SAPIENZA
UNIVERSITÀ DI ROMA

# Techniques and tools for program tracing and analysis with applications to parallel programming


Sapienza University of Rome

Ph.D. program in Computer Engineering

Candidate

Simone Economo
ID number 1270165

Thesis Advisors

Prof. Francesco Quaglia
Dr. Alessandro Pellegrini

Co-Advisor

Prof. Camil Demetrescu


October 2019

**Techniques and tools for program tracing and analysis with applications to parallel programming**

Ph.D. thesis. Sapienza – University of Rome



This thesis has been typeset by LaTeX and the Sapthesis class.

Version: October 31, 2019

Author's email: economo@diag.uniroma1.it

# Abstract

# Acknowledgments

# Contents

CHAPTER 1

# Introduction

In the last twenty years the conceptual hardware organization of computing systems has changed significantly due to the increasing pressure of two factors: the physical limitations of the scaling process of transistors on the one hand; the demand for high-performance, low-power, energy-efficient computing on the other. As a result, complex multi-core and heterogeneous architectures are ubiquitous nowadays. They represent a cost-effective way to employ the increased number of transistors in chips and to support the high degree of parallelism of many HPC applications. These include applications for bioinformatics, molecular dynamics, weather forecasting, quantum chemistry, computational fluid dynamics, cosmology and astrophysics, computer-aided design, computer graphics, machine learning and computer vision. These families of HPC applications exploit, to some extent, algorithms for numerical approximation, discrete or continuous simulation, 2D and 3D rendering, supervised and unsupervised learning, feature detection, object and pattern recognition. Overall, given the current and foreseeable architectural trends, we cannot avoid thinking of hardware parallelism and heterogeneity as concepts pervasively affecting the daily-life operation of software applications. As a result, we are today at a point where hardware has become complex enough to respond to different, advanced quantitative and qualitative desiderata. Unfortunately, exploiting and taming this complexity in software is challenging. Figure 1.1 depicts the reference conceptual organization of software on top of a hardware layer which consists of a combination of CPUs, special-purpose accelerators, different memory hierarchies and interconnections between all these components. At the other extreme, we can find applications which are written in different programming languages, paradigms and models. In between the two layers, a *Software Environment* (SE) encompasses all software which is required to (a) translate an application coded in a source language down to a low-level close-to-machine language, (b) support the execution of the application during its lifetime; (c) support the development of that application. As such, a SE includes many different tools and components: typically, an operating system kernel along with its drivers and modules; a set of standard and third-party libraries which sometimes may form a middleware or a runtime library; compilers, linkers and also instrumentation tools. It can also include other development tools which

assist programmers during the development (and debugging) of applications. In this scenario, the challenge witnessed at the level of environmental software is two-fold: on the one hand, applications demand software technology able to maximize the exploitation of hardware features; on the other hand, the complexity of hardware should have a minimal impact on the complexity of application themselves in terms on architecture and development. This challenge is extremely critical in the transition from peta-scale computing to exa-scale computing—a target that looks foreseeable for the end of the decade. Therefore, when witnessing such a technological shift, it is fundamental to devise techniques and tools to accommodate this growth in complexity at the level of environmental software.

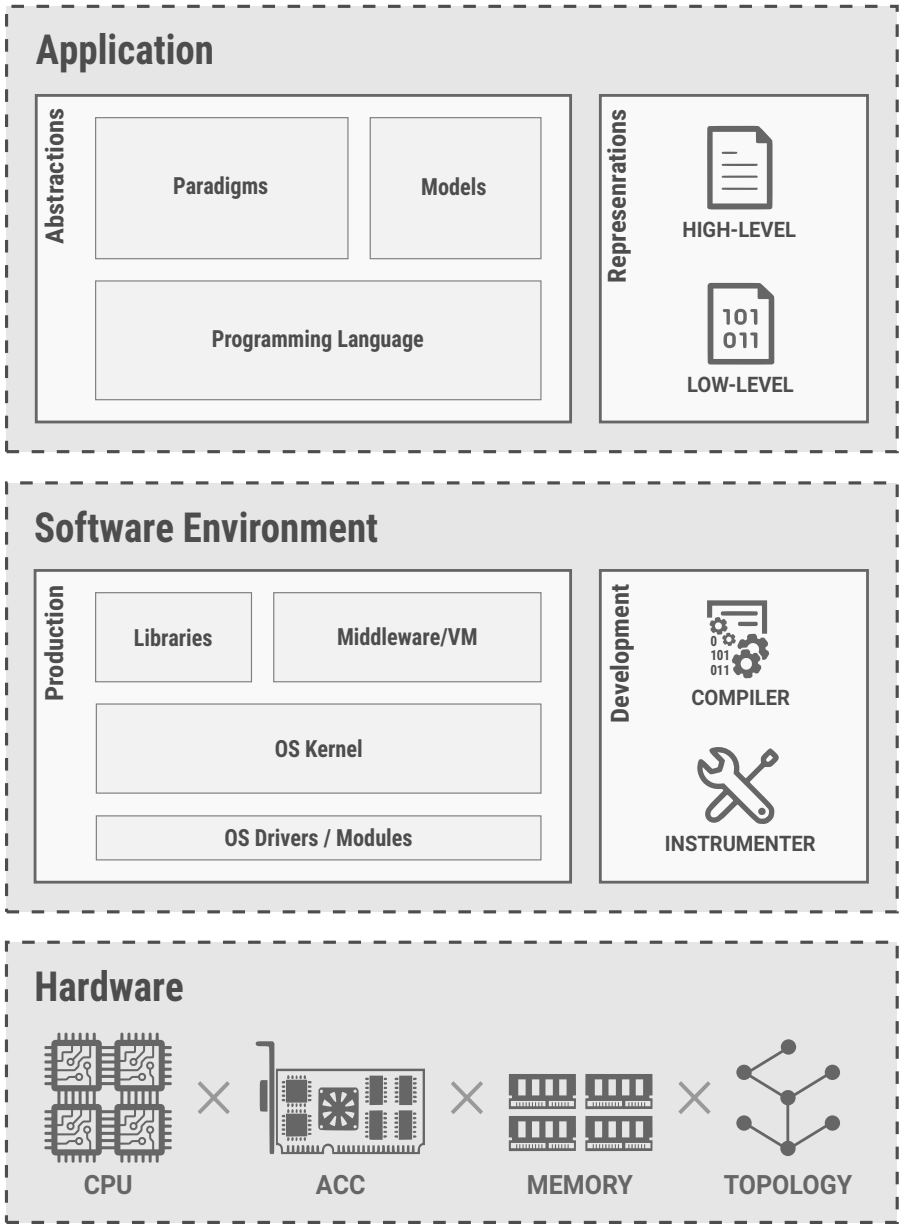



**Figure 1.1.** Reference organization of software on top of different combinations of hardware.

The methodology proposed in this thesis to tame this complexity is grounded on the idea that, to bridge the gap between hardware and software, program tracing and analysis are needed to understand and to infer the relevant traits and properties of applications. These features can then be exploited for the sake of optimizing the execution of the application (e.g., for performance, energy, or power purposes) or to change other aspects of it (e.g., for the sake of correctness or security). Overall, this process, depicted in Figure 1.2, can be framed into a framework that involves: a TRACING step, to collect useful static (i.e., prior to execution) and dynamic (i.e., at run-time) data about the application; an ANALYSIS step, to process the collected data and derive useful pieces of information; an EXPLOITATION step, which uses the collected information to improve the execution of the application in quantitative or qualitative ways, either off-line (i.e., prior to execution), or on-line (i.e., at run-time).

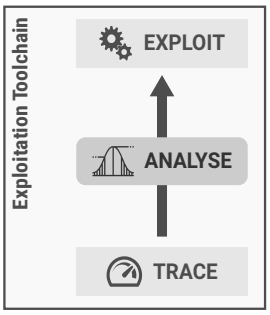



**Figure 1.2.** Functional representation of an Exploitation Toolchain.

An important aspect of this methodology is that the above-mentioned steps can be executed prior to execution or at run-time, thus solving different problems and raising different issues, as we will see in the following chapters. For example, in Profile-Guided Optimization (PGO) tracing is achieved at run-time, the analysis steps is performed after execution, and the exploitation step is carried out before executing the application. The first step is typically instrumentation-driven and requires to run an instrumented version of an application to collect some dynamic traits of its execution (e.g., edge profiles describing the likelihood of taking code branches); the second step processes and feeds back the information gained in the first step to the compiler; the compiler can then exploit this information to produce a better compiled version of that applications so as to improve its execution at runtime. Another example of compile-time exploitation is that of debugging an application looking for potential correctness and security bugs, and then feeding back the outcome of the analysis to the user or an automatic tool to fix the encountered issues. In this case, the tracing and analysis steps can be performed both off-line and on-line. When the exploitation is performed at run-time, this implies that the other steps are, too, executed while the application is running. This methodology is usually employed for optimization purposes, to maximize the performance of the application throughout execution, in presence of changing requirements or mutable properties of the applications. For example, the operating system can trace the access to virtual pages in order to understand which are the hottest pages, so as to relocate them to high-bandwidth memory banks. In this case, all the steps are performed at run-time, for run-time exploitation purposes. Another example is that

of tracing low-level hardware events in terms of control-flow predictions units (e.g., the return-address stack) to detect possible hijacking attack schemes and terminate the process for the sake of security.

A major focus of this thesis is on software environments for parallel applications. Such software environments are usually conservative in the way they manage the parallelism of the underlying hardware. For example, they allow to parallelize independent portions of code, but little effort is put into trying to avoid conflicts between these independent parallel regions (as for the case of parallel *transactions*), or to avoid parallelization errors between dependent parallel regions (as for the case of parallel *tasks*). It is worth stressing that these issues should not be a concern for application developers working at the end of the software stack. It is the software stack itself that should provide these guarantees. The reliance on environmental software is essential to software developers at any level of competence, and will also remain essential in the future as it is the only reasonable way to allow for an easy programming, maintenance and deploy of applications. For this reason, another major effort of this thesis is to present tracing and analysis technique that can be implemented into existing software environment in a transparent manner. When presenting our contributions in terms of tracing and analysis ideas, great effort is put into stressing the difference between the *techniques* we propose, and the *tools* that implement them. The fact that such techniques can also be embodied into existing software stacks serves as an evidence to this transparency objective.

