Simulation Driven Performance Analysis for Software Optimization

Josef Weidendorfer

Simulation Driven Performance Analysis for Software Optimization

Habilitation Thesis

Josef Weidendorfer

Abstract

Simulation of computer systems (or components thereof) traditionally is used in the design and evaluation of new hardware configurations and enhancements. To get a good understanding of design trade-offs for objectives such as performance, throughput, or power consumption, it is essential to use a model as accurate as possible and to simulate it in a cycle-accurate manner. As basis for the evaluation, typically, a predefined set of applications is used which is supposed to be representative for future expected use of the hardware.

In contrast, we propose architecture simulation for software optimization which has completely different requirements and goals. It is meant to be used as measurement method to guide analysis and optimization of application code. While it can not replace measurement on real hardware, it may complement conventional measurement methods with attractive benefits such as reproducibility, non-existing measurement overhead, and advanced performance metrics. While the simulated architecture model has to be relatively simple for acceptable slowdown, this also has important benefits. Results are potentially easier to comprehend and can be valid for a set of architectures, hinting at architecture-independent optimization possibilities.

This work gives an overview of the kind of architecture simulation useful for performance analysis and optimization purpose, its benefits and potentials, but also disadvantages and limitations in contrast to established analysis methods. Using cache simulation as important case study, tools for simulation and visualization were developed to show the potential of simulation driven performance analysis. Derived optimization strategies and resulting improvements are presented for various applications. Finally, ideas for further tools using simulation driven analysis are sketched.

Contents

1 Introduction

Good exploitation of the computational capacity of computer systems is an obvious worthwhile goal. With appropriate algorithms used to solve a problem at hand, well tuned programs can give users the highest return from invested hardware resources. This is especially true in the field of High Performance Computing (HPC). Here, large and expensive systems are used to run applications with high demands for computational power. Examples are simulation codes to solve problems in (astro-) physics or chemistry, to do weather forecasts, or to optimize product designs for various industries. Getting more performance out of a system allows for more precise simulations or a higher number of simulations to be executed within the same time or power budget. This directly maps to better scientific outcome.

It would be nice if compilers could translate programs into such efficient code that the full computational power of HPC systems always became available to the users. This is wishful thinking. For example, modern HPC systems consist of parallel structure on all kind of levels, from vector processing units, superscalar pipelines, and multi-core processors up to a large number of nodes with distributed memory. All these parallel resources must be fed simultaneously with enough data to keep the execution units going. To this end, complex and deeply layered memory hierarchies are required. On some levels, processor hardware itself can be quite good in exploiting parallelism, such as with instruction level parallelism exploited by superscalar pipelines. On other levels, compilers often do a good job, such as with automatic vectorization or parallelization of simple loop nests. However, providing data at the right time is a problem where compilers often cannot really help much. For example, to improve local memory access behavior, often, code transformations aware of global program behavior are required. Similarly, exploiting multiple cores and nodes within an HPC system via thread-level parallelism may be restricted due to load imbalance and synchronization/communication overhead. It is often impossible to overcome these problems by automatic means.

Thus, to get the best performance out of an HPC systems, manual tuning is essential. Due to the high diversity of hardware, it is very difficult to provide generic guidelines for program optimization. The most important ones are

- usage of vendor libraries if possible, and
- instructing the compiler to do its best by using the correct command line flags.

If these simple rules do not result in expected performance, analysis tools are required. They allow to get insight into how well a program makes use of available hardware resources. In the best case, they not only provide information about the relation of a bottleneck to program source, but they also present how much better one can get.

Performance analysis tools enable users to understand program behavior by measuring of what is going on in a target computer system during execution. Result of such measurement may be a stream of time-stamped events. These events can correspond to the execution of floating point operations, outcome of branch prediction, or cache behavior on memory accesses. Using additional hardware resources for the measurement often is impossible or too expensive. Therefore an analysis tool usually runs on the system which is the target for measurement itself. In this case, it is essential that any measurement overhead is kept to a minimum. Any overhead shows up as impreciseness in the measured data, reducing the informative value. For these reasons, performance analysis based on measurement on real hardware has important restrictions:

- it only can tell about bottlenecks which exist on the system where the measurement is done, and
- any measurement technique must make sure to not disturb results by too much overhead.

Regarding the second point, specific hardware performance counters are available in most modern processors which enable low-overhead measurements. The counters can be configured to monitor various hardware events without influencing execution. Measurement techniques either read out these counters at specific points in time, such as when a function is entered or left, or they use the counters for statistical sampling. To this end, an interrupt can be raised after a given number of events has been seen. However, any post-processing beyond simple aggregation or filtering of measured data can easily get too expensive.

A correct interpretation of results from hardware performance counters can be difficult. The reason is that they often reflect micro-architectural implementation details which are not obvious. Events may be counted during speculative execution of code which afterwards gets rolled back due to wrong branch prediction. The number of cache misses counted for successive accesses to the same memory block may depend not only on the performance and available parallelism in the memory hierarchy but also on the size of buffers within the pipeline for out-of-order execution. In addition, hardware performance counters often appear on the errata list for a processor. As they are not important for functionality, they are not tested as well as other processor components before chip production.

Simulation for Performance Analysis

As just discussed, there are restrictions to be aware of when designing performance analysis tools which measure the behavior of real hardware. This is especially true when using hardware performance counters to get more insight. These issues can be avoided by instead observing the execution behavior of a program running on an appropriate architecture model. Events of interest can be defined in a way that makes them easier to understand than corresponding hardware events. Model parameters can be modified to check how the program would run on another hardware configuration. There is no measurement overhead which can disturb measurement results, enabling novel measurement techniques. However, the real benefit is the ability to allow for measurements which are not available on real hardware. As arbitrary details of the model state can be monitored, specific analysis can be supported. For example, detection of *false sharing*[1] requires history about byte-wise usage information of a memory block within caches of different processor cores. This information is not available on real hardware. Therefore, tools relying on information from real hardware cannot detect instances of *false sharing*.

An obvious drawback of simulation is that maintaining simulation state may be complex. It can render a performance analysis tool driven by simulation to be unacceptably slow. However, for tuning opportunities which promise large improvements, a simulation slowdown of a factor of 100 may be fine. If the use of a simulation-based tool enables optimization that e.g. convert main memory accesses into L1 cache hits, it definitly is worth the time spent in analyzing the bottleneck. Thus, if an analysis tool can help in finding the right code modifications, a developer may accept long runtimes. An important factor for the speed of simulation is the complexity of the architecture model. For specific analysis it may be enough to use a simplified model. For example, if it is already known that a code has bad cache behavior, simulation of pipeline execution is less important, as bad cache behavior will be the dominant effect for performance. Model simplification can make results easier to understand. Moreover, there often is not enough documentation available to allow construction of precise models of commercial multi-core processors.

For traditional performance analysis tools, there are established techniques keeping overhead small. These techniques also apply to tools based on simulation.

- *Filtering.* Events are ignored which are expected to not be important. For example, L1 instruction cache accesses due to program execution flow are usually not relevant for the allover access behavior of a program regarding a multi-level cache hierarchy.

- *Selection.* Only program parts or phases are monitored that are assumed to be relevant to or representative of the whole program execution. Selection typically is done in iterative analysis procedures. Overhead saved can be invested to enable more detailed measurement.

- *Sampling.* Events which are not statistically relevant are also not important for performance. This observation allows for statistical measurement, taking into account only a subset of events. Sampling can be done either on event basis (focussing only on every 1000-th event) or on time interval basis (for every 1 s of execution time, event collection is done only in the first 1 ms).

- *Aggregation during measurement.* This reduces the amount of data to process and store. Aggregation may be done for classes of events, for events triggered by the same code, or events relating to the same data structures. Aggregation is done when detailed information is not needed for bottleneck detection (e.g. timely

[1]False sharing describes the slowdown effect from frequently transferring a memory block between caches of different processor cores, triggered by frequent writes to different bytes of that block by both cores. That is, the cores do not actually share data. The slowdown is easy to avoid by putting data regularly accessed by different threads into their own memory blocks.

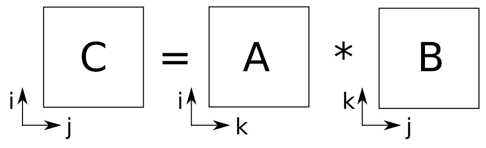


Figure 1.1: Indexes used for matrix-matrix multiplication example.

ordering of events). Further, it can be used for getting a first overview of the performance characteristics in the beginning of an iterative analysis procedure.

In general, the mentioned overhead reduction techniques are also very effective in decreasing simulation time. For example, due to sampling or selection, larger time intervals may exist in which simulation is not required. In this case, simulation can be switched off temporarily and replaced by much faster functional emulation. However, the serious slowdown of cycle-accurate simulation cannot be overcome by these techniques. The need for relying on simplified models and corresponding limited accuracy of results means that architecture simulation should be seen as complementing traditional analysis tools.

1.1 Example: Tuning of Matrix Multiplication

To make it more clear how architecture simulation can help in performance analysis, let us have a look at the example of matrix-matrix multiplication (MM). We use square matrices of side length *size*. For the operation $C = A \cdot B$, Fig. 1.1 shows the variable names used for the matrix dimensions. If we assume that matrix C is initialized with zeros, the following code shows a possible implementation:

```
for (i=0; i<size; i++)
    for (j=0; j<size; j++)
        for (k=0; k<size; k++)
            c[i][j] += a[i][k] * b[k][j];
```

For matrix multiplication, it actually does not matter in what order the products of elements from A and B are added to elements of C. The given implementation just shows one out of several possible orderings, which can be given the name “IJK” corresponding to the variable names used in the loop nests.

What ordering results in the best performance over all sizes, and why? For this example, we simply can test all possible orderings. But that does not give any insight. It would be nice to have an analysis method that can tell us why some orderings are faster than others. We could use that insight to come up with good solutions for similar codes. Fig. 1.2 shows achievable performance in MFlop/s for matrix side lengths from 400 to 600 on an Intel Core i5 processor with 3 MB of L3 cache at 2.4 GHz, using double precision. As can be seen, IKJ does much better than JIK. With 8 bytes per matrix element, at size 600, the required memory for all three matrices is $3 \cdot 600^2 \cdot 8$ bytes = 8.2 MB. This does not fit into the last-level (L3) cache of the used processor,

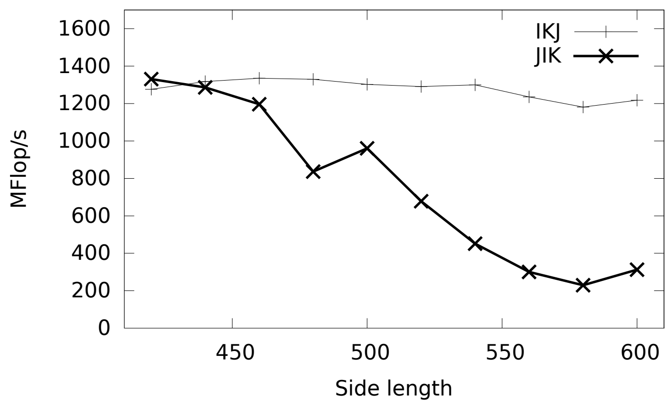


Figure 1.2: Performance of MM for loop orderings IKJ and JIK.

so the need for main memory accesses may be the answer to slower performance for larger sizes.

To check that, we use PAPI [73] to measure cache misses[2]. Fig.. 1.3 indeed shows a sharp increase in L3 misses for JIK for larger sizes, but it does not explain why IKJ is so much better. Further, L3 misses only increase for side lengths larger than 560. Thus, main memory accesses cannot explain the bad performance of JIK for smaller sizes.

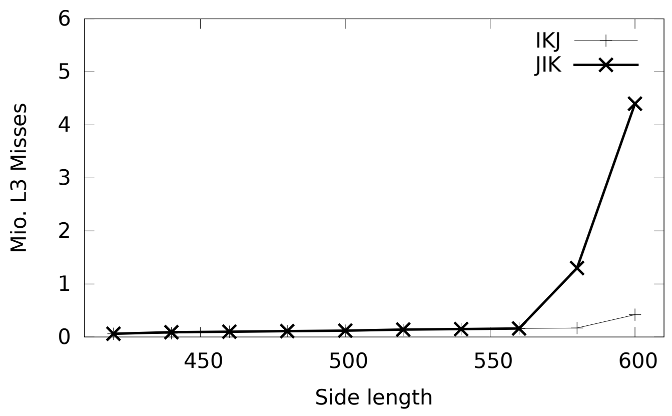


Figure 1.3: Number of L3 misses for loop orderings IKJ and JIK.

To verify that performance changes are caused by the memory hierarchy, it makes sense to look at L1 misses, as given in Fig. 1.4. And indeed, L1 misses for the JIK case show a similar shape as seen in the performance numbers of JIK. However, the reason for the much higher miss rates of JIK in comparison to IKJ is not easy to understand.

The problem is that cache misses only show symptoms of bad behavior, similar to runtime numbers. At least, the numbers support the hypothesis that the problem is caused by the memory hierarchy.

[2]`PAPI_L1_DCM` and `PAPI_L3_TCM` for L1 data cache misses and L3 total cache misses, respectively.

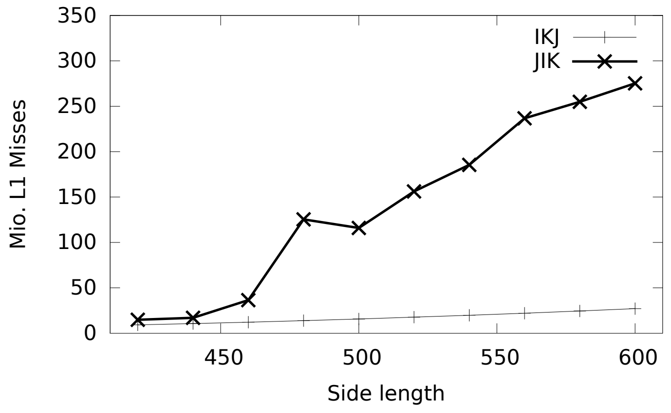


Figure 1.4: Number of L1 misses for loop orderings IKJ and JIK.

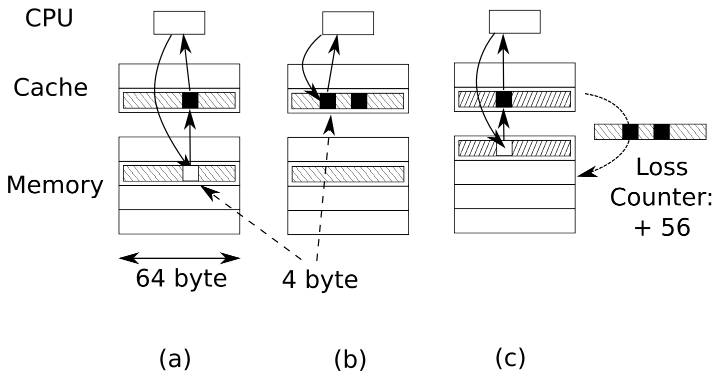


Figure 1.5: On every eviction, number of unused bytes add up to the Loss metric.

Metrics for Better Insights

For a metric that provides better insights, it is helpful to remember that caches always transfer data on a block level. More data may be loaded into a cache than actually are used. To describe this fact, we would be interested in a metric "Amount of data loaded into cache which are *not* actually used". If this number gets high at given points in time during execution of a program, it is a clear indication that memory accesses or data layout needs to be re-arranged. Actually, not using bytes loaded into cache both wastes expensive cache space and further reduces the already limited bandwidth to main memory.

However, processor caches do not remember which bytes actually were accessed by a program. Adding such a feature is not expected to be done by commercial processor manufacturers, as resources would be quite high for something that is not used most of the time. But it is easy to add collection of this meta-information to a cache simulator. For every cache-line, we maintain a mask which tells whether given bytes were accessed by the program. When a line gets evicted, we increment a counter by the number of bytes not used.

Fig. 1.5 shows an example. First, a memory block is read from main memory into the cache, with all cache lines still invalid (a). Then, another access to that line is done, setting further bits of the usage mask for this line (b). On eviction, the number of unused bytes are derived from the usage mask, incrementing the metric by 56 bytes in the example (c).

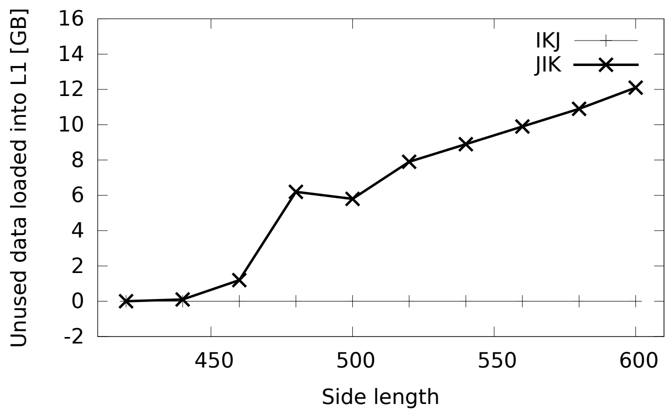


Figure 1.6: Amount of data unnecessarily loaded into L1 cache, for IKJ and JIK.

The implementation of this metric in a cache simulator is presented in Sec. 3.5. Fig. 1.6 shows that the shape of the curve for JIK for this metric is very similar to the number of L1 misses. Now the problem is easy to understand. Some matrix data were accessed in a very sparse way. The L1 cache is too small for the sparsely accessed memory blocks to still be in L1 at later accesses of these blocks. In the case of JIK, the curve shows that no data got loaded unnecessarily. We can extend the metric to maintain multiple loss counters for different data structures, and increase the loss counter for the data structure which covered most of a cache line on eviction. In our MM example, it would show that the sparsely accessed matrix is B. Looking at the code, we now see that B actually was accessed column-wise, which is a bad access pattern as long as cache-lines covering one matrix column do not fit into L1 cache. This hints at a possibility to make JIK fast. The column-wise access should only be done on a limited number of rows to enable cache-lines to stay in L1.

1.2 Design of Analysis Tools using Simulation

The example presented shows some main benefits of architecture simulation over real measurement. What issues can prevent architecture simulation to be a useful technique in practice? To this end, it is instructive to think about the development steps involved for a corresponding tool.

- First, one needs to decide about the architecture model to simulate. What aspects of real machines are important, what can be modeled in simplified ways, and what can be ignored? It is important to find a good trade-off between accuracy and complexity, as simulation speed may be crucial for usability of the tool.

- To simulate what is happening during program execution, we must be able to observe actions that drive the simulation. For that, we can use specific monitoring hardware, OS-supported methods (using processor exceptions, e.g. via breakpoints or hardware performance counters), interpretation, or instrumentation (injection of measurement code) to capture the execution flow of a program. The methods differ in cost (is there a need for specific hardware?), performance (is there a slowdown?), power (which/how much actions can be captured?), and interference (can the simulation driven by observed actions influence subsequent execution flow?). Further, interpretation and instrumentation allow for capturing actions either on the source or binary level. Actions need to be annotated in a way that allows relation back to source. For faster speed, actions may be selected, filtered, or sampled. This can be inherently coupled with the capturing method (sampling with hardware performance counters), or be done explicitly e.g. by instrumenting only selected program parts.

- Observed actions need to be forwarded to the simulator. This step can be decoupled from the previous step by writing actions into a file (a *trace*), enabling replay of the same stream of actions for different architecture model parameters. However, the amount of data may be prohibitive. Also, this restricts the simulation. We cannot model behavior where simulation outcome influences the stream of observed actions, e.g. for timely interleaving of events from multiple threads. For the latter, tight coupling of action generation and simulation is needed (*execution driven simulation*) in contrast to *trace-driven simulation*.

- Depending on the metric to collect, the simulation needs to maintain additional state. Events important for the metric need to be monitored. The source context of actions must be forwarded as annotation to observed events. For faster speed, parallel simulation can be done (*parallel discrete event simulation*, PDE). Typically, different components of an architecture can be simulated in parallel. However, correct timing of interactions and access to shared resources must be handled correctly. There are trade-offs between accuracy and simulation speed possible by relaxing time constrains.

- Events observed within the simulation typically need to be post-processed and aggregated to get manageable amounts of measurement data (e.g. by counting or histogram generation). This aggregation has to take into account source context annotation, to enable useful presentation of results.

The above steps provide a comprehensive list of design choices for simulation-based analysis tools. For different use-cases and metrics to collect, different tradeoffs can be made to come up with a tool which is usable in practice.

Focus on Memory Hierarchy

The overhead of simulating a complex architecture model can render an analysis tool impractical to use. For that reason, it is important to think about the "sweet spot"

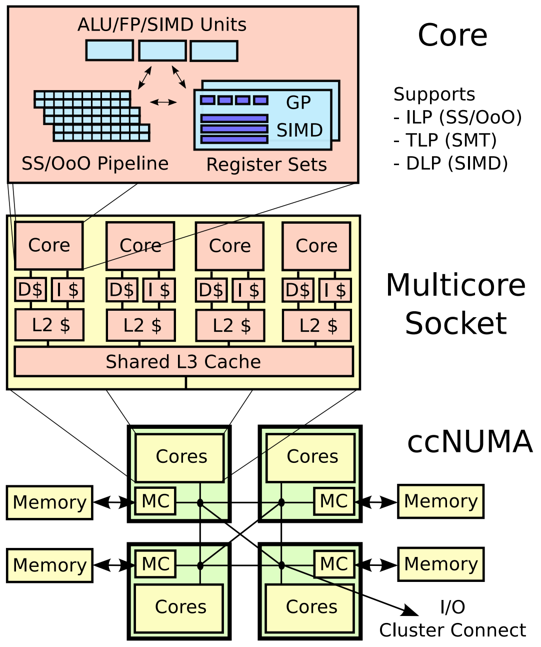


Figure 1.7: Components of a modern server with 4-socket multi-core CPUs. Each core supports instruction-level (ILP), data-level (DLP), and thread-level (TLP) parallelism by superscalarity (SS) and out-of-order execution (OoO), vector units (single instruction multiple data - SIMD) and simultaneous multithreading (SMT), respectively.

of usable architecture simulation for performance analysis. To this end, the following questions should be answered.

- Which components of a modern computer system are often the limiting factor for performance in such a way that it is worth to do simulation for a better understanding of the issues?
- Can architecture models including the identified components be simple enough to still expose the issues behind eventual performance bottlenecks, without the simulation getting too slow for acceptance of a corresponding tool?

Fig. 1.7 provides an overview of a modern server or HPC node consisting of multiple multi-core CPUs (without accelerators[3]). We can identify three levels within such a system:

- On the inner level, superscalar execution pipelines with out-of-order functionality drive scalar and vector processing units. They execute instructions from multiple hardware threads. Performance issues may be not-exploited data level parallelism (DLP) available within vector units, data and control conflicts, as well as resource conflicts due to shared use of resources by pipeline stages and

[3]Accelerators such as GPUs are interesting to be integrated into a model for a computing node, as their usage pattern of running a huge number of often independent small kernels should allow good performance estimations by relative simple and fast running simulations. However, this is future work.

hardware threads. With the x86 AVX extension for vector processing, eight double precision operations (using FMA, multiple-add) are possible in the best case. Control and data dependencies usually are resolved by the out-of-order pipeline unless branch prediction is not working or memory accesses cannot be served by the L1 cache. Assuming L1 cache hits, the worst performance issue for programs caused by this level typically is not above around a factor of 10 compared to best performance.

- The next level shown in the figure is the chip package with multiple cores and a hierarchy of cache levels including a shared last-level cache. Performance issues here relate to increased latency and reduced bandwidth for memory accesses (assuming L3 hits) and communication among cores. With a latency of around 25 cycles for L3 accesses and L3 bandwidths of around 40 GB/s per core one may observe a worst-case slowdown of around a factor of 10 in comparison to the best case (ie. L1 hits). That is, we may see performance effects in the same order as seen within a core. However, for total performance achievable by a CPU, one has to take the effect of idling cores due to missing parallelism or load imbalance into account.

- On the outer level, there are various chips on the main board with own memory modules and an interconnection structure. Memory can be accessed from all chips in a coherent fashion, but may have different latency due to different data paths used. This is an example of a ccNUMA system (cache coherent non-uniform memory access). Performance issues are caused by communication which gets slowed down due to bandwidth limitations and contention as well as increased latency. We assume latency in the order of hundreds of cycles as well as bandwidths limited to around 20GB/s per chip. In a bad case, all cores of such a server system may want to access one memory module at the same time. Thus, comparing with the best case (L1 hits), worst-case slowdowns at this level can easily go up to a factor of 100.

In summary, it is easy to see that memory accesses can be responsible for huge slowdowns. They can produce issues resulting in ten times larger slowdown than any effects within cores. To be able to simulate how many accesses actually go to main memory, we need to observe memory access actions of a program, and follow their way through the memory hierarchy. It should be enough to only simulate the L3 cache as long as we are only interested in performance bottlenecks due to main memory accesses. However, due to limited associativity of caches, it can happen that accesses miss the L1 and L2 cache even though most L1 or L2 cache lines are not frequently utilized. This requires the inclusion of a lower cache level in the architecture model.

The importance of the memory hierarchy for all-over performance of programs is expected to hold true in the future. This can be explained by the cost of operations in terms of energy consumption[4]. According to the DARPA ExaScale Computing

[4]Energy consumption relates to heat dissipation which can destroy circuits. Cooling as countermeasure is only possible to some degree. Thus, ultimately, energy consumption has a hard limit. It

Study [39] (p. 154), the energy needed for a floating point operation alone is expected to go down by a factor of around 7 in ten years, from 55 pJ/op in 2008 to 8 pJ/op in 2018. However, processors consist not only of computational units. Data transport has to be taken into account (within cores and the memory hierarchy). As the amount of energy needed to drive signal paths on-chip does not benefit as much from future technology, data transport will become more expensive in relation to computation. More specifically, for an assumed future processor architecture, the study expects 7 pJ per clock cycle for every 1 millimeter distance on-chip for a 64 bit signal path to deliver new data.

For the memory hierarchy of real processors, the effects induced by the cache itself can be simulated quite accurately, as vendors typically document important parameters. This even may include the kind of hardware prefetchers used[5]. In contrast, a good performance approximation of a superscalar out-of-order pipeline requires much more parameters to be known and documented, which often is not the case. Further, model complexity is increased immediately by an order of magnitude, making simulation unfeasible for performance analysis.

In a paper proposing a relatively accurate simulation technique for out-of-order processors called *interval simulation* [27], the authors compare cycle-accurate simulation with their approach and a simple “1-IPC model” (assuming a fixed execution speed of one instruction per cycle apart from memory accesses). While interval simulation can achieve 24% accuracy compared to cycle-accurate simulation, the 1-IPC model is still “only” 60% off. Such simulation errors are well acceptable for a performance analysis tool which focuses on memory hierarchy effects.

Targeting Parallel Programs

The first part of this work looks at architecture simulation targeting sequential code sections (Sec. 3). For any modern analysis tool, it is important to work with multi-threaded code. The presented tool is able to do that. However, there is a difference between being able to run multi-threaded code on the one hand, and using an architecture model that gives meaningful results for multi-threaded code on the other hand.

One can classify performance bottlenecks of a parallel program into issues which also appear with sequential code, and issues only happening with parallel code. This last point includes communication/synchronization overhead, not being able to use all resources because of load imbalance, or multiple threads slowing each other down due to shared resource usage. To reflect these effects, an architecture model is required that is more complex than a model that only focuses on issues in sequential code. Thus, it is useful to follow a two-step strategy. One mode of the performance analysis tool uses a simple model which allows to show bottlenecks for sequential code. Another

is expected that some percentage of transistors has to be switched off at any point in time for future processors, called the “Dark Silicon” effect.

[5]Intel documents the type and number of hardware prefetchers used in their processors in their Software Developer Manuals [56].

mode uses a more complex model which enables bottleneck detection important in the context of parallel code.

This strategy goes hand in hand with a useful tuning approach. First, one should go for the best sequential performance, e.g. by running an MPI program with one task only. In a second step, one can focus on parallel performance issues. This is recommended because applying sequential optimization often results in more complex parallelization strategies. Similarly, to be able to apply the right optimization for best parallelization (load balancing, reducing pressure from single threads on shared resources, and so on), one should use well tuned sequential code as basis. This is equivalent to the reasoning that performance analysis is meaningless on code compiled with "-g -O0" (ie. without compiler optimization).

For the reasons stated, we devote separate sections in this work to architecture simulation for sequential code on the one side (Sec. 3), and parallel code on the other side Sec. 4).

Acceptable Slowdown

To decide about model complexity that can be used in a tool, the achievable speed of simulation is important. We assume that a user takes advantage of the fact that architecture simulation is stable and reproducible, and makes use of *Selection*. Thus, only small representative sections of code are actually simulated.

Given this scenario, a slowdown of up to a factor of 100 is very well acceptable in practice. Even interactive programs with a graphical user-interface remain responsive on modern machines.

1.3 Overview and Contributions

Architecture simulation enables collection of metrics which can provide better insights into the execution behavior of a program. However, in the example given in Sec. 1.1 we actually have chosen the right metric for the problem at hand.

What are good metrics? This depends on which components of a processor should be covered. For the memory hierarchy, numbers of hits and misses may not be helpful. We will discuss metrics for memory accesses together with cache optimization in Sec. 2.1.1.

For both good usability and trustworthiness, we want the analysis tool to be able to analyze programs without special preparation steps such as recompilation. Thus, we require that the tool is able to supervise the execution of binary code. On the one hand, this makes it easier to use. On the other hand, the analysis includes any optimization done by the compiler, and thus reflects production runs of the code as much as possible. Source code is only needed for the analysis tool to present visualizations to the user which relate measurements back to functions, source lines or data structures[6]. In our work, the architecture simulator actually represents a virtual machine

[6]This relation back to source requires the binary to include debugging information. Modern compilers allow for both optimized code generation and including debug information, e.g. via "-O3 -g".

providing an environment for user processes to run. This is adequate for HPC code as long as programs do not spend relevant time in I/O phases. For HPC, one can expect that OS scheduling is not important, but threads/processes are fixed (*pinned*) to cores. Thus, there is no scheduling behavior of the OS to observe, and the simulator becomes independent from the OS used. Providing an observable process environment can be achieved by ISA (instruction set architecture) virtualization. This uses either interpretation or binary translation techniques to allow the observation of actions in program execution. Binary translation can be done either statically by binary rewriting (using tools such as ATOM [104] or MADRAS [114]), or dynamically. The latter technique is called DBI (dynamic binary instrumentation), and corresponding tools are Valgrind [86] and Intel Pin [76]. In contrast to interpretation, usage of DBI promises relatively low overhead, but some rules need to be obeyed to use them efficiently. In Sec. 2.4 we introduce the mentioned DBI tools.

Architecture simulation itself may be trace-driven or execution-driven. To be faster, the simulation itself should run in parallel. Speedup of parallel simulation is usually heavily bound by synchronization for temporal ordering. Relaxations are possible. We discuss simulation methodologies in Sec. 2.3, and present our parallel implementation for bandwidth bottleneck analysis in Sec. 4.3, which is relaxing on time order constrains.

For a performance analysis tool, "good" metrics (that is, metrics that pinpoint at the reason for a performance bottleneck) need to be coupled with appropriate presentation to the user. The previous example actually was a special case. It only looked at a few lines of code. But even in that example, the metric "cache space wasted due to sparse access" becomes much more useful by adding information about which data are accessed in such a sparse way (in the example, it was matrix B). Thus, analysis tools need to show context information for events which are identified to cause a performance bottleneck. The context should show the code position and possibly the data structure being involved. However, more context information may be needed. Both code position and data structure context can be refined or coarsened for best usability. We will discuss this in Sec. 3.6. An important aspect for presentation of collected metrics is the kind of visualization used. We will discuss a prototype implementation for visualizing measurement results of the architecture simulation, and provide some experiences in Sec. 3.2.

Analysis that relies on observing the execution of a program can be time consuming, and counter-measures for long runtimes under supervision of a performance analysis tool were already given. A different strategy is to use analytical modeling. Simple analytical performance models can be used to find good implementation strategies even before an algorithm is written down in a programming language. However, with existing code, this code needs manual inspection and understanding to find out the input parameters for the analytical model. We give examples of different proposed simple analytical performance models for HPC in Sec. 2.3.4.

To sum up, the novel contributions of this work are:

- A robust call-graph capturing technique for x86/Linux at binary level was developed, coping with arbitrary recursions depths, different exception handling mechanisms, complex signal handler interactions, and threads. This is used for exact relation of events to source, and described in Sec. 3.1.3.

- To better understand the effects of hardware prefetching as well as exploitation of spatial locality within a cache hierarchy, simple but effective extensions of a standard cache simulator were developed (Sec. 3.4 and 3.5).

- A new technique we call *argument-controlled profiling* is introduced. By embedding function parameters into calling context information, new ways of presentation and analysis of recursive functions are possible (Sec. 3.6). This is helpful in recent HPC codes employing e.g. cache oblivious algorithms and data structure traversals following space-filling curves.

- For estimation of false sharing effects, an analysis technique and corresponding prototype was developed (Sec. 4.2).

- For analysis of parallel code with regard to the memory hierarchy, a specific bandwidth requirement and dominance analysis was developed (Sec. 4.3). In this context, the big advantage of simulation is that we can focus on just one potential bottleneck (here bandwidth limitations) by ignoring other effects. This is impossible with tools measuring real hardware.

Context of this Work / Acknowledgment

Since doing research as a PostDoc research assistant starting in 2003 in the DFG project DiME (Data Local Iterative Methods For The Efficient Solution of Partial Differential Equations) — a cooperation among the Computer Architecture Chair of Prof. A. Bode at Technische Universität München and Prof. U. Rüde at FAU, Erlangen — I was fascinated by the possibilities of dynamic binary instrumentation (DBI). Shortly before, an open-source tool for DBI, Valgrind, authored by Julian Seward and Nick Nethercote, became available. Valgrind had a simple cache simulator included, but the presentation of results was very basic. I extended the cache simulator by adding code for dynamically constructing the dynamic call graph of a program execution on-the-fly, and wrote an appropriate graphical visualization front-end (KCachegrind). The simulator was used within DiME to evaluate various cache optimization strategies, and got extended over time. It got used in analysis and optimization of various codes in students work and PhD theses.

The fascination for DBI frameworks also found its way in teaching at LRR. At this time, a book was published that very well explained the basic techniques of runtime ISA translation and runtime instrumentation (Smith/Nair: Virtual Machines [101]). Based on that, I started a seminar on Virtualization Techniques. It soon evolved into a complete lecture, with the first exercise for students showing the inner workings of Valgrind.

I like to thank Prof. Dr. Arndt Bode for his support over all the recent years, and all colleagues, especially Carsten Trinitis.

2 Background

This chapter explains some key concepts. Later sections of this work refer back to this chapter if needed.

In Sec. 2.1, we shortly recapitulate why and how caches work, provide basic strategies for cache optimization, and finish with a discussion of metrics that should help programmers find the right optimization strategies for their code. In Sec. 2.2 we describe performance analysis tools for sequential bottleneck analysis. We discuss what users should expect from a performance analysis tool. Further, there is an overview of existing tools using traditional techniques. We later refer to this discussion when describing the design for our simulation based sequential performance analysis tool in Sec. 3. The kind of architecture simulation discussed in this work in a more general term is called *discrete event simulation.* We will present the basics in Sec. 2.3 as well as differences to analytical modeling. For both, we present current research work around multi-core architectures, which uses simulation on the one hand and analytical models on the other hand. We conclude this background section with a discussion of instrumentation techniques, especially dynamic binary instrumentation (DBI), in Sec. 2.4). Two frameworks (Valgrind and Pin) are discussed which make the implementation of DBI tools easier by providing architecture-independent instrumentation APIs and infrastructure for dynamic code generation.

2.1 Caches, Optimization, and Locality Metrics

Computers have different types of volatile memory cells. On the one hand, there is Static Random Access Memory (SRAM) with cells using flip-flop circuits typically consisting of six transistors. This design is fast but requires relatively much space and power. SRAM is used within processors for example for register space, to store temporary values within a core pipeline, and fast buffers. On the other hand, there is a huge demand for cheap memory with high density to store data needed during program execution. For that, Dynamic Random Access Memory (DRAM) cells are used, consisting of just one transistor with a capacitor. Due to a different manufacturing process and for increased flexibility, DRAM cells (main memory) reside in own chips separate from processors. Processors should be fast and main memory large. For a processor to not stay idle most of the time waiting for data, it makes sense to have memory buffers on the processor die which get data copied over from slow main memory as needed. For programmers to not have to explicitly copy data back and forth, these buffers are made transparent by moving memory blocks automatically from/to main memory by using the same address space as main memory. They work as *caches.*

How can these caches bridge the gap between slow main memory and the need to have data quickly available in a processor? A key here is the policy which decides about how/when to move data between main memory and cache. The basic reason why small but fast buffers can work at all is the so-called *Principle of Locality.* It also provides insight for a good automatic strategy to make use of available buffer space. Every program exposes locality in its memory access behavior. If a program uses the same data multiple times, this is called *temporal locality.* Further, many algorithms traverse trough sequences of data which often are stored nearside in address space. This behavior is called *spatial locality.* To make good use of locality behavior, caches keep copies of recently used data as long as possible — thus exploiting temporal locality. To exploit spatial locality, they work on a memory block level to bring nearside data into cache at once. Caches can be used as buffer to load data in advance if it is known or expected that this data is used soon. While locality in memory accesses makes repeated accesses to same or nearside addresses fast, prefetching of data results in the first use to be as fast as bandwidth to main memory allows.

Cache designs always involve trade-offs between size, speed, and cost (power consumption/die area). For that reason, cache configurations most often have multiple levels. Small fast caches get backed by larger slower caches. A cache which can hold n copies of memory blocks is much faster if only a subset of the n cache-lines has to be checked on a cache access. Thus, cache lines are grouped into sets (whose size is called *associativity*). The address of a cache access determines the set in which all cache lines are probed for a hit. There are a lot of questions regarding the best cache configuration. Just to mention some examples:

- what is the best cache size taking trade-offs for access speed and cost (power consumption/die area) into account?
- what is the best size of cache lines (that is the amount of data transfered at once into/out of the cache)?
- what is the best size of a cache set?
- what existing copy of data should be evicted to make room for a new block to be placed into a cache set?
- how to predict future memory accesses for automatic prefetching?
- how many levels of caches to stack on each other?
- should caches be private or shared among cores in a multi-core processor?

For general purpose processors, manufacturers rely on large internal benchmark suites. These are expected be representative for future users. Based on these benchmarks, simulations can help to find the best configurations. For more details about cache structure and configurations, we refer to standard computer architecture books [53, 105, 109].

2.1.1 Cache Optimization

An HPC programmer wants his/her code to run as fast as possible. Cache configurations in general purpose processors are chosen to make the average case fast. For a given code, there still may be ways to further improve cache behavior resulting in higher performance.

The cache miss ratio, that is, the number of cache misses related to the total number of accesses is an indicator for the effectiveness of a cache. To get an idea about the miss ratio, an analysis tool can make use of the performance counters built into modern processors to show how often the cache was missed, in relation to the total number of memory accesses.

