in Kripke the processor count shows no effect on locality, the problem size per process did influence the locality in about a third of memory accesses. Table 6.1 shows for an exemplary instruction group how the distributions of the byte stack distances behaved when scaling the application. We see that, while the procesor count does not have a clear influence on the statistics, a bigger problem size per process leads to higher and wider spread values of the byte stack distance. As explained in Chapter 4, it is natural for the locality to have a high variance, and it is imporant to analyze and model both the median and the range of the values.

**Table 6.1:** Statistics describing distribution of byte stack distance of one of the instruction groups in *Kernel_3d_DGZ::LTimes* in Kripke.

**(a)** Median

| Processors | Problem Size $8^0$ | $8^1$ | $8^2$ | $8^3$ | $8^4$ |
|---|---|---|---|---|---|
| $3^3$ | 12 | 28 | 112 | 1,105 | 4,318 |
| $4^3$ | 12 | 26 | 142 | 586 | 4,277 |
| $5^3$ | 13 | 35 | 86 | 560 | 4,315 |
| $6^3$ | 11 | 32 | 126 | 548 | 4,268 |
| $7^3$ | 11 | 34 | 76 | 543 | 4,267 |

**(b)** 95 % Interquantile Range

| Processors | Problem Size $8^0$ | $8^1$ | $8^2$ | $8^3$ | $8^4$ |
|---|---|---|---|---|---|
| $3^3$ | 152 | 4,416 | 44,462 | 401,939 | 2,780,825 |
| $4^3$ | 746 | 5,832 | 46,242 | 350,680 | 2,848,907 |
| $5^3$ | 1,485 | 5,873 | 46,263 | 377,271 | 2,830,687 |
| $6^3$ | 787 | 5,955 | 42,397 | 371,909 | 2,740,447 |
| $7^3$ | 929 | 5,778 | 47,188 | 358,353 | 2,749,297 |

In the same instruction group, the line stack distance, a metric for spatial locality, shows a similar behavior: Whereas the median is constant, the range increases with increasing problem size. This is a behavior we already observed in our matrix multiplication example, where it was caused by nested loops with bad locality. It is natural that this can also be found in more complex programs.

The byte stack distance samples of the instruction group mentioned in Table 6.1 accounted for about 11 % of the application's total byte stack distance sample count. In total, we gathered more than 11,300 byte stack distance samples for each configuration, meaning that a fraction of only 0.9 % of memory accesses suffices to be considered for modeling by our toolset.

Kripke's locality influence of the processor count is negligible, so weak scaling may have no influence on memory access times. However, the problem size has a quadratic impact on the variance of both temporal locality and spatial locality distribution. Thus, its locality models are similar to what we observed in our bad implementation of the matrix-matrix multiplication, and we can assume that its performance worsens for large problems.

## 6.2 LULESH

The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) code is a mini-app that calculates simplified 3D Lagrangian hydrodynamics on an unstructured mesh. It is written in C++ and supports a variety of parallelization paradigms, among which we focus on MPI. In our test runs, the problem size defines the volume of the cube per process.

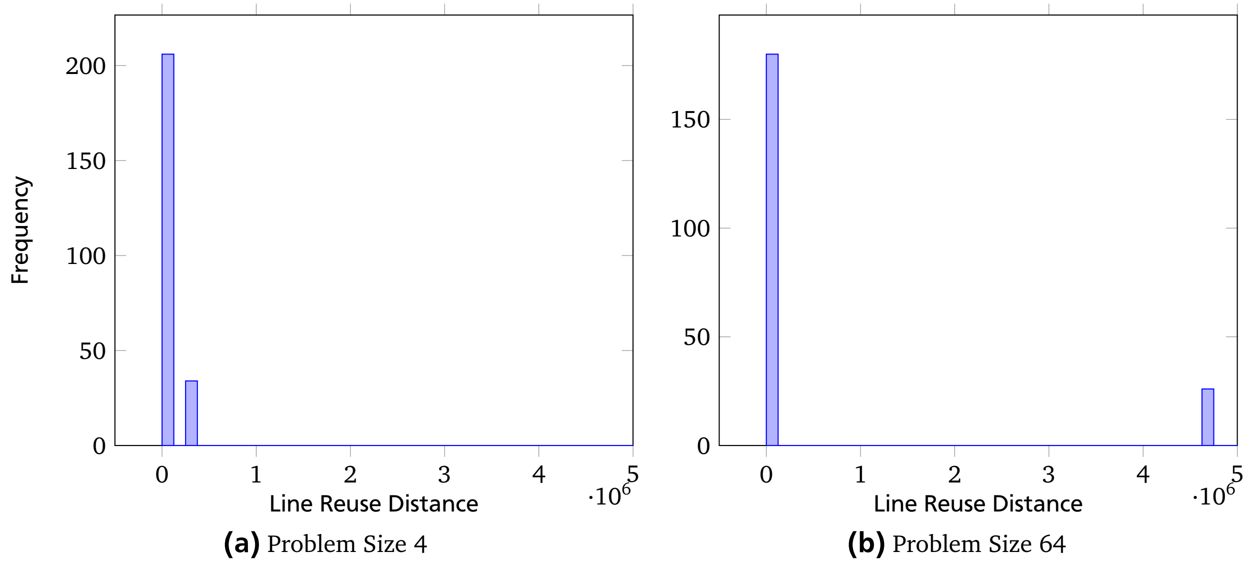
About two thirds of the sampled memory accesses are in function *CalcPositionForNodes*, where six arrays are iterated consecutively in a data-local fashion. About one third of sampled memory accesses occur in the function *ApplyAccelerationBoundaryConditionsForNodes*, where arrays are accessed in a consecutive fashion. Both instruction groups show constantly low metric values, not affected neither by problem size nor processor count.