The rest of this thesis is structured as follows. In Chapter 2 we provide an in-depth explanation of techniques and technologies to perform program tracing. Then, in Chapter 3 we proceed to analyze a transparent tracing technique for generic software that attach to the low-level representation of an application, for either on-line or off-line exploitation purposes. Chapter 4 discuss a high-level probabilistic model to analyse and estimate the abort probability of applications based on the Transactional Processing (TP) paradigm, for the sake of performance and on-line exploitation. Last, in Chapter 5, we present an algorithm to analyse the parallelization of applications written using task-based programming models, in order to evaluate their correctness. Chapter 6 concludes this thesis and hints at possible future research work. Overall, the work presented in this thesis is based on the following original contributions:

- (pending review) **S. Economo**, S. Royuela, E. Ayguadé, and V. Beltran, "A tool-chain to verify the parallelization of OmpSs-2 applications", *34th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*
- **S. Economo**, E. Silvestri, P. Di Sanzo, A. Pellegrini and F. Quaglia, "Model-Based Proactive Read-Validation in Transaction Processing Systems", in *Proceedings of the 24th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*
- **S. Economo**, D. Cingolani, A. Pellegrini, and F. Quaglia, "Configurable and efficient memory access tracing via selective expression-based x86 binary instrumentation", in *Proceedings of the 24th IEEE International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*

The research work that lead to this thesis has also produced the following additional publications:

- E. Silvestri, **S. Economo**, P. Di Sanzo, A. Pellegrini and F. Quaglia, "Preemptive Software Transactional Memory", in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*
- **S. Economo**, E. Silvestri, P. Di Sanzo, A. Pellegrini and F. Quaglia, "Prompt Application-Transparent Transaction Revalidation in Software Transactional Memory", in *Proceedings of the 16th IEEE International Symposium on Network Computing and Applications (NCA)*

CHAPTER 2

# Basics of program tracing

Program tracing is the process of recording information about the execution of a program. It is achieved by intercepting some events of interested during execution and storing information about these events. An event can be anything ranging from a low-level access to memory through a machine instruction, to the invocation of a high-level transaction banking service. The outcome of tracing is a log of data about a program's execution called *trace*. It can be a temporary trace living in memory, or a file living in secondary storage. Usually, a trace is represented as a sequence of records, each of which represents a particular instance of an event of interest and stores contextual information about it. For examples, in the case of memory access, we can store information about the involved address, while for transaction banking we can save information about a bank account's current balance. Traces are usually analysed for optimization or debugging purposes, and can be exploited either on-line or off-line. Classical applications of programs tracing include (a) the evaluation of alternative memory subsystem designs, in terms of performance and energy consumption [52, 89, 16]; (b) the detection of security vulnerabilities and performance inefficiencies in programs [83, 62]; (c) the characterization of program execution patterns over time (e.g., to optimize the placement of thread/data affinities in multi-socket systems [30, 44, 68]). Logically, the process of tracing programs can be split into two different steps.

- The Selection step is the first one. During this step, the code in the program which, at run-time, produces events of interests is decorated with some *tracing logic*. The tracing logic is responsible for saving information about the event and will only be executed at run-time. This step can be performed *ahead-of-execution* (AOE), i.e., during or after the compilation stage), or *just-in-time* (JIT), i.e., while the process is already executing application code. It can be manual or automatic, and can target events in application code or within a run-time library or other middle-level software.

- The Extraction step executes the tracing logic at run-time and extract information on the particular event instance being intercepted. Typically, the tracing logic at this step consists of storing these pieces of information taken

from a single event instance into a temporary storage area. As soon as a certain number of records is stored, the whole batch is flushed to the actual trace. The kind of information that is extracted at this time depends on the abstraction at which the previous step is performed. *High-level* information concern abstract source-level concepts, such as the type and scope of structured objects in memory or calls to high-level API services. *Low-level* information relates to a level of abstraction which is very close to that of the underlying hardware, or that of an intermediate code representation that is more concrete than the abstraction provided to the developer (e.g., JVM's bytecode). For example, it can be the physical address associated to an accessed virtual memory location, or the current state of CPU architectural registers.

Typically, tracing is performed in the context of a more general framework that involves more stes, as we already mentioned in Chapter 1. Indeed, our work is based on a tight cooperation between program tracing and analysis, and between the analysis part and a further exploitation step that tries to make good use of the results of the analysis part:

- The Analysis step processes the records in the trace to infer some properties of (the execution of) an application. Its complexity is arbitrary and typically depends on the purpose for which tracing was employed in the first place. This step can be performed *on-line*, i.e., while the application is being executed, or *off-line*, i.e., after the application has terminated. On-line analysis is timely, as it produces results that can be exploited to affect the execution of the application while it is running. However, it can also affect the execution of applications negatively, depending on the space-time complexity of the analysis itself. Off-line analysis is usually more heavyweight and not timely, and the results it produces can be exploited to support application development or the execution of application at a later time.

- The Exploitation step takes the results produced by the analysis and performs some corrective actions on the application being traced. Corrective actions can be of any kind, e.g., for optimization, debugging, or other purposes. Like the previous step, depending on when memory traces are exploited, these actions are executed on-line, thus affecting the current application instance, or off-line, hence affecting future application runs. In the first case, corrective actions are performed by a run-time component that actively monitors and assists the execution of applications. In the second case, corrective actions are usually bundled into a compiler, another external tool, or are performed manually by the user, prior to application execution.

Tracing is distinct to both program logging and program profiling. Even if there is not always a clear distinction between tracing and logging, and despite the fact that some of the same technologies are used for both, program logging tends to be a concern for the users of a program, while program tracings tends to be a concern for developers. While the main purpose of program logging is that of diagnostics and auditing, program tracing is used for debugging or other specialed purposes, such as program-wide or system-wide optimization. Program logging is typically lightweight

and used to produce a summary of the execution of the program, including errors and warnings. Logs are also more standardized and easier to read, often including sentences in a (localized) natural language. Because of this, looking at logs also doesn't give any insight about the code of a program. On the contrary, program tracing is more heavyweight and is used to produce a more precise description of the execution of the program. Traces are therefore larger and more noisy than logs, less standardized and not meant to be read by users (usually not including sentences). Tracing data may also include sensitive information about the code of a program.

Program profiling are program tracing are more similar. They both serve to aid program optimization and use the same technologies and techniques. Both profiling and tracing can be used to guide the design and optimization of programs during development, at compile-time, or at run-time. However, the purpose of profiling is to provide a (statistical) summary of the events of interested (a *profile*). For each instance of an event, an accumulator over an aggregation operation (e.g., a sum or an average) is maintained. The resulting profile therefore contains a number of entries that is proportional to the number of event classes of interested. Program tracing, on the contrary, is used to produce a stream of recorded events. The number of events in a trace is orders of magnitude higher, as each record contain information for each instance of an event. Overall, a profiler can be applied at any scale to derive quantitative metrics about the execution of a program with respect to a particular runtime condition or the current availability of system-wide resources. Typical profiles provide an estimation of the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Profiling and quantitative metrics can also be used to characterize a program at any scale. For example, profiling data can be used to provide insight into transaction workloads in distributed applications. However, there are situations in which a summary profile is not sufficient to understand what is happening during execution. For example, performance and correctness problems in parallel programs due to synchronization issues typically depend on the specific timing of specific instances of events, thus requiring a full trace to get an understanding of what is happening.

In the rest of this chapter we proceed to illustrate the different metrics, techniques and technologies which are used to support program tracing and analysis. In the remaining chapter we make extensive use of the ideas in this chapter to frame our original contributions in terms of techniques and tools.

## 2.1 Metrics

To evaluate different tracing and analysis techniques and tools, it is important to define metrics of reference. In this thesis we mainly consider the following measures:

- **Accuracy**. Intuitively, it is a measure of how trustable is the information being produced at the end of the analysis step. It can be defined as the proximity of tracing results to the true value of what is being traced. Sometimes a 100% accuracy is not necessary for certain kind of tracing purposes (see, e.g., optimization). In other cases (such as for the sake of security) lacking trustable information can be critical.

- **Exactness**. It is the degree to which analysis can provide results that can be traced back to individual events. The higher the exactness, the higher the amount of information that can be used to establish which event affected a particular analysis outcome. Depending on the specific tracing purpose, it might not be necessary to have exact results. For example, knowing that a single location is accessed one or more times may be unnecessary for some cases (e.g., debugging).

- **Overhead**. It is the amount of space/time that is spent at runtime for tracing events. Note that this performance measure is particularly important when it comes to dynamically optimizing the execution of applications. A 100%-accurate process that incurs a 50x execution time overhead penalty at runtime becomes useless for whatever use of tracing on-line or off-line, especially when it comes to tracing long-running applications. Similarly, if the memory overhead of tracing is too high, the system will run out of memory before being capable of tracing applications with an already high memory demand.

- **Perturbation**. It is a measure to describe a modification in the behavior of a program when it is subject to tracing. For example, programs that are susceptible to kernel-level scheduling decisions tend to be very sensible to any form of tracing which introduces even the slightest form of overhead. Devising a non-invasive form of tracing is important for many tracing applications not to modify their behaviors. Failing to do so would put the quality of the trace at risk and may lead to observing artificial and/or compromised program executions [60, 20, 92].

- **Portability**. How easy it is to port a given tracing technique to different applications, different execution environments (e.g., operating systems), different computer architectures. In some cases, such as industrial settings, it might be sufficient to devise a tracing technique which is bound to a particular platform. However, in many other scenarios (including academic ones) a portable technique is typically more interesting—and valuable, too. At the same time, less portability means having the opportunity to exploit more features of the underlying machine/platform, or to collect more information regarding each event.

Throughout this thesis, we provide different definitions for the measures previously described, depending on the particular application scenario or tracing purpose. An important aspect relates to how these metrics are—or can be—correlated via a trade-off curve, and how easy is to change the shape of this curve or move along it to explore different configurations. Generally speaking, increasing the accuracy or the exactness of tracing will almost certainly increase the overhead and the perturbation of the process. Decreasing the overhead and the perturbation of tracing typically also decreases the accuracy and the exactness of analysis. Therefore, an interesting challenge is to try to push the trade-off curve toward high benefits (in terms of accuracy and exactness) and low costs (in terms of overhead and perturbation) points. A few common techniques in the academic and industrial literature to pursue this goal are illustrated in the following section.

## 2.2 Techniques

While program tracing techniques and tools can be described in terms of the logical steps introduced at the beginning of this chapter, different techniques can be superimposed over this logical process, depending on the objective of tracing and on which metrics we wish to optimize. Each technique affects the trade-off between the different metrics in a unique way, thus allowing to customize the tracing process to the specific objective. We proceed to explain which are these techniques, and how they influence the above metrics.