However, cache misses are difficult to interpret. This is the case because miss numbers do not directly relate to elapsed time. Modern processors can overlap computation with memory accesses. Further, at miss time, a data fetch may or may not already be triggered by a previous access or automatic prefetching. The following categories can be defined:

- *Cold/compulsory.* A memory block was accessed for the first time.
- *Capacity.* An old copy of the accessed memory block was evicted due to too small cache size.
- *Conflict.* An old copy of the accessed memory block was evicted due to too small associativity.
- *Concurrency.* An old copy of the accessed memory block was evicted by a transaction of the cache coherence protocol.
- *Prefetch inaccuracy.* An old copy of the accessed memory block was evicted by too early/aggressive/imprecise prefetching.

Different optimization strategies can be used to improve the effectiveness of a cache. In the following, we shortly mention such strategies:

- Increasing the temporal locality. This is done by re-arranging accesses such that accesses to the same data happen closer in time. For example, if elements of an array are accessed multiple times, it is beneficial to split the array into subsets that fit into the cache. Then, do the multiple accesses to one subset before proceeding to the next subset. This is called *loop blocking.* It also works for arrays with multiple dimensions, and results in deep loop nests. To exploit multiple cache levels, the blocking may be done in a hierarchical fashion[7]. Better grouping of accesses to the same data often works by *fusing* different loops which access this data, or by *splitting* loops to not access too much data in a loop such that data gets evicted.
- Increasing the spatial locality. This is done by re-arranging data structures such that their layout results in dense memory accesses. For different program phases, layout conversions may have to be considered.

[7]The order of accesses then approximates so-called *space filling curves.* Resulting algorithms are often called to be *cache oblivious*, as block sizes do not have to be tuned to best match cache sizes.

- Avoiding conflict misses. This can be done by introducing artificial “holes” within data structures, called *padding*. Thus, data items which previously had to be stored in the same cache sets (of limited size) afterwards are mapped to different cache sets.

- Avoiding concurrency misses. To this end, different data to be accessed from different threads should not be stored nearside in address space. Access to shared data should be reduced e.g. by introducing private copies which get synchronized less frequently. Further, thread placement can be controlled such that threads accessing different data do not share cache space to avoid pollution effects.

- Improving prefetch behavior. To enable more effective automatic prefetching, a data layout should be used that makes future memory accesses easy predictable.

Often, the mentioned optimization strategies map to cache miss classes. Thus, it would be nice if analysis tools could always tell the reason for a miss. However, hardware does not remember why a cache line was evicted, so it is not possible to map a later miss access to the evicted data to its reason.

One solution to find the correct optimization strategy is *expert knowledge*. An expert often can tell the best optimization strategy to use for a given piece of code. Another solution is *auto-tuning* [33]. The code which exposes a lot of cache misses is replaced by multiple, parameterized code variants which follow different cache optimization strategies. An automatic search process tries to find the code variant and corresponding parameter settings which result in best performance. This may need re-compilation, and typically consumes a lot of time.

Much simpler is to use a cache analysis tool which is able to tell what optimization strategy most probably will work for a given bottleneck detected. An analysis tool which employs cache simulation can store enough meta-information to detect cache miss classes. Going further, such an analysis tool is not constrained to classifying cache misses.

Access Locality

Locality behavior of memory accesses is the reason that caches work. Thus, it makes a lot of sense to use a metric in an analysis tool that shows access locality directly. Bad locality always maps to memory accesses which expose this bad locality. If a tool detects bad locality, it can provide a relation to source by showing the memory accesses (both source lines and related data structure) which are responsible.

In the most generic way, Denning [35] defines locality in terms of the distance $D(x, t)$ from a processor to an object x at time t. Distances can be in space (related to network hops or memory addresses) or in time. The locality exposed by a program execution is an aggregation of the distances. For example, a simple scalar metric is the reciprocal of the sum of distances of accessed objects. More detailed metrics use distance histograms. Note that the generic definition of locality using distances between processors executing a object accesses and the location of accessed objects is system dependent because accessed objects may migrate around due to some data movement strategies. E.g., locality implicitly captures the used cache replacement policy implemented in a system.

There is value in making locality metrics more independent (abstract) from specific architectures or systems. This way, program modifications which improve such locality metrics promise to also provide improvements for all kind of different systems, as long as there is some relation between the abstract distance definition and the concrete distances found in a system. Vice versa, if applications expose a behavior resulting in higher locality in terms of some abstract distance definition, architecture designs can exploit this by implementing corresponding data movement techniques.

An example of an abstract spatial locality metric is *the average length of non-zero strides performed by a program*[8] as proposed in [123]. The relation to real caches here is that such strides can be detected to control automatic prefetching. In fact, any kind of prefetching strategy proposed in literature (such as address-correlated prefetchers [125]) relies on and works due to some spatial locality of applications, and it can be used to define a corresponding spatial locality metric.

For metrics which measure temporal locality, the terms *Reuse Distances* or *Re-reference Intervals* [59] are commonly used. They use time-related distances between two successive accesses to the same memory block[9]. For histograms, if a memory block is accessed for the first time, the specific distance “infinity” gets used. There are definitions of reuse distances that use time, the number of memory accesses, or the number of cache misses that happen between two accesses to the same memory block. These definitions are often employed in the context of hardware proposals because measuring such distances is relatively easy to implement in hardware. This allows for hardware to implement locality-aware mechanisms such as automatic partitioning of shared cache levels in multi-core processors. However, such distance definitions do not really help in approximating real cache behavior. The following section describes a better metric.

2.1.2 Stack Reuse Distance Histograms

A temporal locality metric should be able to tell whether a given access is about to hit or miss a cache. For that, the *Stack Reuse Distance* is used, introduced in [13]. It tells how many *different* memory blocks are referenced between two accesses to the same block. For a cache with full associativity and *Least Recently Used* (LRU) replacement policy[10], the stack reuse distance exactly can predict whether a memory access will

[8]This is much more abstract than the “Amount of bytes actually used in cache lines before eviction”, as given in the motivating example within the introduction of this work. That locality metric depends on a specific cache configuration.

[9]Here, we assume a cache configuration with a given cache line size. A memory block has the same size as a cache line and is aligned at addresses being multiples of the cache line size. We note that this makes the temporal locality metric dependent on the cache configuration. It also catches some spatial behavior as accesses to the same memory block are treated to be the same. To get pure temporal behavior, a block size of one byte or word can be used.

[10]In an LRU replacement policy, to make room for new data, the cache line that was not accessed the longest time is evicted.

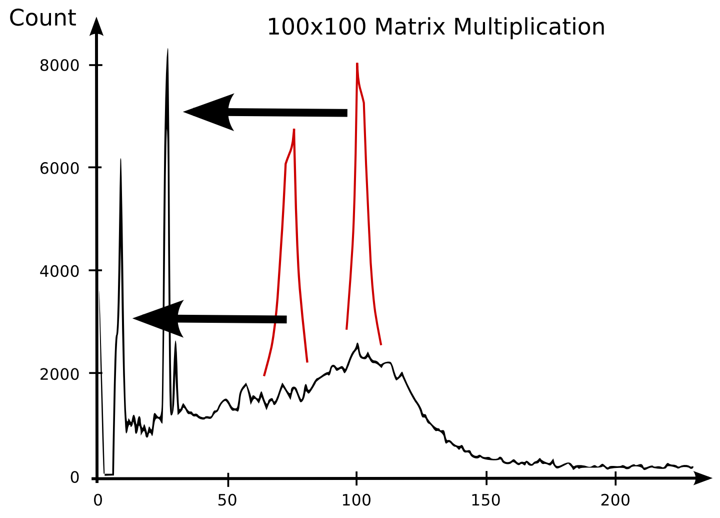


Figure 2.1: Reuse histogram of blocked and original (only hinted in red) variants.

be a hit or miss [19]. This results in an astonishing property. Collected just from one program execution, the reuse histogram can tell how many memory accesses will be hits for *any* given cache size. Thus the histogram can give hints about the best cache size to use for a program. Further, it explicitly shows the size of the average memory footprint[11]. As the stack reuse distance is the most sensible definition for a reuse distance in the context of performance analysis, authors often skip the term “stack”, and just use “reuse distance”. The cumulative variant (from 0 to 100 % by summation of bucket counts) of a reuse distance histogram is called *Miss Rate Curve* in literature [108].

Fig. 2.1 shows an example of a reuse distance histogram for matrix multiplication. Spikes are typically created by loops accessing arrays and may make up most of the accesses in a program. The smaller the distance in the histogram the better the locality. The histogram in the figure shows a cache optimized blocked matrix multiplications. In red color, spikes from a non-optimized version are shown. These are much farther to the right showing bad locality[12].

To collect a stack reuse distance histogram it is required to maintain a stack of memory blocks ordered by access recency during program execution. For a given point in time the top position always refers to the most recently accessed block. Further, the depth in the stack represents the number of other memory blocks accessed by the program since this memory block was accessed itself. The steps to process a new access to a memory block are

1. searching for the memory block in the stack, using the depth as reuse distance of this access (using distance “infinity” if not accessed yet),

[11]The footprint $F_n(t)$ of a program at a given point t in time of execution is the set of accessed memory blocks in the recent n cycles.

[12]This visualization was developed in a students work at TUM (ERA-Großprojekt SS11).

2. updating the histogram, and
3. moving the memory block to the top of the stack.

With M memory blocks accessed and N memory accesses in a program execution, the complexity of calculating the reuse distance histogram in a naive way is $O(N \cdot M)$. There are faster algorithms. Using a tree data structure for the stack reduces complexity to $O(N \cdot logM)$ [3, 18, 19, 87]. If exact distances are not required, and distances are grouped into a constant number of bins, complexity goes down to $O(N)$ [63]. However, the invisible "large constant factor" in the complexity depends on the number of bins which may make this algorithm slow in comparison. Assuming mostly regular memory accesses in a program, it is possible to use time-based reuse distances from sampled accesses for an estimation of the stack reuse distance. The time intervals for reuse can be measured on real systems with hardware breakpoint registers without slowing down program execution [37, 15]. This method was patented and commercialized as part of a tool now sold by Rogue Wave [102]. The approximation heavily relies on the assumptions of regular access patterns which is plausible for HPC codes.

An important property of a reuse distance histogram is that it only depends on the program independent from architecture parameters (apart from the memory block size). However, this property gets lost if we want to extend reuse distance histograms to reflect access locality for multi-threaded codes. Different interleavings of the access sequences from different threads result in different histograms, making them dependent on time behavior of threads. Further, when trying to model multiple caches private to the cores within a multi-core processor, the reuse distance now will depend on the size of the private caches, destroying the original property of telling locality independent from architecture parameters.

Still, it is very useful to have approximations for reuse distance histograms for multi-threaded programs in the context of multi-core processors. Different proposals [96, 128] with corresponding measurement techniques [95] exist. As mentioned above a direct extension of reuse distance for multi-threaded code is not possible. Instead, sensible approximations must be found. This fact can be exploited by chosing an approximation that allows the creation of the histogram to become faster. For the histogram to describe the behavior of a shared last level cache, the ordering of accesses in smaller caches private to cores does not matter. Pericas et al. [89] suggest to filter out stack accesses and update the histogram only after every n accesses by using the memory footprint of these last n accesses (with n being smaller than the number of cache lines in the private caches).

In the context of this work, collecting and visualizing stack reuse distances in a performance analysis tool based on cache simulation is an obvious approach. Actually, Figure 2.1 shows the result of the $O(N)$ algorithm (with N being the number of memory accesses done by a program run) as implemented by TUM students supervised by the author. However, the focus of the work presented in later sections is not on reuse distances as this is already a very well researched topic[13].

[13]We think that new contributions need to make use of static information from compilers, both for maintaining and representing the stack on a coarser level.

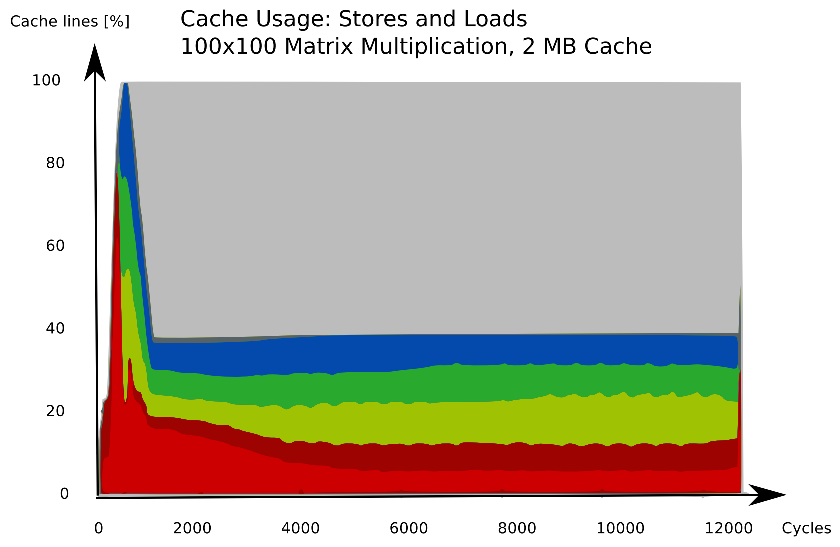


Figure 2.2: Sharing behavior of multi-threaded matrix multiplication.

Other Multicore-Aware Metrics

A metric for the contention on shared connections in a multi-core processor is interesting for a performance analysis tool. We propose this in Sec. 4.3 as bandwidth requirement analysis. Another shared resource on multi-core processors are shared caches. For thread placement decisions on multi-socket systems, it is important to understand data sharing behavior of threads. On the one hand, if data sharing happens, it is beneficial to place threads on cores within the same socket. This way, one thread may prefetch data for another. On the other hand, if threads work on separate data, it is better to put them on separate sockets as this effectively enlarges cache space available to the threads. In [77], authors propose corresponding shared data metrics.

As example for a shared data metric, Fig. 2.2 shows the visualization of an OpenMP matrix multiplication[14]. The x-axis shows time, and the y-axis the usage of cache lines by 8 threads. The colored strips represent private cache-line usage while the much larger gray area at the top shows shared cache-line usage. Obviously it is beneficial to keep the threads on the same multi-core processor.

2.2 Performance Analysis Tools for Bottlenecks in Sequential Code

According to D. Knuth, “premature optimization is the root of all evil” [65]. Micro-optimization in the implementation phase is not only bad for code readability but it is also a waste of time for the developer. Optimization generally should be done after the implementation phase on bug-free code. For the most effective use of developer time it is helpful to concentrate on optimization of code parts which take most of the runtime. Performance analysis tools help in finding these code parts and may give hints about

[14]This visualization was done within the same students work as already mentioned before.

code modifications which result in better performance.

Further motivations to use performance analysis tools are:

- Checking the correctness of assumptions on runtime behavior. If a tool is able to provide the exact number of times a function was called, this allows the programmer to compare his expectations with the outcome of the tool. Differences may point at logical errors.
- Finding the best algorithm for a problem. Implementing different algorithms to the same problem in distinct functions allows the user to compare the time spent in these functions and helps in identifying the best algorithm in a given context.
- Helping in familiarizing with code new to the developer. Often, program parts where most of time is spent are also crucial for the provided functionality. Thus, building an understanding first for the functions identified by a performance analysis tool can be a good strategy to work into new codes. Providing a call graph is essential for this use case.

2.2.1 Bottlenecks in Sequential Code

The code sections pinpointed by a performance analysis tool are the ones where any improvements will have the largest impact. However, an adequate optimization strategy depends on the type of the performance bottleneck found. Bottlenecks in sequential code can be categorized as follows:

- Logical errors. They do not influence the correctness of the program but have negative impact on performance. This includes redundant calls to functions with idempotent results such as multiple initialization. If the tools shows that time-dominant functions are called excessively often, or that loop counts inside of a function are unexpected high, a logical error may exist. To be able to check for that, the tool needs to collect call and jump/loop counts.
- Algorithm with bad complexity. The solution is to test if other algorithms result in better performance.
- Bad runtime behavior caused by sub-optimal exploitation of hardware resources, resulting in slow code execution. Reasons for bottleneck may be
  - memory accesses missing the cache, thus waiting for data from slow main memory;
  - unpredictable control flow changes, resulting in control conflicts in the processor pipeline due to non-working branch prediction;
  - data dependencies limiting instruction level parallelism; and
  - other issues depending on micro-architectural limitations.

With modern processors, bad memory access behavior is by far the biggest problem, as a cache miss can last hundreds of processor cycles. The tool needs to be able to show the exploitation of processor resources. This usually needs hardware to support the collection of adequate event types, and the tool being able to access this hardware support.

In the scope of high performance computing, not being able to exploit expensive resources (e.g. floating point execution units) due to one or multiple of above-mentioned bottlenecks is bad, as it reduces efficiency and results in higher costs. While the first two bottleneck categories are usually taken care of after the implementation phase of a program, the last category needs further analysis every time the hardware changes. Especially for hardware-aware optimization, it is important to understand why a given code results in stalls at some points in the micro-architecture in order to be able to find adequate optimization strategies. Cache optimization often results in much higher performance improvements then optimization trying to avoid pipeline stalls. However, every code modification should be checked for real improvements. A reduction of some event count measured by a tool alone is not worth the introduced higher complexity of "optimized" code.

2.2.2 Performance Measurement Techniques

A tool for sequential performance analysis typically allows to measure event types such as clock ticks (i.e. time), function calls, percentage of bus utilization, or cache misses. These events have to be related to the code region where events happen, or even better, also to the full call path starting from `main` down to the code region. This allows the developer to identify the context of event occurrences more easily, especially when a function is called from different places, or when events happen in library functions inaccessible to the programmer. For the latter, any code changes obviously need to be done at a higher level of the call chain.

Storing the occurrence of every single event is usually not possible because of the overhead and the high amount of data this would produce. As the tool usually runs on the same hardware as the program to be measured, resource consumption of the tool itself should be kept at a minimum to not influence the measurement, thus destroying the usefulness of the measurement itself.

There are different solutions for minimizing the overhead of a performance analysis tool:

- Online aggregation. Instead of storing a sequential stream of time-stamped events (an *event trace*) often resulting in unacceptable I/O overhead itself, we only increment counters when events happen. Multiple counters can be used for different code positions, resulting in a histogram of event counts. A code position can refer to different granularities from machine code instruction addresses or source lines to functions and compilation units, and allows to build an aggregation hierarchy. A list of counts at same granularity is called a *flat profile*. Further, as long as there is no aggregation done which propagates event counts along function calls, all event counts are said to be *exclusive*. However, such propagation is useful to visualize the dynamic call relationship among functions. Event counts which include counts from called function are called *inclusive* costs. For more detailed context, event occurrences can be related to all the functions in the call path up to `main`, which is called (*call path profiling* [50]). The advantage of online aggregation is that the size of the measured data depends on code size and not on runtime (which is the case for full event traces). However, aggregation for inclusive counts or call path relation can result in unacceptable overhead if not done carefully.

- Statistics instead of exact counts. For a program execution, according to the *law of large numbers* [94], the distribution of events happening at different code positions does not change if we only look at a random subset of $1/n$-th of all the events as long as the resulting counts are sufficiently large. This is called *Sampling*. An approximation of this technique is to look at every n-th event occurrence instead of choosing a random subset[15]. Here, n is called the *sample interval*. This strategy is supported by all processors with hardware performance counters. A counter for a specific event type can be configured to trigger an interrupt calling into the tool after a given number of events happening. The advantage is that the overhead of the measurement tool is adjustable, and there does not need to be any *instrumentation* of the target binary for the tool to work. Instrumentation refers to code modifications needed for basic functionality of an analysis tool. Any such instrumentation results in eventual overhead which may disturb the measurement.

- Architecture simulation. This systematically avoids any influence of the measurement tool on the measurement. However, for practical reasons, the simulation slowdown has to be acceptable. This usually means that the model has to be quite simple, and only part of the micro-architecture of a processor is covered.

2.2.3 Call Trees and Call Graphs

It is important to concentrate on code regions where most time is spent in typical program executions. Finding the distribution of time spent in different code sections is called *profiling*. Performance analysis tools may use different representations which are introduced in the following. Let us look at the example given in Fig. 2.3. There are three visualizations all representing the same program execution. Nodes denote invoked functions, and the directed edges denote calls. The dynamic call-tree (DCT) on the left (see Fig. 2.3 (a)) represents a full trace from left to right. It has a size which is linear to the number of calls happening. For profiling, the much smaller dynamic call graph

[15]We note that this technique is susceptible to *aliasing* effects, that is, if code paths have an iterative behavior which relate to the choosen sample interval of n events of the given type, the approximation may degrade. To counter this effect, it usually is enough to slightly randomize the interval to be n on average.

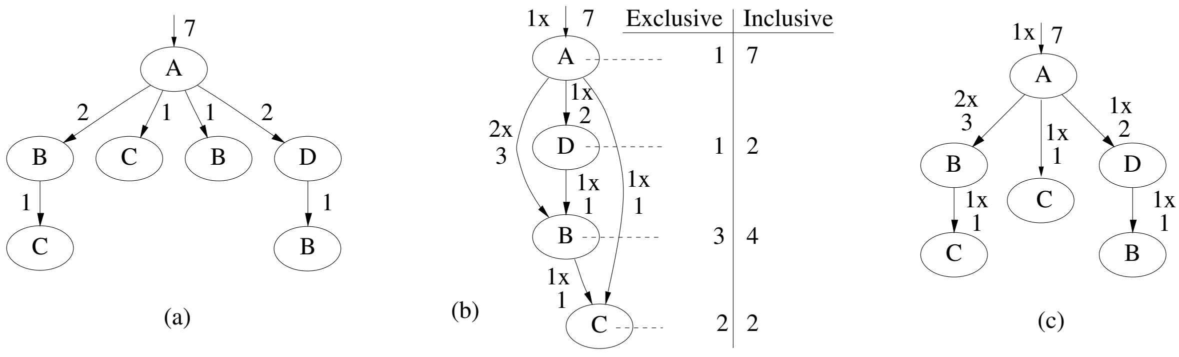


Figure 2.3: A call tree (a) with its call graph (b) and calling context tree (c)

(DCG) typically is enough (Fig. 2.3 (b)). However, a DCG contains less information. In Fig. 2.3 (b), one cannot see that D → B → C actually never happened.

Therefore, [5] proposed the calling context tree (CCT), see Fig. 2.3 (c). There, each node represents the occurrence of a function in a call chain, a context, starting at the root node[16]. During profiling, relating events to nodes and edges is done. Events can be memory accesses, cache misses, or elapsed clock ticks, for example. Attributes of interest for nodes are the number of events occurring inside the function (*exclusive* or *self* cost), and additionally in functions which are called from the given function (*inclusive* cost). For edges, events are interesting that occurred while the program was running in code called by this edge, as well as the number of times the call was executed. In Fig. 2.3, event attribution is shown, assuming one event during execution of each function.

2.2.4 Examples of Performance Analysis Tools

The best known tool for performance analysis of sequential code probably is GProf [45]. It uses sampling based on time intervals (available in every OS) and compiler instrumentation to get the exact count of function calls. This allows to come up with a call graph with heuristically calculated inclusive costs. For the propagation of inclusive costs of a function to its callers, GProf uses the measured call counts and uses the assumption that every call to the function had the same cost. This assumption can be quite wrong e.g. if the cost spent depends on the value of parameters or global state. Further, both the program as well as all used (shared) libraries need to be recompiled as preparation for using GProf. Given a specific command line flag, the compiler inserts (*instruments*) code to collect call counts at the beginning of each function. Depending on the function size, the overhead may vary and be significant. Due to the mentioned drawbacks, it is difficult to recommend the usage of GProf although it is supported by most compilers.

With the availability of hardware performance counters in processors, the sampling

[16]In the CCT, recursive calls or mutually recursive functions are handled specially to avoid a size linear to the number of calls occurring at runtime.

method is most commonly used today by tools from hardware vendors such as Intel. Alpha processors were the first to make use of hardware performance counters in this way, and the corresponding tool was called DCPI [6]. VTune [55] which is available for Intel processors allows sampling both system-wide and per process. It has a mode for collection of the call graph using binary instrumentation. This mode is meant to be combined with measurements from low-overhead sampling for better estimation of inclusive cost. For this to work, the two required program executions have to show the same runtime characteristics for results to be meaningful. OProfile [71] and Perf [34] are available on Linux and also do sampling with the help of performance counters. Due to security reasons, system-wide profiling always requires `root` permissions. However, Perf's per-process sampling works for regular users. For visualization, tools either provide simple lists on the terminal (such as GProf), or a specific graphical user interface (Intel VTune). HPCView [83] uses a web browser as front-end for visualization.

2.3 Architecture Simulation and Modeling

In this section we give a short introduction into simulation techniques, implementation issues, and its relation to complexity degrees of architecture models. A more detailed overview of simulation methods is presented in [61]. Simulation using architecture models can be very precise. Sometimes this is not needed, and analytical models may be enough. We give a short survey in Sec. 2.3.4. We note that simulation can use estimations from analytical models to allow for faster speeds.

There exist various simulation techniques. Architecture simulation typically uses so-called *discrete event simulation* (DES). The simulation is assuming that changes of the model state only happen together with events at fixed points in time. This is in contrast to *continuous simulation*, where state is updated for activities within time slices [10].

Sequential implementations maintain a global *simulated time*. Simulation routines for each component of the model can return the number of time steps during which the state of the component does not change. This allows for jumps in simulated time, resulting in faster simulation. However, different components may influence each other. The simulation main loop has to keep track of which components have to be called in reaction to events from other components. This is important to not violate causal order when components communicate with each other.

Usually, simulators are driven by the execution of a benchmark program to run in the simulated environment. If a full system including a processor with interrupt controller, memory, and I/O devices is to be simulated, one can put a full OS installation (like an x86-64 Linux Debian distribution) with the benchmark programs on a drive image and boot into the simulated environment from that. Using a simulated serial interface, programs can be started from a console prompt within the simulated environment. To make this process faster, system states can be stored into snap-shots, and simulations directly can start from these snap-shots. To skip uninteresting phases such as system boot or program initialization, simulators can switch to pure *functional simulation*. That is, only state relevant for program execution is updated within a simpler machine model. For example, instead of simulating pipelines or caches, only the effect of instructions is emulated by updating the architecture-visible register file and memory.

2.3.1 Execution- vs. Trace-Driven Simulation

The simulators described in the last section can be classified as being *execution driven*. That is, the instructions whose execution should be simulated are determined by the simulation itself. If for some reason a model does not include the instruction fetch stage of a pipeline, it is not possible to have the simulation being execution driven. Instead, one needs a sequence of events which triggers actions in the model. In this case the simulation is *trace driven*. The trace itself needs to be recorded in an execution driven way independent from its later use of triggering the simulation we are interested in. Due to being decoupled from the architecture model, the trace generation cannot capture sensible time information. Therefore, trace generation can be done using fast functional simulation without time stamps.

Advantages of trace driven simulation are:

- no need to have the architecture model to include the control unit to observe the sequence of actions from the execution of a program;
- fast trace generation by functional simulation;
- simulation can be done multiple times using the exact same trace for isolated analysis of effects from single model parameter changes.

Let us assume that we are only interested in the number of hits/misses produced by a synchronous one-level cache in response to data accesses of a single-threaded user process (that is, requests are served one by one by the cache). In this case, the trace required to feed the cache simulation does not need to contain the sequence of all instructions executed in the program. Instead, it is enough for the generation of the trace to record the sequence of addresses from memory accesses of the program run.

The execution driven functional simulation that is required for trace generation often can be approximated by adequate instrumentation of the code to be simulated, running the resulting binary natively on the host (the machine where the simulation is done). This technique is called host-compiled simulation [75]. In the mentioned example the instrumentation has to produce address sequences for the system to be simulated which may be different to the host (e.g. 16- vs. 32-bit addresses). In cases where the host and simulated memory system are the same, it can be enough to manually add code to the inner loops of the benchmark which generates the needed trace. However, this still can be seen as (a special case of) execution driven function simulation.

Drawbacks of Trace-driven Simulation While there are quite some benefits of trace driven simulation as mentioned above, there are also drawbacks. First, traces can become extremely large. The processing of traces (storing to/loading from external storage) can take a significant amount of time compared to the simulation itself. To avoid the I/O bottleneck with traces, it is beneficial to both generate the trace and directly process it for simulation at the same time. Trace generation can forward events to the simulation in a FIFO structure. However, this technique does not change the fact that we still do trace driven simulation. The term “execution driven” requires the simulation to be able to influence trace generation which is not the case here.

The other drawbacks of trace driven simulation relate to the fact that simulation cannot/does not influence trace generation. This may result in simulation errors. Let us assume that the component we are interested in may be able to influence the order or timeliness of instruction execution. For example, with a unified cache storing both data and instructions it may depend on the timeliness of instruction fetches whether these fetches result in eviction of needed data, thus deciding about hits vs. misses of data accesses. If a model to be simulated includes multiple cores, the state of a shared cache level may influence the timeliness of synchronization transactions among threads running on the different cores. Thus, the cache state may decide which one of the threads will acquire a mutex first and can change the execution characteristics of a program completely. The mentioned examples show that it is impossible to do accurate trace driven simulation when a simulated component can influence execution characteristics. However, for some use cases, the resulting error is small enough that trace driven simulation is acceptable.

There is also a problem if the simulation involves multiple instruction streams with threads or processes synchronizing each other[17]. In this case, any recorded trace will represent some execution order. However, the time constrains of the simulation feeded by the traces may result in behavior which prohibits the order found in the traces. As resolution, a violation of causal order or even neglecting synchronization may be needed for progressing the simulation. If resulting errors are not acceptable for accuracy, trace driven simulation cannot be used.

There is a specific issue with trace-driven simulation involving multiple instruction streams if the trace generation is done simultaneously to simulation to avoid the I/O overhead of huge trace files. In this case, trace generation is done by piecewise writing of observed events for each instruction stream into separate buffers (it makes no sense to order the events of different instruction streams as the order of consumption of these events by the simulator is unknown to trace generation). Trace events are generated in chunks for performance reasons. The actual architecture simulation then fetches these events from each of the buffers as required. To clarify the issue, let as assume that there are two threads about to acquire one mutex. Trace generation may decide that the first thread gets the mutex first. No further events are generated for the second thread who waits for the mutex until the first thread releases it. Now, if the simulator needs to fetch events from the second thread but not from the first, and there is not enough space in the buffer of the second thread for the trace generator to advance it to the point where the mutex gets released, this results in a deadlock. The solution is to suspend trace generation whenever synchronizing instructions are detected until the simulation explicitly wants to fetch the next event. This makes

[17]We assume determistic scheduling here. It is unreasonable to assume that recorded traces can reflect dynamic scheduling decisions.

sure that synchronizations happen in the order needed by the simulation. In fact, this reverts the approach to be execution driven whenever synchronisation happens[18].

2.3.2 Parallel Simulation

Simulation can be parallelized to make it faster. This results in a Parallel Discrete Event-based simulation (PDE) [43, 90]. A natural choice is to run the simulation of the model components in separate threads. Now, each thread maintains its own simulated time. Correct handling of events due to communication among components needs synchronization among simulation threads. One component can only jump in its simulated time when it knows that there are no events to process from other components. Often, straight-forward implementations for this issue heavily limit achievable speedup of the parallelization. To reduce synchronization and to allow for larger time jumps, speculative techniques can help. Components simulate up to a given time distance in the hope that there was no event to process from other components. If later it is found that this assumption was wrong, the component must roll-back to a previous state and restart simulation from there. Whether speculation actually results in improved simulation speed depends on the model characteristics. Further, sophisticated speculation schemes may need cascading rollbacks involving multiple components. This can get difficult to implement correctly. Another solution is to accept small causal ordering violations. All components independently simulate up to a given time distance (often called *time slack*), and all possible inter-component events are processed afterwards.

In the scope of architecture simulation, the description of trustworthy PDE simulation is relevant for simulations which use complex models to achieve cycle-accurate or near cycle-accurate results. For simulating multi-core processors, a simulation thread may be responsible for simulating one core and its private cache levels. In this case, the events that result in required synchronization among simulation threads relate either to the shared cache level of the multi-core processor (which may be handled by a separate simulation thread) or to transactions of the cache coherence protocol for maintaining coherency in the private cache levels. Every modification of an L1 cache-line may result in invalidations of L1 lines in other cores. Thus, for a PDE simulation using slack, a slack time should be chosen that still allows to see separate L1 accesses of other cores. This is needed to be able to simulate false/true sharing effects. If it is expected that the program to be executed on the simulated architecture does not share much data (e.g. to get the SPECrate metrics of the SPEC2006 benchmark), speculative simulation with large slack is possible and can provide good speedup numbers.

2.3.3 Examples for Simulation Tools

Architecture simulation is an important tool in the design process for every hardware vendor, but these in-house simulators are usually not publicly available[19]. There exist some academic projects for cycle-accurate simulators. Examples are SimpleScalar [26]

[18]The issue described is important for our tool presented in Sec. 4.3.

[19]AMD provides a system simulator with a very simple CPU model [4].

and Gem5 [20]. Both are used often in literature to validate architecture proposals. An example of a commercial cycle-accurate simulator is Simics [78].

Parallelization of simulation is important. The *SlackSim* platform [30] for parallel simulation of multi-core processors allows flexible trade-offs between simulation accuracy and simulation speed using relaxations as explained above.

Another proposal for fast parallel simulation is to split the simulation into two phases called TPTS (two-phase trace-driven simulation) [70]. First, representative sequential execution instances of functions are chosen which are simulated using precise core and L1 cache models, and afterwards, resulting execution times combined with address traces are simulated for a multi-core processor model with shared cache. The authors mention that they can reach over 140 MIPS (million instructions per second) with 16 simulation threads.

Other proposals try to speed up simulation yet keeping accuracy by ignoring less relevant effects for performance. This is driven by insights from micro-architecture (white box techniques), or by empirical studies with micro-benchmarks (black box techniques). Examples for white-box techniques for simulation of out-of-order processors are the “first-order superscalar processor model” presented in [62] and “Interval Simulation” [40]. In both techniques the model consists of parts covering ideal pipeline throughput on the one hand and exceptions due to long latencies. For this, Interval Simulation splits simulation time into intervals determined by disruptive events such as branch mispredictions and cache misses, and achieves an error rate of 7% on average in comparison to cycle-accurate simulation. The corresponding tool “Sniper” is extended in [27] for multi-core processor simulation, and the authors evaluate various relaxing schemes. For a 1-IPC model (ie. assuming a throughput of 1 instruction per cycle), they find errors of up to a factor of 3. Such fixed IPC models are often proposed to get first approximations for processor designs. Being wrong by a factor of two or more is bad in that domain[20].

Similar to its use in performance analysis tools, *Sampling* is an important technique for architecture design [28]. However, random sampling is often approximated using fixed sample intervals which can have issues due to aliasing effects. For example, this is problematic if a loop-body of a program to simulate has the same runtime as the sample interval. To avoid this problem, program code analysis can be done. The technique was proposed first as SimPoint [99] which takes advantage of sampling for faster simulation. It uses so-called basic block vectors (BBV) to detect when samples fall into same code sections. A similar technique was proposed to reduce memory address trace sizes [51].

To reduce simulation time, it may be enough to only look at specific processor components. An example for simulation of complex cache designs is CMP$im [58], which is used to study novel replacement policies in [59]. Examples for earlier proposed cache simulators are SIGMA [36, 111], MICA [54] and RSIM [88]. All these simulators are trace driven, as they are tailored for parameter studies.

For performance studies, simulators with simple cache models are usually enough. This way, MemSpy [81] analyses memory access behavior related to data structures,

[20]However, being off by a factor of two at the pipeline throughput level can be good enough for performance analysis which focuses on the caches, which easily may contribute factors of 50 and larger.

and SIP [14] provides metrics for spatial locality. SIP only calculates metrics at one cache level. These simulators are quite similar to the one presented in Sec. 3. However, they rely on manual instrumentation.

A target for architecture simulation not mentioned up to now are HPC systems. To enable for relative fast simulation, processor models must be very simple, and the focus shifts to modeling of networks and message passing. The goal is to estimate the scaling behavior of HPC codes on future systems. An example for simulating the execution of MPI applications on large HPC systems is Dimemas [106]. Going further, xSim [38, 21] allows applications to run on millions of cores. While Dimemas uses traces generated by Valgrind [86], xSim runs the real code of thousands of MPI tasks within one thread of the simulator, and uses real measurement for time estimation between MPI activities.

2.3.4 Analytical Modeling

To estimate the performance (or other properties such as energy consumption) of a program execution, an alternative to simulation is analytical modeling. This may be interesting for various reasons:

- the preciseness of architecture simulation is not needed,
- parameters for precise models are not exactly known,
- an application to analyze does not exist yet as compiled binary, or
- the slowness of simulation is prohibitive.

Analytical models use statistical properties from pieces of code for estimating their performance behavior on a given target machine by using appropriate mathematical equations [113]. Simple analytical models describe performance behavior e.g. by relation of machine parameters and code properties using simple correlations. More complex models may involve state machines, networks of queues, or Markov chains.

Analytical Modeling for Processor Design

For processor design, analytical models typically are used to allow for parameter studies with a huge number of parameter settings. For example, to explore the design space for 6 parameters with 10 possible settings each, we already need to do one million time approximations. Smarter search methods for optimal designs may be used, but doing simulations for this scenario is extremely time consuming. In [48], the authors use models based on probability theory and Markov processes to compute miss probabilities of cache accesses, solely relying on statistical properties of applications from Spec2000 and NAS parallel benchmarks. They report a low average prediction error of just 1.5%, and use their model to estimate performance of a large number of cache replacement policies with different parameters.

In [72], authors do analytical modeling of cache behavior to find the right parameters in embedded processor designs for multimedia. They propose probabilistic cache states to model the evolution of cache content during program execution.

For novel architecture proposals, it is often important to be able to estimate time, power, and die area overhead. This is dependent on a manufacturing process, and exact simulations on the transistor level are impossible to do for academic research teams. To enable such estimations, [127] proposed to use analytical models for such predictions, and for a corresponding tool called CACTI, they published model parameters as part of Technical Reports [100]. In [80], the models were extended to account for power leakage.

Analytical Modeling for Performance Optimization

Analytical performance models were proposed for finding the right parameter settings for performance optimization strategies[21]. In [84], the authors use Markov chain based throughput prediction to estimate the performance of small but relevant subsets of program code. For an electromagnetics application they achieve average prediction errors below 10% when comparing to real measurements.

Analytical models are proposed to predict the behavior of shared caches in multi-core processors. In [29], authors propose models to predict cache miss numbers. In an evaluation using the Spec benchmark suite they show error rates of only 3.9%. Similarly, analytical models are used in [7] for contention and conflict estimation. The authors predict the performance of applications parallelized at the loop level. Results may help compilers or programmers to choose the best parallelization strategy. The paper uses probabilistic miss equations for intra-loop and inter-thread re-references derived from source code and shows good results for performance prediction.

To come up with adequate analytical models for the performance behavior of various loop nests in HPC applications, it can be observed that the number of different memory access patterns usually is limited. In [92], the authors use this observation by doing real measurements of various micro-benchmarks and use results to predict loop execution times in applications. From the source of full applications, they manually derive the sequence of loops and their mapping to micro-benchmarks and get good performance estimations when the loops are memory bound.