The remaining instruction groups are routines of the MPI library or the compiler's runtime library libgcc, each of them accounting less than one percent of sampled memory accesses. They access data in a local fashion regardless of the tested configuration parameters.

We conclude that LULESH does not have any locality-related scalability bugs.

## 6.3 MILC

The MILC – MIMD Lattice Computation – tool set is a set of codes for studying quantum chromodynamics (QCD) via parallel simulations of the SU(3) lattice gauge theory on a four-dimensional lattice. It consumes a major fraction of the CPU cycles in US DOE and NSF computing centers. MILC is considered a highly scalable application. In this work, we analyze the application MILC/su3_rmd.




**(a)** Problem Size 4

**(b)** Problem Size 64

**Figure 6.1:** Histogram of line reuse distances of an instruction group in MILC. We can clearly see that the variance is influenced by the problem size. Processor count does not vary between the two presented histograms.

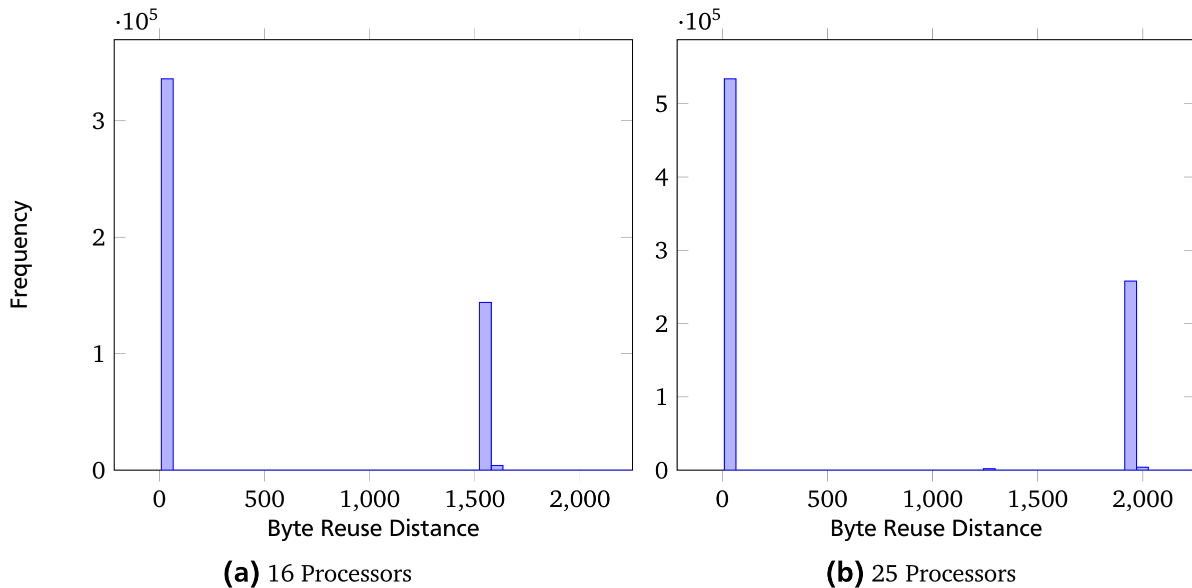
We identified 109 instruction groups in MILC for which we had a sufficient sample count to model the distribution of our metrics. Many of these instruction groups are made of more than one instruction, showing the complexity of MILC. Thanks to the high sample count we gathered, an instruction group only had to account for 0.06 % of samples for the line reuse distance and 0.02 % of samples for the byte reuse distance to be modeled, i.e. to reach the threshold of 100 samples we consider necessary for a reliable model. This proves our toolset's capability to analyze complex programs, where the memory accesses happen in many different instruction groups.