### 2.2.1 Filtering

An important trade-off exists between tracing accuracy and tracing overhead. When an application is traced by intercepting all the events of interest, and all instances of these events, the anaysis step can produce information that is 100% accurate, thorough, and reliable. However, collecting such traces could pose a huge time and space overhead on the selection and extraction steps. Most of the times, these overheads are unbearable and defeat the purpose of tracing itself. At the same time, not all events are relevant for all tracing applications. For example, when tracing application-level code it is not useful to trace events occurring in third-party libraries or even the kernel. If the application is made of tasks (see Chapter 5) or atomic transactions (see Chapter 4) then it might be useful to restrain the tracing to the only events performed within these units of work. For these reasons, tracing is usually implemented through filtering techniques that collect a "relevant" subset of the memory accesses performed by the application. Filtering deals with reducing the number of events that flow through the tracing process. A filter can be seen as a condition $\phi$ that is applied to an event $e$ and the current *context* $\sigma$ to decide whether that event should be passed to the next tracing step or should be discarded. Depending on when the filter is evaluated, i.e., prior to or during execution, the entities $e$ and $\sigma$ can represent different things. For example, when tracing memory accesses, $e$ represents an access of interest in the program, and $\sigma$ can be the current program execution state in terms of register values and values of all active memory locations in its virtual address space.

**if** $\phi(e, \sigma)$ **then**
    DISCARD($e$)

Filtering by itself can occur at any stage of the tracing process. When performed at the SELECTION stage, it allows to spare EXTRACTION time and ANALYSIS time on irrelevant accesses. The later filtering is performed in the tracing process, the more time we spend dealing with irrelevant information in previous steps. Notice, however, than it is not always possible to perform filtering at SELECTION or EXTRACTION time. For example, if we need information about the run-time execution state of an application (such as the value of a particular location in memory) to know whether a particular event instance is relevant or not, we can't perform filtering during an off-line SELECTION step. If we the information that we need is still not known in that particular moment during execution, then we may need to postpone filtering until the ANALYSIS step.

### 2.2.2 Aggregation

Another way to reduce the overhead of tracing at the expense of accuracy is by means of aggregation techniques. Many times, at the end of the ANALYSIS step the information that is generated doesn't need any reference to a particular instance of an event that was intercepted during execution. Stated differently, the ANALYSIS step typically produces information whose granularity is higher than that of the single event instance. For example, if we want to know if there have been memory accesses to a given region of memory (e.g., an OS page), or how many, we don't need to know where these accesses have occurred. Therefore, as soon as we encounter two or more accesses falling within that region, we can discard them and only pass through limited/aggregated information about them (e.g., their number). As a different example, if two accesses are performed by the same instruction at different times to contiguous locations in memory, and the time at which they occurred is not relevant, we can represent them as a single access having a larger span on memory. When doing this, we spare tracing overhead because we are reducing the set of accesses on which the tracing process will deal with, at the cost of losing some information about each single accesses. Aggregation therefore deals with grouping events which have something in common in order to focus the subsequent steps to a reduced set of entries which convey a smaller but still meaningful amount of information. It is composed of a condition $\alpha$ on the current event $e$, a set of past aggregated events $\mathcal{E}$, and a context $\sigma$. When the condition $\alpha$ is true, an AGGREGATE operation merges $e$ into $\mathcal{E}$.

**if** $\alpha(e, \mathcal{E}, \sigma)$ **then**
    AGGREGATE$(e, \mathcal{E})$

Aggregation, in a way similar to filtering, allows to reduce the time spent on processing events at subsequent stages in the tracing process. This is because the number of records that are passed to the next stage after using aggregation is reduced. However, just like filtering, it is not always possible to aggregate information coming from different events at early stages. For example, we can aggregate accesses to contiguous addresses during the EXTRACTION step because the address information is the already available. However, if the information must be derived through a complex sequence of operations, aggregation must wait till the next ANALYSIS step.

### 2.2.3 Buffering

Communication between the different tracing stages can occur according to two different temporal schemes, namely synchronously or asynchronously. The first case, the naïve one, expects each event record to be passed through to the next step as soon as the current step has it available and processed. It is suitable whenever we require a timely stream of events that must be consumed on the fly. However, albeit straightforward, this scheme tends to pose excessive overhead while tracing the application. For example, the application might be competing for a lock or performing some time-critical operations that should not be delayed. For this reason, a generally better form of tracing involves buffering. It entails storing the collected events into a temporal storage that is later flushed to the actual trace or another

temporary storage at an appropriate moment in the execution of the application—preferably if outside of its critical path, e.g., through a background process. The natural justification to this asynchronous scheme is that sometimes events need not be consumed on the fly. Eploitation schemes which are expected to run off-line don't require minimum latency between the materialization of an event in the trace and its exploitation. As for on-line exploitations schemes, it is typically better to wait for a few events of a kind before taking corrective actions, e.g. by means of aggregation schemes. While this increases the "time-to-analysis", which is the delay experienced by a single event before being consumed by the analysis step, this also allows to reduce the perturbation on the application.

## 2.3 Technologies

Generally speaking, tracing can be performed using either hardware or software facilities for selecting the events relevant for the analysis, although it also depends on the abstraction at which tracing is performed (low-level vs high-level events). Moving from hardware to software-based technologies, we observe that getting accurate traces becomes more and more prohibitive in terms of runtime tracing overhead and perturbation. This is because selecting a single event is more expensive (and, in a sense, more explicit) as we move from low-level solutions to high-level ones. At the same time, it becomes easier and easier to collect high-level information about events of interests, because the techonologies we use operate at the same level of abstraction. The portability of these technologies, too, increases when abstracting further and further from the underlying machine. In this section we provide an overview of the technologies at our disposal to implement program tracing, discussing their advantages and drawbacks. In the following chapters we make use of the technologies explained in this section to provide working implementations of the tracing technique we propose as our original contributions. It is worth stating that the implementations we provide, depending on the purpose, are not necessarily optimized toward low-overhead and low-perturbation tracing. Sometimes, achieving high accuracy and/or high exactness is more important, as well as developing more portable tools. For this reason, understanding the technologies available nowadays is fundamental to know how to move from theoretical techniques to concrete tracing tools.

### 2.3.1 Hardware-based

Hardware-based technologies work at the level of electric signals or (micro)instructions as seen by a processor. In the first case, solutions are based on monitoring signals coming from on-chip memory controllers [7, 17, 20]. At this level, events are intercepted simply by snooping electric signals being transmitted over communication lanes (e.g., memory buses). The major advantage of these tracing technologies is that it is possible to collect very accurate traces at literally no runtime cost. However, these technologies have some drawbacks in terms of portability and capability of performing filtering and/or aggregation operations:

- They are usually expensive and very difficult to port to other architectures,

since they entail physically attaching additional circuitry to the outgoing bus interfaces of on-chip controllers.

- They can only record very limited information on the events being recorded. This is because the kind of events being intercepted exist at the lowest possible level and any high-level information in generated on-chip and therefore is not available. For example, when recording memory access, usually just the physical memory address involved in the operation. Any higher-level information, such as the virtual address associated to a physical address, cannot be recovered at this level.

- They can only see a subset of the events, just depending on how much of the events take place on-chip. For example, when tracing memory accesses, the kind of events which can be intercepted depends on how much of the memory hierarchy is installed on-chip. In processors with on-chip first-level caches, it is impossible to snoop the accesses that hit the primary cache and so only L1 cache misses can be observed. In processors where all caches are on-chip, we can only see accesses served by the main memory.

On modern processors, hardware-based tracing makes use of Performance Monitoring Units (PMUs) (also called *hardware instrumentation*), which in turn rely on sampling [64, 59] and operates at a higher level than electric signals. When enabled, hardware sampling picks one event of interested every $N$ executed machine instructions (or $T$ clock ticks) and reports to software additional information related to the event, for further analysis. The accuracy and overhead of this technique both depend strongly on the frequency of sampling, which is tunable by the user on most architectures. A too low frequency may lead to poor tracing accuracy, while a too frequent sampling may give rise to significant execution slowdown. The major advantage of hardware instrumentation is its reduced intrusiveness with respect to the traced application, since all tracing is performed by dedicated firmware and hardware registers in the processor. Moreover, it induces no perturbation on the traced program. PMUs have also some limitations, which generally involve limited capability for filtering and/or aggregating events, and scarce portability:

- Despite the freedom to control the frequency of sampling, PMU-based tracing is essentially arbitrary, meaning that the sampled event is chosen without any knowledge of the traced program. For example, the sampled event might not be representative of the behaviour of the program.

- The trade-off between the sampling frequency and the tracing accuracy (or the tracing overhead) is unknown in general. As a result, it is difficult to understand which is the right sampling period for a given program, even only for a specific input.

- Different families of architectures, or different products within the same family, may exhibit incompatible PMU interfaces with different sampling capabilities. Because of this, PMU-based sampling is usually non-portable across architectures.

### 2.3.2 Software-based

Software-based tracing technologies can be classified according to the abstraction level. When performed in the kernel, tracing is performed on processes rather than programs, meaning that it exploits operating system handlers as injection points for the required tracing logic. For instance, on many conventional operating systems it is typical to intercept page faults (hence, page-fault handlers) to acknowledge that a process $p$ has issued an access to a memory page $m$. As a result, most of the literature proposals work at the granularity of virtual pages [30, 44, 90], typically ranging from 4KB to 2MB in size. Kernel-level tracing is completely transparent to the process since it requires no modification to the original program. It is also more portable than hardware-based tracing because the operating system abstracts away many details of the underlying architecture and the majority of low-level handlers are exported to all supported platforms. Despite that, it has some shortcomings, mostly in terms of event granularity and filtering:

- It incurs more overhead than hardware-based techniques, since the additional tracing logic is implemented using machine instructions rather than special-purpose hardware and/or firmware.
- It is impossible to trace events at a level lower than that seen by the operating systems. For example, it is impossible to trace memory accesses at a granularity lower than that of virtual pages, due to the way modern conventional operating system treat physical memory.

Some solutions rely on application-external libraries, like MPI/OpenMP [19, 50, 88] or transactional middleware. These approaches have the advantage of being transparent to the applications that use these libraries and also being able to understand more about their semantics, thus being able to produce event records containing high-level information. Not only that, but they tend to introduce small overhead since the cost of invoking library services is spent anyway. However, there are also some disadvantages in terms of portability:

- These approaches are usually limited to very specific applications (or application domains) that make use of libraries (e.g., MPI for coarse-grained parallelism or OpenMP for fine-grained one).
- Only events related to explicit library calls can be traced. Therefore, all those events that occur outside of library invocations cannot be traced.