The previously mentioned work has a black-box view for a processor and uses micro-benchmarks to derive simple but accurate analytical models for known main memory access patterns for a given architecture. In [49], the authors do the same but propose an analytical model taking streaming access patterns also for different cache levels into account. In their model called ECM (Execution-Cache-Memory), they additionally use a vendor tool[22] for static cycle estimations for inner loop bodies which always assumes L1 hits on memory accesses. Using measurements from micro-

[21]Finding parameter settings for best performing code variants is called *auto-tuning*. State of the art in auto-tuning does not use a performance model but real measurements. However, machine learning techniques could use real measurements for automatically deriving good analytical models.

[22]Intel Architecture Code Analyzer

benchmarks, they calibrate their model such that it delivers adequate performance and power predictions.

The Roofline Model

An analytical model that was proposed in [126] is the so-called Roofline Model. It is much simpler than the analytical models mentioned before and it is often used to get a first understanding of the performance characteristic of kernels used in applications.

The model relates the performance measured in a kernel to the peak performance that should be achievable in the best case, taking the data transfer behavior of the kernel from/to main memory into account. For better comprehension, the authors propose a corresponding diagram as visualization. A kernel here is one loop nest. Typically, HPC codes have only a few kernels where most of the time is spent, so the analysis can be done for each kernel separately. A kernel may be parallelized and running on the full system. In this case, the model relates achieved performance to the peak performance of the complete system.

The proposed visualization (see also Fig. 2.4) shows on the x-axis a computational intensity. This is a constant property of a kernel and specifies the number of computations done for each byte transferred either from or to main memory. For HPC code, a computation usually means “floating point operation”, but this depends on the code. The y-axis shows performance. Both axes are logarithmic.

For each kernel in the application, the user must determine the computational intensity, the number of computations done, and the time spent in the kernel. From the latter two values, the achieved performance can be calculated. By using these numbers, each kernel is represented by one point in the diagram. The theoretically achievable performance of the system for varying computational intensities is also drawn into the diagram. This value will be limited by bandwidth to main memory for smaller computational intensity and by the theoretical computational peak performance of the system otherwise. The shape of the achievable peak for different values of computational intensities gives the model its name. This shape is a property of a given target system.

Fig. 2.4 shows an example diagram. Let us assume that a kernel needs to load all elements from one vector and sum up the products of four successive elements each. For each element of the vector, this kernel does four operations and one load from main memory (one element gets loaded, the others are already fetched and stored in registers). This gives 4 double precision operations for every 8 bytes loaded from main memory and thus a computational intensity of 1/2. The corresponding point is shown in the diagram. For this case, we see that we cannot exploit peak performance of the system with this computational intensity. The kernel is memory-bound. Still, we are quite far away from what the system should be able to perform. As the kernel is memory bound, we know that there must be a problem with getting the best memory performance. To make the diagram more useful, different lines can be drawn for theoretically achievable performance given that some issue exists. Examples for such issues are shown in the diagram. For this system, they are “missing software prefetching”

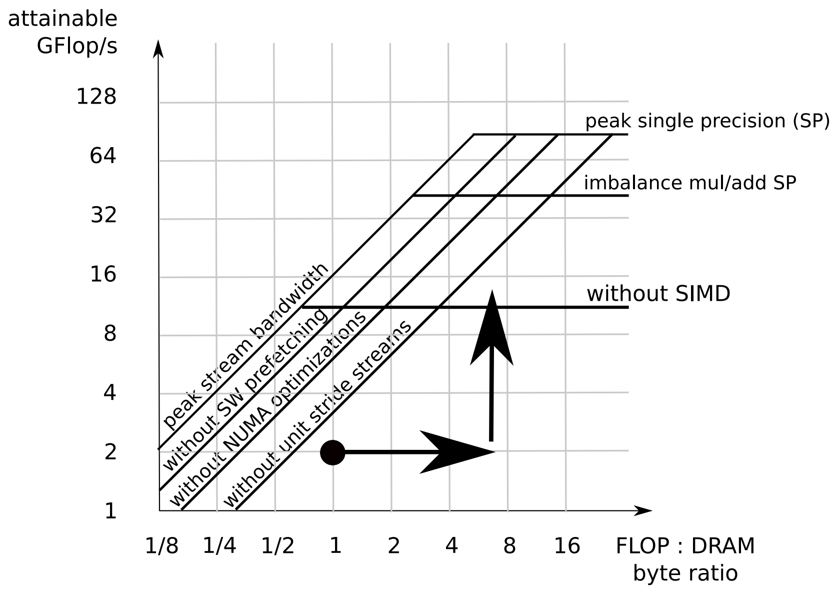


Figure 2.4: Roofline diagram for an some multi-core processor.

and "missing NUMA optimization". For the given kernel, we see an option for faster performance even if the memory issue cannot be solved. By increasing computational intensity (ie. moving the point representing our kernel to the right), we have more distance from the bandwidth limitation line(s). This should give us higher performance. We also can see from the diagram at which point this strategy does not work any longer: when the code starts to become computation bound (here at a computational intensity of 8).

Similar to the system limitation for code that does not exploit prefetching, we can draw lines for lower peak performance because of some issues in the kernel such as wrong balance of operations or not using vector instructions. Regarding the first, the peak performance may be only achievable by using FMA (fused multiply add) operations. As our kernel does not do the same number of adds and multiplies, we are restricted to lower performance. Most modern processors have vector operations. If our kernel is written in a way that vector operations are not possible or the compiler does not generate vector instructions, we may be restricted by another factor of 2 (e.g. for x86 SSE) or 4 (if the system supports four lanes for double precision floating point vector instructions as with x86 AVX).

The simplicity of this model invites for a lot of extensions. Here, we list two options:

- We can draw lines for bandwidth limitations of cache levels. This is useful if the user knows that memory accesses in a kernel never should go beyond a given cache level. In this case, the computational intensity regarding data transfers from/to this cache level has to be determined.

- We can draw different lines for computational peak performance for different operation types/execution units. If the user knows the computational intensity regarding to a given operation type, he will be able to understand system limitations in more detail.

The model deliberately ignores some effects. For example, it cannot tell whether the bandwidth of the instruction fetch unit is a limitation, or whether control/data/resource conflicts in the pipeline constitute a bottleneck. Instead, it assumes ideal circumstances. While this shows that the Roofline visualization may not be able to exactly tell which bottleneck prevents a kernel to actually touch the roof, it does not reduce its usefulness. For more detailed bottleneck analysis, other tools are required. If the model shows that a kernel touches the roof-line, we know for sure that further optimization is not possible (at least as long as we do not increase computational intensity with memory bound code).

As final note we mention here that the tool presented in Sec. 4.3 is much related to the Roofline Model. Bandwidth curves show what bandwidth we need for peak performance at each cache level. The tool also can give us computational intensity, as it counts floating point operations.

2.4 Dynamic Binary Instrumentation

To better understand the benefits and drawbacks of dynamic binary instrumentation (DBI), we first give a short overview on different instrumentation techniques.

2.4.1 Instrumentation Techniques

Methods used in performance analysis tools often have the requirement that code needs to be added to a program to analyze, called *instrumentation*. Typically, this is used for measurement (e.g. reading time or hardware performance counters) and updating metrics to collect. It needs to be done at points in time synchronous to program execution. Instrumentation can happen at different levels:

- manually by the programmer,
- by automatic source code modification,
- by the compiler,
- implicitly by using already instrumented libraries,
- by rewriting a binary,
- at runtime, injecting code before program start,
- by binary translation directly before execution of a piece of code.

Only for the first method, the program needs to be changed. For example, any kind of code for producing log information or debugging via `printf` statements can be seen as instrumentation (which also may be done automatically by other means as mentioned above). For HPC code, there exists the PAPI library for reading performance counters [73]. As programmers need to add calls into PAPI explicitly, this is manual instrumentation. For automatic source code instrumentation, a tool must be able to parse a programming language. One can use a compiler toolkit such as LLVM[23], but

[23]`clang.llvm.org`

there exist specific tools for the task such as OPARI2 [74] or ROSE [93]. An example for the fourth option is the usage of an MPI library that generates profiles or traces for the sending/receiving of messages. Most performance analysis tools for MPI work this way [64, 44]. An example which uses instrumentation generated by the compiler is GProf [45], a profiler distributed with GCC. A program that is compiled with “-pg” generates profile data on execution.

The first four options need a recompilation or at least re-linking[24] to enable instrumentation as preparative step before using a performance analysis tool. If the program makes good use of libraries, one must take care of enabling instrumentation also for these. Otherwise, measurement results may be useless. For example for GProf, system libraries need to be recompiled/available as specifically instrumented versions. This requirement may be prohibitive when commercial libraries are used (for HPC, this mostly relates to specifically tuned processor vendor libraries such as Intel MKL). Finally, a drawback of an instrumentation process done before code generation is that compiler options may result in measurements going wrong. Further, the tool will have problems showing details at machine code level. For example, it cannot observe every memory access.

The last three instrumentation options mentioned above do not have these issues. However, for binary rewriting, there still is a preparation process. Every used library has to be “rewritten” for inserted instrumentation, too. Examples for tools working at this level are ATOM [104] and MADRAS [114]. Also, Intel VTune [55] uses this technique. In contrast, code injection is rarely used. An example is DynInst [25]. This technique relies on patching code in-place. There exist corner cases which are difficult to handle. For instrumentation of the entry of a function, the original code is moved to allow overwriting it with a jump instruction (which is 5 bytes on x86). The jump transfers control flow to custom instrumentation, then to the moved code, and finally jumps back into original code. However, the jump instruction may take more space then the function prologue of the ABI (Application Binary Interface) of the target architecture. In this case, the approach does not work if there is a jump target in the bytes overwritten by the jump instruction. The probability of this case increases with optimized code. The technique gets impossible when a complete function is smaller than the jump instruction. It has to be noted that this is only an issue with the x86 ISA with its variable length instructions. RISC ISAs do not have these problems for code injection.

The last option given above is doing dynamic binary instrumentation (DBI) directly before execution of specific pieces of code at runtime. An overview of the basic technique is presented in the following section. Tools employing this technique can be used with original binaries without recompilation. Further, there is no problem with instrumenting commercial libraries provided only in binary form. For these reasons, this instrumentation technique definitely is the most comfortable way for users. Still, DBI tools have an unavoidable overhead, as every piece of code executed will have to pass the instrumentation step (even if no instrumentation is needed). Thus, DBI is not a good choice for regular performance analysis tools which want as low overhead

[24]Switching libraries can actually be done with other tricks such as with LD_PRELOAD.

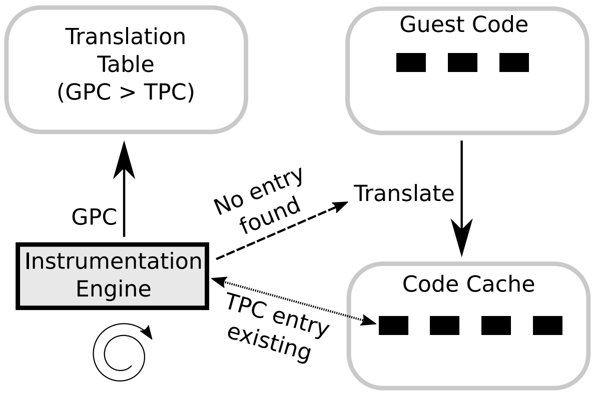


Figure 2.5: Components used in Dynamic Binary Instrumentation.

as possible. However for architecture simulation, the simulation itself usually is the slowest component. DBI overhead often can be neglected in this case.

2.4.2 Basics of Dynamic Binary Instrumentation

Dynamic binary instrumentation uses the same technique as Just-In-Time (JIT) compilers which translate byte code to native code such as the Java Virtual Machine (JVM) or the .NET infrastructure from Microsoft. The same method is used also by *binary translation* for fast emulation of processor ISAs on processors with another ISA (as done in QEMU [12]). A guest program (the program we want to observe) is processed by the operating system in the usual way. The binary and global data are loaded into a new process address space. For the DBI tool to come into play, it usually gets loaded as the first shared library and takes control via the initialization function of that library. Alternatively, it may do the loading of the binary itself (re-implementing this feature of the OS). The DBI tool then allocates two important data structures, a *translation table* and a *code cache* as shown in Fig. 2.5. Now, an emulation of the guest code starts by loading a virtual guest PC register with the address of the first guest instruction to emulate. Control is given to a loop in the *instrumentation engine* which runs until termination. Guest code is processed piece-wise as *basic blocks* (BB, an instruction sequence ending in a control flow change) or larger *super blocks* (similar to BBs, but possibly with conditional side exits in the middle). The translation table is looked up whether there already exists an instrumented code block in the code cache for the guest BB to run. If one is found, a direct jump into the code block happens. If there exists no instrumented version for the given BB yet, a processing of the guest BB is triggered which includes custom instrumentation as needed by the performance analysis tool. The resulting generated code block is inserted into the code cache with a corresponding entry in the translation table. At the end of every generated code block the virtual guest PC is set to the next guest instruction to execute, and a jump back into the main loop of the instrumentation engine is done.

We note that this technique never executes original code and thus cannot have the problems of code injection mentioned above. As both the original guest code and global data structures are exactly at the same addresses as in normal execution, this works transparently even with guest code generating code itself (given that code blocks are invalidated by corresponding mechanisms). For more details of the generic procedure, see [101].

2.4.3 Example Tools: Valgrind and Pin

Example tools specifically written for dynamic binary instrumentation are the Open-Source tool Valgrind[25] [86] and Intel Pin [76]. Valgrind is famous because of its memory definedness and correctness checker `memcheck`. Pin is provided by Intel free of charge. We present a short overview of the basic features and differences of both tools, as we used both tools for different prototypes of performance analysis tools in this work.

Traditional performance analysis tools do measurement or metrics collection in instrumented code. In contrast, the instrumentation for architecture simulation observes program execution and passes on events to the simulation as needed. To this end, both Valgrind and Pin provide adequate features. They enable the development of tools via well documented APIs (Application Programming Interface) in an architecture-independent way [26].

Valgrinds API mainly consists of three functions the tool developer has to write. The first is called at the very beginning to allow allocation of needed data structures. The second function is called at guest program termination to allow writing measurement results to a file. The third function is the most important one. It is called whenever a piece of guest code has to be instrumented. For this, an architecture independent intermediate representation (IR) called VEX is defined. Guest code is translated into VEX IR and forwarded to the tool's instrumentation function as sequence of VEX instructions. This sequence can be arbitrarily modified (by removing/inserting instructions). The resulting VEX IR sequence will be translated back to native machine code before it is stored into the code cache and executed. VEX IR is designed RISC-like as load/store ISA. This makes it easy to catch all instructions doing memory accesses. The instrumentation pass can allocate an arbitrary amount of temporary registers which get mapped to native registers by a register allocation pass. A specific VEX instruction is provided which results in a call back into a tool-provided C function whenever the instrumented code is executed. However, inserting plain VEX instructions lead to much faster code, as done by Valgrinds `memcheck` tool.

Pin's API is different. It does not define a virtual ISA for architecture independence. Instead, it provides an inspection API for guest instructions. While Pin internally also instruments code piece-wise (using extended basic blocks called *traces*), the tool developer is free to define own instrumentation functions which get called either for every instruction, for every basic block, or for every trace. This allows for very compact and easy to understand tool source code. However, only new instrumentation can be inserted between original guest instructions. The only instrumentation possible are

[25]`www.valgrind.org`

[26]Valgrind supports x86, x86-64, ARM, PowerPC and MIPS. Pin supports x86 and x86-64 (and did support Itanium ISA).

call-back requests into own C functions as there is no way to specify new instructions. To still enable fast execution, Pin can inline small callback functions into instrumented code. It uses register liveness analysis to avoid unnecessary register spills. In contrast to Valgrind, Pin does not change original instructions and usage of registers.

The effect of the different API approaches of Pin and Valgrind is that Pin's instrumentation phase and resulting code is very fast for simple instrumentation such as incrementing a counter for every guest instruction. Valgrind is much slower for this case. For more complex instrumentation, Valgrind gets faster than Pin as more time is spent on producing efficient code including register re-allocation.

Both Pin and Valgrind can execute multi-threaded code. Pin does locking of its data structures during instrumentation, but the instrumented code itself can run in parallel. The tool writer is responsible for correct synchronization as tool functions may be called concurrently from instrumented code. Valgrind unfortunately does not allow instrumented code to run in parallel (yet). Multiple threads are scheduled to always run sequentially in time slices. One reason for this design is that the above mentioned tool `memcheck` would not benefit from real parallel execution in threads. For each guest memory access it needs to access a shared data structure. The locking overhead and resulting serialization would make the tool slower than in the current design which does not require any locking to be done in the instrumentation.

In summary, Valgrind is nice due to its source being available. However, for the research on performance analysis for multi-threaded code, Valgrind quickly becomes very slow and is not able to simulate parallel activities. Experiments with very small time slices make Valgrind even slower. Pin instead has no problems with running multiple guest threads in parallel.

For more details, we refer to the cited references for Valgrind and Pin.

3 Simulation Driven Analysis of Sequential Memory Access Behavior

This chapter presents our performance analysis tool-suite based on cache simulation [117, 115]. The tool focuses on highlighting the memory access behavior within one thread. Running multi-threaded code is possible, but the simulation reflects a single-core system where threads are serialized by running in time-slices. The tool-suite consists of a simulator for a single 2-level cache hierarchy based on dynamic runtime instrumentation as well as a GUI front-end for presentation of the simulation results.

We first describe the motivation and concepts of the simulator which emphasizes on the importance of relating events to source by maintaining call stacks. This is also reflected in the visualization of results that is shown afterwards. The second part of this section discusses various simulator extensions which further highlight the benefits of architecture simulation.

We refer to Sec. 2.2 about a basic discussion on what users expect from a sequential performance analysis tool, what kind of bottlenecks there are, and a short list of related tools using more traditional techniques in contrast to simulation.

Goals of Simple Cache Simulation

Before describing the details of the simulation, we list the goals we want to achieve in contrast to analysis tools using real measurements. One important concept is to not try to exactly model the cache hierarchy of a real system, but to use a simple cache model. These are the goals for the cache simulation tool (we will refer to these goals later in this section):

1. The results should be easy to understand and, with the help of event attribution at instruction level, also easy to reconstruct. This is possible using a simple cache model which follows the expectations of users. Results usually match an a-priori analytical analysis of the cache behavior of a given code. This allows to better estimate the improvements one can expect from caches beforehand, and therefore it results faster in satisfying optimizations.

2. The tool should make it easy to reproduce results. For comparison of the effectiveness of a code optimization, it is far better to be able to rely on stable and reproducible measurement results. Results on real processors can vary heavily with the same code from execution to execution, depending on history and other background system activity. This usually leads to the need for averaging runtime results of time consuming, multiple runs. Instead for the simulation, it typically should be enough to look only at a few iterations of dominant kernels. This easily can compensate for slower simulation time.

3. Optimization strategies derived from tool results should be relatively architecture independent. This wish is reflected by using a cache model which approximates the memory hierarchy of different systems quite well. Optimization strategies

which work in a simple cache model typically also result in higher architecture independence of the improvements observed. Simulation allows to easily modify basic model parameters such as cache sizes. This allows to verify the effects of cache optimizations over different cache parameters. So-called *cache oblivious* algorithms are expected to exploit caches without knowing their concrete capacity. This promise easily can be checked. It may happen that optimizations derived from simple cache models do not show much benefits on specific hardware. However, the goal here is to guide users to generic optimizations.

4. In contrast to real measurement, instrumentation overhead does not influence the results of simulation. This fact can be exploited by collecting much more information about source relation of events than would be possible in tools doing measurements on real hardware. We want the tool to collect enough context information such that users easily can understand bottlenecks even in large code bases using deep call stacks. This raises the question of how much context information is useful. As measurements is aggregated by context, a large number of contexts results in amounts of data to collect which may get prohibitive. On the other hand, longer call chain contexts are very useful to break so-called recursive cycles which makes presentation of inclusive cost impossible.

5. Although using simulation, the tool should not be too slow to be inacceptable to use. By using a simple cache model, we make sure that simulation is relatively fast. Before the simulation to reach interesting parts of the code, it may be needed to run initialization code not relevant for performance analysis but still taking time in the simulation. For such cases, the tool should allow to dynamically switch to fast functional simulation to “skip” uninteresting code parts.

6. Simulation allows to have a spectrum of slightly different cache models. We want the tool to default to a simple and easy to understand model. However, there are properties of real caches which make results more difficult to understand but are crucial to performance. With simulation, we can add the flexibilty to optionally add model extensions. Users which are aware of the added complexity can enable these extensions for more exact results. By comparison of results from the more complex with the simpler model, the effectivness of the enhanced cache extensions can be studied.

The goals for the supporting GUI visualization are straight-forward. It should enable easy browsing of results, making good use of the collected information about the execution context of bottlenecks identified by the simulation.

3.1 Callgrind: a Call-Graph building Online Cache Simulator

Callgrind was developed by the author of this work to meet the goals mentioned above. It is a performance analysis tool based on architecture simulation. The simulation is execution driven and is done together with event aggregation on the fly simultaneously to the execution of the target code. The simulation is feeded by dynamic binary instrumentation at runtime. This allows the tool to work on unmodified compiled code. The dynamic binary instrumentation is provided by Valgrind. See Sec. 2.4 for a description of this instrumentation technique and a short introduction into Valgrind.

Callgrind is based on the cache simulator Cachegrind (part of Valgrind) and its cache model. It adds the ability to track any calls and returns happening in the execution of a program run. While Cachegrind provides a flat profile of the number of cache events happening in functions (ie. *exclusive* costs) in its output, in addition to that, Callgrind provides *inclusive* costs with the help of call tracking. In contrast to GProf, which heuristically calculates the inclusive cost from call counts, Callgrind directly collects it by storing the value of a global event counter when entering a function and subtracting it from the counter value at function exit.

Together with a graphical visualization of the call graph this allows to see the cost distribution starting from `main()` and going down the call chain to the function where most cost is spent. When these costs are spent deep down in some 3rd-party library, it is easy to recognize the own code which is responsible for calling the library. This exactly is the position where changes are required for improvement.

3.1.1 Cache Model and Events

The cache simulator models a synchronous two-level inclusive cache with separate L1 instruction and data caches and a unified L2 cache. *Synchronous* means that the simulator always handles cache accesses at once and there cannot be multiple access requests simultaneously in completion. In an *inclusive* cache hierarchy data fetched from main memory traverses all levels from large to small, allocating cache lines in each of the levels. This ensures that data evicted from smaller levels usually can be reloaded from larger cache levels. This means that cache lines of smaller levels are contained in higher, larger cache levels. However, the cache model does not maintain strict inclusiveness, which would mean that whenever a cache line gets evicted from larger levels it also has to be evicted from the smaller levels. This may happen when associativity of the larger level (here L2) is not enough to hold copies found in the two L1 caches.

Writes always allocate space in both L1 and L2. Caches distinguish between write-through and write-back behavior. The latter marks L1 cache lines when being modified. This allows multiple writes to be merged without involving the L2. Data only gets written back to L2 when the cache line gets evicted. However, there is no difference in cache hit and miss counts regardless of write-through or write-back behavior. As the simulator only collects hit/miss counts, the model covers both designs at once.

Both L1 and L2 caches work with user-level addresses (ie. virtual addresses). There is no simulation of a translation lookaside buffer (TLB). At every memory access, the simulator first checks the L1 cache (depending on the access either the L1 instruction or L1 data cache) for the cache-line holding the accessed address and on a miss it also checks the L2 unified cache. Whenever there was a miss in L1 or L2, space for the according cache-line is reserved, possibly evicting another line. The replacement strategy used is LRU.

This cache model resembles a simplification of the cache hierarchy used e.g. in most Intel processors[27]. Callgrind (and Cachegrind) by default check the host processor[28] for L1/L2 cache sizes, cache line length, and associativity. The idea is that the user probably wants the simulation to be similar to the reality on the processor of the machine the simulation runs on. However, these cache parameters can be explicitly given on the command line, too.

Events generated by the cache simulation are L1 hit, L2 hit, or L2 miss. For each of these three results, the type of access (instruction read, data read, or data write) is noted. Thus, there are nine different possible event types. In the output, counters for these events are given per source line and optionally per instruction address. The set of event types does not specify the kind of eviction triggered by a miss. For an L2 write-back cache, the dirtiness of a cache-line (ie. modified or not) would have influence on the cost (bus occupancy) as a modified line needs to be written back. However, because the events do not distinguish between different miss kinds, the cache model thereby does not specify whether the L2 cache is write-back or write-through: both cache types result in the same event counts.

It is important to note that the cache simulation cannot say anything about the stall time in the processor core because of cache misses. This would need a cycle-accurate simulation not only for the cache, but also for the CPU micro-architecture, for the system bus, memory controller, and DRAM chips. Aside from the fact that hardware documentation for a given processor is not available in such detail, the simulation would not be practically useful any more because of the simulation slowdown. However, as memory accesses often slow down application performance in practice, a relative reduction of L2 misses often maps to faster runtime performance. Quantitatively, one can construct a heuristic giving worst-case cache latency by measuring the worst-case miss latency of an L1 and L2 miss on a real machine, e.g. with the Calibrator tool[29], and use this as cycle estimation. This worst-case *derived*[30] event is provided as "cycle estimation" event (CEst) within the visualization GUI. To adjust cache latency values for a given processor, the coefficients in the formula of this derived event can be adapted. However, this worst-case heuristic can be wrong because of the asynchronous behavior of real caches or other application activity that can hides cache/memory access latency.

Comparison with Real Processor Caches

The following discrepancies between this simple cache model and a real processor can be found:

- Synchronicity. Any real cache hierarchy can have multiple requests in the fly,

[27]Intel processors use inclusive caches, with the recent Core micro-architecture switching from write-through to an write-back L1 cache. However, this makes no difference to our cache model. However, AMD processors always had *exclusive caches*. There, on a miss, the processor always loads data directly into L1, and lines evicted from L1 are stored in L2 which is acting as *victim cache* resulting in different content in L1 and L2.

[28]via the CPUID instruction

[29]`http://monetdb.cwi.nl/Calibrator`

[30]A derived event is an event not measured directly, but calculated from other events

handling them in an asynchronous way. Adding this property to the cache model would drastically raise the complexity of the simulator resulting in higher slowdown. However, this also would make event counts to become far more difficult to interpret and understand. E.g. 10 accesses in a row to the same not-loaded cache line result in up to 10 L1 misses and one L2 miss for the asynchronous case, depending on the timing of the accesses and the out-of-order capability of the execution pipeline. However, we expect that a typical user would be confused by such results. Showing one L1 miss and one L2 miss followed by 9 hits in this case is easier to understand.

- No simulation of *hardware prefetchers* by default. Every cache implementation nowadays includes automatic prefetching of data by predicting future accesses. Thus, program accesses eventually find the needed data already in the cache depending on the correctness of the prediction and the timeliness of pre-loading. The latter may depend on the bus load. The influence on application performance can be significant. However, the existence of hardware prefetchers is architecture dependent. Optimization based on the simple cache model will result in faster performance also on processors without prefetchers.

- No difference in events between write-through/write back L2. Cache hierarchies in all processors have a write-back behavior on the last level to reduce bus activity. Thus, writing back dirty evicted cache lines can have a performance impact. However, this happens on writes, and typically, a write transaction can be done completely in the background not influencing the runtime (in contrast to a memory load, where the value is needed for further processing).

While these differences to real processor hardware exist, any optimization which results in a reduction of cache misses within the simple model also should result in reductions of misses on real hardware. Further, it typically also results in faster runtime. However, we do not advocate for the cache simulator to replace real measurements, but guide architecture-agnostic generic optimizations. To evaluate final improvements of a optimization, real runtime measurements are always needed. Actually, the event counters of the simulation often come close to numbers of actual hardware performance counters [85].

Extensions to the Cache Model

The basic cache model of Callgrind, as described in the previous section, is the same as found in Cachegrind. In Callgrind, there are further options to extend this simple cache model in different ways. The motivation for making these extensions optional is given as goal number 6 in the beginning of this section.

- Explicit specification of write-back behavior for the L2. Each of the three L2 miss events (instruction read, data read, data write) are further subdivided into an L2 miss event with dirty vs. non-dirty state, respectively. For the cycle estimation derived event using worst-case cache latency, the formula can be extended to assume more time spent for the three additionally added L2 miss events due to writing back the modified dirty line before loading the new line. The motivation here is that there are two bus transactions instead of one[31]. While the cache extension is useful, the results often do not differ much from the basic cache model[32]. This underlines the decision to not include write-back behavior into the default cache model.

- Addition of a best-case hardware prefetcher. Every processor nowadays has mechanisms which try to predict future memory accesses and prefetch data which are expected to be accessed in the future by a program. In the optimal case, a prefetch of a memory block is completed when it is needed, thus fully hiding memory latency. More details on the hardware prefetcher extension is provided in Sec. 3.4.

3.1.2 Important Callgrind Features

Here, we discuss various Callgrind features and relate them to the goals raised in the beginning. Callgrind consists of two main features: (1) the tracking of calls for building up the call graph, getting call counts, and enabling collection of inclusive cost (goal 4), and (2) the cache simulator, which already was described. The first feature is useful on its own. As the cache simulator results in much slowdown, it is switched off by default (goal 5). Without any options, only the number of executed instructions executed (`Ir` for "instructions read from cache", which gives the same value as in the simple cache model) and function calls are collected.

Control Flow Collection

When analyzing a function alone, the control flow (e.g. number of loop iterations) is important for analysis (relating to goal 4). This needs the collection of (conditional) branches executed. The branches can be visualized next to the source code. However, there is an issue: as performance of the compiler optimized version has to be analyzed, there can be quite some restructuring done between source code flow and assembly code, resulting in confusing source annotation. For better understanding of flow control, the visualization supports control flow annotation for the assembly code (unconditional branches are shown in blue, whereas conditional branches are shown in red).

Avoiding Slowdown for Uninteresting Program Phases

Goal 5 asks for acceptable slowdown. Often, there is uninteresting initialization going on at program start. Such program phases can be executed with Valgrinds minimal

[31]Unfortunately, the previously mentioned Calibrator tool does not measure L2 misses with dirty line eviction

[32]There is an interesting exception: Searching for prime numbers using the algorithm "Sieve of Eratosthenes" has a lot of unnecessary writes where the latency can not be hidden by processors. By getting rid of them, real caches show doubled performance. To show this effect, the write-back simulator extensions has to be switched on.

slowdown of only a factor of 2 – 3 (which is the slowdown of running a code in Valgrind without any additional instrumentation). To toggle instrumentation mode, different options exist. One way is to specify a function name in a command line flag of Callgrind. Another way is to switch instrumentation mode in the target program by using so-called *Valgrind Client Request*. These are hooks compiled into the guest code which can forward information to a Valgrind tool[33].

Cycle Avoidance and Call Chain Contexts

Inclusive cost is useful in profiling as it allows users to get an overview of bottlenecks and go down call chains from main to the relevant code parts. However, as already mentioned in goal 4, so-called recursive cycles can make it impossible to visualize inclusive costs. In the following, we explain this problem and describe the solutions implemented in our tool.

An issue of general concern to profiling tools is how to handle mutual recursion. Recursion happens when one or multiple functions call each other in a mutual way. They form a so-called *cycle*. This is problematic for program analysis because inclusive cost is not defined for calls inside of a function cycle within a call graph (where a function is represented by a single node). Inclusive cost by definition includes both the cost of a function itself and the cost spent in all called functions. However, a recursive function calls itself, so this definition makes no sense. What we could do is to treat each recursion level of a recursively called function as different node in the call graph. This avoids cycles and has no issues with inclusive cost. Further, we could collapse all recursion levels of one function into one node for the real call graph. But there is a drawback: we need to keep track of all contexts of recursion levels of the function in the tool, which may be prohibitive due to resource consumption.

Instead, we do a compromise. Callgrind cannot distinguish all calling contexts in a program as they may get quite long especially with recursive calls. Instead, we ask the user of the tool to specify how long the calling contexts are allowed to get which are to be maintained. If we look at an extended call graph for a program run which has a node for each maintained calling context, we still may find cycles. Detection of such cycles is done after simulation in the visualization. If there are cycles, we do not show the measured inclusive costs (as they are bogus). However, the user can rerun the simulation specifying a longer calling context which may solve the problem.

For small cycles, not showing inclusive cost usually is no issue for the analysis. For large cycles covering much of a programs functions, the benefit of inclusive cost visualization is important. Further, the visualization uses inclusive costs to hide uninteresting portions of the call graph. Thus, with large function cycles, the call graph visualization itself is rendered useless.

To further complicate the issue, one has to realize that a profile does not provide information about the timely order of calls happening. Thus, for the visualization, we cannot detected whether existing calls really happened in a recursive fashion. For example, assume the recorded call chains $A \rightarrow B$, $B \rightarrow C$ and $C \rightarrow A$. This results

[33]It works by inserting a specific NOP sequence that is detected by Valgrind.

Command		Call chain length limit n_{max} 0	1	5	10	20
bzip2 libm.so.6	Nodes	408	850	1 329	1 332	1 132
(Compressor)	Arcs	538	688	1 113	1 113	1 113
cc1 ct_main-i.c	Nodes	1 519	5 157	22 060	41 164	44 899
(C compiler)	Arcs	6 741	10 881	34 282	52 191	54 722
konqueror	Nodes	21 500	55 829	251 449	420 871	507 507
(KDE Browser)	Arcs	51 052	90 629	315 838	470 032	544 487

Table 3.1: Number of nodes and arcs in reduced CCTs

in the recursive cycle $A \rightarrow B \rightarrow C \rightarrow A$. The visualization has to assume that this cycle really happened, and can not provide inclusive cost any more. This is true even if there really only existed the call chains $X_1 \rightarrow A \rightarrow B \rightarrow Y_1$, $X_2 \rightarrow B \rightarrow C \rightarrow Y_2$, and $X_3 \rightarrow C \rightarrow A \rightarrow Y_3$ (the assumed recursion here is called a *false cycle*). The best way is to treat each cycle with its functions forming the cycle as an artificial function with calls going into and out of the cycle. GProf does the same [45]. For cycle detection the visualization has to detect all strongly connected components in the call graph. We use Tarjan's algorithm [112] for that.

Showing large cycles is not really useful for the user. At least for false cycles, there are ways to avoid them. For that, instead of a function represented by one node in the call graph, we introduce separate nodes for each different context the function was executed in. Here, a context may specify the call chain of the function. Another refinement of the context could be to add the recursion depth of a recursive function. This strategy not only removes cycles, but also provides more information to the user. To not be confusing, a visualization of a call graph should collapse multiple contexts of the same node into one visual node by default, and only show a more detailed call graph on demand. Our visualization (KCachegrind) is able to do that.

Using the full call chain from `main` as context of functions can easily produce a huge number of nodes in the call graph. This can be reduced by only using the trailing n_{max} functions in a call chain to specify the context. In Sec. 2.2.3, CCTs were introduced. Callgrind is able to produce a "reduced" $\text{CCT}_{n_{max}}$ by using trailing call chains of maximal length n_{max}. Table 3.1 gives some numbers on the size of these reduced CCTs for various UNIX applications. Thus, users should be careful to chose a small n_{max} for a simulation run, when the originally produced call graph has cycles.

Support for Multi-threaded Code

While mainly targeting sequential performance analysis, Callgrind can produce different profile output for each thread of a multi-threaded program[34]. Similar to the calling context, the thread in which a given bottleneck is happening can be important information for the user, thus relating to goal 4.

There are two points one has to have in mind when using this feature:

[34]This mode is switched on with "–separate-threads=yes".

- The cache model is not changed. That is, Callgrind will simulate the same L1 instruction, L1 data, and L2 unified caches for all threads. This is similar to the situation where there are multiple hardware threads running on one core (such as with Intel Hyper-threading), also using the same caches. If one is only concerned about the results of the cache level directly above memory (last level cache (LLC) on-chip, here L2), the cache model can approximate shared LLC caches on recent multi-core processors quite well.

- Valgrind, the runtime instrumentation tool that is used by Callgrind, handles multiple threads not by running them simultaneously, but it does time-slice scheduling. That is, threads get serialized. The reasons for that is that Valgrind was mainly developed for being used for correctness checking. For such tools, maintaining required shared meta-data would require much synchronization among simultaneously running threads. By serializing threads, this synchronization is not needed. For Callgrind, this means that we get results which reflects time-slicing multiple threads on a single-core. This usually is not what we want: with time slices, threads can make full use of L2, while in a multi-core chip, the threads running on multiple cores would compete for space.

In summary, multi-threading support can be useful to some degree, but does not reflect multi-core scenarios, as simulation does not support simultaneously running threads. This is the reason that for the extended cache model used in Sec. 4 to study bottlenecks in multi-core processors, we cannot use Valgrind, and instead rely on Intels Pin [76], which allows for simultaneous simulation of program threads within simulator threads.

3.1.3 Implementation Details

In 2.4.3, the instrumentation process in Valgrind tools is described. For the cache simulation, we have to catch read and write operations in the IR (intermediate representation) of the piece of code that is to be instrumented. More specifically, calls to the simulator are inserted on every guest instruction, every data read, and every data write. Separate counters for each original guest instruction are maintained and incremented whenever an event occurs in the cache simulation. To be able to collect data separately for different contexts of a function, these counters actually are maintained per calling context and thread separately.

An important task for the instrumentation is to enable the construction of the dynamic call graph of the executed program. For every call site in the program, we want the list of call targets to be noted down. For each possible (call site/call target) tuple, i.e. a call arc, the call count as well as the cost spent inside the called function has to be stored. Again, these data has to be maintained per calling context and thread separately.

Thus, to be able to relate cost to functions and calling contexts, we need a reliable way to maintain the current calling context for every thread. This in turn needs mechanisms to get reliable notifications whenever a call or return is happening. In the following we go into details of the issues and solutions.

Robust Call-Graph Capturing

In the x86 ISA, there exist different functions to change the control flow. As we are interested to maintain the current calling context at function granularity, it may seem to be enough to catch executed x86 CALL and RET instructions, and update the current calling context accordingly. However, using CALL/RET instructions for control flow change among functions actually is only a convention which compilers and manually written assembler code may or may not follow. E.g., the effect of CALL and RET can be emulated with PUSH/POP and JMP instructions. Similarly, CALL and RET can be used to emulate simple jumps. Usually, there are always parts of language runtime libraries which are hand-crafted assembler code, and there may be a reason to not follow the conventions. An ubiquitous example is *tail recursion optimization*. To avoid two returns in a row (and save stack space), a function may reset the stack pointer and jump into another function instead of calling it. Further, returning from a chain of function calls at once may be done by adjusting/resetting the stack pointer before RET. This actually is the typical way exception handling works on the binary level for C++. Callgrind never should crash in such situations and always come up with reasonable call graphs. Not getting notified just once about a return can destroy the usefulness of the resulting call graph.