In 24 instruction groups, accounting for about 37 % of samples, our models show a constant median line stack distance but a variance depending linearly on both processor count and problem size per process, showing that spatial locality significantly worsens for a few percent of the memory accesses. However, the effect of problem size was larger by magnitudes than the effect of processor count. Figure 6.1 shows the line reuse distance histogram of one of these instruction groups, where we can clearly see how the variance gets higher for a bigger problem size. This also justifies why it is important not only to model a centralized statistic, but also a statistic representing the distribution's variance. As already mentioned in the previous chapters, we can expect this kind of distribution of the line reuse distance to be a common case. Two of these instruction groups are in the MPI library, together accounting for 0.5 % of the samples. The same instruction groups show high values of byte stack distances, with both median and 95 % interquantile range linearly growing with the problem size.

The remaining instruction groups show a constantly bounded spatial locality, i.e. low values and constant values of both the median and the 95 % interquantile range regardless of the processor count or problem size.

The temporal locality of about half of the memory accesses grows linearly with the problem size, and some of them show a slight influence of the processor count, too.

Thus, we can assume that the problem size per process can only be moderately increased without degrading memory access times.




**(a)** 16 Processors

**(b)** 25 Processors

**Figure 6.2:** Histogram of byte reuse distances of an instruction group in OpenFOAM. We can clearly see that the variance is influenced by the processor count. Problem size per processor does not vary between the two presented histograms.

## 6.4 OpenFOAM

OpenFOAM is a widely used open source computational fluid dynamics code developed by OpenCFD Ltd. It supports the calculation of a broad variety of effects. We analyzed the executable icoFoam, applied to a test case with two-dimensional domain. It calculates the flow of an incompressible, isothermal fluid. The problem size defines the volume and therefore the number of cells per process. Considering other metrics than we do here, OpenFOAM shows a limited scalability [7].

In OpenFOAM, about half of the memory accesses show a dependence on the processor count and problem size in their variance of byte stack distance and byte reuse distance, however yielding an almost constant and low median value. Figure 6.2 shows histograms of the byte reuse distance of one of these instruction groups for two different processor counts. In this kind of distribution, the median represents the most common behavior, i.e. the lower peak on the histogram. The 95 % interquantile range represents the distance from the median to the higher peak. The frequency of these values of worse locality clearly shows how useful it is to model a statistic describing the distribution's variance, rather than considering them outliers. Our statistics describes their locality if they have a frequency of more than 2.5 %, allowing us to model this behavior. Such distribution of the temporal locality metrics is natural when a loop, where the same data is accessed with a good locality, is executed multiple times, with many memory accesses happening in between.

In all analyzed instruction groups, our modeled statistics indicate that the spatial locality metrics do not seem to be affected neither by problem size nor by processor count. This means, that the locality of at least 97.5 % of the cache line reuses is not influenced by the variation of our configuration parameters.

We can conclude that OpenFOAM's scalability might also be limited by the way memory is accessed. However, given that there is probably only negligible influence on its spatial locality, and the influence on temporal locality is only linear to the problem size and processor count, we can assume that OpenFOAM's locality-related scalability bugs are of no concern.