When it comes to application-level tracing, a well-known method to perform tracing transparently to a program is *software instrumentation*. It is a technique which resorts to transparently patching the code of a program in order to implement the desired tracing functionality. When patching occurs prior to run-time, instrumentation is called *static* or *ahead-of-time* (AOT) [53, 66]. When patching is otherwise carried out while the application is executing, it is called *dynamic* or *just-in-time* (JIT) [35, 69]. Depending on the abstraction at which it is performed, we distinguish between different kinds of software instrumentation:

- *binary-code instrumentation* is achieved when patching object files (such as executables files) and is quite common for classical compiled languages such as C, C++ or Fortran;
- *byte-code instrumentation* is similar to binary-code instrumentation, but works on specific byte-code file formats that are typical of JIT-compiled or emulated languages such as Java or Python;
- *source-code instrumentation* applies directly to source code.

Regardless of the specific type of instrumentation being used, this approach has some interesting properties. Being performed entirely inside application code, it has no dependence on external libraries, the operating system or even the underlying computer architecture, therefore it is a more portable technique. Additionally, it allows to explore interesting trade-offs between tracing accuracy and tracing overhead, since it can exploits relevant static and/or dynamic information on the program. Nonetheless, instrumentation tends to be very invasive in terms of overhead and perturbation:

- Instrumentation techniques tend to be very heavyweight and are likely to perturb the behavior of applications in a significant way [63]. This is because the tracing logic is explicitly injected into the program by means of additional instructions or high-level language statements.
- Static instrumentation, being completely carried out at compile or link-time, cannot exploit relevant information characterizing the actual execution flow of programs, which is something that dynamic instrumentation is able to do.
- Moving from binary-code to source-code instrumentation, we gain high-level information on the program but lose some other properties. For instance, source-code instrumentation cannot be performed at runtime and requires the source code of the program, which is not always available. Binary-code instrumentation is typically able to operate on final executables, but has to deal with a limited amount of information in terms of an application' semantics.

CHAPTER 3

# Tracing generic applications

When targeting general (multi-threaded) applications, the absence of any reference programming model or paradigm makes it difficult to predicate on the high-level semantics of the application, thus forcing tracing tools to rely on less abstract representations of the program (see Section 2.3). This chapter focuses on memory access tracing, a particular form of tracing that is responsible for the interception of the memory references issued by an application at runtime. The outcome of this process are one or more sequences of memory accesses called *memory traces*, just depending on whether the user is interested in tracing per-process or per-thread memory accesses. Independently of whether one relies on hardware or software-based technologies to operate at this abstraction level, tracing all the memory accesses may still lead to excessive runtime overhead. This is especially true for approaches based on software instrumentation, which require to insert additional code into the application to select and extract information about all memory accesses which can be performed. An additional drawback of these techniques is that they cannot exploit relevant information about the actual execution flow of programs and are typically forced to instrument all accesses in the program. On the other hand, arbitrary filtering the accesses to be traced, with the purpose of reducing runtime costs, may lead to produce traces that aren't sufficiently accurate. For example, one might use dynamic sampling (e.g., via PMUs or dynamic binary instrumentation) to only instrument a subset of the accesses of the application. However, sampling is essentially arbitrary, meaning that the sampled instructions might not be representative of the memory access pattern of the program. In these scenarios, tracing can give rise to low-quality samples, while still paying the cost for taking them. Therefore, given the absence of any hint regarding what is relevant to tracing and what is not, and at which granularity, the tracing techniques described in Section 2.2 have limited effectiveness.

In this chapter we look at a flexible tracing technique which can be applied to generic (multi-threaded) applications to implement memory tracing tools for off-line or on-line purposes. Our approach relies on binary software instrumentation and only performs static analysis of code. Even if we are forced to face the classic limitations of static and software-based approaches—namely, the absence of any

information to predict the flow of execution and the need for inserting additional instructions in the program—we are able to exploit a few static properties of code to explore the trade-offs between tracing exactness, tracing overhead, and tracing accuracy. Our selection algorithm is based on a hybrid filtering/aggregation technique that is able to selectively instrument a subset of the memory-accessing instructions that are identified as the "most representative" of the application for a desired tracing granularity. The filtering part gathers instructions in groups and only instruments one instruction per group. The aggregation part assigns to the instrumented instruction an access count that is equal to the size of the whole group. The number of instrumented instructions and the granularity parameter are user-customizable. We also present an abstract addressing mode which fuels (and makes it possible) the rest of the analysis. Its purpose is to disambiguate the manifold nature of memory address expressions in CISC (Complex Instruction Set Computing) and RISC (Reduced Instruction Set Computing) architectures. By relying on this model, we can reason on memory-referencing instructions unambiguously and regardless of the specific Instruction Set Architecture (ISA). Additionally, we can abstract away from the conventions used by compilers and low-level programmers when enconding memory accesses into these expressions.

Overall, the work presented in this chapter involves the Selection and Extraction parts of the tracing process illustrated in Chapter 2. The original contributions of this chapter can be summed up as follows:

- We propose a static selection algorithm and a hybrid filtering/aggregation technique to select a subset of representative memory-referencing instructions, while being able to count how many times an access to a memory address has been issued.
- The trade-off between tracing accuracy and overhead can be controlled through a parameter called *instrumentation factor* which is customizable and whose value depends on user-specific requirements.
- Our instrumentation scheme can be configured in such a way to fit the desired *tracing granularity*, which is too customizable by the user depending on the tracing requirements.
- Our analysis builds upon an abstract addressing mode that disambiguates the weak semantics of concrete ISA addressing modes, thus being portable to different architectures and programming conventions.

## 3.1 Baseline formalisms

Our tracing method is based on selecting for tracing those memory-referencing instructions, i.e., instructions targeting memory as source or destination, which are believed to be more representative of the overall execution. In Section 3.2 we provide a precise definition of what it means for an expression to be representative. Depending on the purpose, the user can instruct our algorithm to select a varying number of memory-accessing instructions to instrument. This is similar to how other tracing techniques based on sampling work. However, differently from them, our

selection is driven by static code analysis only. Additionally, it is less arbitrary than typical sampling because it selects instructions based on deterministic rules and properties of instructions in code. In this section, we illustrate some preliminary analysis we perform on code before running the tracing algorithm. Then, we present an abstract addressing mode to unambiguously describe memory expressions encoded into memory-referencing instructions.

### 3.1.1 Preliminary static code analysis

As hinted, our tracing technique is based on binary software instrumentation, a technique that operates on (and transforms) the binary code of a program before the code itself is run. Our approach to select the memory-referencing instructions to be instrumented relies on rules that compare instructions prior to executing them. However, the problem of choosing representative accesses using only static analysis is complex because memory accesses are encoded within a binary file as *memory-address expressions*, i.e., linear combinations of register variables and constants which are evaluated to actual addresses only at run-time. The complexity of dealing with expressions lies in that the value a register will hold at runtime may be unpredictable. As an example, it may depend on an unforeseeable quantity such as program inputs, or it may materialize as any of multiple values resulting from taking different execution paths. Overall, static analysis forces us to see the "static" memory access trace of a program, i.e. the one observable prior to execution, as *a graph of ambiguous memory-address expressions*. In contrast, the "dynamic" memory access trace, which is the one materialized by a thread at runtime, is *a sequence of finalized (virtual) memory addresses*. Because of the presence of registers and the impossibility to know the actual control flow of an application prior to execution, memory address expressions are subject to the following issues:

1. *Address multiplexing*. A single expression can encode different addresses over time. This can happen because the values held by registers can change in between two executions of the same memory-referencing instruction.

2. *Address aliasing*. Different expressions can encode the same address at the same time. In general, it is impossible to know whether two different registers will contain the same value prior to runtime.

To effectively deal with address expressions prior to executing them for tracing purposes, our analysis relies on a well-known program representation called *Control-Flow Graph* (CFG) [4]. It is a directed graph of all the possible ways in which a thread running a program can flow from one instruction to another at runtime. A node in the CFG is called a *basic block* and represents a contiguous sequence of instructions with one single entry point at the first instruction and one single exit point at the last. An edge between two blocks in the CFG establishes that the program's control flow is allowed to move (along a thread) from the last instruction of the source, straight to the first instruction of the destination. A basic block may reach multiple destinations and be linked to multiple sources, though it does not admit sideways entrances or departures. Therefore, if control flows into a basic block,

it must go through it from the first instruction all the way to the last[1].

Basic blocks represent atomic execution units for the program and therefore are easy to manipulate and to reason about. For this reason, in our approach we target the selection of what memory-referencing instructions to instrument on a per-basic-block basis. If we constrain the rest of the analysis to the scope of a single basic block, then we can eliminate the problem of address multiplexing. Indeed, when we consider only what happens within a basic block, each memory address expression can only represent a single memory address at a time. The fact that multiple executions of the same basic block can produce different addresses for the same expressions becomes irrelevant, because the scope of our analysis involves only what happens in each single execution of that basic block independently. At the same time, by following the linear flow of instructions in a basic block, we can detect whether there are instructions which change the value of a register, as a way to mitigate the effects of the address aliasing problem. To do this, we rely on a simple form of register analysis inspired by the Static Single Assignment form (SSA). When starting the inspection of a basic block, all the architectural registers are deemed stable. We begin scanning the instructions forming each basic block in program order, from first to last. As soon as an instruction that overwrites the value of one or more registers is found, these registers are assigned a new *incarnation number*, i.e., an increased version number which indicates that we have logically encountered new register values. Two register parameters appearing in two different memory address expressions are then considered as *equal* if and only if they use the same register identifier and the same register incarnation number. Compared to other forms of register analysis based on full data-flow analysis (see, e.g., [51]), which operates on the entire CFG, ours only requires a single pass over each basic block. Additionally, our analysis is stateless, meaning that it doesn't remember the values of past incarnations of a register. Clearly, this approach may lead to false negatives. For example, it cannot recognize when a sequence of value-changing operations yields an unchanged register value as the final result. However, in exchange for this, our register analysis is much faster and doesn't depend on the full CFG of a function. Therefore, it can be used both at run-time and in all those cases where functions are disassembled in blocks, as in the case of memory tracing achieved via dynamic instrumentation approaches (see Section 2.2).

By the end of our preliminary static code analysis, we are presented with a graph of basic blocks. Each basic block might contain one or more memory-accessing instructions which encode the address to be accessed as a memory-address expression made up of registers and constants. Each register is uniquely idenfitied by its name and its incarnation number. We can therefore focus on memory-accessing instructions and parse their memory address expressions to understand whether they are relevant or not for tracing. The next step is that of defining a formal addressing mode to represent expressions in a more abstract and less ambiguous way.