To this end, every control flow change has to be observed and correctly mapped to call/return semantics if required. The most robust solution found was to maintain a shadow stack which mostly mirrors the sequence of stack frames of the real stack. Each entry of the shadow stack stores a pointer to the corresponding real stack frame, allowing the synchronization of the two stacks. On a control flow change, the real stack pointer is used to check whether/how many stack frames from the shadow stack should be pop'ped. This works fine with exception handling (or use of longjmp in C). If there was no return detected, the control flow change actually may be a function call. We expect this to be the case either when a CALL instruction gets executed, or the jump target is the first instruction of a function as determined from debug information. This copes nicely with tail recursion optimization. The resulting call graph will show a call arc as specified in a high level language even though no new stack frame was allocated. This example shows a visualization problem of call graphs. A jump from one function to another cannot be represented. This is also an example where our shadow stack does not exactly mirrow the real stack. However, we automatically generate multiple returns on a following single RET in code, as we pop entries from our shadow stack until the stack pointers match.

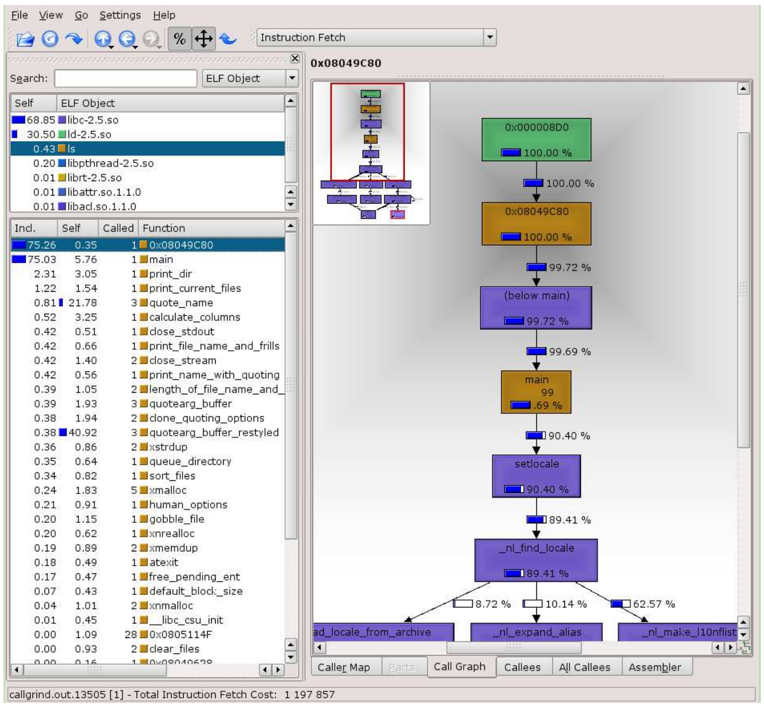
A shadow stack has to be maintained for every kernel- and user-level thread in an application. To this end, Valgrind notifies Callgrind about thread context switches. Similarly, Callgrind gets notified about entering and leaving from a signal handler. As signal handlers get called asynchronously to program execution, we do not want them to appear as part of the call graph of a regular thread. Instead, we want signal handlers to appear as roots (nodes in the call graph not called from anybody), requiring their own shadow stack. Signal handlers actually are complex. A nesting of signal handlers may be active in the context of a thread. Further, signal handlers are not required to return control flow to the operating system with a regular RET instruction. Instead, they directly can jump back into regular program code. For this reason, we put the shadow stacks of signal handlers on top of the shadow stack of the thread in which context they are executed. This way, a jump into regular thread code (with corresponding stack pointer change) will result in the signal handler frame to be pop'ped from the shadow stack. Thus, we get notified about the termination of the signal handler.

With recursive functions, call stacks can become arbitrarily deep. The number of different calling contexts easily can get very huge. As Callgrind maintains a lot of event counters per context, we must be careful to not get into an out-of-memory situation. To avoid that, for the actual calling context, Callgrind only takes the top most functions from the shadow stack into account, with the maximum number being a user-adjustable parameter.

An important implementation issue are calls into functions found in shared libraries. On Linux, they usually get resolved by the dynamic linker the first time they take place. After resolution, the linker directly jumps to the real function. As mentioned above, a jump among functions usually is resolved as call to handle tail recursion optimization. However, this can result in a call graph very difficult to read, as the first call to any shared libary function always gets routed via a central function (the runtime linker) acting as *dispatcher*. This blurs the real call relationships. Here, is is very useful to detect the specific situation of a call into the runtime linker. This allows the following jump into the real shared library function to be handled as being a return to the parent followed by the call to the function. Detecting this special case requires debug information for the runtime linker. Unfortunately, this is not provided by all Linux distributions. We had to implement a search of possible binary templates for reliable detection of the runtime linker code without corresponding debug information.

The described shadow stack handling for robust call graph capturing does not come for free and reduces the speed of the simulation. However, we found it to have much less effect on runtime than the simulator itself. It is well worth the detailed calling context information we get in return.

The current implementation works well on both x86 and its 64-bit extension x86_64. However, Valgrind also supports ARM and PPC instruction sets. Unfortunately, the described call-graph capturing often gives bogus context information for these instruction sets. The reason is that both ARM and PPC use a so-called *link register* to store the return address instead of the stack when calling into another function. This reduces the number of memory accesses required when calling into leaf functions (functions which do not call further functions). However, our shadow/real stack synchronization approach requires the stack pointer to change on returns from functions. This does not happen with said ISAs, resulting in failure of call-graph capturing. A solution may be to also store return addresses as part of the shadow stack, and assume a return if a jump to the return address at the top of the shadow stack is detected, even without a stack pointer change. However, this is future work.



Self	ELF Object
68.85	libc-2.5.so
30.50	ld-2.5.so
0.43	ls
0.20	libpthread-2.5.so
0.01	librt-2.5.so
0.01	libattr.so.1.1.0
0.01	libacl.so.1.1.0

Incl.	Self	Called	Function
75.26	0.35	1	0x08049C80
75.03	5.76	1	main
2.31	3.05	1	print_dir
1.22	1.54	1	print_current_files
0.81	21.78	3	quote_name
0.52	3.25	1	calculate_columns
0.42	0.51	1	close_stdout
0.42	0.66	1	print_file_name_and_frills
0.42	1.40	2	close_stream
0.42	0.56	1	print_name_with_quoting
0.39	1.05	2	length_of_file_name_and_
0.39	1.93	3	quotearg_buffer
0.38	1.94	2	clone_quoting_options
0.38	40.92	3	quotearg_buffer_restyled
0.36	0.86	2	xstrdup
0.35	0.64	1	queue_directory
0.34	0.82	1	sort_files
0.24	1.83	5	xmalloc
0.21	0.91	1	human_options
0.20	1.15	1	gobble_file
0.20	0.62	1	xnrealloc
0.19	0.89	2	xmemdup
0.18	0.49	1	atexit
0.17	0.47	1	free_pending_ent
0.07	0.43	1	default_block_size
0.04	1.01	2	xnmalloc
0.01	0.45	1	__libc_csu_init
0.00	1.09	28	0x0805114F
0.00	0.93	2	clear_files

callgrind.out.13505 [1] - Total Instruction Fetch Cost: 1 197 857

Figure 3.1: A screen-shot of the KCachegrind window, showing the ordered function list on the right and the call graph visualization around the active function (center of the dark background shading) on the left

3.2 Graphical Visualization: KCachegrind

In the following, we shortly describe our visualization tool KCachegrind and explain basic concepts. Fig. 3.1 shows the general layout of the KCachegrind GUI.

We wanted to give users a convenient graphical user interface to browse the results of our simulation tool. For this, we were mainly interested in how to make sure that the visualization is scalable. This is, the user should be able to easily browse simulation results from programs with huge call graphs.

The usage concept is similar to a web browser. There is always one item (function, shared library, source line) active and visualized in different ways. From the various visualizations, one can select a new item to become ative and afterwards go back to the old one. To select new items, there is a list of functions on the left, and more detailed visualizations of the active item on the right. Further, presentation is guided by a selected event type (data read accesses, cache misses, time estimation) which defines the ordering of items in lists.

Most important for supporting fast navigation to performance bottlenecks is that the list with functions can be ordered either depending on inclusive or exclusive costs. However, for better overview, the call graph visualization proved to be even more useful. As the call graph of the full program may be too large, KCachegrind never shows a global but only a local view of the call graph around the active function. To only show the interesting neighborhood around the active function, inclusive cost (of

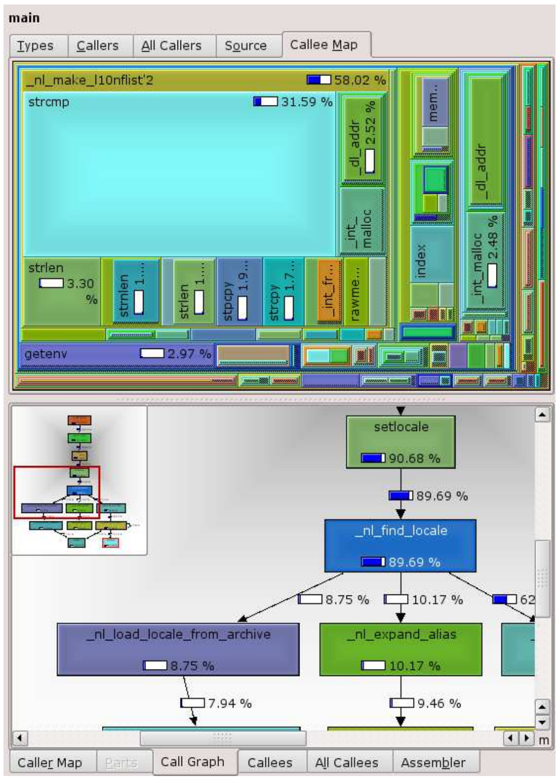


Figure 3.2: Visualization panes showing the Callee Map

the currently selected event type) is used, stopping for example when call graph nodes would represent less than 1% of the active function. Actually, this adaptive suppression of uninteresting nodes solves any scalability issues with huge call graphs.

An interesting option for scalable visualization of performance data is the so-called *Tree Map* visualization as shown in Fig. 3.2. In contrast to the call graph view, it shows the call relationship as nesting of rectangles, but in addition to that, inclusive costs become visible. Each function is represented by a rectangle with the area proportional to its inclusive cost. Called functions are drawn inside of the caller rectangle. Thus, it is easy to spot functions with large inclusive costs even if these are hidden deep down a call chain. An interesting property of the tree map here is that it adaptively reserves space to show details exactly along the call chains which lead to performance issues represented by large inclusive costs.

3.3 Usage Example

In the following, we show the influence on cache behavior of iterating over a matrix either row- or column-wise. The example code is shown in Fig. 3.3. A Callgrind run with cache simulation produces the following output:

```
weidendo@lapbode:~tmp> valgrind --tool=callgrind \
          --simulate-cache=yes --dump-instr=yes \
          --collect-jumps=yes ./matrix
```

```
==19136== Callgrind, a call-graph generating cache profiler.
...
Events    : Ir Dr Dw I1mr D1mr D1mw I2mr D2mr D2mw
Collected : 31122763 13042032 6021808 569 1125785 125317..

I   refs:      31,122,763
I1  misses:           569
L2i misses:           564
I1  miss rate:        0.0%
L2i miss rate:        0.0%

D   refs:      19,063,840  (13,042,032 rd + 6,021,808 wr)
D1  misses:     1,251,102  ( 1,125,785 rd +   125,317 wr)
L2d misses:       370,406  (   245,101 rd +   125,305 wr)
D1  miss rate:        6.5% (       8.6%   +       2.0%  )
L2d miss rate:        1.9% (       1.8%   +       2.0%  )

L2 refs:        1,251,671  ( 1,126,354 rd +   125,317 wr)
L2 misses:        370,970  (   245,665 rd +   125,305 wr)
L2 miss rate:         0.7% (       0.5%   +       2.0%  )
```

At the end of the run, the event types together with the total counts are shown, and afterwards some statistic about cache miss rates. The shown percentages always relate to number of instruction reads, data reads, and data writes issues by the processor (Ir/Dr/Dw). Therefore, “L2 rate” figures provide the rates of the full 2-level hierarchy.

When visualizing the resulting profile in KCachegrind, it is easy to see that each of the two matrix sum functions has exactly the some number of instruction executions (absolute values probably will be different depending on architecture, compiler, compiler version, version of C runtime library and so on). On the authors machine these are 11.007.012 instructions. However, it is more interesting to compare miss counts among the two versions. While `column-wise` shows a lot of L1 misses, L2 misses actually are lower than for `row-wise`. Depending on cache size and latency, the question what is best seems to depend on the architecture. However, when adding the hardware prefetcher with “–simulate-hwpref=yes”, the real advantage of `row-wise` becomes visible: all accesses can be prefetched. However, it must be noted that this is a best-case estimation.

Results for a 3D Multigrid Solver

The cache simulator was originally used in the context of the DFG project DiME (Data Local Iterative Methods For The Efficient Solution of Partial Differential Equations), which studied cache optimization of memory-intensive, iterative numerical code.

The following experiment refers to a standard and to a cache-optimized implementation of Multigrid V(2,2) cycles involving variable 7-point stencils on a regular 3D grid with 129^3 nodes. The cache-efficient version is based on optimization such as array padding and loop blocking [66]. Table 3.2 shows simulated and real events obtained

```
double A[1000][1000];

double rowwise() {              double columnwise() {
  int i, j;                       int i, j;
  double sum = 0.0;               double sum = 0.0;

  for(i=0;i<1000;i++)             for(i=0;i<1000;i++)
    for(j=0;j<1000;j++)             for(j=0;j<1000;j++)
      sum += A[i][j];                 sum += A[j][i];
  return sum;                     return sum;
}                               }

int main() {
  int i, j; double sum1, sum2;

  for(i=0;i<1000;i++) for(j=0;j<1000;j++) A[i][j]=1.0;

  sum1 = rowwise();
  sum2 = columnwise();
  return (sum1 == sum2);
}
```

Figure 3.3: Example code for Callgrind usage

	Simulation			Real Measurement		
	Instr. exec.	L2 misses	Run-time	Instr. retired	L2 Lines In	Run-time
Standard	11 879 M	751 M	1 865 s	11 879 M	777 M	40.4 s
Optimized	11 666 M	361 M	1 798 s	11 666 M	383 M	27.1 s

Table 3.2: Simulation and profiling results for the 3D Multigrid code

on a Pentium-M with 1.4 GHz with corresponding wall clock runtimes. In both cases, the advantage of cache optimization is obvious by effectively reducing the number of cache misses/cache line loads by 50%. Reduction of runtime gives similar figures.

Simulation runtimes show that the slowdown of runtime instrumentation and cache simulation is quite acceptable. In the example, running the standard code has a slowdown factor of 46, and running the optimized version has a slowdown factor of 66. The general observation here is that code which does not exploit the cache can be simulated faster than well optimized code. The reason is that the simulator itself exploits caches quite well and can hide latencies of non-optimized code. In contrast, simulation of already optimized code resuls in the cache simulator and the user code to compete for cache space.

3.4 Extension: Analyzing Hardware Prefetching

One benefit of caches is that they exploit exposed data access locality of programs. As already mentioned in Sec. 2.1, another advantage is buffering of pre-fetches. In the following, we describe the addition of hardware prefetch simulation and a cache optimization using software prefetching [121]. The latter method resulted in a framework which allows to send commands for software prefetching to a separate prefetcher thread. For matrix multiplication and a Jacobi solver, we showed the benefits in [122].

Suppose a memory access needs 50 ns. For a memory connection able to provide 10 GB/s, 500 bytes (50 ns · 10 GB/s) can be transferred waiting for one memory access. However, that memory access fills only one cache line, and thus, serialized memory accesses would only exploit 13% (64/500) of available bandwidth (assuming 64 byte cache lines). This should make it clear how essential prefetching is to exploit bandwidth resources. While programmers/compilers may do prefetching in software, it has drawbacks: the additional instructions consume both decoding bandwidth and hardware resources for processing outstanding loads. However, modern processors do automatic hardware prefetching for specific memory access patterns such as sequential streams, also in the form of strided accesses (where the difference between successive addresses are fixed). See [16] for an overview of well known prefetching algorithms. Advanced algorithms are presented in [11]. Hardware manuals for processors often describe the kind of prefetchers provided [56], although without much detail.

While it is helpful to layout data structures in such a way that accessing them enables hardware prefetchers to do their work, this is not always possible. In that case, it still may be worthwhile to fall back to software prefetching.

To this end, hardware performance counters, showing hardware prefetch activity, can help. In the following, we implement a prefetcher algorithm as extension of Callgrinds cache model, and compare its result with the ones from an Intel processor. We expect that the simple stream detector algorithm implemented is in similar form available in all modern processors. Thus, simulator results are not tied to a concrete processor model.

We test our simulator within the scope of a software prefetch technique. When the performance of a code section is limited by available bandwidth to main memory, prefetching itself cannot do any better. The first step to improve this situation is to use loop blocking [67]. Instead of going multiple times over a large block of data not fitting into the cache, we split up this block into smaller ones fitting into the cache, and go multiple times over each small block before handling the next one. Only the first sweep over a small block has to fetch data from main memory. Still, this first traversal is limited by main memory bandwidth. As further sweeps on the small block do not use main memory bandwidth at all, we can use that to simultaneously do software prefetching on the next block. However, now two blocks need to fit into the cache. We call this technique *Interleaved Block Prefetching*.

3.4.1 Stream Detection Algorithm

The implemented hardware prefetch algorithm gets active on L1 misses: for every 4 kB memory page, it checks for sequential loads of cache lines, either upwards or downwards. When three consecutive lines are loaded, it assumes a streaming behavior and triggers prefetching of the cache line which is a given prefetch distance away (e.g. 5 lines in advance). Prefetching another line is only done when the next access into the detected stream is observed. Further, prefetching is stopped when we reach the 4 kB page boundary (going further would need a TLB lookup which prefetchers usually avoid). As the simulator has no way to do timings, it is assumed that prefetched lines become available in L1 immediately.

The best prefetching distance to use (here 5 cache lines) depends on hardware parameters of the system and empirical assumptions on program code. It should be large enough such that when the real access to this cache line by the program is expected, the line actually is already loaded from main memory into L1 cache. In the above example of a 10 GB/s connection to main memory with 50 ns memory latency, we would need to always have 8 cache line loads outstanding for saturating the connection. Taking into account that not all memory accesses are part of a stream and further, multiple streams could be done by the program simultaneously, 5 cache lines in advance should be enough. If we choose the prefetch distance to be too large, we run into another problem. With automatic hardware prefetchers, the accuracy of prediction is very important. With a prefetch distance of 5 cache lines, whenever a prediction goes wrong, we load 320 bytes too much into L1, wasting both bandwidth and polluting L1 cache capacity.

Although the described design is arguable, it fulfills our requirement that every sensible hardware prefetcher should at least detect the same streaming behavior as our prefetcher. Further, the simulation will show code regions where such simple hardware prefetchers have problems and should be supported by software prefetching.

3.4.2 Results on a 3D Multigrid Solver

The following results have been measured for a cache-optimized Fortran implementation of Multigrid V(2,2) cycles, involving variable 7-point stencils on a regular 3D grid with 129^3 nodes. The Multigrid cycles does two smoothing steps on the finest grid, resulting in two sweeps over the 3D grid in the non-optimized version. These two sweeps get merged in the cache-optimized version. When comparing the two versions, we indeed can observe both for the simulated as well as real measurement that little more than half the number of L2 misses / L2 lines read are done in the cache-optimized variant in contrast to the non-optimized variant. This shows that the blocking works quite well.

In search for further improvements using prefetching, Table 3.3 shows values for the most relevant two functions (RB4W and RESTR) as well as for the complete run. RB4W is the red-black Gauss-Seidel smoother, doing the main work, and RESTR is the restriction step in the Multigrid cycle. We first show the number of simulated L2 misses, first column with hardware prefetching switched off, second column with hardware

	Simulated			Real		
	L2 Misses		Pref.	L2 Lines In		Pref.
	Pf. Off	Pf. On	Req.	Demand	Pref.	Req.
Summary	361	277	110	241	130	373
RB4W	233	226	-	201	47	270
RESTR	108	37	-	28	76	92

Table 3.3: Measurements for the Multigrid code [MEv].

prefetching switched on (i.e. only misses that are not caught by the prefetcher), and third column the number of prefetch actions issued. For the latter, we only show the total number, as the simulator does not split this up between source functions. For real measurements, the Pentium-M can count “L2 lines read in” due to L2 demand misses (“demand” means “triggered by program code”), shown in column 4. Further, we measure “L2 lines read in” because of requests from the hardware prefetcher (in column 5). The last column shows the number of prefetch requests issued by the real hardware. All numbers are given in millions of events (M-events). We note that the simulated numbers cannot directly be compared with real numbers, as they show different things. For real hardware, we cannot switch of the prefetcher. But the “L2 lines loaded in” on real hardware, when summing up demand- and prefetch-triggered numbers, match quite well with the total number of simulated L2 misses (prefetching off).

Simulation results show that the function RESTR is hardware-prefetcher friendly, as L2 misses go down by a large amount when the prefetcher is switched on in the simulation (from 108 MEv to 37 MEv). In contrast, for the first function, RB4W, our simulated prefetcher does not seem to work. Looking at actual hardware, the prefetcher in the Pentium-M works better than the simulated one: it reduces the numbers of L2 demand misses (column 4) even more than the simulated one (column 2). Especially it is doing a better job in RB4W (201 MEv instead of 226 MEv in the simulation). Still, as in the simulation, hardware prefetching does not really work for RB4W (only 47 MEv L2 loads are triggered by the hardware prefetcher instead of the 201 MEv loads triggered by the program itself). This shows that our simulation can show the same trend as will be seen in real hardware regarding friendliness to stream prefetching.

3.4.3 Interleaved Block Prefetching

Pure block prefetching means prefetching multiple sequential cache lines before actually using them afterwards. In contrast, interleaved block prefetching (IBPF) prefetches a block to be used in the future interleaved with computation on the previous block. One has to be careful to not introduce conflicts between the current and the next block.

Interleaved block prefetching is best used with the standard loop blocking cache optimization. When a program has to work multiple times on a huge array not fitting into cache, the data have to be loaded every time from main memory. When data dependencies of the algorithm allow to reorder work on array elements such that multiple

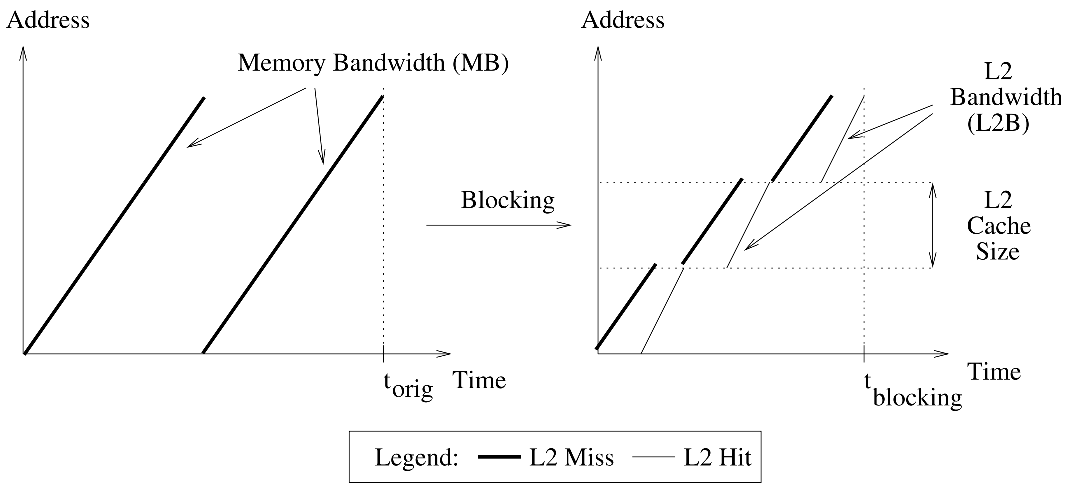


Figure 3.4: The blocking transformation.

passes can be done consecutively on smaller parts of the array, blocking is possible. This improves temporal locality of memory accesses, i.e. accesses to the same data happen near to each other. By using block sizes that fit into cache (usually the cache above main memory), this means that only for the first run over a small block, it has to be fetched from main memory. Fig. 3.4 depicts this scenario on a 1-dimensional array. With the assumption that the code is memory bound and just accessing the given data, we can estimate the performane benefit. The time $t_{blocking}$ is substantially less than t_{orig}. With blocking, the second iteration of each block can be done with L2 cache bandwidth $L2B$ instead of memory bandwidth MB, and thus gives us a speedup of $Sp = \frac{MB}{L2B}$. More general, blocking of N iterations can be done by loading the first pass of each block from main memory and the other $N-1$ passes for L2. Thus, we get

$$t^N_{blocking} = t_{orig} \cdot \left(\frac{1}{N} + \frac{N-1}{N} \cdot \frac{L2B}{MB} \right).$$

As further runs on the small block do not use main memory bandwidth at all, this time can be used to prefetch the next block. This needs the block size reduced such that two blocks fit into cache. Further, apart from the first block, we can assume a block to be already in the cache when work on it begins. With N passes over this block, we can interleave prefetching of the next block with computation over these N passes. For the Multigrid code, $N = 2$, i.e. the resulting pressure on memory bandwidth should be almost cut in half. Fig. 3.5 shows this scenario. If we suppose that the needed bandwidth for prefetching is below the memory bandwidth limit, almost all the time we can work with the L2 bandwidth $L2B$ depending on the relation of the first block size to the whole array size as shown in the figure. Note that for the first block, memory pressure will be higher than before because in addition to the first block, half of the second block gets prefetched simultaneously.

In the following, a 1-dimensional streaming micro benchmark is used: it runs multiple times over an 8 MB-sized array of floating point values of double precision, doing one load and 1 or 2 FLOPS per element (running sums). Table 3.4 gives results with

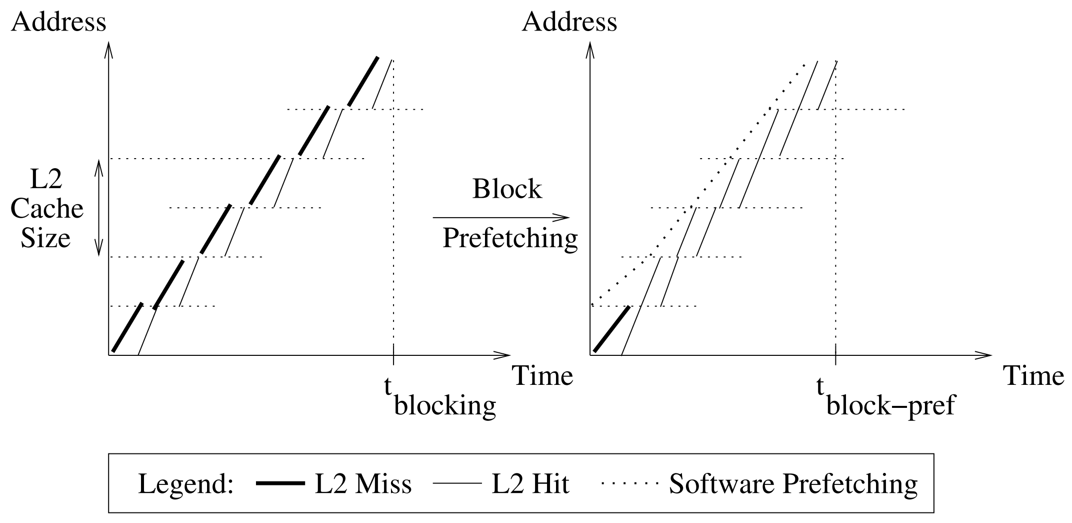


Figure 3.5: The interleaved block prefetching transformation.

1-dimensional blocking for different numbers of N. With pure blocking, $N = 1$ is a degenerated special case, as data is not reused. Still, block prefetching, which is the same as normal prefetching in this case, obviously helps. Again, these results were measured on a 1.4 GHz Pentium-M with DDR-266 memory, i.e. a maximum memory bandwidth of 2.1 GB/s.

For each benchmark and $N \in \{1, 2, 3, 10\}$ we show the runtime in seconds and the achieved (netto) bandwidth from main memory. The netto bandwidth is calculated from the runtime and the known amount of data loaded. A brutto bandwidth that gives the bandwidth needed from memory to L2 can be calculated from the number of memory read burst transactions done, measured by a performance counter. The benchmark only loads data; cache line loads are translated to burst requests. The brutto bandwidth is always between 50 and 100 MB/s higher than the netto bandwidth in this benchmark and therefore not shown in the table.

Measurements show that IBPF improves runtimes by up to 46 % compared to runtimes without IBPF ($N = 2$, 1 Flop/Load). The effect of IBPF on the hardware prefetcher can be seen in the numbers for “L2 lines read in” due to demand accesses (ie. triggered by the program) on the one hand and hardware prefetch requests on the other hand (shown only for the 1 Flop/Load case). Obviously, the software controlled prefetching virtually switches off the hardware prefetching.

One has to note that interleaving prefetching with regular computation can increase instruction overhead: in the micro-benchmark, an additional loop nesting level had to be introduced for inserting one prefetch instruction each time $N * 8$ floating point elements are summed up: with a cache line size of 64 bytes, 8 double-precision values fit into one cache line, and with blocking of N iterations, the prefetching stream advances with $1/N$ of the computation stream. In more complex cases or with other CPUs, the instruction increase can cancel the benefit of interleaved block prefetching. Further, inserting prefetch instructions for single accesses complicates the code considerable. To target both these issues, it would be good if future architectures provide other means

	Without Pref.				With IBPF			
N	1	2	3	10	1	2	3	10
1 Flop/Load								
Runtime [s]	5.52	**3**.92	3.45	2.73	4.89	**2**.67	2.50	2.44
MFlop/s	194	274	311	393	220	402	430	440
Netto [GB/s]	1.48	1.04	0.79	0.30	1.68	1.53	1.09	0.34
Demand [MEv.]	4.2	2.1	1.4	0.4	134	67	45	13
Pref. [MEv.]	130	65	44	13	0.45	0.54	0.27	0.15
2 Flops/Load								
Runtime [s]	7.20	5.24	4.81	3.23	6.86	4.29	3.60	2.84
MFlop/s	298	410	446	665	313	501	597	756
Netto [GB/s]	1.14	0.78	0.57	0.25	1.19	0.95	0.76	0.29

Table 3.4: Results of a micro benchmark.

for software controlled prefetching such as prefetching instructions for memory ranges.

3.5 Extension: Cache-line Utilization

An important benefit of a simulator is that the model state can be observed in detail. This allows for much more insights than what we could measure with real hardware. In the following, we propose metrics derived from cache simulation whichs go beyond simple cache hits/miss counters. Among other benefits, they enable programmers to understand intra-cache-line usage [120].

Cache misses are symptoms of memory accesses with bad locality. Let us assume an example of bad locality. Some data at address *a* is accessed multiple times in a row. But the time inbetween each of the accesses is so long that every access to *a* actually results in a cache miss. However, the cache misses which evict the data at address *a* may happen due to accesses to another address *b* which conflicts with address *a*. Accesses to address *b* actually could be quite frequent with good access locality.

A traditional performance analysis tool (doing measurements on real hardware) easily can identify the cache misses. It will show the memory accesses to address *b* as the reason for the cache misses, as these are the ones which missed the cache. However, as just mentioned, access behavior to *b* may be fine. What we really want is that the tool shows us that we should improve access behavior to *a*.

Performance analysis tools using real hardware measurements cannot do this as they are only activated after a cache miss already happened. At this time, they cannot get the information about what cache line was evicted. Instead, they only can tell which memory block missed the cache and triggered some eviction of old data. In fact, typically, tools only show the code position of the memory access which triggered the eviction which is equally wrong. In effect, the analysis tool shows misleading information.

This example may seem to be contrived, and in fact, memory blocks missing the cache (ie. detected by the analysis tool) always also are subject to eviction themselves.

Thus, the described misleading information by performance analysis tools may not be a big issue in practice. Still, there is a difference between data accesses which trigger evictions and data which gets evicted. By relating cache misses to code positions where the *evicted lines were loaded*, a performance analysis tool would be more precise by actually pointing at the memory accesses which need to be optimized.

Fig. 3.6 shows our example in more detail together with an optimization which gets rid of two misses. We want the tool to show a (or the code accessing a) as hint for the programmer.

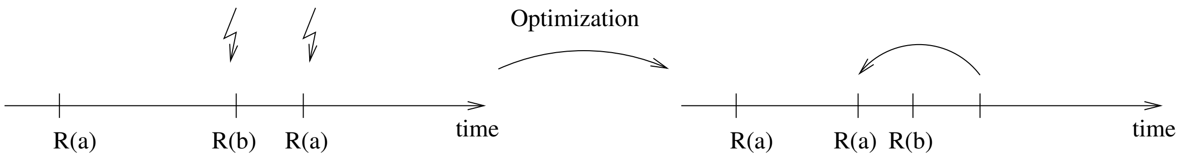


Figure 3.6: Optimization by reordering accesses of a (b conflicts with a)

Cache reuse metrics are good to provide more information about cache utilization. The average temporal reuse of a memory block by a cache is given by the number of times the cache was accessed for the memory block divided by the number of times the block was loaded from the next memory hierarchy level. Spatial locality can be measured by the percentage of bytes accessed in a loaded memory block before it is evicted again. Finally, an interesting metric is the stack reuse distance of two sequential accesses to the same memory block: this is the number of memory blocks accessed in-between [18]. This metric, assuming full associativity, is independent of cache size, and gives a precise prediction of when an access will be a cache miss (see Sec. 2.1.2). All these metrics are useful if related to the code positions where the memory block was loaded, and not where the block was evicted (the latter is not applicable to stack reuse distances).

The extension to Callgrind, described in the following, provides metrics for temporal reuse and spatial use of cache lines related to the code positions where they where loaded.

3.5.1 Additions to Cache Simulation

To allow for cache block utilization metrics and its relation to the code positions that loaded the block, a further data structure was added for every cache block of the L1 and L2. This structure contains the following data for a memory block which currently resides in the corresponding cache:

- the code position of the instruction that recently has filled the block (i.e. the call chain and instruction address),
- a reuse count, and
- a block mask, able to hold the bytes that were touched at least once (a 32bit mask is used, and thus, for 64 byte cache lines, this mask gives a 2-byte granularity).

```
double A[1000][1000];

double rowwise() {               double columnwise() {
  int i, j;                        int i, j;
  double sum = 0.0;                double sum = 0.0;

  for(i=0;i<1000;i++)              for(i=0;i<1000;i++)
    for(j=0;j<1000;j++)              for(j=0;j<1000;j++)
      sum += A[i][j];                  sum += A[j][i];
  return sum;                      return sum;
}                                }
```

Figure 3.7: Example: 2D Sweep for Reuse Metrics

For every data reference, the runtime instrumentation provides the address and length of the access. These are used in the cache simulation. For an L1 hit, the reuse count and the mask of the according reuse info structure is updated. For an L1 miss, the mask is converted to a byte count, and this and the reuse count are added to counters which are attributed to the stored code position, called "Temporal Reuse L1", and "Spatial Reuse L1". The to-be-evicted block can also be found in the L2. By maintaining according pointers, we can update the reuse metrics for the block in L2: the reuse count is added to the L2 reuse count, and spatial use masks are OR'ed. Afterwards, the L1 reuse counters are initialized for the new data which got loaded on the L1 miss. When a line is also evicted from L2, reuse metrics are similarly attributed to counters called "Temporal Reuse L2", and "Spatial Reuse L2".

It would be quite easy to relate the reuse metrics to code position tuples, showing both the code which loaded the data that got evicted and the code which did the memory access that caused the eviction. This is future work.

3.5.2 Results for Example Code

Cache reuse metrics can pinpoint the following problems:

- code regions with low spatial access locality,
- code regions with low temporal locality.

Low spatial and temporal locality is only important if the number of memory accesses in these regions make up a large portion of all accesses of the application run.

By default, the cache simulator uses cache attributes for L1 and L2 which are the same as on the machine where the simulation is run. In the following examples, we use a Pentium M with 32 KB L1 and 1024 KB L2 cache size; line size is 64 bytes and associativity is 8 for both.

The code examples in Fig. 3.7 iterate over a 2D matrix in different index order. Table 3.5 shows the difference of cache reuse metrics annotated to code using good (`row-wise`) and bad (`column-wise`) index order. Loads for index variables are not visible because they are allocated to registers. Thus, apart from read accesses to

	L1 Loads	L1 Spat. Reuse	L1 Temp. Reuse	L2 Loads	L2 Spat. Reuse	L2 Temp. Reuse
row-wise	62 500	100 %	16	62 500	100 %	16
column-wise	999 992	6 %	1	60 190	100 %	16

Table 3.5: Iteration over a 2D matrix

	L1 Loads	L1 Spat. Reuse	L1 Temp. Reuse	L2 Loads	L2 Spat. Reuse	L2 Temp. Reuse
unblocked	1 123 993	68%	2.65	176 398	94%	15
blocked	219 513	81%	13.0	175 791	100%	16

Table 3.6: Results of unblocked and blocked variant.

`matrix[i][j]`, no L1 misses are observed. For the code with bad index order, we observe that quite some columns fit into the L2 cache. Therefore, the L2 load number for the bad index order is smaller than the L2 loads for the good order. However, the reuse of the L2 cache is optimal in all cases. Thus, this example shows bad behavior only in respect to the L1 cache.

```
for(i=0;i<1000;i++)          for(ii=0;ii<1000;ii+=50)
 for(j=0;j<1000;j++)          for(jj=0;jj<1000;jj+=50)
  a[i][j] += b[i][j] *          for(i=0;i<50;i++)
             b[j][i];            for(j=0;j<50;j++)
                                  a[i+ii][j+jj]   +=
                                    b[i+ii][j+jj] *
                                    b[j+jj][i+ii];

       (unblocked)                   (blocked)
```

Figure 3.8: Example: Both row-/column-wise access, unblocked/blocked

The second pair of example codes in Fig. 3.8 shows code that needs both row- and column-wise access at the same time. The only option for cache optimization here is to do blocking as shown on the right. Table 3.6 shows the differences of cache reuse metrics for the unblocked and the blocked version of this code. L2 reuse metrics show a quite good behavior because of the small matrix size.

3.5.3 Visualization

For larger programs, a user has to locate the code regions with bad locality before being able to look at details such as outlined above. The bottlenecks “bad locality” and “low reuse” need to be easy visible in large lists showing corresponding “cost” of functions. The best approach is to map bad behavior to large cost as we do with cache miss counts. This way, the user will find the bottlenecks at the top of the function list if this list is sorted by descending costs.

More concretely, to identify code with low temporal locality, we use as cost

$$C_{tempReuse} = MissCount/TemporalReuse\,,$$

and for low spatial locality

$$C_{spatReuse} = MissCount * (1 - SpatialReuse)\,.$$

3.6 Extension: Refined Context Presentation

An advantage of simulation based performance analysis is that we can spend some effort at runtime in providing good context information, as this never influences measurement results. Callgrind already allows for call chains as context. These actually are encoded into function names. In the following, we show how the context can be refined in more complex ways for specific use cases [69].