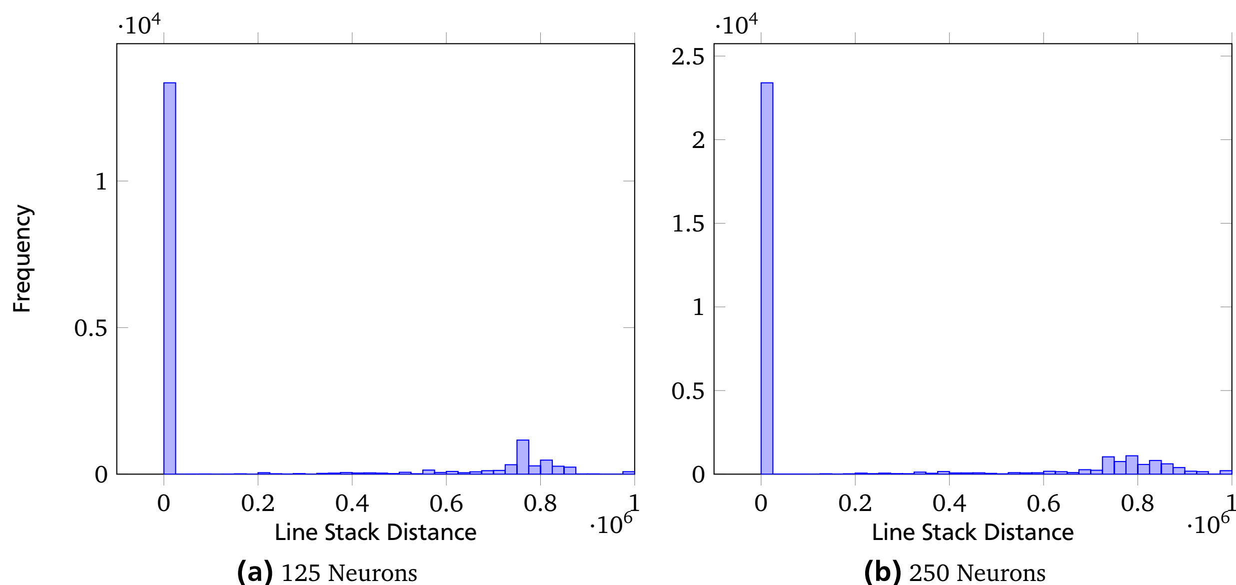



**Figure 6.3:** Whole-program line stack distance distribution of Relearn on two processors.

## 6.5 Relearn

Relearn [19] simulates the dynamics of the connectome in the brain, that is, how connections between individual neurons are formed and deleted. This is also called structural plasticity. The code is written in C++ and parallelized with MPI. In comparison to the original version, optimized memory management makes the code used in this study far more scalable. The problem size parameter in our tests defines the number of neurons per process to be simulated.

Figure 6.3 shows the whole-program distribution of the line stack distances in Relearn for two different problem sizes. We can see that the problem size parameter does not signficantly affect locality.

The four most significant instruction groups, together accounting for about 88 % of all byte stack distance samples, show a rather bad temporal locality with a high median and a high range of byte stack distance and a line stack distance with a low median but a high variance. However, neither of these statistics show a correlation with the problem size or program count, thus it cannot be considered a scalability issue. The remaining instruction groups are in external libraries and do not show scalability bugs neither.

# 7 Outlook

We provide a method allowing to automatically assess how scaling impacts an application's utilization of the processor caches.

Our approach inherits the current limitations of empirical performance modeling tools, or more specifically Extra-P. Kashif et al. expanded Extra-P such that it can detect segmentation in performance measurement and estimate the point where behavior changes, rather than assuming the data to have single-function models [14]. This may be an improvement also affecting locality models since implementations could switch from one algorithm to another according to problem size or number of processes. However, segmented modeling is still limited to single-parameter models.

We gathered distributions of reuse distances and stack distances. Of these distributions, we modeled the median, as a statistic for the most-commonly sampled value, and the 95 % interquantile range, as a statistic for the variance. As justified in theory and shown in the previous chapter, this is an appropriate approach, because histograms of reuse distances and stack distances are naturally almost-always made up of two peaks, the lower values being significantly more frequent. Models of the median and the range describe how the positions of these peaks change. Methods could be investigated to model the evolution of the distribution itself, rather than a few descriptive statistics. In some cases, this could allow better predictions of cache behavior.

As described in Section 5.3, our toolset is split up into several programs which have to be executed to analyze applications. As a future work, our toolset could be integrated more closely into Extra-P, making it easier to use.

# 8 Conclusion

Empirical performance modeling is a technique to efficiently analyze how parts of a program behave when parameters such as problem size or processor count vary. We extended Extra-P, a tool for automatic performance modeling, with the capability of generating empirical models of the locality of memory accesses for each instruction group, allowing the user to assess the cache utilization of the code at a granularity finer than loops.

We discussed and evaluated the reuse distance and stack distance as powerful instruments to quantify the locality of data accesses. We extended ThreadSpotter such that it exports distributions of reuse distance and stack distance values for each set of instructions within a loop accessing the same data. Our software processes its output, intelligently combining the measurements, calculating statistics describing the metric's distributions, and transforming it into suitable input data for Extra-P. Its sophisticated and well-established regression algorithm then creates models fitting the data and describing the influence of parameters to the memory locality, allowing to easily assess how scaling impacts the cache utilization.