[1]Sideways entrances and/or departures may happen due to hardware interrupts (e.g., for re-scheduling), as well as software interrupts (e.g., signal handling). However, they are well beyond the scope of user space program analysis, therefore they do not need to be captured by this representation.

### 3.1.2 The BID addressing mode

The rules that explain how to combine register values and constant values within a memory address expression depend on the specific hardware architecture and processor. Each *addressing mode* constraints the possible combinations in different ways. For example, in x86 processors the most expressive addressing mode is called *SIB addressing* (Scale-Index-Base) and involves expressions that combine four quantities. The *base register* $b$ specifies the base address from which the address computation starts. The *index register* $i$ is used as an index value whenever $b$ points to an array-like object or a regular structure. The *scale* is used in conjunction with the index register to specify the step size and can assume one of the values in $\{1, 2, 4\}$ (as well as 8 in 64-bit mode). To obtain the final memory address, a *displacement* $d$ is added to the result, as in the following expression:

$$EA_{x86} = b + i \times s + d$$

A similar form of addressing is the *RIP-relative addressing* (for x86-64 only). It is constituted by an immediate value which is used to displace from the current position in the program. It's like SIB addressing, but there's no index (hence, no scale) and the base register is bound to be the program counter. Lastly, in *absolute addressing* the final address is already available in the expression. It is a form of SIB addressing where there are no index and base registers.

Different processor families and ISAs (Instruction Set Architecture) have similar addressing modes involving a base register which can be, usually exclusively, combined with a constant (hence, a displacement), a register (hence, an index register), or a scaled register (hence, an index register and a scale). Some simple EPIC (Explicitly Parallel Instruction Computing) and RISC processors can go even further and only admit a base register, with no possibility to specify an immediate, nor a register offset.

$$EA_{others} = \begin{cases} b + i \times s & \text{e.g., ARMv8} \\ b + i & \text{e.g., ARMv8, Sparc, PowerPC} \\ b + d & \text{e.g., ARMv8, RISC-V, MIPS, Sparc, PowerPC} \\ b & \text{e.g., Itanium} \end{cases}$$

To make our analysis portable to different processor and architectures, we discard concrete addressing modes in favor of a more abstract model called *BID addressing*, where only base, index and displacement are considered. A *BID expression* is denoted as a tuple $\{b, i, d\}$ such that:

- The *base* parameter $b$ refers to the initial address of a memory object, such as an array or an arbitrary structure.
- The *index* parameter $i$ is used to iterate over a contiguous sequence of entries in an array-like object.

- The *displacement* $d$ is used to move within an arbitrary structure to access its fields.

To compute the effective address out of a BID expression, it is sufficient to aggregate the above parameters as in the expression:

$$EA_{BID} = b + i + d$$

These parameters resemble the ones used in many different ISAs and are general enough to represent all the different addressing modes which are available in modern processors. Observe, however, that depending on the way address expressions are exploited by compilers and low-level programmers, the semantics of each parameter which forms the expression may change from instruction to instruction. For example, one instruction may encode the most-significant bits of an address in a constant, while another instruction might pass them into a register and use a constant for the least-significant bits. As another example, on Linux-x86-GCC, statically-allocated objects can be accessed by putting the base address of a data object into the SIB displacement, since that address is already known at compile-time. On the contrary, the base address of a dynamically-allocated object is not known until the same object effectively materializes in memory. Therefore, its base address is concealed behind a register. Assembly programmers can further distance themselves from this scheme and encode memory address in arbitrary ways—e.g., by deliberately using a single base register as an accumulator for the final address, for instance via the x86 `LEA` instruction.

To mitigate the effects of this ambiguous parameter semantics, we allow each parameter $p_e$ in the tuple $\{b_e, i_e, d_e\}$ to map to either a register or an immediate value. To discriminate the two cases, we introduce the notion of *BID expression template* $\tau_e$, which is a tuple where each symbol indicates whether the corresponding parameter in the expression $e$ corresponds to a register $R$ or to an immediate $I$. The addition of templates makes it possible to distinguish between different compiler and programmers conventions in terms of usage of parameters in a memory address expressions, but it must be supported by additional information in order to be effective. Typically, the ability to discern between templates strongly depends on compiler choices and the environment under consideration. For Linux-x86-GCC, for example, when the object file is an ELF executable then we can compare the SIB displacement parameter with the range of addresses allocated to static data regions. If the object file is an ELF relocatable, we can check for the existence of relocation information associated with the x86 instruction. In Section 3.3 we propose a way to instantiate the BID addressing mode and its templates feature for the case of x86-64 processors and relocatable-object files.

We note that a BID expression $e$ can never lack its base $b_e$, since it gives always the most significant bits of an effective memory address. On the other hand, missing quantities (index and/or displacement), if any, are modelled in our analysis through a *nil* value, just to represent the fact that they do not contribute to the actual value of an expression $e$. Therefore, a BID expression can represent not only x86 addressing modes (included RIP-relative and absolute ones), but also less complex schemes like those used by other ISAs such as RISC and EPIC ones. Not only that,

but we stress that a more complex static analysis scheme could allow to recover full BID expressions out of simpler RISC/EPIC expressions. For example, it could transform an expression involving a base register $r$ only into a full BID expressions with a base, an index, and a displacement, by tracing back the operations which performed side-effect on $r$ in the same basic block, or across multiple basic blocks.

In the rest of the chapter we will make use of BID expressions and templates to devise a selection algorithm that can be ported to different architectures, while also making it possible to predicate on address expressions independently of the programmer/compiler idiom used to encode memory accesses via the available addressing modes. Our selection algorithm works by defining two relations over pairs of different BID expressions: the equivalence relation and the similarity relation.

### 3.1.3 Equality between BID expressions

Our first objective is that of finding BID expressions which are equal, in order to establish a trade-off between tracing overhead and tracing exactness. When we find a group of expressions which are equal, we have the option of selecting for tracing a single element to represent the whole group. Therefore, a representative access is an access encoded in an expression which has duplicates in the basic block. An access is more representative than another in the same basic block if it can represent a bigger group of duplicate expressions. We can therefore choose the most representative expressions for each basic block and only instrument these representative ones. Upon instrumenting, we can assign to the chosen expression an access count that is equivalent to that of the group. In this way, by only instrumenting a single expression for each group, we reduce the overhead of tracing at expenses of the exactness, since we lose information on the uninstrumented duplicates.

Since BID expressions are composed of constants and variables, respectively represented by immediate and register parameters, we need to compare BID expressions on a parameter-by-parameter basis. Intuitively, *BID equality* holds for two expressions if and only if they share the same template and their parameters $b$, $i$, and $d$ have the same values. If templates mismatch, then the expressions are deemed incomparable. If the templates are equal, parameter-wise equality is defined as expected for immediate values, while it relies on names and incarnation numbers for registers (according to our preliminary static analysis). Formally speaking, given a basic block $b$ that contains a set $\mathcal{E}_b$ of memory address expressions (associated with memory-referencing instructions), we introduce the *BID equivalence relation* $= \subseteq (\mathcal{E}_b \times \mathcal{E}_b)$ between any two BID expressions $(e, e')$ belonging to $\mathcal{E}_b$ as follows:

$$e = e' \iff (\tau_e = \tau_{e'}) \wedge (b_e = b_{e'} \wedge i_e = i_{e'} \wedge d_e = d_{e'}) \tag{3.1}$$

By Equation (3.1), all the BID expressions of the basic block $b$ can be clustered into different classes, depending on whether they are equivalent or not. Let $\mathcal{C}_b$ be the set of *candidate BID expressions* for a basic block $b$ , formed by choosing a single representative from each equivalence class over the set of all expressions $\mathcal{E}_b$. Tracing a single candidate for each equivalence class means instrumenting the instruction which uses that BID expression. Each instrumented access via an expression $e$ is associated with an access count which represents the cardinality of the corresponding

equivalence class $[e]$. The access count is precise and accurate because, by definition, all instructions in a basic block $b$ are executed exactly once at run-time.

As hinted before, the instrumentation of a single element for each cluster implements a form of static aggregation at the basic-block level which generates a trade-off between tracing overhead and tracing exactness. Since all memory accesses issued within a given group can be traced by only instrumenting the access issued by a single representative element of the cluster, tracing all these accesses has the same cost as tracing one. As for the exactness, we note that by only instrumenting a single representative access, we are losing information on the instructions that performed the non-instrumented accesses in the same cluster. Depending on the kind of analysis that we wish to perform, this kind of aggregation can be considered acceptable or not. In Section 3.2.3 we provide a few accuracy measures that are compatible with this aggregation scheme. Observe that by aggregating accesses prior to execution during the SELECTION step, we can also spare the overhead of processing each access individually during the EXTRACTION and ANALYSIS steps.

### 3.1.4 Distance between BID expressions

We have defined a notion of equivalence between BID expressions and proposed an aggregation technique which allows to maintain an access count for each distinct access in a basic block. However, BID expression pairs can only be evaluated via the equivalence relation, hence there is no way to compare two expressions apart from checking if they encode the same address. Most specifically, we don't have a way to determinate how similar are two BID expressions which are not equal. This notion can affect the selection of candidate accesses to select for instrument, by giving proprity to those accesses that are considered to be more different, or less similar, than others. At the same time, the granularity according to which memory access tracing is carried out can play a role in determining what elements of $\mathcal{C}_b$ are more relevant. For example, two or more elements, although being associated with different BID expressions, might target the same memory region at a coarser granularity (e.g. an operating system virtual-page, a block of dynamically-allocated memory, a cache-line aligned chunk, etc.). If the user is only interested in tracing the accesses at such granularity, then instrumenting a single instruction as representative of all those targeting the same region will be enough.