For program codes consisting of millions of lines of code, writing fast code is a complex challenge. Profiling is the tool of choice to identify the code parts and to get hints on how to do an optimization. However, use of direct or indirect recursion can make the results of a profiler difficult to understand. To provide useful data such as inclusive cost, functions which are part of recursive cycles need to be suppressed by introducing artificial cycle functions comprising all functions of this recursive cycle. They are identified by strongly connected components in the dynamic call graph [45]. Unfortunately, this throws away a lot of information and, sometimes, renders results useless for analysis. The solution is to relate event counts measured by the profiling tool not only to the function name, but to a more elaborated context which better describes the code position. A similar argument holds for non-recursive functions whose runtime behavior depends significantly on their arguments.

Thus, it is useful to enhance the function context by incorporating a program state. To distinguish a function with respect to a state, it has to be specified by the user before entering the function. A natural choice is to use the value of function arguments. In the output of the profiling tool, the same function with different states then is treated as individual symbols. Thus, cost is also assigned to different symbols. This is the main idea of *argument controlled profiling.*

3.6.1 Detailed Context in other Analysis Tools

The simplest useful result of a profiling tool is the *flat profile.* All performance cost such as time (clock ticks) or cache events is related to code which triggered the cost. However, entries in a flat profile may point to code parts which reside in a library unavailable for modifications, or they carry no potential for optimization in themselves. Here, tracking back the call stack for tunable code parts, i.e. lines which invoke the expensive functions, is valuable. For this, the context is enriched by the call path: it becomes a calling context. A lot of papers in the area of profiling tools discuss the efficient collection of full call paths (starting at the entry point of the program), using precise yet low-overhead measurement strategies [5, 103, 42, 129, 23]. Many profiling tools provide the calling context on request [71, 97, 117, 107, 31].

For tools performing real-time measurements, there is always a trade-off between accuracy and completeness because of the influence of measurement overhead. A good compromise is to allow selective instrumentation. The developer helps minimizing overhead by providing hints on what type of information she is looking for before the measurement starts. The TAU Performance System [97] implements so-called *phase-based profiling* [79]. The developer can indicate logical phases of her program by calling a profiling interface (API). These phase names, arbitrary strings, can be constructed to contain function arguments. In contrast to our approach, this technique requires the user to manually modify the source code and recompile it. Another selective instrumentation method is *incremental call path profiling* [17], where specified functions are dynamically instrumented to do full stack walks for determining the calling context.

Sampling is a strategy not based on instrumentation. It allows the overhead to be adjustable, but provides statistical results. Only every n-th occurrence of an event of interest is collected. To get the calling context at sample points, stack walks at arbitrary points of execution need to be supported. Some tools such as Intel's PTU [31] rely on debug information generated by the compiler. However, not all compilers produce reliable debug information. By contrast, hpcrun [107] uses binary analysis to generate meta information needed for stack walks.

A lot of the tools mentioned above preserve recursive invocations in the calling context. However, if it is known that the behavior of a recursive function does not depend on recursion depth, an overwhelming amount of unneeded information is generated.

3.6.2 Usage and Example

Argument controlled contexts allow to distinguish profiling results of the same function according to function arguments. Yet, this can not be done for every argument-value combination of every function, as this would lead to an explosion of profiling results. Instead, the programmer needs to specify that she wants a distinction for a given function according to selected values of arguments.

In our prototype extension of Callgrind, we support the command line option

`--separate-par=`*function*`:`*num*[`:`b_1[`,`b_2...]]

Here, $function$ is the name of the function for which we request the creation of an enhanced context: the *num*-th 4-byte-value on the stack, interpreted as 4-byte integer value (which also works for booleans), is to be incorporated into the context name. Optionally, b_i values can be given, representing bucket borders. E.g. for value x and borders $(5, 10)$, the ranges $x <= 5$, $5 < x <= 10$, and $10 < x$ are distinguished.

We consider the following 2-dimensional Fibonacci-like function

```
int fib2(int i, int j) {
  if ((i<2) || (j<2)) return 1;
  return fib2(i, j-1) + fib2(i-1, j) + fib2(i-1, j-1);  }
```

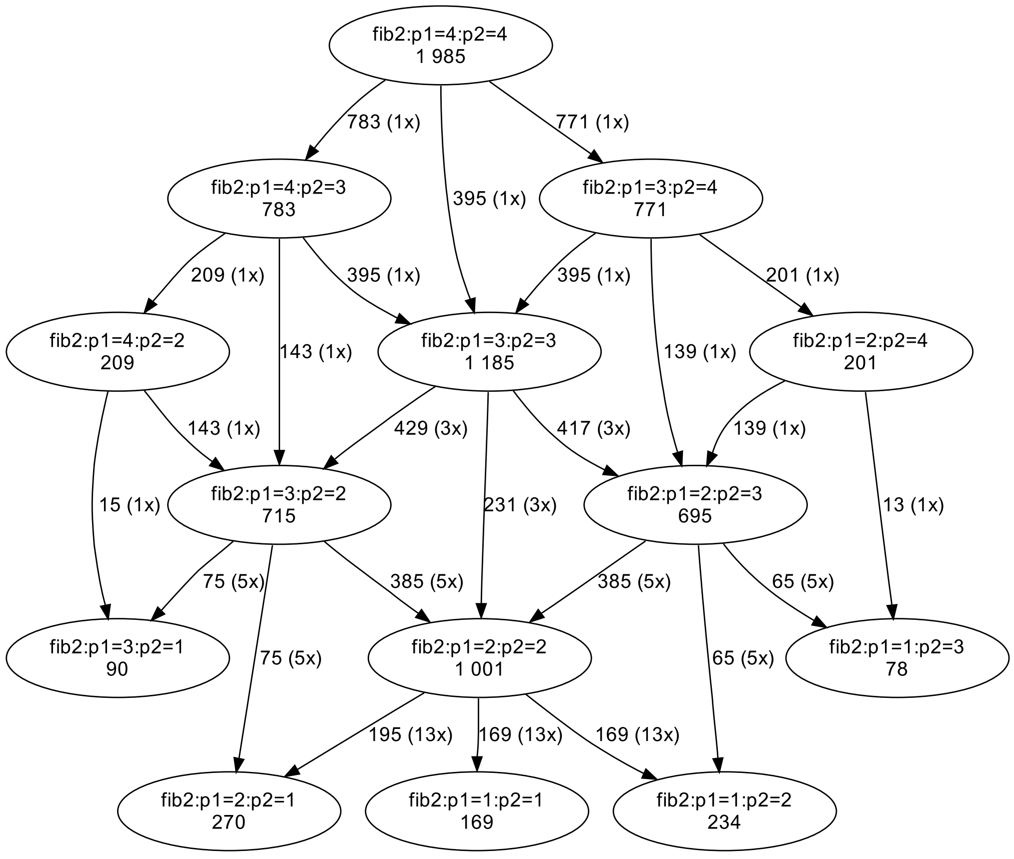


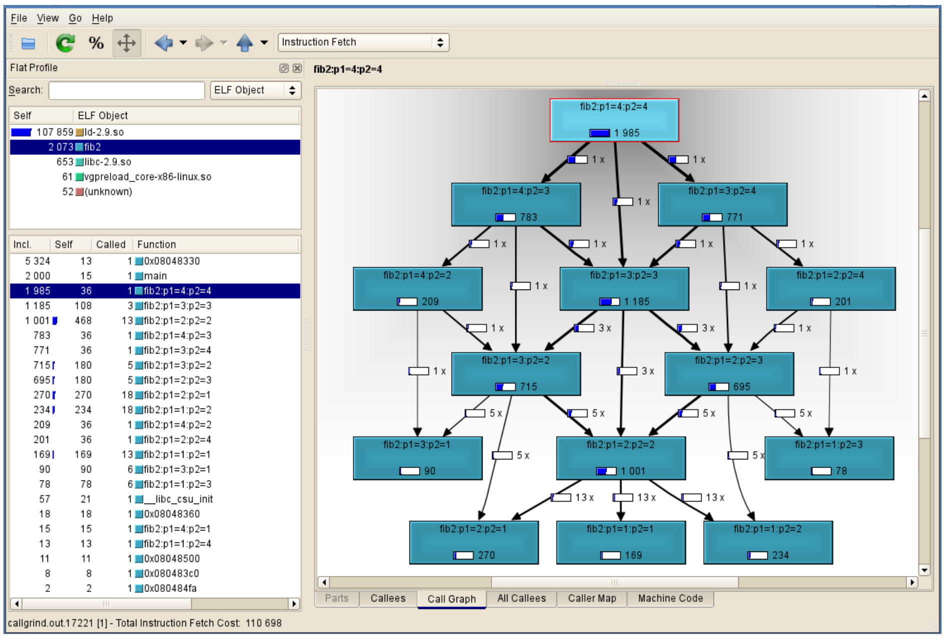
Figure 3.9: The call graph of `fib2(4,4)`.

together with the `main` function calling `fib2(4,4)`. Executing

```
> valgrind --tool=callgrind \
          --separate-par=fib2:1 --separate-par=fib2:2 ./fib2
```

produces among others profiling data visualized in Fig. 3.9, if we select the symbol "`fib2:p1=4:p2=4`" to be displayed. Each node represents a profile for function `fib2`, but is distinguished by the requested argument values. Edges between nodes represent calls with the call count shown in parenthesis. The number given in the nodes and next to edges is the inclusive cost for the event "Instructions Executed". This is the event collected by Callgrind when cache simulation is switched off.

Fig. 3.9 is generated by the "Export Graph" functionality of our profile visualization tool KCachegrind [117]. The screen-shot in Fig. 3.10 shows on the left a list of functions (with `fib2:p1=4:p2=4` selected). The functions are grouped according to the "ELF object" they reside in. These groups are shown in the list above the functions. The selected group "fib2" is the main executable, i.e. the function list only shows functions residing in the binary "fib2". On the right, the interactive call graph visualization



Self	ELF Object
107 859	ld-2.9.so
2 073	fib2
653	libc-2.9.so
61	vgpreload_core-x86-linux.so
52	(unknown)

Incl.	Self	Called	Function
5 324	13	1	0x08048330
2 000	15	1	main
1 985	36	1	fib2:p1=4:p2=4
1 185	108	3	fib2:p1=3:p2=3
1 001	468	13	fib2:p1=2:p2=2
783	36	1	fib2:p1=4:p2=3
771	36	1	fib2:p1=3:p2=4
715	180	5	fib2:p1=3:p2=2
695	180	5	fib2:p1=2:p2=3
270	270	18	fib2:p1=2:p2=1
234	234	18	fib2:p1=1:p2=2
209	36	1	fib2:p1=4:p2=2
201	36	1	fib2:p1=2:p2=4
169	169	13	fib2:p1=1:p2=1
90	90	6	fib2:p1=3:p2=1
78	78	6	fib2:p1=1:p2=3
57	21	1	__libc_csu_init
18	18	1	0x08048360
15	15	1	fib2:p1=4:p2=1
13	13	1	fib2:p1=1:p2=4
11	11	1	0x08048500
8	8	1	0x080483c0
2	2	1	0x080484fa

callgrind.out.17221 [1] - Total Instruction Fetch Cost: 110 698

Figure 3.10: Screen-shot of KCachegrind with call graph view.

for the selected function is displayed. This is just one of many visualization types supported by KCachegrind, such as caller/callee lists, *tree map* visualizations of the callee tree, or source/assembly annotations.

3.6.3 Implementation

For Callgrind, it is important to be able to relate the effect of every memory access to a context (e.g. function name, calling context, thread number, and so forth). Thus, it is helpful to have the context readily available whenever an event is triggered. The calling context is updated every time a function is entered or left. This is done by code inserted by runtime instrumentation. The first time a function (more precisely, a *basic block*) is executed, Valgrind forwards the code to Callgrind which inserts instrumentation as needed. The instrumented code then is put into a *translation cache* and executed instead of the original. At instrumentation time (i.e. only once per *basic block*), the function name is related to the code by looking it up in the debug information. At execution time, inserted code can easily detect when the currently active function changes. Whether this change was triggered by a call or return event is heuristically determined with the help of the instruction opcode, the value of the stack pointer, and a shadow call stack data structure. Whenever the tool detects that a new function is entered or left, the above mentioned calling context is updated.

We modified the existing implementation in the following way to allow for argument controlled profiling. On entering a function selected by `--separate-par`, the inserted code does not directly push the symbol name of the entered function to the calling context, but generates a new symbol by appending argument value information to the function name. At this point in time, the value of the stack pointer is readily available to allow access to the function arguments. Finally, the new symbol name is used to update the calling context.

3.6.4 Case Study

The tool was used for optimizing a PDE solver working on adaptive Cartesian grids [124]. The solver traverses a tree-like data structure with a recursive algorithm which is split up among several routines. The computational cost may be unequal partitioned among used functions, as the grid changes due to refinement as well as coarsening. We got separate profiling results by making use of two parameters for context refinement: one specified, whether grid changes are required, the other, whether we are about to enter a leaf without further recursive invocations.

The results showed that a specialization of the functions for different scenarios, identified by the parameters, indeed was able to improve performance significantly. With relatively low investment, it was possible to reduce the code's runtime by more than 16%.

3.6.5 Conclusion on Argument-Controlled Profiling

Argument controlled profiling is a useful feature for analyzing code which makes heavy use of recursive functions. Such code structures are e.g. found within sophisticated PDE solvers where the use of recursive data structures is mandatory for the implementation of adaptive grids. Another candidates for argument controlled profiling are compilers or expression tree evaluation working on abstract syntax trees (AST). The approach is valuable in finding often used node type sequences which are candidates for new *super node* types, potentially shrinking the AST—and thus, traversal time—significantly.

4 Simulation Driven Analysis for Multi-threaded Programs

In this section, we first discuss how well performance bottlenecks in multi-threaded code actually can be detected by using simulation with relatively simple architecture models. Then, we look at a specific issue called *false sharing*. This effect significantly can reduce the performance of multi-threaded applications. We show how to estimate the number of false sharing events happening in a given application run.

Finally, we describe a kind of analysis that fits quite well with the constrains dictated by simple architecture models in the context of multi-threaded code. The analysis detects the physical data paths in the memory hierarchy of a multi-core processor which need high bandwidth capacities to not slow down a program. This way, the proposed tool for this analysis can pinpoint code positions which expose the most dominant bottlenecks regarding bandwidth requirements. Programmers can check corresponding code regions for possibilities to introduce cache optimization such as blocking, padding or data layout changes.

4.1 Bottlenecks Approachable by Simple Models

The main issue of simulation based performance analysis is the slowdown experienced by users when executing the program to analyze under supervision of the tool. The focus of this work is about understanding which kind of performance analysis can be done within the restriction of using simple models to keep simulation acceptable fast.

However, performance bottlenecks in the memory hierarchy may depend on temporal interleaving of actions in threads which run on different cores. Without sophisticated complex architecture models including pipeline behavior, it is not clear that effects happening on real machines are reflected in a simulation in such a way that results are useful for performance analysis. That is, if the existence of a given kind of bottleneck is sensitive to the ordering of actions from different threads, any prediction of the relevance of resulting bottlenecks from simulations with simple models is expected to be inaccurate.

What types of bottlenecks are important here? Issues due to low temporal or spatial locality of accesses by single threads of multi-threaded programs can very well be detected with tools focusing on sequential performance. The same is true for issues due to limited associativity of caches or missed chances for prefetching. Of interest here are effects from threads influencing each other because of shared use of resources or because they communicate with each other, including synchronization.

The memory hierarchy of multi-core processors has two kinds of resources that are shared among cores[35]. First, there is memory space (shared last-level cache and main memory modules) and second, there are the physical data paths between components of the memory hierarchy. In respect to memory space, the shared usage of a last-level cache by all cores on a multi-core processor may be significant. If threads are running

[35]We ignore threads running on the same core here (Simultaneous Multi-Threading, SMT), as they are sharing so many resources that we cannot approach their behavior with simple architecture models.

simultaneously on the cores and are accessing distinct data, they actually compete for space. The reduced cache space available typically results in more misses, and the additionally needed memory accesses slow down the threads.

On the other hand, in respect to data transfers on shared paths within the memory hierarchy, the most relevant cause for interference typically is the shared connection to main memory. For a multi-socket system also the shared connections between processor chips become relevant. When sharing a connection, the most probable bottleneck is contention when the bandwidth requirement of a multi-threaded program gets above the available bandwidth limit. Every core using such an inter-processor connection (e.g. due to accessing remote memory of another NUMA domain) increases the observed contention. In the end, the slowdown effect may be proportional to the number of cores using the contended resource[36].

The other type of bottleneck in the cache hierarchy of multi-core architectures is due to communication among cores. In the end, any kind of communication/synchronization means that the involved threads access the same block of memory. Further, the hardware needs to support atomic operations. That is, two accesses to the same memory cell coming from the same core are not allowed to interleave with an access to this memory cell from another core. Communicating via the same memory block always triggers a cache coherency transaction if multiple private caches are involved. Data written by one core may still reside in its private L1 cache in a modified state. If another core is reading this memory block, two things happen. First, the read access passes down to the shared last-level cache and gets forwarded to all other cores whose private caches need to be checked for modifications. Second, if such a modification is detected, it gets written back to the last-level cache and finally gets forwarded to the reading core. Usually, the latency of such a series of actions is much larger than a simple last-level access. If this happens often, programs can experience relevant slowdown even though main memory is not involved.

Finally, a typical parallelization issue is load imbalance. The behavior of the memory hierarchy can trigger load imbalance issues if hit/miss behavior is different between threads even if there are an equal number of memory accesses.

Let us go back to the question raised above: which of the mentioned bottlenecks may be sensitive to the timely interleaving of memory accesses between threads? This relates to the question of when activities in one thread may change an access of another thread from being a cache hit to becoming a cache miss. The miss is either served from main memory (1) or from the private cache of another core (2).

Now suppose that the cache simulation using a simple model got the timing between two processes wrong. Due to that, at one point in time of the first thread, the simulation may see the second thread doing n memory accesses more than what real hardware or a cycle-accurate simulation would have shown. Regarding situation (1), these accesses could have evicted exactly the n cache lines from the shared last-level cache which are needed by the first thread. Similar for situation (2), the first thread may have to do at

[36]It can be observed on recent multi-core processors that one core actually cannot exploit the full bandwidth to memory although all accesses go through the shared last-level cache. A reason may be that the last-level cache only allows a limited number of outstanding memory requests per core.

most n "load-from-other-core" accesses instead of seeing hits into its private L1 cache.

Actually, we expect the case of situation (1) to rarely produce large errors due to a simpler architecture model. The large size of shared last-level caches makes it very unlikely that wrongly simulated timing behavior results in evictions which produce misses instead of hits. However, situation (2) may happen much more often when same memory blocks are accessed frequently by different threads.

In the following section (4.2) we look at a special case called *false sharing.* We describe a tool that was developed in a students work[37] and published in [47]. In that work we rely on estimated bounds due to the problem of not being able to get the exact timely interleaving behavior of threads.

Already at this point, we sum up some conclusion from this study on false sharing. Any multi-threaded program whose performance depends significantly on timely interleaved behavior between threads also exposes this instability on real machines. Thus, exact cycle-accurate simulation would not really help as it only presents one single, possible result. If the unstable performance behavior is not wanted, we need to get rid of the frequent accesses to the same memory blocks by different threads in the first place. To detect these, it actually is not so much important for a tool to be able to show exact numbers of how often a memory block was bouncing between different L1 caches of a multi-core processors; these numbers are expected to be unstable anyway. However, if these events are relevant for performance, even a simulation of a simple cache model should be able to catch enough of these events to make them appear prominently in the measured results. Also, false sharing is usually easy to avoid by separation of data structures used from different threads. For the tool to be useful to users, it should provide hints on the severity of false sharing instances. For this use case bound estimations actually work quite fine.

All other bottlenecks mentioned above are not expected to be sensible to small "time drift errors" (that is, the differences which show up between the results of a cycle-accurate simulator and a simulator using a simple architecture model due to the performance of threads being simulated slightly different). However, the difference of simulated times between cycle-accurate and simple simulation could become large. But there is an important observation: explicit synchronization points between threads actually reset any accumulated time drifts between cycle-accurate and simple simulation. Thus, for programs to simulate where we expect larger "time drift errors" due to rare explicit synchronization, we could introduce further synchronization points. However, this usually is not easy in practice and seems to be inadequate anyway as it modifies original program behavior (potentially slowing it down). Instead, it seems reasonable to accept time behavior which can be slightly off due to simulation of a simple model, and to expect that our performance analysis tool still will identify relevant bottlenecks. We propose our simulation approach to "only" complement real measurements for more insight into possible optimization strategies. A tool that is fast and can identify most important bottlenecks is better than a much slower tool which may be able to also pinpoint minor performance issues. The latter tool simply would not be used because

[37] Stephan M. Günther: *Assessing Cache False Sharing Effects by Dynamic Binary Instrumentation.* Guided Research, Technische Universität München, 2009.

of its unacceptable slowdown.

4.2 Estimating False Sharing Event Counts

Unnecessary sharing of cache lines among threads of a program due to private data which are located in close proximity in memory is a performance obstacle. Depending on access frequency to the data and scheduling of threads to processor cores, this can lead to substantial overhead because of latency induced by cache coherence transactions, known as *false sharing.* Since processor hardware can not distinguish these effects from real data exchange (true sharing), all measurement tools have to rely on heuristics for detection.

Instead of relying on measurement of real or accurately simulated hardware, we propose an approach to estimate *worst case* numbers of false sharing events. The idea is to provide a list of data structures and code positions which is ordered by their worst-case influence on runtime. In addition, we propose a way to map our false sharing estimates to temporal overhead which makes it possible to compare them e.g. to latency due to bad cache behavior in general.

Our approach identifies segments of execution in threads which may run in parallel in an overlapping fashion on separate cores. Currently, we restrict ourself here to segments separated by barrier synchronization due to simpler implementation. However, this does not reduce the usabilty of the approach. Further, it is easy to extend the resulting tool to handle the more general case in the future. Only inside such segments false sharing can happen among involved threads. This allows to handle each set of eventually overlappingly running segments separately. As key concept for the worst case estimate, we do not assume or simulate any order of interleaved accesses by different threads within such segments, but only count the number of loads and stores executed. This has important consequences. First, as temporal order is not relevant for the estimate, we can use a simple machine model, because there is no need for exact simulation of advancing time. Second, the dynamic binary instrumentation framework does not need to be able to run client threads concurrently. Also, there is no need to make the time slice of instrumentation frameworks which serialize thread execution especially small. Third, we keep the amount of data collected for false sharing detection at a manageable level. Detection can be postponed to a post-processing phase. The prototype of our tool, called *Pluto*, consists of two parts. The first is the data collection phase implemented using Valgrind [86] as DBI framework. The second part is the postprocessing with visualization of the results.

4.2.1 Related Tools and Work

False sharing is often mentioned in programmer manuals for parallel shared memory machines, e.g. SGI architectures [32]. However, it is difficult to provide an exact definition of the effect which matches intuition. Our definition is chosen to allow for occurrence estimates via execution-driven traces. In [22], the authors compare different definitions and their practicability on example code. They conclude that the effect on runtime is always difficult to predict. In [119], the intuitive effect of false sharing is captured by a simple benchmark. Latency on various cache architectures is compared including multi-core systems. It is shown that shared caches can drastically reduce the effects of false sharing.

Intel's PTU tool [57] is able to give hints at such events by providing offsets within a cache line for sampled accesses to a given memory block, separated by thread ID. Last level cache miss events are taken into account. For more precise identification, newer systems can even focus on coherency misses. The feature is based on PEBS (precise event based sampling) functionality of modern Intel processors, allowing to identify the data address of memory accesses. However, as only a subset of events is sampled, exact ordering and interleaving of accesses can not be detected, prohibiting direct false sharing detection. Also, aggregated information of accessed offsets can wrongly hint at true sharing where actually false sharing is happening. False sharing detection by architecture simulation is done in [110] and provides very detailed cache behavior, but is based on a machine model which just takes latency of specific instructions into account.

False sharing is also a significant problem for DSM (distributed shared memory) systems. There, the smallest memory unit used for the software caching algorithm (as well as the coherency protocol) has to be a memory page, in order to use the hardware memory management unit. As this is much larger than the size of a cache line in a processor cache, it drastically increases the risk of false sharing. Approaches to reduce this issue were researched in the 90s [41, 9]. In [60], methods for compilers to reduce false sharing effects were proposed, but these do not seem to have been implemented.

4.2.2 Concept

The following section gives a short overview of the notation used. Afterwards, false sharing events are defined for the scope of this paper. Using this formalism an estimate for the number of occurrences is derived. Further, a way to better approach real-world applications is proposed. The last section outlines how synchronization of threads by means of barriers can be taken into account.

Model and Notation

We initially choose a simplified model for our approach where each core has a private cache of unlimited size. This is motivated by the assumption that false sharing in real-world applications involves only a small subset of cache lines which is frequently accessed and for this reason not subject to capacity misses. However, we also outline an approach to consider finite caches in Section 4.2.4. Furthermore, we assume that exactly one thread $t \in T$, with T being all threads of our multi-threaded application, is executed per processor core and that threads are not migrated between cores at runtime. Threads may be synchronized by barriers. In such a case, we refer to a parallel section as a code segment encompassed by barriers. Partial synchronization between only a subset of threads is not considered here. A memory block $b \in B$ is an aligned block of shared memory which equals the size of a cache line. The memory accessed by the application consists of an ordered sequence of these memory blocks.

We denote X as being the set of possible offsets into one memory block and Y the set of allowed access sizes. Then we can denote a single load or store operation by thread $t \in T$ to memory block $b \in B$ with offset $x \in X$ and size $y \in Y$ by $l^{(t)}_{x,y}[b]$ or $s^{(t)}_{x,y}[b]$, respectively. The total number of load or store operations with identical attributes is denoted by capital letters $L^{(t)}_{x,y}[b]$ and $S^{(t)}_{x,y}[b]$, respectively. The total number of store operations $S^{(t)}[b]$ to some block b by thread t is given by

$$S^{(t)}[b] = \sum_{x \in X, y \in Y} S^{(t)}_{x,y}[b].$$

Sometimes the offset x and access size y are not considered. Then we rely only on the summed values $L^{(t)}[b]$ and $S^{(t)}[b]$. For ease of notation we skip the index b whenever possible and then assume only a single memory block.

Definition of False Sharing

Every cache coherency protocol maintains cache states for each memory block and each private cache. The state *Invalid* (I) denotes the fact that a cache does not have a copy of the block. If the cache has a copy, the state is either *Shared* (S) or *Modified* (M) with the distinction that the latter is reserved for when the copy in the cache is written to or is expected to be written to, and therefore it has to be made sure that only one copy does exist in the whole system[38]. If a cache needs to change the state of a memory block due to a memory access being done in the program of the corresponding processor core, the cache communicates this fact to other caches using cache coherence transactions. Let us assume a scenario where two caches have a copy of a memory block in state S, as both cache have read the block before. If one cache wants to modify his copy (going to state M), he has to send a transaction to trigger the other cache to evict (invalidate) his copy from the cache. If afterwards the other core wants to read the block again, this needs another transaction as the modified copy needs to be loaded and states have to be switched to S again. However, multiple accesses to the same block by one core only trigger a state change for the first store after previous loads (S to M).

With this in mind, a definition of false sharing is given as follows: Let two threads $t_1, t_2 \in T$ be running on two distinct processor cores with separate caches. For a memory block accessed by both threads, we refer to a *false sharing event* between these cores if there is a cache coherence transaction triggered by one of the cores resulting in state changes for this memory block within each cache of the involved cores, but this coherence transaction does not serve the purpose of a data exchange between the two threads. With this definition, both reads and writes to a memory block can trigger false sharing as long as state changes for this block in the caches of two cores are involved. E.g. after one thread writes, reading the memory block by another core already refers to false sharing if the bytes actually read are distinct from the bytes written. One can argue that in the situation where one thread "just" has to invalidate its copy, this should not be counted as false sharing. However, also in this

[38]Both S and M states may be splitted up in sub-states for more complex coherency protocols. However, this does not matter here.

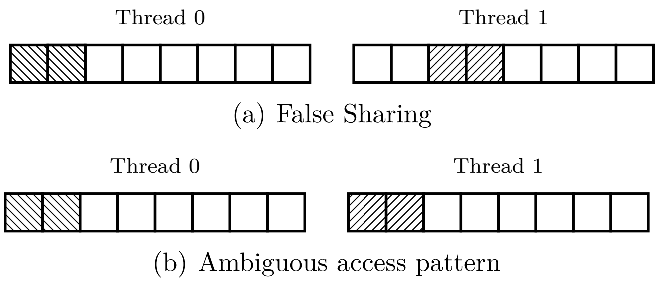


Figure 4.1: Access patterns to the same memory block from two different threads (hatching diagonal up denotes load, hatching diagonal down store).

case, a coherence transaction which serializes actions in the two caches with significant data path occupancy and latency penalties is involved.

Given an individual memory access, we can not make any statement of whether it can trigger a false sharing event or not. It always depends on the sequence of operations. Therefore, in reality, the temporal order of memory references plays an important role. Consider the access pattern as given by Figure 4.1(a). Both threads access disjoint sections which is why no data can be exchanged via this cache line. Whether false sharing events happen, depends on the sequence of memory references:

1. $s_{0,1}^{(0)} \to s_{1,1}^{(0)} \to l_{2,1}^{(1)} \to l_{3,1}^{(1)}$

   In this case, thread 0 writes 1 byte to offset 0 and offset 1 each before thread 1 loads a value. This results in one false sharing event according to our definition since the block is transfered between caches only once (with one coherence transaction).

2. $s_{0,1}^{(0)} \to l_{2,1}^{(1)} \to s_{1,1}^{(0)} \to l_{3,1}^{(1)}$

   Here, the accesses of both threads are interleaved which results in three coherence transactions and consequently three false sharing events.

Note, that such an access pattern is unnecessary and avoidable. Since no data is exchanged there is no point for different threads to access memory locations which are that near to each other. Now consider the access pattern as given by Figure 4.1(b). It is no longer obvious whether false sharing happens or not since both threads access a common location. For example, thread 0 may write to offset 0 and afterwards thread 1 may read from offset 1. Such crosswise accesses would make up a worst case sequence of accesses. Although such a sequence might be deliberately provoked, it is very unlikely that a real-world application would behave in this way. In contrast, accesses of different threads to the same memory block at the same offset and of the same size are a very strong indication for data exchange.

Estimate for False Sharing

For each segment our tool collects the memory accesses happening but does not trace the sequence (which also would produce too much data). Due to the lack of temporal order between memory accesses, we try to construct a worst case sequence which maximizes the number of false sharing events within a parallel section. Assuming two threads ($T = \{0, 1\}$), for each memory block, a first estimate can be made by only considering the absolute number of accesses per thread $L^{(t)}$ and $S^{(t)}$. We do not consider offset and size of references here. This makes it impossible to differentiate between false sharing and possible data exchange. However, if data exchange is treated as false sharing we increase the estimate for the bound which might result in a big error but in general does not underestimate the number of false sharing events. Given a number of memory references, the worst case is an alternating sequence of store and load operations by thread 0 and 1 respectively. When no more loads are available, there might still be a number of store operations of both threads left which can also be arranged in a way such that false sharing occurs. This can be written as a sequence of non-distinguishable operations:

$$\begin{aligned} & s^{(0)} \to l^{(1)} \to s^{(0)} \to l^{(1)} \to \ldots \\ \to\ & s^{(1)} \to l^{(0)} \to s^{(1)} \to l^{(0)} \to \ldots \\ \to\ & s^{(0)} \to s^{(1)} \to s^{(0)} \to s^{(1)} \to \ldots \end{aligned}$$

The first reference causes no false sharing since it is a compulsory miss and any subsequent access causes one false sharing event. We do not consider compulsory misses since there is at most one per memory block accessed. It is important to consider false sharing events involving load operations first because solely load operations can not cause false sharing. In order to obtain a sequence of maximal length, there must not be two subsequent store operations until no more loads are available. The problem of constructing such a worst case sequence can also be seen from the point of set-theory. Since we do not consider time we can also abandon the view of sequences. Instead, we can assume sets of load and store operations for both threads. Now we combine each load of thread 0 with a store of thread 1 and vice versa. Afterwards, there might be store operations left for both threads. These can also be combined. By counting the number of combined operations we end up with a value which equals in length to the sequence above. From both views, we can derive the following formula which gives an estimate of the false sharing events $\tilde{\varphi}$ for a given block $b \in B$:

$$\begin{aligned} \tilde{\varphi} = 2 \cdot & \left[\underbrace{\min\left(S^{(0)}, L^{(1)}\right)}_{(1)} + \underbrace{\min\left(S^{(1)}, L^{(0)}\right)}_{(2)} \right] \\ & + 2 \cdot \underbrace{\max\Big(\min\left(S^{(0)} - L^{(1)}, S^{(1)} - L^{(0)}\right), 0 \Big)}_{(3)} - 1 \end{aligned} \tag{1}$$

Part (1) considers all possible combinations of store operations by thread 0 with load operations by thread 1. Accordingly, (2) considers the inverse case. Remaining store operations (if any) of both threads are combined in (3). If either $L^{(1)} \geq S^{(0)}$ or $L^{(0)} \geq S^{(1)}$ holds, there are no more remaining store operations and part (3) is empty.

Equation (1) does a good job if there is no data exchange in reality. Unfortunately, it can not differentiate between false sharing and data exchange at all, which is why the result may turn out to be a massive overestimation. Nevertheless, it may be useful if the programmer knows where data exchange between threads is performed.

Considering Offsets and Sizes

So far, we did not consider the offset x and size y of a memory access. This is inevitable to determine if threads access disjoint parts of a memory block. Now we try to derive an estimate for the maximum number of data elements *exchanged* between two threads. This estimate $\tilde{\theta}$ is useful to make a statement regarding the reliability of $\tilde{\varphi}$ calculated according to Eq. (1).

Again, we assume only two threads and no temporal order of the memory references. If we find a pair $(s^{(u)}_{x,y}, l^{(v)}_{x',y'})$ of references within the same block with $u \neq v$ (ie. different threads) and $\{x, \dots, x+y-1\} \cap \{x', \dots, x'+y'-1\} \neq \emptyset$ (ie. overlapping byte ranges), this might have been a data exchange in reality. However, checking for overlapping ranges makes the estimation very complex. In practice, if there is real data exchange, compilers ensure that this happens using the same data type on the reading and writing side. Thus we do not check for overlapping ranges but assume data exchange only to happen when both offsets and sizes match, ie. $x = x'$ and $y = y'$. If this assumption goes wrong in special cases, we underestimate data exchange. This is not critical for the purpose of our tool.

The matching process is illustrated by Figure 4.2. Two loads of thread 0 match with two stores of thread 1 in size and offset which we count as a total of four true sharing events. In a similar way one load of thread 1 matches a store of thread 0 which is counted as two true sharing events. The justification for doubling the number of events is that we designed the estimate $\tilde{\theta}$ in a way such that $\tilde{\varphi} - \tilde{\theta} = 0$ holds if the access patterns to a block of two threads are very similar. For example, one thread may write a number of values which are afterwards read by another thread. This would result in an ambiguous case and a high value of $\tilde{\varphi}$, although it is very likely that both threads exchanged data. At the same time, the similar access pattern would be considered by $\tilde{\theta}$. If $\tilde{\theta}$ turns out to be in the order of magnitude of $\tilde{\varphi}$, we know about the ambiguous case. Strictly speaking, we can not make any statement whether or not the result of $\tilde{\varphi}$ is meaningful. If, on the other hand, $\tilde{\theta}$ is more than an order of magnitude smaller than $\tilde{\varphi}$, it is very likely that false sharing occurs, unless there are some temporal constraints in reality which prevent false sharing at all. The estimate $\tilde{\theta}$ for a given memory block $b \in B$ can be calculated as follows:

$$\tilde{\theta} = 2 \cdot \sum_{x \in X, y \in Y} \left[\min\left(S^{(0)}_{x,y}, L^{(1)}_{x,y}\right) + \min\left(S^{(1)}_{x,y}, L^{(0)}_{x,y}\right) \right] \tag{2}$$

The inner term of the sums is very similar to parts (1) and (2) in Eq. (1) except for the added offsets and sizes. Since alternating store operations of both threads can not serve the purpose of data exchange, these are not considered as true sharing and are therefore omitted in (2).

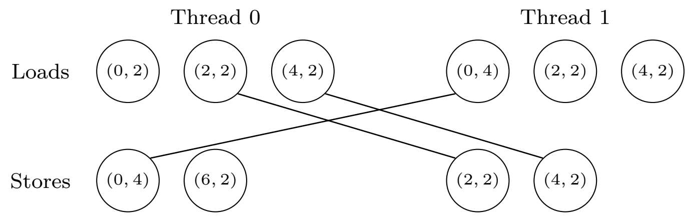


Figure 4.2: Finding pairs of matching load and store operations. The tuples (x, y) denote the offset $x \in X$ and size $y \in Y$ of each memory reference.

The first estimate $\tilde{\varphi}$ for false sharing does not consider offset and size of memory references. Now that we have an estimate for the number of data exchanges, it appears tempting to obtain a better estimate

$$\tilde{\varphi}' := \tilde{\varphi} - \tilde{\theta}. \tag{3}$$

Although, this approach delivers astonishingly exact results in first tests, as we will see in Section 4.2.4, these might must be seen critical. The general problem here is that the validity of the result completely depends on the unknown sequence of memory references. It is possible to construct a sequence such that $\tilde{\varphi}'$ fails. However, such a sequence is unlikely to be produced and dampening $\tilde{\varphi}$ if matching memory references exist appears reasonable for real-world applications.

Considering barriers

Barriers can be quite easily considered as long as they affect *all* threads. Synchronization between a subset of threads is much more difficult and part of future work. For now, we assume a number of parallel sections defined as a code segment which can be executed in parallel and independently by an arbitrary number of threads and which is encompassed by barriers. Anything discussed so far holds within a single parallel section. Subsequent parallel sections can be taken into account by introducing an index s to the set of memory blocks B where s denotes the parallel section. Let S be the set of parallel sections and B_s the set of memory blocks which have been accessed within section $s \in S$. We note that at most one false sharing event (or one data exchange) can occur between subsequent parallel sections. This happens when a core accesses a memory block the first time within this section which has been accessed by another core in the previous section provided that no capacity miss occurred in the meantime. The number of false sharing events (or data exchanges) between any section s and its successor $s + 1$ is bounded by $|B_s \cap B_{s+1}|$. Estimates $\tilde{\varphi}$ and $\tilde{\theta}$ which take multiple parallel sections into account can be expressed by $\tilde{\varphi}_s$ and $\tilde{\theta}_s$ per parallel section $s \in S$:

$$\tilde{\varphi} = \sum_{s \in S} \tilde{\varphi}_s + \sum_{s=1}^{|S|-1} |B_s \cap B_{s+1}| \tag{4}$$

$$\tilde{\theta} = \sum_{s \in S} \tilde{\theta}_s + \sum_{s=0}^{|S|-1} |B_s \cap B_{s+1}| \tag{5}$$

4.2.3 Implementation

The measurement part of our approach collects statistics about memory references for each parallel section. These are written to file when reaching the end of such a section. Currently, we expect the programmer to notify our tool on passing borders of sections. This is done via a so-called *Valgrind client request*. To be able to calculate the estimate for one section, we need to count accesses to any given memory block. Number of accesses have to be collected separately for each thread and depending on whether they are store or load accesses. Further, to be able to calculate estimates not only for false sharing, but also for potential true sharing events, and to relate the estimates to code positions for visualization, distinct access counts are also aggregated for different access offsets and sizes, as well as different code positions at which accesses are happening. Instead of/additionally to the collection of code positions we could relate detected false sharing events to data structures. However, this is quite complex to implement in a general useful way, and future work.