We motivated on a simple example how code can have issues of bad locality when the problem size scales, and showed how our novel toolset can find these issues, in the end allowing the developer to fix them and so to improve the application's performance.

Then, we went over to complex HPC applications, namely Kripke, LULESH, MILC, OpenFOAM and Relearn and applied our toolset on them, generating models for their locality for the parameters problem size and processor count. Our software noticed that indeed some of these programs suffered from locality-related scalability bugs in parts of their code. This analysis was contributed to a work proposing a method for requirement analysis and examining the scalability of these applications [7], which proves both the applicability and relevance of our method.

Our methodology can be used to assess how scaling an application on larger machines or larger problems impacts the data locality, and thus its cache utilization and overall computing performance. Thanks to modeling the locality with a fine granularity of instructions or few instructions, rather than modeling the whole program, it helps developers find exact spots in the code where the locality is influenced by scaling. If debugging symbols are available, the instruction addresses are mapped to source code lines, making it easy to fix the scalability bugs discovered by our toolset. With its automated, approach it is easy and feasible to integrate it into the development workflow of HPC applications.

# 9 Bibliography

[1] ARM. Cortex-A15 technical reference manual, 2011.

[2] Erik Berg. *Efficient and Flexible Characterization of Data Locality through Native Execution Sampling*. PhD thesis, Acta Universitatis Upsaliensis, 2005.

[3] Erik Berg and Erik Hagersten. StatCache: A probabilistic approach to efficient and accurate data locality analysis. In *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS*, pages 20–27. IEEE, 2004.

[4] Erik Berg and Erik Hagersten. Fast data-locality profiling of native execution. In *ACM SIGMETRICS Performance Evaluation Review*, volume 33, pages 169–180. ACM, 2005.

[5] Erik Berg, Håkan Zeffer, and Erik Hagersten. A statistical multiprocessor cache model. In *Performance Analysis of Systems and Software, 2006 IEEE International Symposium on*, pages 89–99. IEEE, 2006.

[6] Alexandru Calotoiu, David Beckingsale, Christopher W. Earl, Torsten Hoefler, Ian Karlin, Martin Schulz, and Felix Wolf. Fast multi-parameter performance modeling. In *Proceedings of the 2016 IEEE International Conference on Cluster Computing (CLUSTER), Taipei, Taiwan*, pages 1–10. IEEE Computer Society, September 2016.

[7] Alexandru Calotoiu, Alexander Graf, Torsten Hoefler, Daniel Lorenz, and Felix Wolf. Fast-track requirements engineering for exascale co-design. In *Proceedings of the 2017 IEEE International Conference on Cluster Computing (CLUSTER), Hawaii, USA*. IEEE Computer Society, September 2017. (submitted).

[8] Alexandru Calotoiu, Torsten Hoefler, Marius Poke, and Felix Wolf. Using automated performance modeling to find scalability bugs in complex codes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 45. ACM, 2013.

[9] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. In *ACM SIGPLAN Notices*, volume 38, pages 245–257. ACM, 2003.

[10] David Eklov and Erik Hagersten. Statstack: Efficient modeling of LRU caches. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 55–65. IEEE, 2010.

[11] Erik Hagersten. StatCache/StatStack statistical cache modeling. 2013.

[12] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[13] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. Cache-aware roofline model: Upgrading the loft. *IEEE Computer Architecture Letters*, 13(1):21–24, 2014.

[14] Kashif Ilyas, Alexandru Calotoiu, and Felix Wolf. Off-road performance modeling – how to deal with segmented data. In *Proc. of the 23rd Euro-Par Conference, Santiago de Compostela, Spain*, Lecture Notes in Computer Science, pages 1–12. Springer, August 2017.

[15] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[16] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. PAPI: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, volume 710, 1999.

[17] Scott Pakin and Patrick McCormick. Hardware-independent application characterization. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, pages 111–112. IEEE, 2013.

[18] ParaTools. Threadspotter. `http://www.paratools.com/threadspotter/`, 2017.

[19] Sebastian Rinke, Markus Butz-Ostendorf, Marc-André Hermanns, Mikaël Naveau, and Felix Wolf. A scalable algorithm for simulating the structural plasticity of the brain. In *Proceedings of the 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Los Angeles, CA, USA*, pages 1–8, October 2016.

[20] TOP500.org. Top500 supercomputer sites. `https://www.top500.org/`, 2017.

[21] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[22] Yutao Zhong, Xipeng Shen, and Chen Ding. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(6):20, 2009.