To take into account these new requirements, we introduce the notion of *BID expression distance* between two candidates $e$ and $e'$. We define a distance function for every template $\tau$ and claim that two expressions $e$ and $e'$ belonging to $\mathcal{C}_b$ and associated with different templates cannot be related through a distance. The distance function takes two expressions belonging to that template as arguments and produces a natural number as output to correlate them according to a notion of distance. The computation of a distance also takes into account the *tracing granularity* $\kappa$:

$$\delta_{\tau,\kappa} : \mathcal{C}_b \times \mathcal{C}_b \longrightarrow \mathbb{N} \tag{3.2}$$

Two expressions $e$ and $e'$ belonging to the same template $\tau$ such that $\delta_\tau(e, e') = 0$ are called *similar*. Formally speaking, given a basic block $b$ that contains a set $\mathcal{C}_b$ of

candidate BID expressions, we introduce the *BID similarity relation* $\approx_\kappa \subseteq \mathcal{C}_b \times \mathcal{C}_b$ between any two BID expressions $(e, e')$ belonging to $\mathcal{C}_b$ as follows:

$$e \approx_\kappa e' \iff (\tau_e = \tau_{e'}) \wedge \delta_{\tau_e,\kappa}(e, e') = 0 \tag{3.3}$$

Notice that the similarity relation expressed in Equation (3.3) is not transitive. Given an expression $e$ and two other similar expressions $e'$ and $e''$, $e$ can be similar to both $e'$ and $e''$, but the two expressions $e'$ and $e''$ are only similar with $\kappa' = 2\kappa$. Because of this, the similarity relation is not an equivalence relation and we can't derive equivalence classes over $\mathcal{C}_b$. Nevertheless, we can still exploit the notion of similarity to provide a selection algorithm that takes into account the granularity $\kappa$. Most notably, the concept of expression distance is useful to select expressions in such a way to "sample" the highest number of accesses to distinct $\kappa$-byte memory regions in the address space. For example, it may allow to estimate the paged working set of a scientific application so as to optimally place threads and data on the different processor sockets, or to estimate the cache-resident set of a thread which feeds a software prefetching module.

As a hint of how a distance function should be implemented, consider two expressions $e$ and $e'$ belonging to $\mathcal{C}_b$ which have the same template. We can measure their distance by comparing their characterizing quantities $\{b_e, i_e, d_e\}$ and $\{b_{e'}, i_{e'}, d_{e'}\}$ on a parameter-by-parameter basis. To this extent, we can define a *score triplet* as a tuple of the form $\langle s_b, s_i, s_d \rangle$, where each value is a score associated to the respective parameter $p \in \{b, i, d\}$. We define with aliases $s_3$, $s_2$ and $s_1$ the scores assigned to each parameter in decreasing order. Then, for each parameter $p$ characterizing $e$ and $e'$, the idea is that of incrementing the computed distance between the two expressions by a value $s_p$ whenever $p_e \neq p_{e'}$ (for register parameters) or $|p_e - p_{e'}| \geq \kappa$ (for immediate parameters). To obtain a valid scoring system, a triplet must satisfy the constraints:

$$s_1 < s_2 < s_3$$
$$s_1 + s_2 < s_3$$

Additionally, a valid distance function should always make sure that two expressions $e$ and $e'$ in $\mathcal{C}_b$, for which (i) the immediate parameters differs by a value less than $\kappa$ and (ii) all the register parameters are equal, should be similar.

### 3.1.5 BID and x86-64

For the x86-64 instruction set, we decided to model address expression by resorting only to two BID templates:

1. the `RRI` template, which is suitable for address expressions used to access dynamic objects or objects relative to the current instruction pointer;

2. the `IRR` template, which better matches address expressions associated with accesses to static objects, and which gracefully reduces to absolute addressing when both the index and the displacement are *nil*.

Observe that x86 and other ISAs don't allow to specify more than one immediate parameter in an expression. It wouldn't make sense for at least one reason: if all values are immediate and known, they can be computed as a single immediate value before evaluating the address expression (by a compiler or by Assembly programmers); if they are unknown or don't fit in the value range of the immediate parameter, they can be placed inside a register. As for having two register values in each template, it comes directly from the presence of a base register and an index register in SIB addressing. Mapping the `RRI` template to SIB addressing is easy, as the base parameter corresponds to the base register and the index parameter corresponds to the result of multiplying the index register with the scale. The displacement parameter is the same as the displacement value of x86. As for the `IRR` template, the only difference is that the base parameter corresponds to the x86 displacement, while the displacement parameter corresponds to the x86 base register.

Generally speaking, assigning the correct BID template to each SIB expression is undecidable, as it pertains to the semantics of the application. Stated differently, it is impossible to know where are the most significant bits of an address by simply looking at its expression in code. However, in practice, it is sufficient to look at the displacement in SIB expressions to get an idea of what is the correct template for them. For example, we can try to match the displacement to the offset of static data regions in the executable. If there is a match, then it will be likely that the expression is accessing a static object; otherwise, if the displacement value is significantly lower and doesn't look like an address, we assume it will access an object at a location which depends on the base register. This problem is easier to solve when we are working with relocatable-object files (see Section 3.3).

## 3.2 Selection algorithm

We have defined the notions of equivalence and distance between BID expressions, and proposed an aggregation technique which allows to maintain an access count for each distinct access in a basic block. This allows us to establish an important trade-off between tracing exactness and tracing overhead. However, we still don't have a way to establish a trade-off between overhead and accuracy. Ideally, this trade-off should be passed by the user, to make it possible to trace applications according to different requirements and have expectations in terms of the overhead and the accuracy of the tracing process.

To meet this requirements, we first call *instrumentation coin* our currency to buy tracing of a single access. Its *cost* is constant and is determined by a finite-time contribution to the tracing overhead. Specifically, it is determined by the overhead of inserting additional instructions in the basic block to instrument the representative access. For this reason, it doesn't change with respect to the access to be instrumented. Then, our selection algorithm consists of spending a user-tunable number of instrumentation coins on those accesses that give the highest payoff in terms of value. Conceptually, our problem can be modelled as a simple (0,1)-knapsack constructed as follows. Let $\mathcal{C}_b$ be the set of candidate BID expressions for a basic block $b$ over the set of all expressions $\mathcal{E}_b$. Let $m_b$ be $|\mathcal{C}_b|$, which the total number of equivalence classes. Let $v_e$ be the value of instrumenting an expression $e$ from $\mathcal{C}_b$.

Let $\omega$ be a cost-controlling parameter in $[0, 1]$. To select the access to instrument from the set $\mathcal{C}_b$ we conceptually solve the linear programming problem below, noting that the total weight of our knapsack problem is defined as $W_b = \zeta(\omega \cdot m_b)$:

$$\max \sum_{e \in \mathcal{E}_b} v_e x_e \quad \text{such that} \sum_{e \in \mathcal{C}_b} x_e \leq W_b$$
$$x_e \in \{0, 1\} \qquad v_e \geq 0 \tag{3.4}$$

To choose which elements to select for tracing, we don't actually solve the knapsack problem in Equation (3.4), but an augmented iterative version which takes into account the possibility to influence the choice on the next expression $e'$ to put in the knaspack, after that $e$ was chosen in the previous iteration. To explain why, we first need to understand the meaning of the parameter $\omega$. It is user-defined and is called *instrumentation factor*. It allows to specify a fraction of memory accesses that must be instrumented in each basic block. A value of 0 means that no access will be instrumented, while a value of 1 means that all access will be instrumented. However, while $\omega$ specifies *how many* accesses will be instrumented, the *choice* of which accesses to instrumented depends on the value $v_e$ assigned to each expression. To favour expressions which represent a higher portion of accesses, we need to give a higher value to those expressions which are associated with bigger equivalence classes. However, in order to also be able to pick expressions that are less similar to one another, we need to take into account the reciprocal distance between them. We chose to compute the value $v_e$ as a tuple of $|[e]|$, which is the size of the respective equivalence class to which $e$ belongs, and $\bar{\delta}_e$, which is the average distance of $e$ towards all the other same-template candidates. To represent the fact that, the access count being equal, expressions in $\mathcal{C}_b$ associated with greater average distance from the others will exhibit a higher value of instrumentation (because of the granularity $\kappa$), we define a total order on values $v_e$ for each template as follows:

$$\begin{aligned} v_e = \langle |[e]|, \bar{\delta}_e \rangle \quad v_e < v_{e'} \quad &\text{iff} \quad |[e]| < |[e']| \text{ or } |[e]| = |[e']| \wedge \bar{\delta}_e < \bar{\delta}_{e'} \\ v_e = v_{e'} \quad &\text{iff} \quad |[e]| = |[e']| \wedge \bar{\delta}_e = \bar{\delta}_{e'} \end{aligned} \tag{3.5}$$

This definition of $v_e$ allows us to prioritize expressions that represent a higher number of accesses or that have a higher probability of encoding an address which is less similar to all the others in the same basic block. However, this definition alone is not sufficient to prioritize distant expressions. In fact, whenever we select the next expression to put in the knapsack according to this definition of $v_e$, we might inadvertently pick an expression that is similar to another already in the knapsack. To favour expressions that, even if associated with a smaller value, represent accesses that are considered more distant, we need to temporarily disable any expression $e'$ which is deemed similar to the last chosen expression $e$. Overall, to select the most valuable expressions in a basic block given an instrumentation factor $\omega$ and a tracing granularity $\kappa$, we proceed as follows:

1. We gather all representatives into $\mathcal{C}_b$ and evaluate values $v_e$ for each expression $e$ in $\mathcal{C}_b$.

2. We solve a residual instance of the (0,1)-knapsack problem over the current $\mathcal{C}_b$:

   - Every time we select an expression $e$ from $\mathcal{C}_b$ to be put in the knapsack, we remove from $\mathcal{C}_b$ both $e$ and any other expression $e'$ in $\mathcal{C}_b$ that is similar to $e$. All the expressions $e'$ become part of another set $\widehat{\mathcal{C}}_b$.
   - If there is still space in the knapsack, we set $\mathcal{C}_b = \widehat{\mathcal{C}}_b$ and $\widehat{\mathcal{C}}_b = \varnothing$. Then, we solve the next residual instance of the (0,1)-knapsack problem.

We can therefore define an access to be the "most representative" at any point during the current (0,1)-knapsack problem instance if (i) it has the highest value $v_e$ amongst all expressions in $\mathcal{C}_b$ and (ii) it doesn't belong to the set $\widehat{\mathcal{C}}_b$.