The counters are put into a global hash table. All items listed above are part of the key for the hash table. An issue with this approach is that the full key has to be stored for every counter to check against at hash table look-up. To keep memory requirements low, the number of hash table entries should be minimized. E.g., code positions can just be function names, and only for selected functions also include line numbers. Similar access types could be mapped to one entry with multiple counters. Another way to reduce entries comes from the fact that our estimate is zero if there were only load access to a memory block, or the block is private to a thread. This information sometimes can be detected (memory areas with read-only permission), or could be provided by the programmer with client requests. Currently, the following heuristic is implemented: If a memory block was never written to before, load accesses are ignored. In principle, this is problematic for our estimate as early loads could be rearranged with late stores of another thread to produce false sharing events in a worst-case ordering. Therefore, we print out a statistic of ignored loads to blocks where there was a store afterwards. If this number is low in relation to resulting estimate, the heuristic worked for a given run of an application.

To sum up, our tool's instrumentation adds a callback to a helper function for every data access happening in supervised code. This function looks up the counter in the global hash table of access types and either increments a counter of an already existing entry, or creates a new entry when the given access type was observed first. At the end of an execution unit (or at program termination), all entries are sorted by memory block, and for each block, it is first checked whether accesses are only loads, or accesses are only from one thread. If this is the case, no output has to be generated for this block. Otherwise, all access counts are written to an output file. To denote a code position, we write source name and line to this file, if debug information is available. We could do something similar for memory block addresses, e.g. use the symbol name for static/global variables, or otherwise use the code position of the memory allocation which includes a given block. Currently, we just output the address. Finally, all counters are set to zero (currently, we remove all hash table entries), and a marker is written to the output file indicating that a new execution unit has started.

After the run, the postprocessing tool (`plutoparser`) reads in the file, sums up access counts per memory block and calculates estimates as explained in section 4.2.2. With a specified average time overhead for false sharing events, temporal estimates can be calculated from the occurrence estimate.

4.2.4 Results

To confirm our approach, we have tested with two self-written applications which provoke false sharing and a real-world application. The first test is a synthetic benchmark with a very regular access pattern designed to trigger false sharing. This can be used to estimate the penalty per false sharing event. The second application is a simple implementation of a prime sieve parallelized using OpenMP. The real-world application calculates sparse matrix-vector products and is part of an imaging process. All tests have been performed on a system equipped with two Intel Xeon X5355[39] processors.

Synthetic False Sharing Benchmark

Two threads modify elements of a data array at 64 byte strides. In order to trigger false sharing, both threads have an offset of 4 bytes to each other, which corresponds to a single data element. A second version of this benchmark avoids false sharing by accessing data elements located in disjoint parts of the array. The access pattern is illustrated by Figure 4.3. The test is repeated for a number of iterations. The benchmark is written in a way such that the number of operations performed by each thread remains constant and independent of array size. For this purpose, the number of iterations is adapted dynamically. We perform the test with $6.25 \cdot 10^7$ operations, which results in twice as much memory references, since modification of an element implies a load and a subsequent store. To avoid inaccuracies due to thread migration at runtime and to get an estimate for the worst-case penalty, we pin the threads to cores located on different sockets.

Figure 4.4(a) shows the time per memory reference for both cases with and without false sharing. The samples were obtained by time measurement and afterwards converted to clock cycles. Apart from a break-off at small array sizes, which we can not really explain, one can clearly see that false sharing causes a massive penalty. By calculating the difference between both runs, we end up with the overhead caused by false sharing for a single memory access. This is shown by the continuous line in Figure 4.4(b).

The average penalty lies around 125 cycles per reference which gives us the penalty $\tau = 125$ per false sharing event on this system. The dashed line shows the time estimate as given by our tool when using this penalty, which is reduced to a trivial result due to the constant number of memory references and the regular access pattern. To validate our assumptions about the number of false sharing events, we checked the results by means of hardware performance counters (using pfmon). On the one hand, we can

[39]Quad-core processor, 32KiB L1 cache, 8MiB L2 cache split into two times 4MiB for two cores per socket, 2.66GHz core frequency.

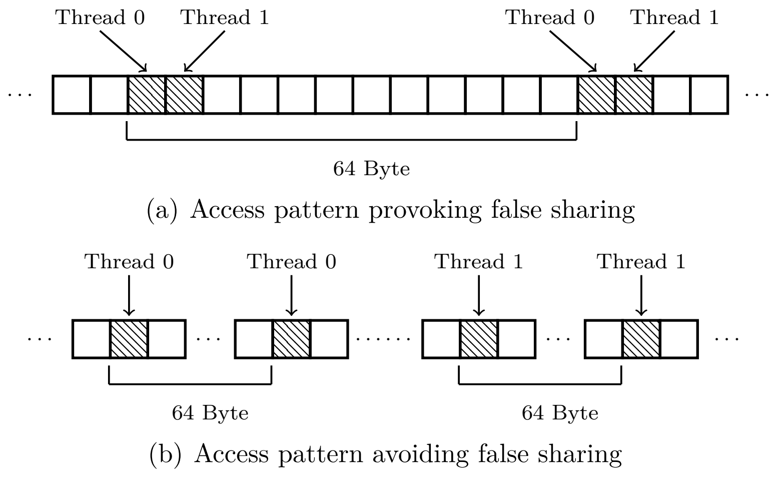


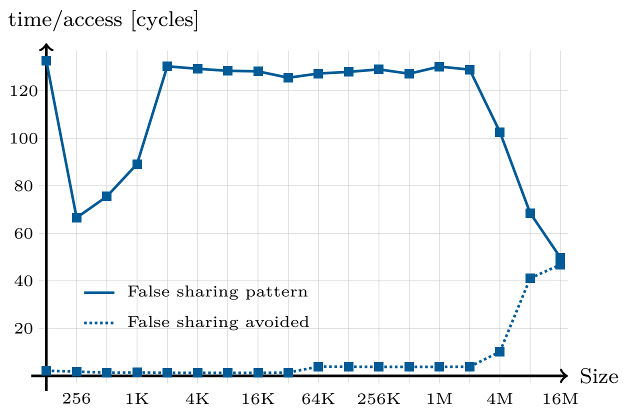
Figure 4.3: Access patterns to provoke or avoid false sharing. The type of access depends on the selected mode.

measure the number of memory accesses for loads and stores using `L1D_CACHE_LD:MESI` and `L1D_CACHE_ST:MESI`, respectivly. The sum is the number of memory references. On the other hand, the counter `L2_LINES_OUT:ANY` gives us the number of cache lines evicted. According to our definition, false sharing happens with coherence transactions. Evictions are generated when data written on one core is read by another core (state change from M to S). This covers half of our references. If we assume that the number of false sharing events is exactly twice the number of evictions and the penalty of a false sharing event is τ, we can print the cycle overhead of false sharing per memory reference (measured from counters as given above). This is shown by the dotted line in Figure 4.4(b). Apart from lower array sizes, we see that the lines match quite well. We note that the parameter τ of course is hardware dependent which must either be known or measured using a test application like this one. Otherwise, inaccuracies induced by a default value must be accepted.

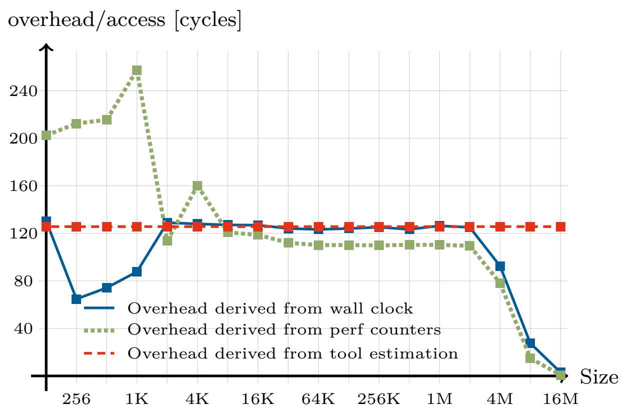
Although the estimate as given by our tool is trivial for this case one problem is apparent: Since we do not consider capacity misses, we are not aware that for arrays larger than 4MiB false sharing does no longer cause overhead. This is of course not true in general but a consequence of the regular access pattern and the huge number of iterations. Assuming uniformly distributed accesses, the probability of false sharing would decrease with size but not drop to zero almost instantly after the cache size is exceeded. However, our approach would benefit from considering finite caches.

Prime Sieve

Now that we have an estimate for the penalty induced by false sharing at least for our test system, we can turn to a more realistic application. We implemented a simple variant of the prime sieve by Eratosthenes. The idea behind the algorithm is to calculate all multiples of $n \in \{2, 3, \ldots, N\}$ up to a given bound N and to remember which numbers are multiples of another one. This is implemented by using a large bit



(a) Time per memory reference



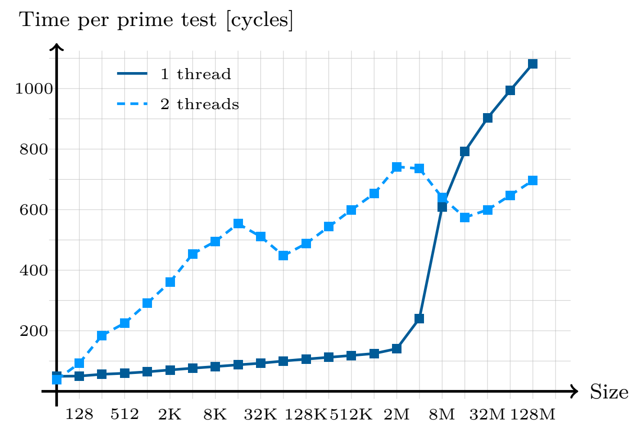
(b) Overhead per memory reference

Figure 4.4: Clock cycles and overhead due to false sharing per memory reference.

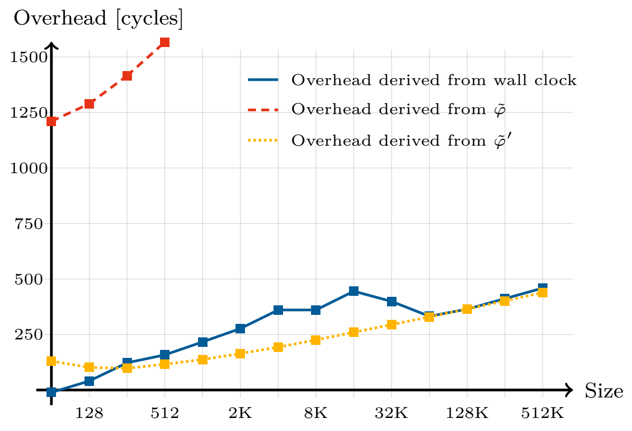
mask initialized to zero where the n-th bit represents the number n. Afterwards the algorithm calculates the multiples starting at $n = 2$ and sets the corresponding bits to one. At the end, all bits which are not marked represent a prime number. Of course there is a huge overhead in this naive implementation since most numbers are marked multiple times. However, it can be easily parallelized using OpenMP and completely avoids the need for synchronization between threads working on the same array. The relevant code section is shown in Listing 1. We used a static parallelization to avoid any overhead induced by task assignment. The chunk size is set to one to ensure a tight interleaving between threads. This makes sense since otherwise OpenMP would divide the loop anywhere around $^N/_2$ which would lead to an unequal load balancing (there are fewer multiples smaller than N for large numbers than for small numbers). In this case, we pinned the threads to cores located on different sockets again. Figure 4.5(a) shows the average time per prime test measured in clock cycles for a given array size. Since the algorithm has no linear complexity, the time per prime test increases for larger numbers. Parallelization fails for array sizes up to 8MiB which is double the size of the L2 cache. This is explained by the irregular access pattern of the algorithm

```
#pragma omp parallel private(i,mask,x) {
#pragma omp for schedule(static,1) nowait
   for( i=2; i<N; i++ ) {
      x = 2*i;
      while( x < max_size ) {
         mask = 128 >> ( x % 8 );
         ptr[ x/8 ] |= mask;
         x += i; }}}
```

Listing 1: Parallelized loop of the prime sieve.



(a) Time measured in clock cycles for 1 and 2 threads.



(b) Overhead of threaded version measured in clock cycles.

Figure 4.5: Time and overhead per prime test.

since the stride sizes increase with the numbers tested. Memory blocks which make up the latter part of the array are accessed more frequently than other ones, e.g. the first cache line is never accessed again after the first 512 numbers have been processed (64 byte times eight numbers per byte). The continuous line in Figure 4.5(b) shows the difference between the dual and single thread variant and therefore the overhead induced by parallelization. The dashed and dotted lines indicate the false sharing es-

```
#pragma omp parallel for schedule(dynamic)
for(r = 0; r < rows; r++)
   for(i = rowptr[r]; i < rowptr[r+1]; i++)
      y[r] += val[i] * x[col[i+1]];
```

Listing 2: Forward projection of an MLEM iteration.

timates $\tilde{\varphi}$ and $\tilde{\varphi}'$ multiplied with the penalty of an false sharing event τ as derived from the previous experiment. It turns out that $\tilde{\varphi}$ overestimates the penalty by more than an order of magnitude[40]. This is obvious for two reasons: First, modifying values involves reading and writing and therefore a pattern of subsequent loads and stores for each thread. The estimate $\tilde{\varphi}$ assumes the worst case, since it does not know the sequence of references. Secondly, the access pattern is not strictly sequential. The stride sizes vary which is why not every modify results in a false sharing event even for small array sizes. The result as given by $\tilde{\varphi}'$ is much more accurate since it is considerably reduced by matching accesses between threads. Nevertheless, there remains a sufficiently large estimate to reflect the overhead induced by false sharing. Although these results are promising, it must be kept in mind that $\tilde{\varphi}'$ is no longer an estimate for an upper bound, but an attempt to counter the pessimistic overestimation of $\tilde{\varphi}$. Therefore, it might happen that it underestimates the overhead.

Image Reconstruction Algorithm

In a cooperating with the department for nuclear medicine at “Klinikum rechts der Isar”, Munich, we are working on tuning medical image reconstruction code. To this end, we work on cache optimization and parallelization of various algorithms used in this field [68]. As third example, we look at the OpenMP parallelization of the MLEM image reconstruction algorithm [98]. This algorithm runs so-called forward and backward projection phases in an iterative loop. Each projection is a sparse-matrix vector multiplication. We concentrate on the forward projection, as this is subject to false sharing (see Listing 2). The matrix is stored in standard compressed sparse row (CSR) format, consisting of the three arrays `rowptr`, `val`, and `col`. The parallel for loop is partitioned according to the rows of the matrix. As the number of entries per row varies a lot, dynamic scheduling needs to be used for load balancing. This scheduling uses a default chunk size of 1. Thus, different threads frequently modify elements of the array `y` in an interleaved fashion. The sparse matrix has a size of 2.7 GB and the involved dense vectors have around 800 k entries.

The first row of Table 4.1 shows the runtime of this code for two threads pinned to cores on different sockets. A chunk size of 16 eliminates false sharing if array `y` is properly aligned (second row of Table 4.1). A sampling based profiling tool shows that less than 2% of time is spent in scheduling related routines in the OpenMP runtime. Therefore, OpenMP scheduling overhead is not falsely interpreted as false sharing. When comparing wallclock times between version “d1” (the version triggering lots

[40]The slope of $\tilde{\varphi}$ is roughly linear for array sizes greater than 256KiB, not exponential as one could assume by looking at the section in Figure 4.5(b).

	Wall-clock time [s]	FS est. $\tilde{\varphi}$ [MEvents]	time est. [s]	Tool runtime [s]
d1	0.901	12.70	0.297	168
d16	0.616	0.17	0.008	155

Table 4.1: Results for an MLEM forward projection.

of false sharing) with “d16” (the version for which we assume that almost no false sharing is happening), there is a difference of $0.901 - 0.616 = 0.285$ seconds. As can be seen, the time estimate of our tool (showing a difference between the versions of $0.297 - 0.008 = 0.289$ seconds) is astonishingly accurate for this example using the false sharing event penalty of $\tau = 125$ cycles, as determined in Section 4.2.4. The last column of Table 4.1 shows the wall clock time needed for the forward projection phase run in our tool. This currently shows a slowdown of around factor 200. However, we expect that we can significantly reduce this slowdown.

4.2.5 Conclusion on False Sharing Estimation

We propose an approach to estimate cache false sharing effects. The goal is to come up with a practical tool pinpointing code positions and data structures responsible for bad scalability of shared-memory applications because of false sharing. To this end, we use data collected in an execution-driven way via dynamic binary instrumentation to derive estimates for cache line sharing events. By ignoring temporal order of accesses for these estimates, we avoid the need for time-consuming simulation of complex multi-processor systems. A worst-case false sharing estimate is provided by assuming access orders that maximize the number of false sharing events. This rough figure is refined by a best-case estimate for true sharing events. In practical cases, the difference of these estimates seems to work nicely, and we obtain promising results for real-world applications. We provide a micro-benchmark which allows to measure false-sharing latency on real machines, and use this to get temporal overhead of false sharing.

However, our estimate sometimes can be far off from reality. Currently, we assume that every access can trigger a false sharing event, even if in the meantime a capacity miss might have occurred which prevents false sharing. Our worst-case estimate matches read and writes references with exactly the same offsets and access sizes. This may need to be refined for some applications. Further, automatic detection of synchronization needs to be added, and more than just global barriers should be supported.

4.3 Bandwidth Requirement Analysis

As discussed in Sec. 4.1, bottlenecks that are very sensitive to timely interleaving behavior of multiple threads are difficult to approach with simulation of simple cache models. But for such bottlenecks, being very accurate is not really important for a performance analysis tool. This section proposes a cache simulation based analysis tool which concentrates on the effects due to limited bandwidth within multi-core processors. The work was also published in [116].

A lot of applications are bound not by compute capability but by limited bandwidth available in a system. It is predicted that for future architectures, the energy required to transfer a byte will go down much slower than the energy required for a calculation due to technology advances [46]. Thus, bandwidth requirements will play an even more important role for the performance achievable by an application.

As the distance of a data transfer is significant for cost regarding both energy and time, the exploitation of a memory hierarchy using caches is essential for good performance. Further, with multi-core architectures, contention on shared links can be a major obstacle for performance. For applications with low computational intensity, cache optimization is good because it makes sequential code parts running faster. However, there are further, probably much more important benefits due to the reduced amount of data transfers in the memory hierarchy. This reduces contention, allows scaling with the number of used cores, and reduces energy consumption in general. The latter is especially important in two cases: first, when energy consumption as well as required efforts for cooling infrastructure are a significant part of the total cost of the system such as for HPC systems; second, for systems with energy constrains such as limited battery power (mobile computing, embedded systems).

Detection of whether a given bandwidth limitation actually is a bottleneck is astonishingly tricky. The reason is that practically achievable bandwidth depends on whether prefetchers are active, whether the accesses are sequential, strided, or random. Slowdown due to memory accesses may appear on the physical data paths to the memory modules, between chips, or within the cache hierarchy on-chip. Using hardware performance counters to understand these issues is difficult. Miss counts often are almost useless, and prefetcher behavior and bus utilization can not easily be attributed to source code.

We propose an analysis method for the bandwidth requirements of applications which decouples bandwidth analysis from other effects. This is done using a machine model whose performance is purely bound by only one resource limit, such as peak instruction throughput or peak computational performance. The model assumes that there are no stall cycles due to data dependencies or other latency, and that any peaks in bandwidth requirements are smoothed out by corresponding hardware or compiler techniques such as instruction reordering or prefetching for best utilization. This way, the analysis is able to show steady bandwidth requirements of hot code paths. As visualization we propose *bandwidth curves*. These show which portion of an application's runtime a given bandwidth is required on a given physical data path to exploit the systems peak performance. When mapping back to systems with real limitations, bandwidth curves make it is easy to see the slowdown induced by a bandwidth limitation and provide an upper bound for the performance achievable. For our proposed analysis method, we developed a corresponding tool. It uses runtime instrumentation (via Intel's Pin tool [76]) to work with compiled binaries. Execution is simulated within the assumed model, and bandwidth curves are generated. To show the usefulness of our approach, we present a case study in which cache optimization is applied to a memory bound iterative solver. The positive effect of the optimization is easily visible in the curves.

Relation to Other Work

Most performance analysis tools today use hardware performance counters to show behavior regarding the memory system [55, 34]. Typically, hit and miss counts are provided, as well as utilization of the memory bus (percentage of cycles the bus is busy). On newer processors, it is possible to get a statistical distribution of memory access latency (e.g. the data load latency analysis on Intel processors). However, contention detection can be difficult by using such measurements.

Temporal and spatial locality metrics of memory accesses (introduced in Sec. 2.1.1) try to capture the behavior of an application in regard to its ability for exploiting caches. However, none of these metrics provide any information about bandwidth requirements. The roof-line model [126] (described in Sec. 2.3.4) cannot provide contention analysis at various points of the memory hierarchy and does not apply to complex applications. To the best of the author's knowledge, there are no tools concentrating on bandwidth requirement visualizations similar to the one proposed here.

4.3.1 Performance and Machine Model

A real machine has a lot of different resources, such as core pipeline with computational units, the memory system with cache hierarchies, buses, and I/O. Each of them has its own limits, and for different phases of an application, different resource limitations may be relevant for the observed performance. However, it is often difficult to understand whether a given resource really is the dominant bottleneck, as the difference between theoretical peak performance and real exploitable performance is often influenced by micro-architectural effects.

To overcome these obstacles for performance analysis, we propose an abstract machine model which allows us to focus on application requirements regarding bandwidth demands. We assume that the performance of the application is fully determined just by the restrictions of one resource. For numerical applications, a good choice for the latter is the achievable computational capacity of a machine in GFlops per second. This way, the bandwidth requirements of the application can be expressed in the amount of bytes moved per floating point operation executed which is known in literature as *computational intensity* of a given program phase. Another choice to determine the base performance of the application may be that the machine can execute a given number of instructions per clock cycle and is therefore restricted by the throughput of the instruction decoder. Thus, we assume that the application always runs at some constant peak performance. Within this model we observe the resulting bandwidth requirements. A comparison with real bandwidth limitations allows us to check whether some limitation is a real bottleneck.

Our tool simulates the execution of a binary on a machine model. It observes how much data is moved over specific physical data paths in the system (e.g. from core to L1) in a given time unit which is determined by the execution of one instruction or one floating point operation. Let T be the number of time units for the complete program. For $t \in \{1, \ldots, T\}$ let r_t be the number of bytes transferred in a given direction at time t on a given data path in the memory hierarchy, e.g. number of bytes read by a core from its L1 cache. Then $V = \sum_{t=1}^{T} r_t$ is the total amount (volume) of bytes transferred in this direction by the program on this data path (e.g. the amount of data read by the core from its L1). If we strictly use this model, we will observe a number of bytes moved in some time units and nothing in others. However, real systems will smooth out such peak requirements by either starting transfers early or reordering instructions while waiting for data to arrive. Thus, a real system will exploit the available bandwidth capacities and not stop using the bandwidth only because the application does not execute a move instruction in a given time unit. It is important to model this behavior. To this end, we average the bandwidth requirement using a sliding window of a given time length W. Thus, the smoothed number of bytes moved at time t is

$$\bar{r}_t = \frac{1}{W} \sum_{w=0}^{W-1} r_{t-w}$$

(with $r_t = 0$ for $t < 0$). A good value for W may be in the order of the latency of a memory access (a few hundred cycles) as such latency is covered by hardware prefetchers. Summing up these smoothed bandwidth requirements over all time units of a program run gives again the total amount of data transferred (with $r_t = 0$ for $t > T$ or $t < 0$):

$$\sum_{t=1}^{T+W} \bar{r}_t = \sum_{t=1}^{T+W} \left(\frac{1}{W} \sum_{w=0}^{W-1} r_{t-w} \right) \overset{(i:=t-w)}{=} \frac{1}{W} \sum_{w=0}^{W-1} \left(\sum_{i=1-w}^{T+W-w} r_i \right) = \frac{1}{W} \sum_{w=0}^{W-1} V = V \ .$$

Thus, if we plot $\bar{r}_t$ for $1 \leq t \leq T$, the area under the curve is the amount of data transferred.

To understand bandwidth requirements of multi-threaded code, it is useful to assume that every thread is bound to run on a dedicated core, and there is no oversubscription. Thus, for an application with 100 threads we use a machine model with 100 cores. Thus, every data shared among threads has to be transferred over physical data paths. This results in a worst-case scenario regarding contention, which should help the analysis to find bottlenecks. In addition, it makes our model independent from scheduling and migration decisions in the operating system. Thus, it makes analysis more general. The model should approximate multi-core systems. We assume a single memory module shared by all cores. For the shared connection to the memory, we simply sum up the bandwidth requirements of all cores within each time unit of our simulation. This strategy may not match reality as code executed on different cores on real hardware may be slowed down by various effects depending on the actually executed code. However, for HPC code, it is quite likely that code with similar behavior is executed (e.g. the same loop body when using OpenMP parallel for).

The model works fine with simulating arbitrary caches in the data paths. Caches exploit locality properties of memory access patterns, and thus work as a kind of filter for bandwidth requirement analysis. They potentially reduce the requirements. But

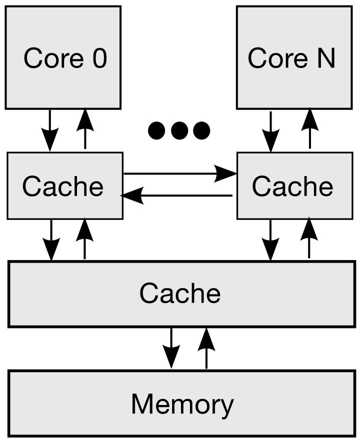


Figure 4.6: A multi-core system with connections interesting for bandwidth analysis.

caches also play another role in multi-core chips: they allow for fast communication among cores by keeping the data to be communicated on-chip. By simulating a cache coherence protocol, we capture bandwidth requirements between cores. For our analysis, we are only interested in the amount of data transferred between cores, and we currently ignore the number of coherence transactions we would need in a specific cache coherence protocol. This allows us to go with an simple protocol such as MSI (e.g. the E state in MESI does not exchange data).

Fig. 4.6 shows the configuration of a typical multi-core chip with a separate cache for each core and a shared cache for all cores. Bandwidth figures are useful for each of the data paths denoted by arrows as each of them could become a bottleneck. The arrows between the private caches denote communication among cores where the data needs to be fetched from the other core. For this, available bandwidth typically is lower than to/from the shared cache as coherence actions are involved.

Our model captures bandwidth requirements of data paths in one multi-core chip. However, it easily can be extended to match multi-socket systems with a single memory module. Extending the model further for systems with multiple memory modules is straight-forward, but needs the simulation of allocation strategies for memory pages. This is future work. Similarly, extending the analysis to systems with different memory name spaces (such as clusters) is possible.

4.3.2 Bandwidth Curves

To understand whether a given bandwidth requirement actually constitutes a bottleneck on an approximated real system, a comparison with bandwidth restrictions of the system must be done. Up to now, we assume the bandwidth requirement analysis to produce a trace of averaged bandwidth. That is, for each interesting data path, we get a number for each time unit over the whole program run. As such an amount of data easily gets prohibitive, we need to aggregate data online. Instead of storing $\bar{r}_t$ for each $t \in \{1, \ldots, T\}$, we reorder the time steps by permutation $o : \{1, \ldots, T\} \rightarrow \{1, \ldots, T\}$ such that $\bar{r}_{o(t)} \leq \bar{r}_{o(t+1)}$. Let us denote the permuted rreordered time steps as $t' = o(t)$.

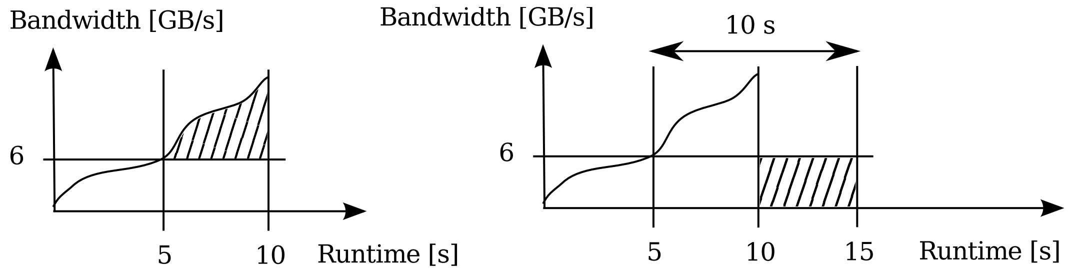


Figure 4.7: Slowdown effect of limited bandwidth shown by a Bandwidth Curve.

We note that these time steps do not represent ordered time any more. However, $\bar{r}_{t'}, t' \in \{1, \ldots, T\}$ now is monotonically increasing. This has an important benefit. For each bandwidth b, we can find a t'_b such that $\bar{r}_{t'} > b$ for all $t' > t'_b$. The bandwidths used during simulation are discrete and the number of different discrete bandwidth values is relatively low and fixed. We easily can store a counter for each bandwidth value which denotes the number of time units passed where bandwidth requirement was equal to the given bandwidth value. From this, we can reconstruct the monotonically increasing curve $\bar{r}_{t'}$. In addition to this important benefit for online aggregation, a monotonically increasing curve is easy to visualize. Intuitively, this curve shows how long/often the bandwidth requirement (in bytes per time unit) exceeded a given bandwidth b during program run, namely t'_b time units.

For a given bandwidth requirement $\bar{r}_t$, the sum $\sum_{t=1}^{T+W} \bar{r}_t = V$ was defined above as the amount of data in bytes transferred over the given data path. Permutation of elements in the sum does not change its value. Thus, using reordered time step indexes, we still have $\sum_{t'=1}^{T+W} \bar{r}_{t'} = V$ (with $\bar{r}_{t'} = 0$ for $t' > T$). Thus, the area under the monotonically increasing curve $\bar{r}_{t'}$ actually represents the amount of data transferred.

Let us take the example of Fig. 4.7 (left). The x-axis denotes reordered time steps t'. We map the time steps to corresponding time duration in seconds. This is possible when the time units used in our machine model relate to elapsed cycles. If we assume a fixed IPC model (every instruction takes a fixed number of cycles) and we use the execution of a guest instruction as time unit, this is indeed the case. Now, Fig. 4.7 (left) shows a bandwidth curve $\bar{r}_{t'}$ and a line for the bandwidth limitation being $L = 6$ GB/s. In this example, during half the runtime of the program the required bandwidth is larger than L. Let us assume that the program needs T time units to run (here, T corresponds to a runtime of 10 seconds). Let us denote t'_L as the number of time units with bandwidth requirement equal or below L. In the example we have $t'_L = T/2$. This is the crossing point of the 6 GB/s line and the bandwidth curve in the figure at 5 seconds.

Now, let us assume that on a real system, a data path cannot transfer data at bandwidth higher than $L = 6$ GB/s. The hatched area in the figure will result in an observed slowdown, as the data path cannot deliver the required bandwidth. This area has a size of

$$V_L = \sum_{t'=t'_L}^{T+W} (\bar{r}_{t'} - L) \ .$$

As shown on the right, by filling a rectangle of the same area, we can calculate the slowdown exactly in absolute time. The additional number of time units needed for the program, taking the bandwidth limitation L into account, is $T_L = V_L/L$. Even more, we know exactly the time units in our machine model with bandwidth requirements above L, which is $T - t'_L$. Therefore, the number of time units needed in a machine with the real bandwidth limitation consists of two parts: first, t'_L time steps which are not constrained by the bandwidth limitations, running at peak performance of the machine (as long as we assume this data path to be the only bottleneck in the system), second, $(T - t'_L) + (V_L/L)$ time units where we use the complete bandwidth capacity L of the data path. We note that the second component does not depend on the assumed peak performance of our machine model. It is a time span determined the data path capacity L, the number of time units simulated in which the bandwidth requirement exceeds this capacity, as well as the amount of data transferred (which depends on accuracy of our cache simulation).

In the example of the figure, the second component makes up 10 seconds. These 10 seconds are definitely needed due to the bandwidth limitation of 6 GB/s on the real system.

4.3.3 Prototype Implementation

Our prototype tool uses Pin [76] on Linux. In contrast to Valgrind (an open-source framework very similar to Pin), it can run multiple guest threads (the program to be observed) in parallel on a multi-core host system[41]. We model a 2-level inclusive cache hierarchy as shown in Fig. 4.6. Replacement policy is LRU with write-back for L1 and L2. An MSI cache coherence protocol between private L1 caches is implemented. Gnuplot-compatible files for the bandwidth curves of each interesting data path are generated at program termination.

Parallelization of Simulation

Our multi-core cache simulator is an implementation of a Parallel Discrete Event-based simulation (PDE). See Sec. 2.3 for a short description with further references.

For our simulation, we decided to use separate simulation threads for each core, taking care of memory accesses from this core. Thus, a simulation thread is also responsible for all private cache levels (only 1 in our model) as well as accesses going into the shared cache level. The latter means that the data structures which hold the state of the shared cache level themselves are shared among simulation threads. Each program thread executing within the simulated model (in the following, we use the term *guest program* for that) gets pinned to a core. Thus, in effect, each simulation thread is responsible for running one guest thread.

For a good parallelization strategy, three things have to be kept in mind. First, our model does not simulate pipeline behavior with control or data hazards and just assumes a best case throughput. Thus, we already expect huge time drifts between simulated cores in relation to a real machine. Therefore, we should be able to use a

[41] There is work going on to allow for this feature in future Valgrind versions.

time slack for our PDE simulation to be as large as the time drift we expect from the simple simulation model. Second, our methodology for bandwidth request analysis deliberately ignores possible slowdown effects from the cache model, as we want to understand the needed bandwidth for the best case scenario in which the memory hierarchy does not restrict performance. Third, explicit synchronization within the multi-threaded code will make sure that threads do not arbitrarily drift apart from each other.

All these observations allow us to not worry too much about needed synchronization among simulation threads. In fact, we rely on the explicit synchronization from the multi-threaded guest program, and do not do any further synchronization within the simulation itself. Thus, we also do not need to maintain a simulated time at all. Consequently, the actual interleaving of guest threads in our simulation is given by the runtime behavior of the simulation threads. This includes not only the simulation of memory accesses, but also the execution of the instrumented guest threads, as well as any activities within the runtime instrumentation framework (Pin). The main overhead for the latter part is instrumentation of pieces of code before being executed the first time. For HPC code, we expect this to be negligible.

The question remains if the described parallelization strategy is enough to reflect shared cache space usage and contention behavior on shared connections (that is between the shared cache level and main memory), as this is exactly what our tool should be able to show. If we assume that between common synchronization times of the guest threads (e.g. barriers inserted by OpenMP) the memory access behavior is the same, we should be able to reflect the cache space usage and contention effects.

Unfortunately, memory access behavior usually has very dynamic nature. For example, the standard cache optimization “blocking” results in misses for the first sweep over a data block. Other sweeps get hits. Thus, the memory access behavior cannot be expect to be constant even within one loop. However, one should keep two things in mind here. First, we want to reflect the shared usage of the last level cache and the contention on the connection to main memory. The optimization “blocking” usually is not done on this level. Further, we are most interested in OpenMP code that runs the same code on different data in the different threads. It would be interesting to analyze different implementation strategies here. But for now, this is future work.

Implementation Details

Simulation of the private L1 caches of a multi-core processor needs to take invalidations from other cores into account. Because of that, every test of whether a memory access will hit or miss a private L1 itself needs to access information that could have been modified by another simulation thread.

Because such a test has to be done for every memory access of the guest program, it should be fast. We use atomic instructions instead of locking. Actually, on x86, read and write operations of word-size to architecture-aligned addresses are assured to be atomic. For each memory block found somewhere in the memory hierarchy, we maintain a data structure that includes a tag and a bit-mask. Both currently are 32 bit, which allows to use regular read/write operations on a x86 64-bit architecture[42]. The mask specifies which cores have a valid copy of that memory block in L1. The bit-mask allows for fast invalidations and invalidation detection between simulation threads. For both L1 and L2, the cache state includes access recency information within each cache set to enable the simulation of a LRU replacement policy. The recency order is explicitly memorized with an array data structure for each set. Each access may re-shuffle array elements to update the order. Each element includes the tag of the cache line and a pointer to the corresponding data structure including the above-mentioned bit-mask.

On a memory access, the array for the corresponding L1 set is searched for a element with a valid and matching tag. The validity check is done using the above mentioned combined bit-mask/tag information using atomic access. As the L1 array itself is private to the simulation thread responsible for this core, there is no locking needed when the access results in an L1 hit. Further, during the pass through the cache lines of a set, we remember any invalid lines. On a miss, we reuse such an invalid line without triggering eviction of the least-recently used valid line. For memory accesses going to the shared cache level, two implementation variants are interesting. In the first variant, a lock gets acquired to access/modify the state of the shared cache level. In a second variant, we deliberately do not do any locking and accept races instead. Of course, this may result in an inconsistent tag array for an L2 cache set. However, the worst thing happening is a miss result instead of a hit. In this case, we trade accurateness of the simulation with simulation speed.

Next to feeding the memory access simulation, we generate histograms of averaged bandwidth requirements for various connections within the cache hierarchy of a multi-core processor. Our first implementation did histogram updating at every executed instruction of a guest thread, making this task by far the most expensive part of the analysis tool, around 80%. The first improvement comes from the observation that it is not needed to do histogram updates every instruction as observed bandwidths are averaged over quite a few instructions anyway. The handler is still called for every guest instructions, but does averaging and histogram updating only on every 16-th invocation. This way, the tool gets a factor of around 3 faster.

It needs some experience with runtime instrumentation frameworks to spot further options for reducing overhead. Temporarily removing the instrumentation for histogram updates, we could see the tool still getting almost three times faster. However, looking at a profile[43], the histogram update function only took around 10%. The issue here is that a callback into our handler is done for every guest instruction. Therefore, this function appears near to the top of the profile even though 15 out of 16 times, it is only updating and checking a counter. Further, each handler callback requires Pin to generate code to save registers on stack, call the function, and restore the registers afterwards. And unfortunately, this overhead in dynamically generated code does not show up in the profile as this overhead is distributed among all instrumented code. For any improvement, we should not run the callback on every guest instruction in

[42]If we want to simulate for than 32 cores, we need to extend the mask to more than 64 bits. This is still possible without locking by using `lock cmpxchg16b` on 64-bit x86 architectures.