Despite being solved iteratively, the tractability of this linear programming problem is guaranteed by the fact that all items are equally-weighted, meaning that they occupy the same space in the knapsack. This is because, regardless of which access is instrumented, the EXTRACTION phase of the tracing process will produce the same amount of information. Furthermore, the next instance to be solved operates on a residual number of expressions, meaning that its computational cost cannot be greater than that of solving a single instance where all expressions are always considered for selection, as in Equation (3.4). Concretely, our selection algorithm instruments a single basic block in $\Theta(n_b^2)$ time in the worst case, with $n_b$ being the total number of accesses in $b$ (including those which are equivalent), and assuming that the cost of applying $\delta_\tau$ to each pair of expressions is constant. It takes at most $n_b^2$ steps to produce the set $\mathcal{C}_b$, since equivalence classes can be computed in no more than a single quadratic pass over instructions in the basic block. To compute distances between representatives of each class we need $m_\tau^2$ applications of the function $\delta_\tau$ for each template, where $m_\tau$ is the number of candidates belonging to the same template. In the worst case, we only have one template and all expressions are different. A linear scan of $m_b$ steps is needed to compute the average distances for all elements in $\mathcal{C}_b$. The selection algorithm can be implemented by sorting all candidates in $\mathcal{C}_b$ according to their values in decreasing order. Candidates are then read from the priority queue one at a time. If the next available candidate must be discarded given the current granularity $\kappa$, we keep it in the queue and move to the next element. Otherwise, we detach it from the queue in constant time and flag as 'discarded' all others similar elements in the queue. Overall, scanning the queue takes between $W_b$ and $m_b$ steps. The cost of discarding similar elements is linear in their number, provided that they are all logically connected. If this connection can be established while computing distances, overall discarding elements will take between 0 and $m_b - 1$ steps. Finally, the queue will be scanned at most $W_b$ times, when each visit detaches just a single element.

### 3.2.1 Computing the size of the knapsack

Observe that ideally our selection algorithm would select a number of elements $W_b$ to be included in the knapsack equal to $\omega \cdot m_b$. However, in many cases $W_b$ can't be computed this way, as there isn't sufficient precision in a basic block to select exactly that number of elements. For example, if $\omega$ is 0.1 and $m_b$ is less than 10, then rounding must be employed because we can't select exactly 10% of the

accesses in the current basic block. The purpose of the function $\zeta(\cdot)$ employed in Equation (3.4) is to adjust the value of $W_b$ as a function of the desired $\omega$ and the number of equivalence classes $m_b$ in $b$.

It must be noted that, conceptually, the application of the function $\zeta(\cdot)$ to the product $\omega \cdot m_b$ can be seen as equivalent to applying an adjusted instrumentation factor $\omega_b^\zeta$ to the current basic block. This can have a detrimental effect on both the overhead and the accuracy. Since this problem is relevant for any implementation of the described tracing algorithm, it will be further discussed in Section 3.3.

### 3.2.2 Overhead implications

In Equation (3.4), the *instrumentation factor* is directly related to the fraction of accesses that the user wants to be instrumented for each basic block, and has a direct consequence on the overhead of tracing. Assume that $i_b$ is the number of times that a basic block $b$ is executed at run-time, and that $\mathcal{B}$ is the set of distinct basic blocks that are executed. If we assume that the cost $c_b$ of executing a basic block $b$ is constant, the execution time of a program can be expressed as:

$$C_{orig} = \sum_{b \in \mathcal{B}} i_b \cdot c_b \tag{3.6}$$

If the user is interested in tracing only a fraction $\omega$ of the accesses in each basic block, with $c$ being the cost of a single instrumentation coin, then the execution time of the program with tracing is:

$$C_{traced} = \sum_{b \in \mathcal{B}} i_b \cdot (c_b + c \cdot \omega \cdot m_b) = C_{orig} + c \cdot \omega \cdot \sum_{b \in \mathcal{B}} i_b \cdot m_b \tag{3.7}$$

We can get an upper bound for Equation (3.7) as follows. The cost $c_b$ of a basic block can be expressed as its length $|b|$ in terms of instructions, times the average cost per instruction $k_b$ (e.g., in terms of clock cycles). We can decouple the cost $c$ as $c'_b \cdot k_b$, where $c'_b$ is the cost of a single instrumentation coin in terms of this new currency. If we take $c' = \max_{b \in \mathcal{B}} c'_b$, we get a first bound for $C_{traced}$:

$$C_{traced} \leq C_{orig} + c' \cdot \omega \cdot \sum_{b \in \mathcal{B}} i_b \cdot m_b \cdot k_b$$

Then, we can assume that the number of candidate accesses $m_b$ can't be greater than the total number of instruction in $b$. This is a reasonable assumption, as literally all ISAs only allow to perform at most one access to memory per each instruction. This last observation leads to the following upper bound for $C_{traced}$:

$$C_{traced} \leq C_{orig}\,(c' \cdot \omega + 1) \tag{3.8}$$

As a result, the overhead of instrumentation never exceeds $c'\,\omega + 1$, with the cost of an instrumentation coin $c'$ being the only parameter which depends on the particular implementation. If we assume that the cost of executing an instruction is independent of the particular basic block and instruction executed, then $c'$ is directly proportional to the number of instruction which are inserted in code for

each instrumented access. By varying $\omega$ while keeping $c'$ fixed, we can expect the overhead to vary at most proportionally with it. Lastly, when $c'$ or $\omega$ are zero, $C_{traced}$ degenerates to $C_{orig}$ as expected.

Observe that the tracing granularity parameter doesn't have any direct impact on the overhead of the tracing process. Indeed, the number of expressions that are selected for tracing doesn't vary with $\kappa$. The only thing which varies is the set of selected expressions, and the order in which they are selected. This can in principle have some micro-architectural effects that can alter the overhead of the tracing process, for example in term of code layout and exploitation of the in-processor instruction cache. However, these effects should be minimal and are not expected to dominate in any way the effects which are instead produced by varying the instrumentation factor $\omega$.

### 3.2.3 Accuracy implications

The relation between accuracy and the instrumentation factor is less prominent, as it depends on the particular accuracy definition that we wish to define. Let $\mathcal{A}(\omega) \in [0, 1]$ be an accuracy measure depending on the instrumentation factor $\omega$. The following must hold for $\mathcal{A}(\omega)$:

$$\omega_1 \leq \omega_2 \implies \mathcal{A}(\omega_1) \leq \mathcal{A}(\omega_2) \tag{3.9}$$

Equation (3.9) tells us that a necessary condition for each accuracy measure is that its value increases by increasing the instrumentation factor. This means that if the user is willing to spend more instrumentation coins, she will eventually observe a higher accuracy. It is impossible to decrease the accuracy by increasing $\omega$. Ideally, we would also like to pose another constrain on the accuracy:

$$\forall \omega \quad \mathcal{A}(\omega) \geq \omega \tag{3.10}$$

By virtue of Equation (3.10), the rate at which accuracy increases is at least the increase rate of the instrumentation factor. Therefore, the user can immediately relate the instrumentation factor to the overall accuracy achieved at the end of the tracing process for the whole program. Achieving this bound is difficult in general, as it depends on specific accuracy measure and other factors, such as the selection algorithm being employed and the value $v_e$ assigned to each expression. However, it can still be observed empirically, as shown in Section 3.3.

Overall, our selection algorithm is designed to target two different accuracy measures. The first accuracy metric is the *count accuracy* $\mathcal{A}_{count}(\omega)$, defined as the number of accesses which are detected by our algorithm, over all possible accesses. Let $b \in \mathcal{B}$ be a basic block in the program and $i_b$ be its access count at run-time. Let $s_b^\kappa(\omega)$ be the number of selected accesses and $n_b^\kappa$ be the total number of accesses in $b$ when considering accesses to regions of granularity $\kappa$. Then, the count accuracy can be evaluated as follows:

$$\mathcal{A}_{count} = \frac{\sum_{b \in \mathcal{B}} i_b \cdot s_b^\kappa(\omega)}{\sum_{b \in \mathcal{B}} i_b \cdot n_b^\kappa} \tag{3.11}$$

This accuracy measures is designed to be maximized when we are only interested the most representative accesses in terms of access count. Indeed, when $\kappa$ is equal to 1B (the finest granularity), the algorithm never disables any expression and degenerates to a single (0,1)-knapsack problem instance that is solved at once. When this is the case, $s_b^\kappa(\omega)$ is the sum of the top $W_b$ expressions with the highest access count. However, changing the value of $\kappa$ can affect the count accuracy negatively. Given some expressions which are similar, the algorithm instruments a single of them and moves to other expressions, only returning to the one left apart if there is still space in the knapsack. By varying the value $\kappa$, we can affect the number of similar pairs seen by the selection algorithm, which in turn can alter the selection of candidates in $\mathcal{C}_b$. As a result, it is possible for a discarded similar expression $e'$ to have a value $v_e$ which is greater than the one of another expression which is included in the knapsack. This scenario becomes more likely when increasing $\kappa$, as more expressions are considered similar.

To take into account the need for covering the widest range of different accesses to the address space, we can define another accuracy measure, called *single accuracy* $\mathcal{A}_{single}$, defined as the number of accesses to distinct regions of granularity $\kappa$ over the total number of distinct accesses. It measure the coverage of our selection algorithm, intended as the capability to catch accesses to distinct $\kappa$-size and $\kappa$-aligned memory regions in the address space. Let $\hat{s}_b^\kappa$ be the number of distinct selected accesses to regions of granularity $\kappa$ and $\hat{n}_b^\kappa$ be the total number of distinct accesses in $b$. Then, we compute the single accuracy as follows:

$$\mathcal{A}_{single} = \frac{\sum_{b \in \mathcal{B}} i_b \cdot \hat{s}_b^\kappa}{\sum_{b \in \mathcal{B}} i_b \cdot \hat{n}_b^\kappa} \tag{3.12}$$

Notice that if we only consider the average distance between expressions and their similarities, then we favor the single accuracy over the count accuracy. This is equivalent to solving the iterative (0,1)-knapsack by ignoring the size of equivalence classes in the value $v_e$ of an expression. In general, we cannot say whether this approach actually maximizes this definition of accuracy, as we don't know what is the actual distance between expressions in the final address space. However, by discarding similar expressions, we are minimizing the probability of picking two expressions which access the same memory region of granularity $\kappa$. Therefore, favouring distant expressions over similar ones is a way to increase the single accuracy at every step of the selection algorithm.

As already suggested in Section 3.2, our selection algorithm is designed to consider both accuracy metrics at the same time. Indeed, even if $\omega$ affects the number of items $W_b$ in the knapsack, our algorithm can choose which $W_b$ instructions to instrument at freedom. To maximize the number of instrumented accesses, it would be sufficient to simply instrument the top $W_b$ elements of $\mathcal{C}_b$ with the highest access count. By doing this, we would however loose the opportunity to exploit the parameter $\kappa$ to try to observe a wider fraction of the address space of the program. For this reason, the algorithm still takes the top $W_b$ candidates, but discards those which are believed to fall within a memory region of granularity $\kappa$ which was already intercepted.

## 3.3 Experimental assessment

In the following section we evaluate our selection algorithm in the context of ELF object files, the x86-64 ISA, and the Hijacker open source instrumentation framework [66], augmenting its capabilities in order to fully support the proposed algorithms[2]. We also report a set of representative data collected with benchmark HPC applications taken from the PARSEC suite [12].