[43]For this, we use the `perf top` utility from the hardware performance counter support of Linux.

the first place. For this, it is important to understand that Pin does instrumentation on a granularity of a superblock (sometimes called extended basic block, EBB, and in Pin-terminology *trace*). A superblock is a sequence of instructions with one entry and multiple conditional exits. Now, we do a callback only either at exits from a superblock or in the middle if we see that we passed more guest instructions than a given threshold. This threshold parameter specifies a maximal number of instructions after which a callback should be inserted for doing averaging and histogram updates. However, the real number of instructions passed between two histogram updates now depends on code structure and is different for every superblock. The consequence is that we cannot assume a fixed number of guest instructions passed by for averaging, but we need to introduce a parameter for averaging which takes the number of passed guest instructions into account (if the peak performance of our machine model is defined to be a maximal throughput of floating point operations, this parameter actually specifies the number of floating point operations done by the guest instruction passed). If an exit point of a superblock is less instructions “away” from the start of the superblock, the callback may not actually result in a histogram update, as we still maintain a counter to not call the updates too often. This is important for code with small superblocks or early exits. With this change in place, we were able to make the tool again a factor of two faster.

Mapping Results back to Source

In the current implementation, there is nothing about relating simulation results back to source code. This of course is crucial for any performance analysis tool. We shortly mention the planned extensions here. It is easy to map every guest instruction back to function name and source lines using debug information. This feature is provided by Pin. However, the issue to solve here is how to map bandwidth requirements averaged over the execution of multiple guest instructions to single positions in source code. To this end, the question is what a user expects from the tool. He wants to see where in the code large bandwidth requirements come from. If there is an inner loop, all the instructions in the loop should be annotated with the steady state bandwidth requirement. To achieve that, we can maintain min/max/average bandwidth requirement numbers for each guest instruction. After every simulated time unit, for each instruction responsible for the current averaged bandwidth requirement (we can use a ring buffer with the size of the used window to have this information available all the time), we update the corresponding numbers. Even better, we could maintain histograms for each instruction. However, if this proposal is implemented in a naive fashion, the tool would become very slow.

The solution to get faster here is similar to the way of maintaining the bandwidth requirement for the complete program as described above. Instead of updating numbers per guest instruction in every time unit, we do it only after e.g. 1000 time units (ie. using sampling). This usually is enough for a performance analysis tool: the dominant program phases run so long that sampling still provides enough significant information.

Simulation Speed

Table 4.8 shows achievable simulation speed for different applications and different thread counts using an Intel Core-i7 Ivy Bridge quad-core processor and the Intel compiler 13.0 with optimization flags "-O3 -march=native". The L1 cache of the simulator is configured to have a size of 256 kB (associativity of 8), and L2 cache has a size of 3 MB (associativity of 12). The parameter for the threshold to insert a callback for histogram updates is set to 16.

We show numbers for a native run without simulation as well as simulation with and without locking for L2 (see Sec. 4.3.3 above). Also, we provide two lines of results for each application and thread count. In the first line, we show numbers for user time (u), that is the sum of time spent in all threads, and in the second line wall-clock time (w). Next to simulation times, we show slowdown factors relative to the corresponding native runtime in parenthesis. The first application *fib* is sequential recursive calculation of Fibonacci numbers. This is a best case for the simulator as memory accesses are expected to hit the MRU (most recently used) spot in cache sets of L1 most of the time. The other two applications actually are variants of a 2D Jacobi solver presented in more detail in the next section (we run 240 iterations with matrix side length of 2500). The *naive* version is a worst case for the simulator with a lot of memory accesses. The *optimized* version has much less memory accesses but still transfers a lot of data within the cache. Both variants only have a limited amount of data sharing, and they work mostly on their own data.

	Threads		Native	Without lock	With lock
fib 40	1	user	0.40s	28.5s (71x)	28.0s (70x)
		wall	0.40s	28.4s (71x)	28.1s (70x)
naive	1	user	2.06s	81.4s (39x)	85.6s (41x)
		wall	2.09s	81.8s (39x)	86.0s (41x)
	2	user	3.47s	88.7s (25x)	119.3s (34x)
		wall	1.80s	45.8s (25x)	61.5s (34x)
	4	user	6.91s	188.8s (27x)	238.4s (34x)
		wall	1.80s	49.7s (27x)	66.4s (37x)
opt.	1	user	1.11s	70.8s (64x)	74.0s (67x)
		wall	1.14s	71.2s (62x)	74.4s (66x)
	2	user	1.21s	82.0s (68x)	88.0s (73x)
		wall	0.80s	42.7s (53x)	45.5s (57x)
	4	user	1.56s	97.0s (62x)	118.4s (76x)
		wall	0.40s	26.6s (67x)	33.1s (82x)

Figure 4.8: Runtimes and slowdowns of our analysis tool.

For a user of the tool, slowdown factors in regard to wall-clock time are important, as these are the ones which will be noticed. As the numbers show, we manage to stay between factor 25 and 82. This always is below factor 100, which we assumed to be the highest slowdown factor an user would accept as explained in the introduction.

Another nice observation is that our simulation slowdown does not get higher with a larger number of threads. That is, benefits due to parallelization in a multi-threaded application transfer over to the simulation. In other words, the simulation scales quite well. Sometimes, slowdown numbers actually improve with larger number of threads as can be seen with the naive application. We remark that this code, if executed on real hardware, is heavily limited by bandwidth to main memory. When run within the simulation, this pressure goes down and reduces the slowdown of simulation this way.

The results show that the slowdown is higher with code which either does not do much memory accesses or which mostly hits the cache on memory accesses. For *fib* with a slowdown of 70, the simulator actually only has to look up the MRU spot of the corresponding L1 cache set for most memory accesses to see that these actually are L1 hits. Instead, for *naive*, we observe a slowdown of only 40 or better. On the first look, this may be counter-intuitive. However, if one takes into account that the tool itself never can run only in L1, the numbers make sense. For *fib*, which runs on a real processor perfectly in L1, we make the allover memory access behavior worse by running within our tool. For *naive*, it is vice-versa.

If we compare the slowdown numbers between the simulator without vs. with using locking, it is clear that the difference has to go up with higher number of threads as more serialization will happen. The largest difference can be observed with wall-clock times for the naive application at 4 threads (factor 27 vs. 37). Locking is done only within the cache simulation part of the tool and only when the L2 cache is involved (with *fib*, numbers are the same as L2 is not involved there and therefore, no locking is needed). With *naive*, the cache simulation always needs locking which makes up a large portion of total runtime. Therefore, serialization due to locking is most visible in that scenario.

Trading Simulation Speed for Accuracy

Exactly for the case just mentioned (results with *naive*), simulation accuracy should be worse than with the other codes. This should be reflected by changes in the number of L2 misses. Analyzing results in more detail (not shown in the table) one can see that without locking, threads 1 to 3 (the master thread 0 has a slightly higher number due to some initialization code) all have exactly 47.0 million L2 read misses. The numbers are actually exactly the same with locking. Looking a bit more into simulation statistics, we see that locking indeed has a large effect on L1 invalidations. Without locking, the number of invalidations in each thread is between 5 and 7 million. With looking, the invalidation numbers consistently are below a million. However, from reasoning about the algorithm, we would not expect 7 million invalidations in one thread. Thus, we use locking for the results presented in the next section.

4.3.4 Case Study

For the following examples of the bandwidth curves, we analyzed an iterative Jacobi solver over a quadratic domain $N \times N$ with fixed boundary condition using the following

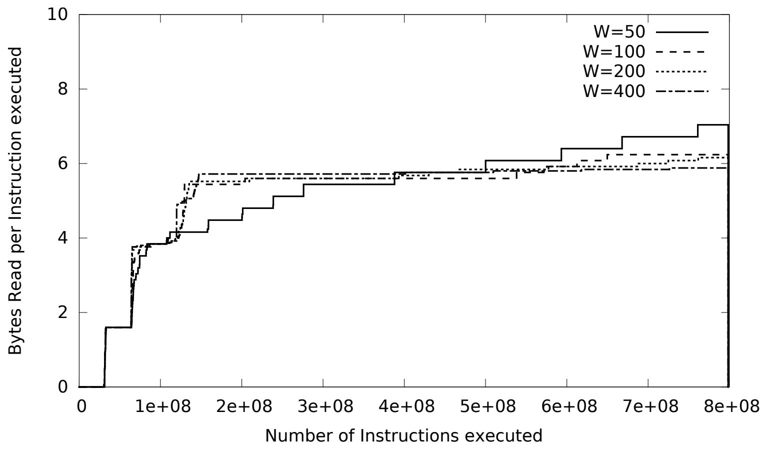


Figure 4.9: Influence of smoothing parameter W (50, 100, 200, 400), naive version.

update rule:

$$v_{x,y}^{i+1} := \frac{1}{4}(v_{x-1,y}^{i} + v_{x,y-1}^{i} + v_{x+1,y}^{i} + v_{x,y+1}^{i})$$

The solver allocates two matrices. One is used to store values at even iterations and the other for values at odd iterations. With $N = 2500$, the alternately used two matrices need 100 MB of memory with every value stored in double precision. The naive version does one sweep simultaneously over both matrices for each domain update. It loads 16 bytes from and writes 8 bytes to memory for each element update. We note that 2 rows fit easily into the on-chip cache. Further, 8 of the 16 bytes loaded are caused by the read-for-ownership done before each write. After some initialization, we run 24 iterations. Intel compiler 13.0 is used with OpenMP enabled.

Influence of Smoothing Window Length

Fig. 4.9 shows the effects of changing the parameter W for smoothing the bandwidth requirements. The 24 iterations need around 800 million instructions. The average number of bytes read is around 6 bytes per instruction (the actual number depends on the code produced by the compiler) most of the time. The lower numbers (around 150 million instr.) come from start-up and initialization. As the solver code in the naive version consists of a tight inner loop with uniform memory access requirements, we would expect the plot to show a simple line throughout the whole program run. However, for $W = 50$, this is not true. Checking the assembler code, the compiler does some unrolling and 50 instructions are not enough to see the steady state behavior for the inner loop. However, the curves for $W = 100$, 200, and 400 are almost the same.

Effects of Caches

In the following, we use $W = 200$ as smoothing window and the cache hierarchy of Fig. 4.6. Each core has a private 32 kB L1 data cache and there is a shared 3 MB L2

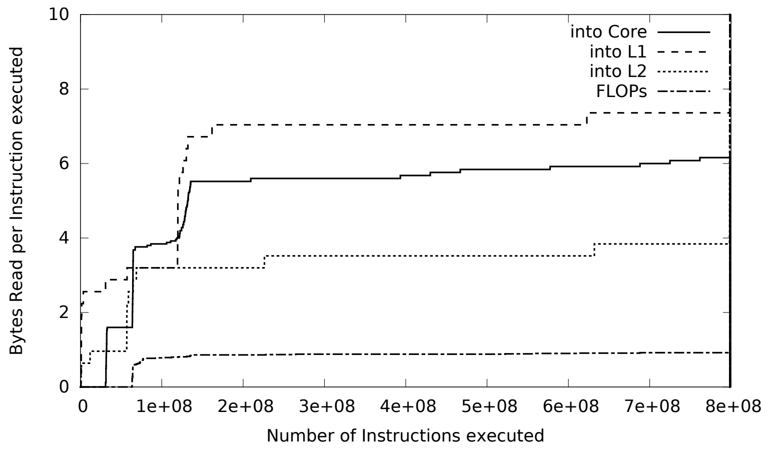


Figure 4.10: Bytes read into Core, L1, shared L2, and FLOPs, naive version.

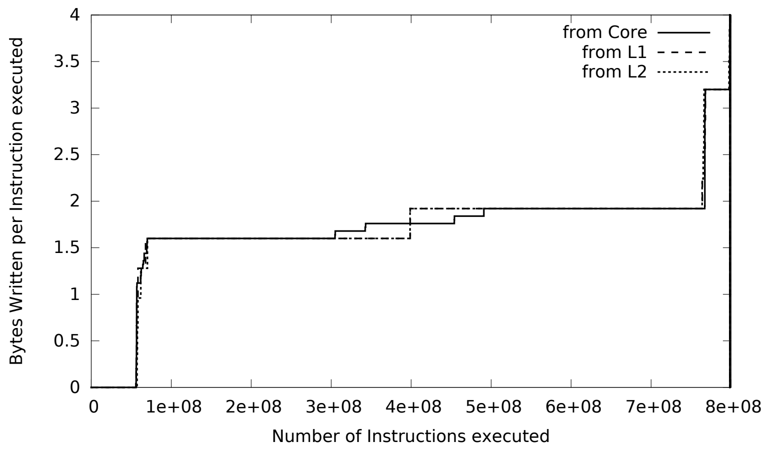


Figure 4.11: Bytes written from Core, L1, shared L2, naive version.

cache. As most HPC codes do fit into L1 instruction cache, we ignore loads into the memory hierarchy due to code fetched by the cores.

As in the previous figure, for Fig. 4.10 we use the number of instructions executed as time unit. For bytes read, the curves show different data paths: bytes read by Core (as in Fig. 4.9), bytes read from shared L2 into L1, and bytes read from main memory. This figure shows that the bandwidth requirement for bytes read into L1 can be higher than for bytes read by the core. This can be explained. With $N = 2500$, the L1 cache is not large enough to hold even one line of each matrix and thus, each time data has to be reloaded. This in itself just means the same requirements by the core and L1. However, each write to a newly allocated cache line also triggers a prior read-for-ownership (RFO), explaining the larger figures for bytes read into L1. We also can measure the number of FLOPs executed per instruction in the same way as the numbers for bytes transferred. As can be seen, the code does around 1 FLOP per instruction on average. As the code does not only consist of instructions doing

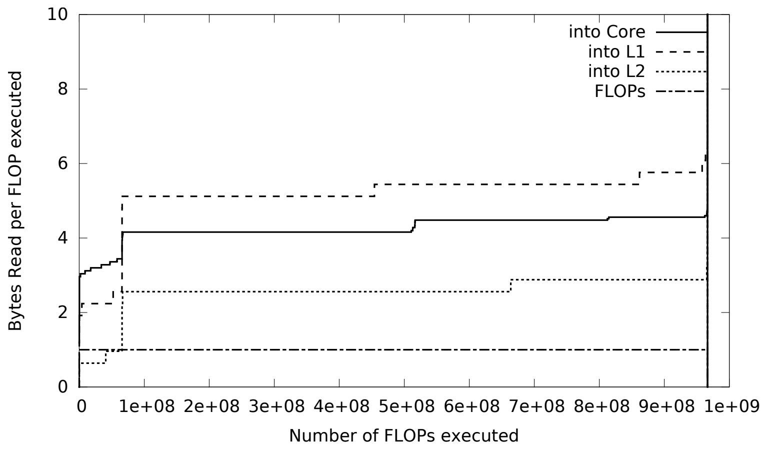


Figure 4.12: Bytes read into Core, L1, shared L2, naive version.

floating point operations, to achieve the observed average, the code obviously is well vectorized (we correctly count multiple FLOPs depending on SIMD instruction from x86 SSE or AVX extensions). In relation to L2 requirements, there are 4 bytes loaded from memory per FLOP executed.

Similarly, Fig. 4.11 shows curves in regard to bytes written. For the writes, the caches can not help at all. The average number of bytes written per instruction executed is around 1.8. The peak on the left (around 3.2 bytes) comes from the initialization phase.

Normalization to FLOPs executed

Up to now, we used as time unit the instructions executed. Fig. 4.12 shows the same Fig. 4.10 (bytes read), but related to FLOPs executed. First, one can see that the FLOP count, normalized to FLOP count, is a straight line at 1, as expected. There are almost one billion FLOPs executed, and there is a strange peak at the very end: as start-up does not do any FLOPs, there is a huge number of bytes read for the first FLOP executed in the program (which is ordered to the left).

Visualization of Cache Optimization

For cache optimization, the domain is recursively split into blocks fitting into L1. However, within one iteration, there is no reuse of data. We must block over multiple iterations to get reuse. This is done by doing multiple passes over the small blocks that fit into L1, making sure that we still maintain all data dependencies. Because of missing dependencies on block boundaries, the second iteration on a block must not update a boundary of depth 1, the third not a boundary of depth 2 and so on. When two neighboring blocks are done, the missing borders in-between are updated. As block splitting is done recursively by bisection (depending on block dimension either horizontally or vertically), doing the border updates after coming back from the bisection

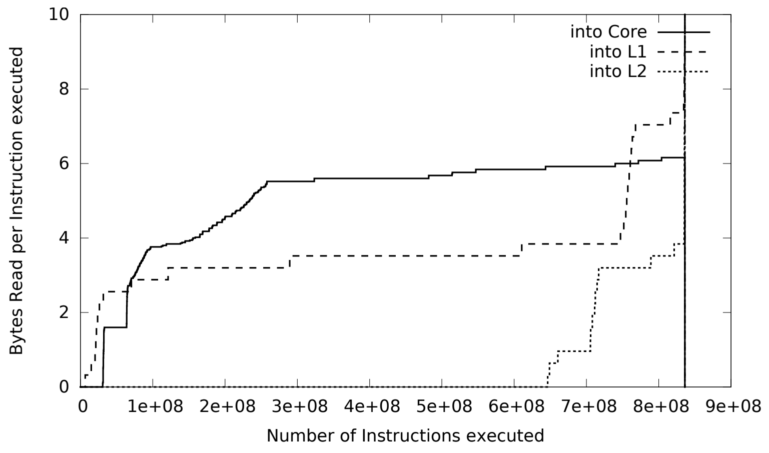


Figure 4.13: Bytes read into Core, L1, shared L2, blocking version.

is straight-forward. The optimized version of the solver has the number of iterations to block as variable parameter.

We show the results corresponding to the naive version (Fig. 4.10) in Fig. 4.13. We use an iteration blocking of 12. With blocking, the number of instruction executed is increased to around 840 million. The curve showing reads from the core did not change much though, in contrast to the figures for bytes read by L1 and by L2. L1 bandwidth requirements is much reduced. Typically for blocking, only for a small portion of runtime the code has the same bandwidth requirements as the unoptimized, original code. Similar effects can be seen for L2. However, most of the time, there is nothing read from memory at all. This shows the effectiveness of the optimization.

4.3.5 Multi-threaded Code

The OpenMP version of the 2D iterative solver splits the matrices into slices of equal size. As each thread executes the same code, the curves are the same for the privately used data paths, and are scaled by the number of threads for the shared connection to memory.

Table 4.2 shows user runtime numbers (threads summed up) for the example code measured on a quad-core Ivy Bridge (2.7 GHz), and the time predicted by our model to be spent at least due to a 15 GB/s read limit from memory. As expected, in contrast to the blocked version, the memory limitation plays an important role for the naive version, and becomes more significant in relation to measured runtime for a larger number of threads.

[s]	runtimes				due to limitation			
threads	1	2	3	4	1	2	3	4
naive	0.23	0.38	0.57	0.76	0.13	0.28	0.43	0.59
blocked	0.13	0.16	0.20	0.26	0.02	0.04	0.06	0.08

Table 4.2: Runtimes of iterative solver and estimations due to memory limitation.

4.3.6 Conclusion on Bandwidth Requirement Analysis

We propose a way to analyze the data transfer requirements of applications independent from other effects. For an example code, the resulting visualizations are quite insightful. The assumed peak resource restriction in the machine model may be too optimistic for good estimations in the absence of bandwidth bottlenecks. But even in this case, our method at least assures that memory bandwidth is not an issue. An analysis tool must provide a relation to source lines/data structures. We plan to use the sampling approach as sketched above. Further, bandwidth requirement analysis can be combined with spatial locality analysis to show how much of the transferred data is actually used by cores. The combination will tell whether data layout changes are useful to achieve better bandwidth behavior.

4.4 Bandwidth Dominance Analysis

Bandwidth curves already provide some insight into how given physical data paths in the cache hierarchy of a multi-core processor get used, and where the major bottlenecks may happen due to extraordinary bandwidth requirements. Further, from a bandwidth curve and a corresponding real bandwidth limitation, we can exactly derive the time Δt_p that the bandwidth limitation at this data path p adds to the best-case runtime of the program (see Fig 4.7). Now, let us assume that due to a cache optimization, we are able to push the bandwidth requirements at data path p below the real bandwidth limitation. Does our program now run faster by Δt_p? Actually, no. At the same points in time of program execution when data path p was a bottleneck, there could have been another data path p' which also was a bottleneck but was hidden by the bottleneck at path p before. Whenever multiple bottlenecks (ie. bandwidth requirements are higher than the real bandwidth limitation at multiple data paths) exist, one always hides the performance issue of the others.

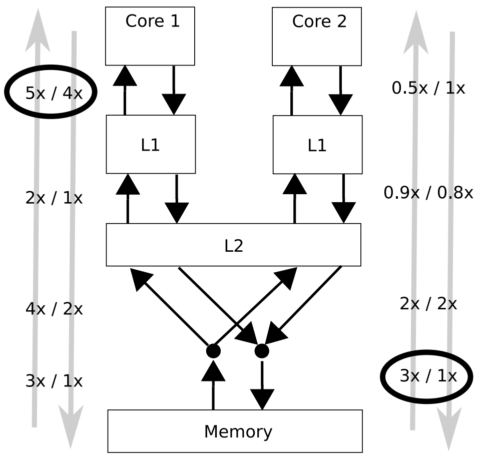


Figure 4.14: Throughput slowdowns for threads on 2 cores.

What is missing in our bandwidth requirement analysis? The memory accesses of one thread may traverse over multiple data paths from core to main memory. For each of these connections, a real bandwidth limitation may result in a bottleneck. If there is more than one bottleneck at the same time, which one is actually relevant for allover performance? Of course, the bottleneck is relevant that results in the highest slowdown factor, as this will hide (or *dominate*) all other bottlenecks. At a given point in time, there may not be any bottlenecks at all. This happens when all bandwidth requirements are below their respective bandwidth limitations. In this case, the performance is determined by the assumed peak performance of the core either given by a maximal throughput of instructions or floating point operations per clock cycle.

Fig. 4.14 shows an example. We use our tool presented in the previous section (Sec. 4.3). For two threads running on two different cores of a multi-core processor at a given point in time, we show for each data path between a core and main memory the ratio between bandwidth requirement derived by the tool and an assumed bandwidth limitation. The two numbers annotated to each data path relate to read/write directions. A ratio equal to or below 1 denotes that the data path does *not* represent a bottleneck. The thread on the left side (running on Core 1) actually gets restricted by bottlenecks from almost all connections (factor "1x" is exactly using available bandwidth). Overall, the slowdown of 5x (encircled in the figure) is determined by the read path from L1 cache. As the thread cannot run faster due to that limitation, the limitations on the other data paths between Core 1 and main memory do not matter. For the thread on the right, the limiting path is the read direction from memory (factor "3x").

There are merge points shown in Fig. 4.14 between the shared L2 cache and main memory. Such merge points do not exist in reality, but allow the model to reflect the observation that for most modern multi-core processors, the data path to main memory cannot fully be exploited from one core only. E.g. on recent Intel Xeon processors, a multi-threaded stream benchmark often can use the full main memory bandwidth only running on at least three cores. On the one hand, this may be deliberate design, e.g. by constraining the number of outstanding memory accesses which come from one core (which needs smaller buffers to remember the outstanding loads), or it may be an effect of a fixed prefetch distance setting of the hardware prefetcher in private per-core caches above the shared cache. In any case, to enable the specification of a significantly lower per-core bandwidth to main memory than is achievable by multiple cores, we need to introduce the two merge points (for read and write direction) with separate data paths above and below. For these paths, different limitations now can be provided.

4.4.1 Model Details

For detecting the dominant bottlenecks in a given multi-threaded program, we start from the analysis tool described in Sec. 4.3. Fig. 4.15 presents all throughput limitations we take care of. Our tool is able to measure not only bytes transferred between components of a multi-core processor, but also floating point operations executed. Similar to bandwidth limitations, we can specify a maximal peak performance of ALUs within cores which may result in a slowdown[44].

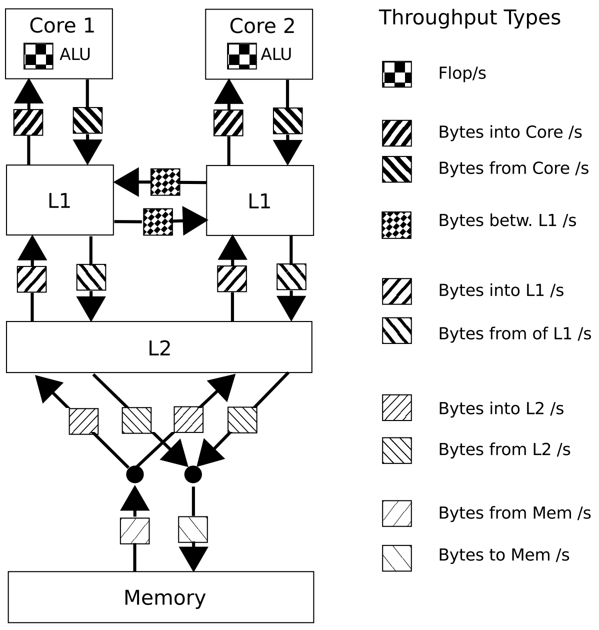


Figure 4.15: Throughput types with shading.

Other connections visible in Fig. 4.15 not discussed yet are the drawn channels between L1 caches. Actually, such connections do not exist on recent multi-core processors but are introduced into the model to allow the specification of bandwidth limitations for the communication among threads. Actually, the limitation reflects latency due to cache coherence transactions for loading a line into an L1 cache that previously was written to from another core and still is remaining in that remote L1 cache. From experimentation we see that the maximum bandwidth achievable for such situations is lower than what a shared cache level (here L2) can provide. The (artificial) data paths between L1 caches are relevant for a thread only for reading data from other L1 caches. There is no writing direction, as cores cannot directly write into L1 caches of other cores. Currently, to keep the model symmetric, we do not allow different bandwidth limitations to be specified for inter-L1 paths between different cores (this may be a useful future extension). Further, we assume that cache coherence transactions do not overlap and thus, we only assume one limitation relevant for all data that gets fetched by cores from L1 caches of other cores.

The need for merge points between L2 and main memory was already mentioned above. These allow to specify lower per-core limitations of the data paths to main memory. In regard to main memory requests, real multi-core processor hardware actually has to employ a scheduling algorithm that decides about the order in which main memory requests are passed to a memory controller, and thus to main memory. One can imagine different policies involving priority or fairness schemes. Such schemes

[44]This could easily be extended to integer operations, that is, to different kinds of execution units. However, we focus on HPC applications here for which FLOPs are most interesting.

could be handled by the model in the merge points. In our implementation, we use a fairness scheme which demands the same slowdown factor for each core whenever real bandwidth requirements are above the specified limit. This behavior is also shown in Fig. 4.14. Both cores are slowed down by the same factor of 3 for the reading direction. This policy means that each core gets a portion of available bandwidth that is equal to the ratio of its current bandwidth requirement and the total bandwidth demand of all cores (a different policy would be one that tries to equally distribute available bandwidth among all cores).

Fig. 4.15 maps bandwidth/throughput limitations to different hatching patterns. We will makes use of these in the results from the case study in Sec. 4.4.4.

4.4.2 Extension to Bandwidth Requirement Analysis

As described in Sec. 4.3.3, guest[45] threads are fixed to cores, and each simulation thread is responsible for one guest thread. Interleaved with the execution/simulation of one guest thread, a simulation thread does bandwidth averaging and histogram updating for all connections relevant to this guest thread. Here, we add the detection of dominant bottlenecks, as all needed information is available.

To determine slowdown factors induced from shared data paths (concretely, the paths to/from main memory), we need the total bandwidth requirement of all threads. To this end, every thread stores its partial bandwidth requirement into a variable that is accessible to the other threads. To get the current, total bandwidth requirement, each thread sums up the partial requirements and calculates a slowdown factor from it (summing up the partial bandwidth requirements implements the fairness scheduling described above). It is important to set the bandwidth requirement of a thread for a shared data path to zero whenever the thread is idle. Pin only drives simulation when running in user level code. However, a thread can become idle within a system call (e.g. when waiting on a barrier via calling the OS). As our simulation is not able to see what is happening in system calls anyway, we simply assume that in such system calls the own bandwidth requirement becomes zero. The same is true when a thread terminates.

Dominance detection of bottlenecks enables the tool to exactly predict the slowdown factor for multi-threaded code. This is done by collecting the dominant bottleneck for each point in time of guest program execution. For this to not result in a lot of data to collect, we use the same trick as with bandwidth curves. We maintain dominance histograms next to the bandwidth requirement histograms. While the latter aggregate bandwidths for each point in time of thread execution, the dominance histogram only remembers points in time where a corresponding data path was a dominant bottleneck. IN a post-processing step all dominance histograms relevant to one guest thread can be merged to calculate the additional time induced by the dominant bottlenecks. Further, this allows to attribute any additional time to corresponding data paths.

[45]The program that is run within the simulated model is called *guest* here.

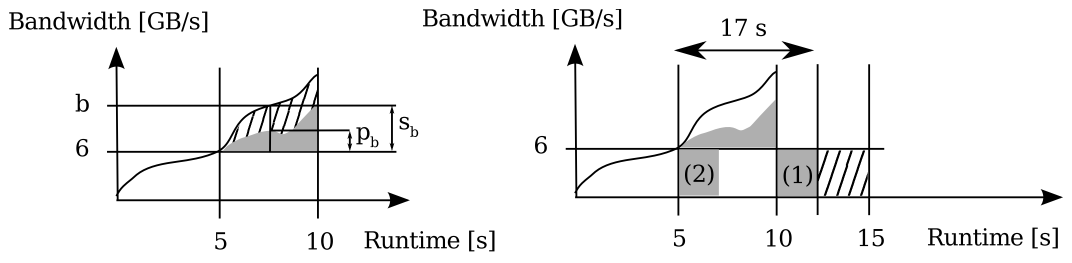


Figure 4.16: Bandwidth curve taking dominance into account.

4.4.3 Visualization

First, we explain additions to the bandwidth requirement curve visualization described in Sec. 4.3.2. There, Fig. 4.7 (left) showed an example of a bandwidth curve which we will reuse and extend in the following. We use the same notation as introduced there. The bandwidth curve is given by averaged bandwidth values $\bar{r}_{t'}, t' \in \{1, \ldots, T+W\}$ with T being the number of time units needed by the given program and W being the window size for averaging ($\bar{r}_{t'} = 0$ for $t' > T$). The amount of data transferred is $V = \sum_{t=1}^{T+W} \bar{r}_t$. Further, we assume a data path bandwidth capacity/limitation of L. The number of time units in the bandwidth curve with bandwidth requirement below L is t'_L.

Now, we want to see how often a given data path actually is a dominant bottleneck. To this end, we extend the bandwidth requirement curves as shown in Fig. 4.16. Similar to the example from Sec. 4.3.2, we assume a limitation of $L = 6$ GB/s which results in a number of time units both below and above L mapping to a time of 5 seconds elapsed, respectively.

In all points in time where the bandwidth requirement exceeds L, we either may have a dominant or a non-dominant bottleneck. Let us look at a concrete bandwidth value b which exceeds the data path limitation, i.e. $b > L$. Bandwidth values are discrete. Let δ be the minimal increment between discrete bandwidth values. With b_{max} being the maximal bandwidth observed, we set $t'_{b_{max}+\delta} = T + W$ as the number of time units needed for the whole program. Thus, $\bar{r}_{t'} = b$ for $t'_b \leq t' < t'_{b+\delta}$. We denote the number of time units in which bandwidth b was required as $\Delta t_b = t'_{b+\delta} - t'_b$. Further, for each bandwidth value b, in our simulation, we measured the number of time units with the data path being dominant. Let us denote this number by Δt_b^{Dom}.

The area in the diagram below the bandwidth curve relates to the number of bytes transferred over the data path. Originally, the complete area above the bandwidth limitation line and below the curve (this was the area using a hatching pattern) was used to calculate the slowdown induced by the limitation. With $\bar{r}_{t'} = b$ for $t'_b \leq t' < t'_{b+\delta}$, this area relates to the amount of bytes given by

$$V_L = \sum_{t'=t'_L}^{T+W} (\bar{r}_{t'} - L) = \sum_{b>L} \sum_{t'=t'_b}^{t'_{b+\delta}} (b - L) = \sum_{b>L} \Delta t_b (b - L) \ .$$

Only a part of the hatched area belongs to time units where the data path is dominant. Using Δt_b^{Dom}, we observe that the amount of bytes transferred during dominant times units is given by

$$V_L^{Dom} = \sum_{b>L} \Delta t_b^{Dom}(b - L) \; .$$

We want to mark the part of the hatched area belonging to dominant time units. Every time step t' belongs to some bandwidth $b > L$ due to $t'_b \leq t' < t'_{b+\delta}$. We know that for each bandwidth b the ratio of dominant time units is $\Delta t_b^{Dom}/\Delta t_b$. For the visualization, we can reflect this ratio of dominant time units at b by graying out a proportional ratio of the hatched area by vertical partitioning for all the time steps relating to b. This is shown in Fig. 4.16 (left). Here, $s_b = b - L$ is the vertical span of the hatched area for all time units belonging to bandwidth b. The vertical span to be grayed out is given by p_b which is calculated according to $p_b/s_b = \Delta t_b^{Dom}/\Delta t_b$. We note that the grayed-out area should match V_L^{Dom}.

Similar to our observations in Sec. 4.3.2, we want to quantify the slowdown induced due to dominant time units. This is simple now: the slowdown is derived from V_L^{Dom} in contrast to V_L. This is shown in the right part of Fig. 4.16. The additional time needed due to the limited bandwidth capacity of the data path again is shown beyond the 10 seconds mark on the x-axis. The dominant part is grayed out, annotated with "(1)". The non-dominant part is still shown as hatched area. The hatched area does not actually induce any slowdown because the original time units belonging to this hatched area are dominated by other data paths. The same is true for the area below the 6 GB/s line between the 5 and 10 seconds mark on the x-axis. Only the gray area annotated with "(2)" will influence final runtime, as only here the given data path was dominant. For the other part of runtime in this region, another data path is dominant and will contribute to the final runtime of the program. The time units belonging to "(1)" and "(2)" can be used for exact calculation of runtime of the program, as here, the given data path was dominant. Further, these time units are not dependent on the accuracy of assumed peak performance of our machine model.

The same visualization can be done for all connections relevant for a thread. The sum of additional time needed for all dominant parts of data paths will give us the total *additional* time due to bandwidth limitations. We note that this total additional time is not dependent on the accuracy of assumed peak performance of our machine model. As such, depending on how much time the program is hitting a data path capacity limitation, we can provide quite good estimations of the time needed on the real system.

As we now have determined dominant data paths and their contribution to total runtime for any point in time where a bottleneck exists due to some bandwidth limitation in our machine model, we can use this information for another visualization providing a better overview than the single bandwidth curves.

Stacked Bar Diagrams

As just described, with the help of dominance histograms and corresponding modification of the bandwidth curve visualization, we can extract exact times determined by different physical data paths in the system, and show their contribution to the total runtime of the threads running in the system.

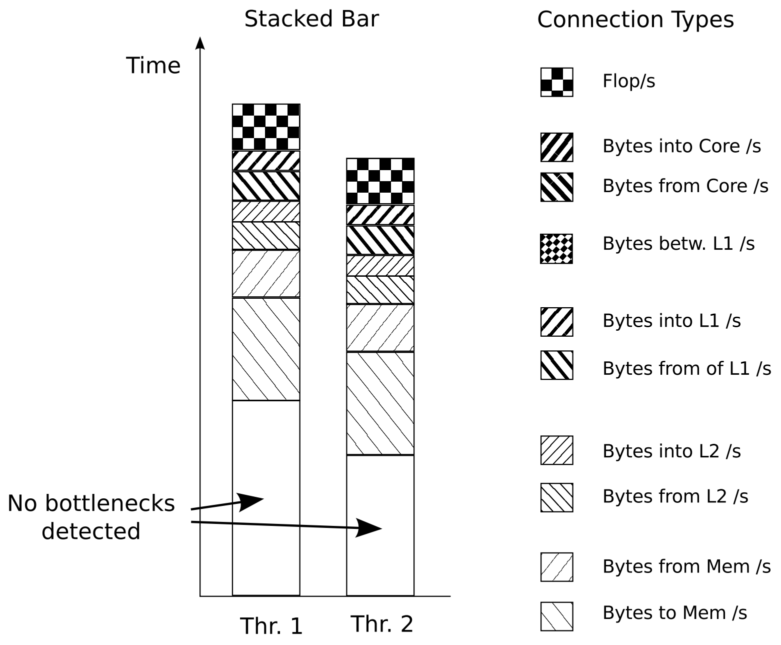


Figure 4.17: Bandwidth curve taking dominance into account.

The time steps within execution of a thread can be classified in two categories. First, for some time steps, there may be no data path in the system where bandwidth requirements hit the data path capacity. Second, for all other time steps, we will be able to identify a dominant data path slowing down execution.

The first category of time steps can be annotated as "running with peak performance". Actually, these are the only portion of thread execution runtime that depends on the (not-so-exact) fixed IPC assumption in our simple architecture model. However, we also assumed that all data can be transferred with highest possible bandwidth due to adequate prefetching. However, this is an unrealistic best-case assumption and the reason that our runtime estimation can be too low.

Fig. 4.17 shows a stacked bar breakdown visualization of time for two threads. The idea to show time breakdown as stacked bars is taken from [52]. As peak performance in our machine model, we use a fixed IPC number, ie. the throughput of instructions executed is limited. The floating point units are seen as some kind of data path with limited throughput. Thus, for the portion of time caused by "Flop/s", we run into the peak performance of the floating point units and not are not restricted by instruction throughput. Here, the threads show slightly different behavior. One thread is running longer than the other. This is quite usual for OpenMP programs, as this programming model uses a fork-join model. There, a master thread often does a bit more work than the other threads, such as initialization. Another reason for different runtime of threads may be load imbalance.

To exclude initialization from the analysis, it would be quite easy to change our tool to allow to run parts of the program without doing measurement.

Performance	4 instructions per clock 2.7 GHz
Cache parameters L1	32 kB, assoc. 8, LRU, 64 byte line size
L2	3 MB, assoc. 12, LRU, 64 byte line size
ALU	4 FLOPs per clock
Core/L1	86.4 GB/s read, 43.2 GB/s write
L1/L2	34.8 GB/s read, 18.9 GB/s write
L1/L1	14.9 GB/s
L2/Mem per core	14.9 GB/s read, 10.0 GB/s write
L2/Mem total	23.0 GB/s read, 10.3 GB/s write

Table 4.3: Used cache parameters and bandwidth limitations.

4.4.4 Case Study

In the following, we present numbers from the same application as in Sec. 4.3, using the extended bandwidth analysis tool with dominance analysis. This enables visualizations which show a break-down of thread runtime with time portions mapped to dominant bottlenecks of physical data paths in the memory hierarchy of a multi-core processors. Instead of showing separate stacked bars for each thread of the multi-threaded application, we sum up all threads and show one bar for a complete application run.

The 2D Jacobi solver is given in two variants. In the first (naive) version, we expect a lot of memory accesses. In the second, we do cache optimization by blocking of 12 iterations each time. The Jacobi solver works with matrices of side length 2500 (resulting in 50 MB matrix sizes) and we run 48 iterations in our simulation. In Table 4.3 we show the cache parameters and bandwidth limitations used. The numbers reflect a quad-core Intel Ivy Bridge processor at 2.7 GHz. The limitations were determined with micro-benchmarks.

First, we show the stacked bar graphs for the memory-bound 2D Jacobi in Fig. 4.18. The top diagram shows absolute sum of user time for all threads. As the time goes up for higher thread numbers, the observed wall-clock time is almost always the same. It is 0.35s for 1 thread and 0.25s for the other thread counts. However, more interestingly, we see that most of the time, writing to memory is the dominant bottleneck with reading from memory on 2nd place. With one thread, the bottleneck is actually the limitation from the data path “memory per core”, as we run only on one core. However, that limitation is still visible with 2 threads for reading.

Fig. 4.19 shows the results for the cache optimized version. Obviously, there is much more time spent now without any bottlenecks in the memory hierarchy (the empty rectangles at the bottom of the bars). Further, some time is now even limited by the ALU (the checker pattern). While there is still some time where writing to main memory is dominant, it is important to look at absolute numbers. The user time of the version running with 8 threads actually is a factor of 4 smaller now. This probably also maps to wall-clock time.

Finally, in Table 4.4, we compare the time we get from the simulation to real user

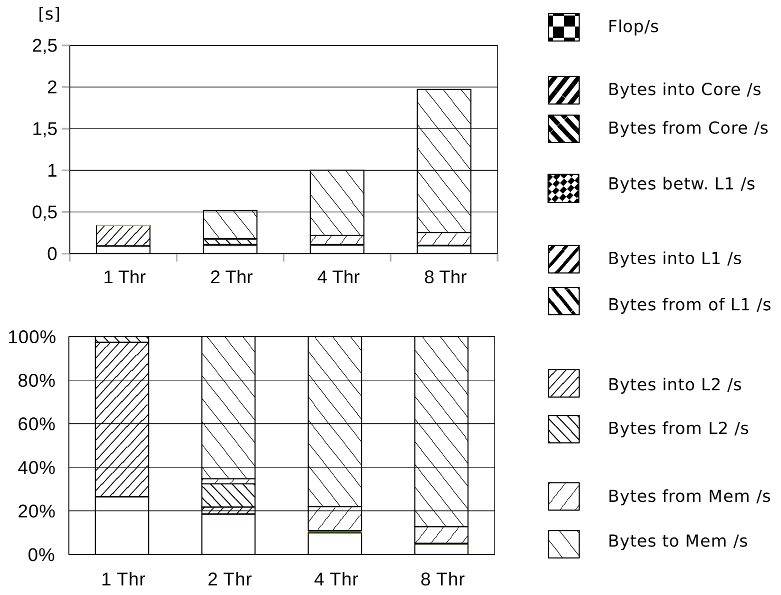


Figure 4.18: Dominance Analysis for naive 2D Jacobi.

		1 thread	2 threads	4 threads	8 threads
Naive	Simulated	0.35s	0.52s	1.00s	1.97s
	Real	0.50s	0.72s	1.47s	3.06s
	Error	31%	28%	32%	36%
Optimized	Simulated	0.18s	0.21s	0.27s	0.45s
	Real	0,30s	0.33s	0.42s	1.01s
	Error	38%	37%	35%	55%

Table 4.4: Comparison of estimated and real runtimes.

runtime on the Ivy Bridge processor. First, this processor only has 4 cores, so the real numbers for 8 threads actually use Hyper-threading. Thus, it is expected that errors between simulation and reality are much higher for the 8-threads case. Further, similarly to the optimized version, the time without any bottleneck in the memory hierarchy makes up quite a large portion of total runtime. For this case we expect the error among reality and simulation to be quite large. The reason being that accuracy here depends heavily on the simple fixed IPC model. As numbers show, this assessment is true.

We see that our simulation consistently estimates runtime to be around 30% below the real runtime. We note that our model only takes data into account which actually was transferred by the program. One source of error could be that we ignore cache coherence transaction. However, as the bottleneck limitations were calibrated from real benchmark code, this reasoning may be wrong.

A much simpler explanation is that we assume perfect prefetching and best exploitation of physical data paths up to their capacities. However, as motivated in the introduction of the method, our bandwidth requirement/dominance analysis only is

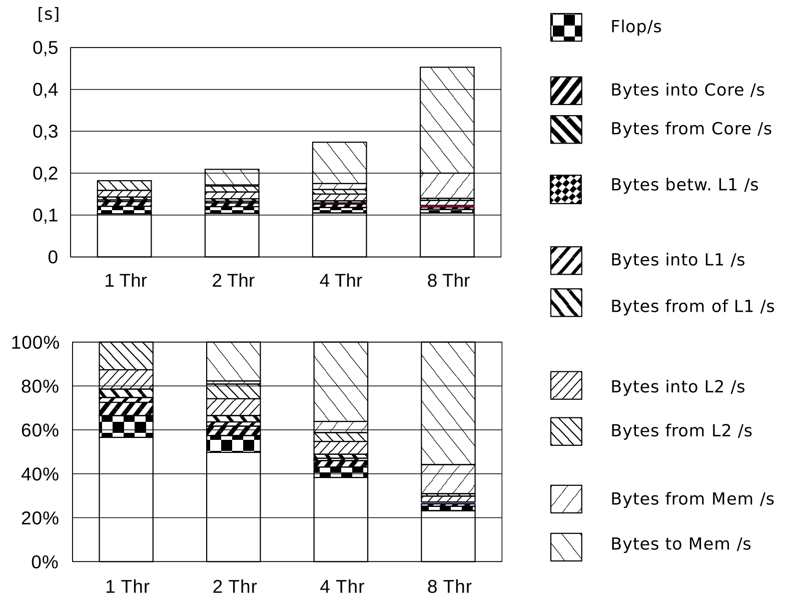


Figure 4.19: Dominance Analysis for optimized 2D Jacobi.

meant to provide (lower) bounds which are derived from bandwidth limitations. If we introduce a constant correction coefficient of 1.3 for possible latency effects, we actually are astonishing near to real numbers.

Finally, we note that inter-L1 paths do not actually exist. If there is a lot of communication among cores, this actually need to be passed between the cores over the shared caches, influencing other data paths within our model. However, for the given 2D Jacobi case study, data is partitioned among threads, and communication among cores only can happen at partition boundaries which should be negligible.

4.4.5 Conclusion for Bandwidth Dominance Analysis

In this section, we presented a machine model that uses the assumption of unlimited bandwidth for data paths within the memory hierarchy. Using results from this model, we derive slowdown factors for bandwidth limited systems. We show that this strategy can provide interesting insights.

First, it explicitly provides slowdown factors by comparing bandwidth requirements of a program with assumed bandwidth limitations in the memory hierarchy. Such numbers cannot be measured on real systems. Bandwidth curves are quite easy to understand for the user of the analysis tool. Second, even though the machine model used is very simple, we can derive relatively accurate runtime estimations. The more bottlenecks there are detected, the more accurate the time estimation will be. As the tool itself has an average slowdown of only around a factor of 50, it seems quite acceptable for a tool that can provide this level of detail.

We did not yet add relation of results to source code. This would be important for an analysis tool to give to users. However, it should be quite easy to add support for source code relation using a sampling strategy. This will allow for interactive bandwidth curves where users select regions and the tool shows the corresponding execution stream that was responsible for the given selected bandwidth. In the end, it may be enough to annotate source lines and data structures with the highest bandwidth requirement they are responsible for.

As use case, we presented a 2D Jacobi algorithm both in a naive and optimized version. To get more experience with the accuracy of our machine model with different kinds of HPC codes in practice, we want to study its behavior in the context of various HPC benchmarks. This may result in extensions of the model. However, we expect such extensions to always be a trade-off between simulation speed and higher accuracy in results only in the case where the code does not hit any bandwidth limitation. It is questionable whether this kind of higher accuracy provides new insights about the usefulness of the approach as performance analysis tool. Further, we argue that this kind of fine-tuning is not really needed. Either we detect data paths to be bottlenecks most of the time. Then, corresponding optimization should be done. In this case, the approach as presented already provides quite accurate estimations coming from the resource requirements and actual resource limitation. Both can be derived from the simple machine model with good preciseness. In contrast, if we see that a code does not hit resource limitations, accuracy will be worse due the simple architecture model. However, in this case, there is not much optimization to be done for which the approach could provide detailed insights.

We note that in this argumentation the purpose of the presented approach is crucial. It is about finding bottlenecks in regard to bandwidth or throughput limitations. Especially, the tool does not help with effects caused by latency. However, this is out of scope here. It needs further research if we can come up with a method that allows for detection of bottlenecks caused by latencies. Finally, we note that we propose simulation only as a complement to analysis tools relying on real measurements, to be able to provide more details than what we could collect on real systems. In this regard, we are convinced that the presented strategy is an important contribution beyond current state-of-the art.

5 Conclusion

This work focuses on the question of how well architecture simulation can complement traditional performance analysis tools which do real measurements. The hope is to provide the user with more details and insight on how to optimize an application which does not run with the expected performance. The target architectures of interest to this work are modern multi-core processors[46].

The challenge is to restrict the simulation to only run simple architecture models which both make sure that simulation slowdown is acceptable in practice, but also that results are easy to understand. The latter is important for users to be able to derive effective optimization strategies. At the same time, the model should be accurate enough for identifying issues which actually are also relevant on real hardware.

To this end, one important observation is that bad exploitation of the memory hierarchy of modern multi-core processors easily can make a difference in performance of factor 10 and more. We expect this to hold true for future systems built according to the von-Neumann architecture. This is the reason why this work concentrates on the memory hierarchy.

5.1 Summary of Research Done

To study the key questions just mentioned, as part of this work, we built several prototypes of tools:

- We implemented Callgrind, a cache simulator using dynamic binary instrumentation that can relate counts for memory accesses, hits, and misses to a dynamically built call graph representation of the execution (Sec. 3.1). While multi-threaded code can be analyzed, Callgrind only simulates a single cache hierarchy. The DBI (dynamic binary instrumentation) framework it is based on time-sliced (i.e. serialized) execution of threads. For an analysis tool, it is very important to get detailed context information, ie. relation to source core. From our own experience and observed usage by others, the tool often is used just to get the call graph itself. This already helps a lot to understand the inner working of a program. By providing function call counts, users can compare their expectations to find errors such as redundant function calls. The slowdown observed (including cache simulation) usually is between 40 and 70. Making Callgrind early available as Open-Source allowed us to collect feedback regarding ease of use and ease of understanding measurement results. By now it is in regular use by Open-Source projects[47].

- Important for usability is a simple graphical front-end which allows for fast browsing of results from the mentioned simulator (Sec. 3.2). By own experience and

[46]The generic observations of this work should easily map to many-core processors but also should be useful for accelerators such as GPUs.

[47]For example, OpenOffice uses it since quite some time for performance analysis, especially for running nightly performance regression tests to quickly detect code changes that negatively influence performance. Michael Meeks emphasizes the benefit of reproducible simulations for this use case [82].

comments from other users, the most attractive and used feature in this GUI is that navigation trough measurements can be done within the drawn call graph. The existence of this GUI motivated others to use it for own profiler tools (for example ADAPTOR [24] and XDebug, a profiler for PHP[48]).

- To allow for more detailed analysis of cache exploitation, Callgrind was extended by a hardware prefetcher simulation (Sec. 3.4), by cache-line usage metrics (Sec. 3.5), and an improved source relation feature (Sec. 3.6). Each of these extensions enables novel performance analysis procedures with corresponding optimization strategies. It proves that cache simulation allows for analysis methodologies that surpass traditional tools which rely on pure hardware measurements.

- To research the benefit of cache simulation in the scope of multi-threaded applications for multi-core processors, we looked at a difficult and unstable effect called *false sharing* which can be a significant performance issue. This is an example of a bottleneck depending heavily on interleaved execution behavior of threads, requiring exact time estimations. Therefore, getting exact counts for false sharing instances in the execution of a multi-threaded program is impossible using simulation of simple architecture models. However, as unstable behavior can also be observed on real machines, cycle-accurate simulation does not help either. Instead, we developed a tool that gives worst-case estimations on how often false sharing may happen (Sec. 4.2). An important aspect of a performance tool is to present bottlenecks by severity. This allows users to first concentrate on bottlenecks most relevant to performance. Our worst-case estimation works quite well in this regard. However, the tool has similar overheads as race detection tools (memory consumption and time overhead depends on the size of code sections between synchronizations) which may make it useless for a lot of applications[49].

- Another major obstacle for performance on multi-core processors is contention on shared physical data paths used by all cores. Using real measurement for detecting the severity of contention is difficult. Seeing that some resource is *almost* utilized to its full capacity does not really tell whether this resource is actually a relevant bottleneck. With simulation, we can find out how much of a resource a program actually needs to run at the computational peak of the system. We call this Bandwidth Requirement Analysis, and we developed an according analysis tool (Sec. 4.3). This tool not only produces easy to understand histograms for different data paths in the cache hierarchy of a multi-core processor but also gives good time estimations as long as bandwidth limitations play a relevant role for allover performance (Sec. 4.4).

From these prototypes and studies, we see that it is definitely beneficial to use

[48]`www.xdebug.org`

[49]A better way to detect false sharing is sampling of addresses in memory accesses using hardware performance counters on modern real hardware. Frequent accesses to the same data from different threads may be instances of false sharing. Only source analysis can verify this, and it also catches real data sharing. However, the strategy is simple enough.

architecture simulation for performance analysis in cases where caches and their exploitation play a relevant role for performance. However, to get a first overview on the performance characteristics of yet unknown code, we recommend the use of traditional tools such as PAPI [73] or the tools around the Performance Events For Linux [34]. These can tell whether the memory hierarchy is a bottleneck. If so, it is possible and worth-wile to use cache simulation for more detailed analysis.

There still are some cases in which cache-focused architecture simulation does not help. The performance of applications can be far from achieving theoretical peak performance when the code is slowed down due to pipeline hazards, such as control or data conflicts. Speculation in out-of-order pipelines may often go wrong due to mispredicted branches. Data dependencies may produce long critical paths resulting in stall cycles. Note that the latency of memory accesses can play an important role for such effects. However, simulation on this level requires complex models which makes it far too slow for usage in a performance analysis tool. False sharing effects fall into a class of bottlenecks which are not approachable by simulation of simple cache models as these effects depend on the fine-grained timely order of accesses to shared cache lines from different threads. The same argument holds as before: the needed model complexity for accurate prediction would make simulation too slow. However, there are approaches not relying on accurate model simulation which can prove useful, such as worst-case analyses.

Usage of Developed Tools

We list use cases here in which we made heavy use of the developed tools. We note that this list is not exhaustive, and we find ourselves used to this way of performance analysis as regular guidance for code optimization.

Development for the cache simulation tool-suite for performance analysis of the memory access behavior described in Sec. 3 started in the context of the DFG DiME-2 project [1]. In this project, the simulator was used to analyze the benefits of different cache optimization for 2D and 3D PDE (Partial Differential Equation) solvers. In this context, we also made good use of the extension for cache-line utilization. For example, this showed the benefit of further improvements within the so-called red-black technique within Jacobi solvers by separating the black and the red matrix elements [2].

In a cooperation with the Klinikum-Rechts-der-Isar, Munich, and in the context of the PhD work of T. Küstner, an algorithm for medical imaging is the subject of research [68]. Most time in this algorithm is spent in huge Sparse-Matrix-Vector operations with the sparse matrix generated by a Monte-Carlo method involving randomness. Due to that randomness in the matrix, standard compression methods improving performance are not possible. The cache simulator was used to understand the benefits of row/-column reordering [8]. The findings from this analysis triggered further research in dynamic code generation techniques to get rid of instructions managing loop nests [118]. Further, this resulted in work which tried to predict cache miss behavior in order to insert software prefetching into specialized, dynamically generated code [91].

5.2 Open Research Questions

Within the context of the topic of this work, there are still some open research questions. We list them here in no special order.

- The tool for bandwidth requirement analysis shows that it is possible to come up with accurate predictions under the assumption that bandwidth limitations are the main reason for performance issues. However, this is not always the case. For example, pointer chasing along a linked list makes the latency of every memory access becoming part of the critical path relevant for performance. Linked lists are seldom used in HPC code. However, it would be nice to detect such obstacles to performance. Data dependence between two memory accesses within a superblock can be detected at instrumentation time of a DBI (dynamic binary instrumentation) tool. Much more difficult is fast handling of data dependences crossing borders of superblocks. In any case, taking into account memory access latency within critical paths could make time estimations better for a larger class of applications than which are "covered" by our bandwidth requirement analysis.
- We presented Stack Reuse Distances in Sec. 2.1.2 as a good abstract metric for temporal locality. In student's work[50], this analysis was actually implemented, but it was not focus of this work. However, there are quite some ideas which may allow for fast estimation of reuse distance histograms especially when instrumentation can be supported by compiler information. The reuse distance histogram is an example of an analysis that embeds information of program behavior for different cache configurations at the same time. It seems possible to do a similar analysis in regard to cache-line usage statistics. At the moment, Callgrind can provide numbers only for a specific cache configuration.
- Recent work on performance optimization strategies which are not in direct relation to this work, focus on dynamic code generation techniques [118]. However, the author does not know any performance analysis tool that can give sensible results in the scope of dynamically generated code. While our cache simulator based on DBI does execute such code, there is no source code relation and thus, results become useless. Dynamic code generation needs to provide information about the relation to source code, and performance analysis tool needs to make use of it.

5.3 Final Remarks

In this work, we show that architecture simulation is not only a tool to facilitate exploration of architecture designs. It can be put into good use for performance analysis purpose, being able to provide more detailed information than what can be collected by measurement on real hardware. However, for this use case, it is crucial to use simple models. We show that there are a lot of different performance analysis approaches

[50]In a so-called "ERA Großprojekt" supervised by the author.

which give detailed insights into performance bottlenecks both for sequential as well as parallel code. The analysis approaches are designed to still provide worthwhile information even though used architecture models are simple and fast to simulate. We note that we found quite some cases where we were astonished ourselves about the accuracy of results for practical use cases.

While simulation based performance analysis obviously provides benefits, it cannot replace analysis tools relying on real measurements. There simply are situations where we want a tool to show specific information about runtime characteristics in the context of a given target architecture with unknown internal micro-architecture. However, simulation based analysis is a perfect complement for such tools if we know from real measurements that some hardware component is responsible for bad performance. In this case, simulation allows to look deeper into the performance issue when using adequate models and metrics.

Finally, we were not able to study adequate models for all kind of different performance bottlenecks, for example in regard to performance issues due to latencies in the critical path of an application. We see a lot of challenges for further research. However, we feel that we were able to significantly contribute beyond current state-of-the-art in performance analysis tools especially with our approaches targeting multi-threaded applications.

References

[1] Homepage of DFG project DiME/DiME-2.
http://www10.informatik.uni-erlangen.de/Research/Projects/DiME.

[2] Abschlussbericht des Projekts Ru 422/7-5 (Kennwort: DiME-2), 2007.
http://www10.informatik.uni-erlangen.de/Research/Projects/DiME/pubs/DiME_final.pdf.

[3] G. Almási, C. Caşcaval, and D. A. Padua. Calculating stack distances efficiently. *SIGPLAN Not.*, 38(2 supplement):37–43, June 2002.

[4] AMD. AMD SimNow simulator.
http://developer.amd.com/tools/simnow/Pages/default.aspx.

[5] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of PLDI '97*, June 1997.

[6] J. M. Anderson, L. M. Berc, J. Dean, et al. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390, November 1997.

[7] D. Andrade, B. B. Fraguela, and R. Doallo. Accurate prediction of the behavior of multithreaded applications in shared caches. *Parallel Comput.*, 39(1):36–57, January 2013.

[8] A. Auweter. Analyzing and optimizing the cache-behavior of a sparse matrix in pet image reconstruction. Master's thesis, Technische Universität München, 2010.

[9] D. Badouel, T. Priol, and L. Renambot. SVMview: A performance tuning tool for DSM-based parallel computers. In *Proceedings of Euro-Par'96*, volume 1123 of *LNCS*, pages 98–105. Springer, 1996.

[10] J. Banks, B. L. Nelson, and D. M. Nicol. *Discrete-event System Simulation.* Prentice Hall, 2010.

[11] M. Bekerman, S. Jourdan, R. Romen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser. Correlated load-address predictors. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 54–63, May 1999.

[12] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[13] B. T. Bennett and V. J. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, 19:353–357, 1975.

[14] E. Berg and E. Hagersten. SIP: Performance tuning through source code interdependence. In *Proceedings of the 8th International Euro-Par Conference (Euro-Par 2002)*, pages 177–186, Paderborn, Germany, August 2002.

[15] E. Berg and E. Hagersten. Statcache: A probabilistic approach to efficient and accurate data locality analysis. In *Proc. of the Int. Symposium on Performance Analysis of Systems and Software*, 2004.

[16] S. G. Berg. Cache prefetching. Technical Report UW-CSE 02-02-04, University of Washington, Februar 2002.

[17] A. R. Bernat and B. P. Miller. Incremental call-path profiling. *Concurrency and Computation: Practice and Experience*, 19:1533–1547, 2007.

[18] K. Beyls and E.H. D'Hollander. Platform-independent cache optimization by pinpointing low-locality reuse. In *Proceedings of International Conference on Computational Science*, volume 3, pages 463–470, June 2004.

[19] G. Bilardi, K. Ekanadham, and P. Pattnaik. Efficient stack distance computation for a class of priority replacement policies. *International Journal of Parallel Programming*, 41(3):430–468, 2013.

[20] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[21] S. Böhm and C. Engelmann. xSim: The extreme-scale simulator. In *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS) 2011*, pages 280–286, Istanbul, Turkey, July 2011. IEEE Computer Society, Los Alamitos, CA, USA.

[22] W. J. Bolosky and M. L. Scott. False sharing and its effect on shared memory performance. In *Sedms'93: USENIX Systems on USENIX Experiences with Distributed a nd Multiprocessor Systems*, pages 57–71, Berkeley, CA, USA, 1993. USENIX Association.

[23] M. D. Bond and K. S. McKinley. Probabilistic calling context. In *Proceedings of OOPSLA'07*, Montreal, Quebec, Canada, October 2007.

[24] T. Brandes. ADAPTOR profiling guide. Institute for Algorithms and Scientific Computing (SCAI), Fraunhofer Gesellschaft, 2004.

[25] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14:317–329, 2000.

[26] D. Burger, T. M. Austin, and S. W. Keckler. Recent extensions to the simplescalar tool suite. *SIGMETRICS Perform. Eval. Rev.*, 31(4):4–7, March 2004.

[27] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 52:1–52:12, New York, NY, USA, 2011. ACM.

[28] T. E. Carlson, W. Heirman, and L. Eeckhout. Sampled simulation of multi-threaded applications. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 2–12, April 2013.

[29] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 340–351, Washington, DC, USA, 2005. IEEE Computer Society.

[30] J. Chen, M. Annavaram, and M. Dubois. SlackSim: A platform for parallel simulations of cmps on cmps. *SIGARCH Comput. Archit. News*, 37(2):20–29, July 2009.

[31] Intel Corporation. Intel performance tuning utility.
http://software.intel.com/en-us/articles/intel-performance-tuning-utility.

[32] D. Cortesi, J. Fier, J. Wilson, and J. Boney. Origin2000 and Onyx2 performance tuning and optimization guide. *Silicon Graphics, Inc*, 1998.

[33] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[34] A. Carvalho de Melo. Performance counters on Linux. In *Presentation at the Linux Plumbers Conference*, Sep 2009.

[35] P. J. Denning. The locality principle. *Commun. ACM*, 48(7):19–24, July 2005.

[36] L. DeRose, K. Ekanadham, J. K. Hollingsworth, and S. Sbaraglia. SIGMA: A simulator infrastructure to guide memory analysis. In *Proceedings of SC 2002*, Baltimore, MD, November 2002.

[37] D. Eklov and E. Hagersten. StatStack: Efficient modeling of LRU caches. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 55–65, March 2010.

[38] C. Engelmann and F. Lauer. Facilitating co-design for extreme-scale systems through lightweight simulation. In *Proceedings of the 12th IEEE International Conference on Cluster Computing (Cluster) 2010: 1st Workshop on Application/Architecture Co-design for Extreme-scale Computing (AACEC)*, pages 1–8,

Hersonissos, Crete, Greece, September 2010. IEEE Computer Society, Los Alamitos, CA, USA.

[39] K. Bergman et al. ExaScale computing study: Technology challenges in achieving exascale systems. Peter Kogge, Editor and Study Lead. DARPA. Technical Report TR-2008-13, 2008.

[40] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Trans. Comput. Syst.*, 27(2):3:1–3:37, May 2009.

[41] V.W. Freeh. Dynamically controlling false sharing in distributed shared memory. In *Proceedings of the 5th IEEE International Symposium on High Perfor mance Distributed Computing*, pages 403–411. IEEE Computer Society, 1996.

[42] N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-overhead call path profiling of unmodified, optimized code. In *Proceedings of ICS'05*, Cambridge, MA, June 2005.

[43] R. M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, October 1990.

[44] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, April 2010.

[45] S. Graham, P. Kessler, and M. McKusick. GProf: A call graph execution profiler. In *SIGPLAN: Symposium on Compiler Construction*, pages 120–126, 1982.

[46] ITRS Technology Working Group. International Technology Roadmap for Seminconductors, 2012 Update, 2012.
http://www.itrs.net/Links/2012ITRS/2012Chapters/2012Overview.pdf.

[47] S. M. Günther and J. Weidendorfer. Assessing cache false sharing effects by dynamic binary instrumentation. In *WBIA '09: Proceedings of the Workshop on Binary Instrumentation and Applications*, pages 26–33, New York, 2009. ACM Press.

[48] F. Guo and Y. Solihin. An analytical model for cache replacement policy performance. *SIGMETRICS Perform. Eval. Rev.*, 34(1):228–239, June 2006.

[49] G. Hager, J. Treibig, J. Habich, and G. Wellein. Exploring performance and power properties of modern multi-core chips via simple machine models. *Concurrency Computat.: Pract. Exper.*, 2013.

[50] R. J. Hall. Call path profiling. In *ICSE'92: Proceedings of the 14th international conference on Software engineering*, pages 296–306, New York, NY, USA, 1992. ACM Press.

[51] R. Hassan, A. Harris, N. P. Topham, and A. Efthymiou. Synthetic trace-driven simulation of cache memory. In *AINA Workshops*, pages 764–771. IEEE Computer Society, 2007.

[52] W. Heirman, T. E. Carlson, S. Che, K. Skadron, and L. Eeckhout. Using cycle stacks to understand scaling bottlenecks in multi-threaded workloads. In *IISWC*, pages 38–49. IEEE Computer Society, 2011.

[53] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[54] H. C. Hsiao and C. T. King. MICA: A memory and interconnect simulation environment for cache-based architectures. In *Proceedings of the 33rd IEEE Annual Simulation Symposium (SS 2000)*, pages 317–325, April 2000.

[55] Intel. Intel VTune performance analyzer.
`https://software.intel.com/de-de/intel-sdp-home`.

[56] Intel Corporation. Intel-64 and IA-32 architectures software developer manuals (SDM).

[57] Intel Corporation. Intel Performance Tuning Utility 3.2, User Guide, chapter 7.4.6.5, 2008.

[58] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob. CMP$im: A Pin-based on-the-fly multi-core cache simulator. In *Proc. Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, pages 28–36, Beijing, China, June 2008.

[59] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). *SIGARCH Comput. Archit. News*, 38(3):60–71, June 2010.

[60] T.E. Jeremiassen and S.J. Eggers. Reducing false sharing on shared memory multiprocessors through compi le time data transformations. *ACM SIGPLAN Notices*, 30(8):179–188, 1995.

[61] L.K. John, D.J. Lilja, B. Calder, L. Eeckhout, J.J. Yi, and J.E. Smith. The future of simulation: a field of dreams. *Computer*, 39(11):22–29, 2006.

[62] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. *SIGARCH Comput. Archit. News*, 32(2):338–, March 2004.

[63] Y. H. Kim, M. D. Hill, and D. A. Wood. Implementing stack simulation for highly-associative memories. *SIGMETRICS Perform. Eval. Rev.*, 19(1):212–213, April 1991.

[64] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The Vampir performance analysis tool-set. In *Parallel Tools Workshop*, pages 139–155, 2008.

[65] D. Knuth. Structured programming with goto statements. *ACM Journal Computing Surveys*, 6(4):268, Dec 1974.

[66] M. Kowarschik, U. Rüde, N. Thürey, and C. Weiß. Performance optimization of 3D multigrid on hierarchical memory architectures. In *Proc. of the 6th Int. Conf. on Applied Parallel Computing (PARA 2002)*, volume 2367 of *Lecture Notes in Computer Science*, pages 307–316, Espoo, Finland, June 2002. Springer.

[67] M. Kowarschik and C. Weiß. An overview of cache optimization techniques and cache-aware numerical algorithms. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies — Advanced Lectures*, volume 2625 of *Lecture Notes in Computer Science*, pages 213–232. Springer, March 2003.

[68] T. Küstner, J. Weidendorfer, J. Schirmer, T. Klug, C. Trinitis, and S. Ziegler. Parallel MLEM on multicore architectures. In *Proceedings of 9th International Conference on Computational Science (ICCS 2009)*, number 5544 in LNCS, pages 491–500. Springer, 2009.

[69] T. Küstner, J. Weidendorfer, and T. Weinzierl. Argument controlled profiling. In *Proceedings of 2nd Workshop on Productivity and Performance (PROPER 2009)*. Springer, 2009.

[70] H. Lee, L. Jin, K. Lee, S. Demetriades, M. Moeng, and S. Cho. Two-phase trace-driven simulation (TPTS): A fast multicore processor architecture simulation approach. *Softw. Pract. Exper.*, 40(3):239–258, March 2010.

[71] J. Levon. OProfile, a system-wide profiler for Linux systems.
http://oprofile.sourceforge.net.

[72] Y. Liang and T. Mitra. Cache modeling in probabilistic execution time analysis. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, pages 319–324, New York, NY, USA, 2008. ACM.

[73] K. London, S. Moore, P. Mucci, K. Seymour, and R. Luczak. The PAPI cross-platform interface to hardware performance counters. In *Department of Defense Users Group Conference Proceedings*, pages 18–21, 2001.

[74] D. Lorenz, R. Dietrich, R. Tschüter, and F. Wolf. A comparison between OPARI2 and the OpenMP tools interface in the context of Score-P. In *Proc. of the 10th International Workshop on OpenMP (IWOMP), Salvador, Brazil, September 2014 (accepted)*, LNCS. Springer, September 2014.

[75] K. Lu, D. Müller-Gritschneder, and U. Schlichtmann. Fast cache simulation for host-compiled simulation of embedded software. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 637–642, San Jose, CA, USA, 2013. EDA Consortium.

[76] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[77] H. Luo, X. Xiang, and C. Ding. Characterizing active data sharing in threaded applications using shared footprint. In *Proceedings of the 11th International Workshop on Dynamic Analysis (WODA 2013)*, New York, NY, USA, 2013. ACM.

[78] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, February 2002.

[79] A. D. Malony, S. S. Shende, and A. Morris. Phase-based parallel performance profiling. In *Proceedings of ParCo'05*, volume 33 of *Parallel Computing: Current & Future Issues of High-End Computing*, pages 203–210, Jülich, Germany, 2006.

[80] M. Mamidipaka and N. Dutt. eCACTI: An enhanced power estimation model for on-chip caches. Technical report, In Technical Report TR-04-28, CECS, UCI, 2004.

[81] M. Martonosi, A. Gupta, and T. E. Anderson. MemSpy: Analyzing memory system bottlenecks in programs. In *Measurement and Modeling of Computer Systems*, pages 1–12, 1992.

[82] M. Meeks. LibreOffice under the hood: progress to 4.3.0, 2014. https://people.gnome.org/~michael/blog/2014-07-29-under-the-hood-4-3.html.

[83] J. Mellor-Crummey, R. Fowler, and D. Whalley. Tools for application-oriented performance tuning. In *Proceedings of 15th ACM International Conference on Supercomputing*, Italy, June 2001.

[84] R. Mitra, B. S. Joshi, A. Ravindran, A. Mukherjee, and R. Adams. Performance modeling of shared memory multiple issue multicore machines. *2012 41st International Conference on Parallel Processing Workshops*, pages 464–473, 2012.

[85] N. Nethercote and A. Mycroft. The cache behaviour of large lazy functional programs on stock hardware. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance (MSP 2002)*, Berlin, Germany, July 2002.

[86] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[87] Q. Niu, J. Dinan, Q. Lu, and P. Sadayappan. PARDA: A fast parallel reuse distance analysis algorithm. *Int. Parallel and Distributed Processing Symposium*, 2012.

[88] V. S. Pai, P. Ranganathan, S. V. Adve, and T. Harton. An evaluation of memory consistency models for shared-memory systems with ILP processors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, October 1996.

[89] M. Pericas, K. Taura, and S. Matsuoka. Scalable analysis of multicore data reuse and sharing. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 353–362, New York, NY, USA, 2014. ACM.

[90] K. S. Perumalla. Parallel and distributed simulation: Traditional techniques and recent advances. In *Proceedings of the 38th Conference on Winter Simulation*, WSC '06, pages 84–95. Winter Simulation Conference, 2006.

[91] M. Plichta. Faster sparse matrix operations by code generation embedding prfetching. Bachelor thesis, Technische Universität München, 2013.

[92] B. Putigny, B. Goglin, and D. Barthou. A benchmark-based performance model for memory-bound hpc applications. In *Proceedings of 2014 International Conference on High Performance Computing & Simulation (HPCS 2014)*, Bologna, Italy, July 2014.

[93] D. Quinlan and C. Liao. The ROSE source-to-source compiler infrastructure. In *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT 2011*, October 2011.

[94] P. Révész. The laws of large numbers. *Probability and Mathematical Statistics*, 4:176 pp., 1968. Academic Press, New York-London.

[95] D. L. Schuff, M. Kulkarni, and V. S. Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 53–64, New York, NY, USA, 2010. ACM.

[96] D. L. Schuff, B. S. Parsons, and V. S. Pai. Multicore-aware reuse distance analysis. In *IPDPS Workshops*, pages 1–8. IEEE, 2010.

[97] S. Shende and A. D. Malony. TAU: The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[98] L.A. Shepp and Y. Vardi. Maximum likelihood reconstruction for emission tomography. *IEEE Transactions on Medical Imaging*, MI-1(2):113–122, 1982.

[99] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. *SIGOPS Oper. Syst. Rev.*, 36(5):45–57, October 2002.

[100] P. Shivakumar and N. P. Jouppi. CACTI 3.0: An integrated cache timing, power, and area model. Technical report, In HP Labs Technical Report WRL-2001-2, 2001.

[101] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[102] Rogue Wave Software. *Rogue Wave ThreadSpotter Optimization Tutorial*, 2011. White Paper.

[103] J. M. Spivey. Fast, accurate call graph profiling. *Software – Practice Experience*, 34:249–264, 2004.

[104] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 196–205, New York, NY, USA, 1994. ACM.

[105] W. Stallings. *Computer Organization and Architecture: Designing for Performance*. Prentice Hall Press, Upper Saddle River, NJ, USA, 8th edition, 2009.

[106] V. Subotic, J. C. Sancho, J. Labarta, and M. Valero. A simulation framework to automatically analyze the communication-computation overlap in scientific applications. *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 275–283, 2010.

[107] N. R. Tallent, J. M. Mellor-Crummey, and M. W. Fagan. Binary analysis for measurement and attribution of program performance. In *Proceedings of PLDI'09*, Dublin, Ireland, June 2009.

[108] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. RapidMRC: Approximating l2 miss rate curves on commodity systems for online optimizations. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 121–132, New York, NY, USA, 2009. ACM.

[109] A. S. Tanenbaum. *Structured Computer Organization (5th Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

[110] J. Tao and W. Karl. CacheIn: A toolset for comprehensive cache inspection. In *Proceedings of ICCS 2005*, volume 3515 of *LNCS*, pages 174–181. Springer, 2005.

[111] J. Tao, M. Schulz, and W. Karl. A simulation tool for evaluating shared memory systems. In *Proceedings of the 36th ACM Annual Simulation Symposium*, pages 335–342, Orlando, Florida, April 2003.

[112] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.

[113] Y. C. Tay. *Analytical Performance Modeling for Computer Systems, Second Edition.* Synthesis Lectures on Computer Science. Morgan & Claypool Publishers, 2013.

[114] Cedric V. MADRAS: Multi-architecture binary rewriting tool. Technical report, University of Versailles Saint-Quentin en Yvelines, 2013.

[115] J. Weidendorfer. *Tools for high performance computing: Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing*, chapter Sequential Performance Analysis with Callgrind and KCachegrind. Springer, Stuttgart, July 2008.

[116] J. Weidendorfer. Data transfer requirement analysis with bandwidth curves. In *6th Workshop on Productivity and Performance. EuroPar 2013 Workshops*, Aachen, Germany, 2013.

[117] J. Weidendorfer, M. Kowarschik, and C. Trinitis. A tool suite for simulation based analysis of memory access behavior. In *ICCS 2004: 4th International Conference on Computational Science*, volume 3038 of *LNCS*, pages 440–447. Springer, 2004.

[118] J. Weidendorfer, T. Küstner, and S. A. McKee. Performance optimization by dynamic code transformation. In *Proceedings of Computing Frontiers 2011.* ACM Press, 2011. Poster Abstract.

[119] J. Weidendorfer, M. Ott, T. Klug, and C. Trinitis. Latencies of conflicting writes on contemporary multi-core architectures. In *Ninth International Conference on Parallel Computing Technologies (PaCT-2007)*, number 4671 in LNCS, pages 318–327, Pereslavl-Zalessky, Russia, 2007. Springer.

[120] J. Weidendorfer and C. Trinitis. Collecting and exploiting cache-reuse metrics. In *ICCS 2005: 5th International Conference on Computational Science*, volume 3515 of *LNCS*, pages 191–198. Springer, May 2005.

[121] J. Weidendorfer and C. Trinitis. Cache optimizations for iterative numerical codes aware of hardware prefetching. In *Applied Parallel Computing, State of the Art in Scientific Computing, 7th International Workshop, PARA 2004, Lyngby, Denmark, June 20-23, 2004, Revised Selected Papers*, volume 3732 of *LNCS*, pages 921–927. Springer, 2006.

[122] J. Weidendorfer and C. Trinitis. Off-loading application controlled data prefetching in numerical codes for multicore processors. *International Journal of Computational Science and Engineering (IJCSE)*, 4(1):22–28, 2008.

[123] J. Weinberg, M. O. McCracken, E. Strohmaier, and A. Snavely. Quantifying locality in the memory access patterns of HPC applications. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, Washington, DC, USA, 2005. IEEE Computer Society.

[124] T. Weinzierl. *A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids*. Dissertation, Institut für Informatik, Technische Universität München, München, 2009.

[125] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos. Making address-correlated prefetching practical. *IEEE Micro's Top Picks*, 30(1):50–59, 2010.

[126] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Comm. ACM*, 52(4), April 2009.

[127] S. J. E. Wilton and N. P. Jouppi. CACTI: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31:677–688, 1996.

[128] M.-J. Wu and D. Yeung. Efficient reuse distance analysis of multicore scaling for loop-based parallel programs. *ACM Trans. Comput. Syst.*, 31(1), February 2013.

[129] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *Proceedings of PLDI'06*, Ottawa, Ontario, Canada, June 2006. ACM.