### 3.3.1 Implementation details

In this section we proceed to illustrate the tools that have been employed to implement the selection algorithm of Section 3.2 and perform the experimental evaluation on the PARSEC benchmark suite.

#### Hijacker instrumentation framework

Hijacker is a static instrumentation tool aimed at providing the necessary support for generic binary code modifications. It addresses this issue by operating at the binary level and modifying the binary image of programs according to some external user directives. Hijacker lays between the compiling stage and the linking one of a typical software compilation tool-chain. Its high-level functioning can be summarized as follows. Hijacker takes as input a relocatable object file and a user-defined configuration file containing rules that drive the instrumentation process. As a result of such process, a new relocatable file is produced as output, carrying the intended alterations to the original program's code and data.

Internally, Hijacker translates the input relocatable object file into a convenient internal representation, with little to none dependence on the original object file format. This internal representation is then altered by the instrumentation engine according to a set of rules, provided by the user via the configuration file, which instruct the instrumentation engine on the specific tasks to be performed. After the whole process is accomplished, the patched internal representation of the program is translated back into the original object-file format and a new object file is produced. Such a file is again a relocatable object, entirely compliant with any compiling framework. This allows the user (or an automated tool-chain) to easily link it against the remainder of the software, including external modules (e.g., static libraries). The whole process of compiling, instrumenting, and linking different object files is shown in Figure 3.1.

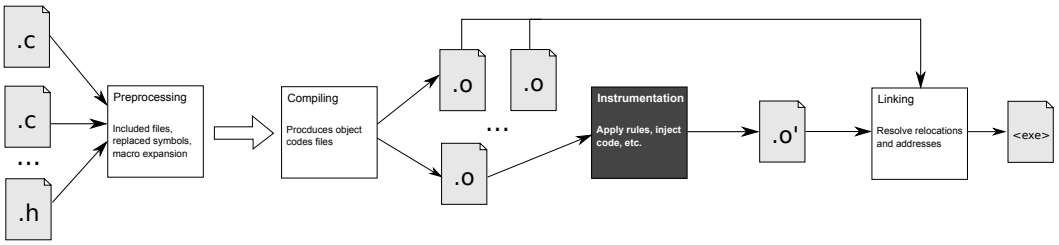



**Figure 3.1.** The process of instrumenting relocatable-object files with Hijacker

[2]Source code available at `https://github.com/HPDCS/hijacker/tree/mascots-16`

A typical invocation of Hijacker would be:

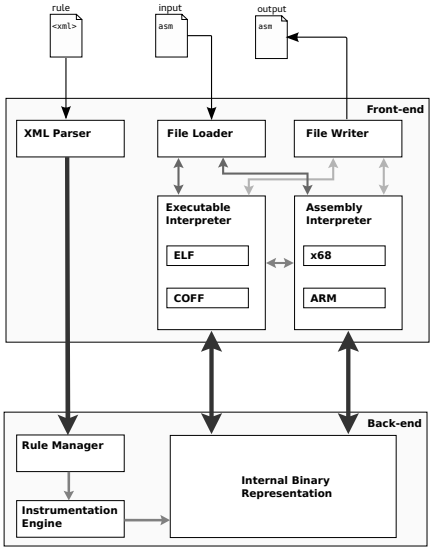
```
$ hijacker -i inputfile.o -c rules.xml -o outputfile.o
```

where `-i` specifies the input file, `-o` the output file and `-c` the supplied configuration file. The main difference between relocatable files and executable ones is that executable files can be directly loaded into a process' address space, while relocatable files contain unresolved memory references among instructions and data which must be finalized by the linker. The choice to work on relocatable objects is driven by three main considerations: (a) the possibility to have at our disposal important meta-data about instructions, data and cross-references between them; (b) the opportunity to separate the instrumentation logic from the linking logic; (c) the potential for flexibility, since in this way modules can be separately instrumented and linked according to a developer's needs. Another way to operate would be to directly instrument executable object files, ready to be launched on the target machine. Although this solution would be more portable—not all tool-chains keep relocatable object files on the file-system, since they are the result of intermediate compilation steps—we would loose all benefits mentioned above.

Hijacker is composed of two main modules, as shown by Figure 3.2a: a *front-end* which directly interacts with the user, interprets and emits objects files; a *back-end* that is basically constituted by a rule manager and the instrumentation engine. The front-end is logically split into a *file loader* module which parses the input relocatable file and a *file writer* module which emits the instrumented file. These modules internally trigger two engines, the *executable interpreter* and the *assembly interpreter*, whenever a format-dependent or machine-dependent stage has to be undertaken (e.g., disassembly). This allows Hijacker to handle different combinations of object formats and instruction-sets—despite currently supporting only ELF and x86. The back-end comprises the rule manager and the instrumentation engine. The rule-driven engine is responsible for parsing the directives contained within the `.xml` file specified by the user. The instrumentation engine is invoked by the rule manager synchronously with the detection of a new rule (that is, as soon as a rule is met in the configuration file.)

The logical bridge between Hijacker's front-end and back-end modules is represented by the *Intermediate Binary Representation* (IBR), an abstract high-level structure maintained for the sake of convenience. Hijacker's IBR is built out of the input relocatable file to operate on a logical view of the program, independent of either the original object-file format or the original instruction set. Stated differently, it abstracts away many format-specific details of the input object file, and decouples architectural details of the underlying instruction set architecture from semantic code aspects. Compared to the offset-based byte-level representation typical of binary program images, our IBR refers to objects in the program (e.g., sections, functions, variables, etc.) according to a pointer-based scheme. This means that, whenever a rule is applied, references among instructions, functions and data are preserved, since they are indeed realized as logical pointers in our IBR. For example, instructions can be freely added to (or removed from) the existing chain, while leaving unaltered all other unrelated links. This scheme is powerful and allows us to implement even complex instrumentation tasks with relatively little effort [3].

[3] It also paves the way for an extensible support of other object-file formats and instruction set




**(a)** Hijacker's architecture scheme

```
1  <?xml version="1.0"?>
2  <hijacker:Rules
3   xmlns:hijacker="http://www.dis.uniroma1.it/~hpdcs/">
4
5   <hijacker:Executable suffix="version1">
6
7    <hijacker:Instruction
8     type="I_JUMP"
9     injectBefore="inject.s" />
10
11  </hijacker:Executable>
12
13  <hijacker:Executable suffix="version2"
14   entryPoint="foo"
15   initFunc="myinitfunc" finiFunc="myfinifunc">
16
17   <hijacker:Inject file="inject1.s" />
18   <hijacker:Instruction type="I_MEMWR"
19    injectBefore="inject2.s" />
20   <hijacker:Instruction type="I_MEMWR">
21    <hijacker:AddCall function="bar"
22     arguments="target" convention="stdcall" />
23   </hijacker:Instruction>
24
25  </hijacker:Executable>
26
27 </hijacker:Rules>
28
```

**(b)** Example of a `.xml` configuration file

Figure 3.2b shows an example of a simple configuration file. The `Rules` tag is the root tag which logically includes all possible directives. It can contain a variable number of `Executable` tags, each of which defines a different instrumentation version of the same original executable. Each instrumentation version can have its own rules and is independent of other versions. We can define several attributes for each executable version:

- The `suffix` attribute is used to specify the name for this executable version. It also works as a suffix to be appended to the name of all functions in this version.

- The `entryPoint` attribute instructs Hijacker to directly invoke the instrumented version of the program, when it is launched, by calling the function whose name is passed as argument. This attribute is only needed if the input relocatable file comes with an actual `main` function that will be automatically invoked at launch time.

- The `initFunc` attribute is optional and allows to install a custom initialization function right before the actual entry point function is about to start. The function accepts no parameters and returns no value. It must be provided (as a function internal to the module or, more commonly, as an external function) prior to the final linking step.

- The `finiFunc` attribute is optional and allows to install a custom finalization function right before the actual entry point function is about to end. The function accepts no parameters and returns no value. It must be provided (as a function internal to the module or, more commonly, as an external function) prior to the final linking step.

families, without the need to modify the internal IBR-based instrumentation engine.

Within an executable version, an `Instruction` tag selects instructions according to their type and allows the user to inject before or after them a desired instrumentation stub. It comes with the following attributes:

- The `type` attribute select the class of instructions to be instrumented. A comprehensive list of the available classes is provided in [66].
- The `injectBefore` and `injectAfter` respectively instructs Hijacker to insert additional code before of after the targeted instruction. Code is passed via an Assembly file whose path is passed as argument to the attribute.

In more complex scenarios, the instrumentation stub can call an external function. This is specified through an additional, nested `AddCall` tag which requires a few extra attributes:

- The `name` attribute specifies the name of the function to be invoked.
- The `arguments` attribute specifies the kind of attributes to the passed to the function. Currently, it only accepts the `target` value, which tells Hijacker to provide the function with disassembled information about the instrumented instruction.
- The `convention` attribute specifies which calling convention must be honored to actually invoke the function.

With Hijacker we also left open the opportunity to manually inject custom Assembly/C code through the `<Inject>` tag. It only accepts a `file` attribute, which represents the path to the file containing the code to be injected. Lastly, Hijacker also admits the possibility to develop custom plug-ins or skins, called *presets*. They can be invoked within each executable version via the `Preset` tag and typically solves specialized instrumentation tasks. The next section describes the preset we developed to implement the tracing technique discussed in this chapter.

### Selective Memory Tracer

Selective Memory Tracer (SMT), or simply `smtracer` is an implementation of the techniques described in this chapter for the Hijacker instrumentation framework, the x64 instruction set, and Linux ELF objects. It allows to instrument relocatable object files by allowing the user to pass the desired instrumentation factor $\omega$ and the tracing granularity $\kappa$. To distinguish between the two x86-64 BID templates described in Section 3.1.5, SMT exploits the meta-data present inside relocatable-object files. Specifically, whenever it encounters relocation information for the displacement value of an x86 SIB memory access expression, it applies the `IRR` template to the expression; otherwise, it switches to the `RRI` template. The presence of relocation information means that the displacement value will be overwritten with a value upon linking. If this is something for the linker, it must be an address to an object whose location is known prior to execution. Therefore, the displacement must contribute to the most significant bits of the effective address at runtime. To derive distance functions for the two x86-64 BID templates, SMT uses the functions shown in Figure 3.3. They

**procedure** $\delta_{RRI}(e, e')$
  $\delta \leftarrow 0$
  **if** $b_e = b_{e'}$ **then**
    **if** $i_e = i_{e'}$ **then**
      **if** $|d_e - d_{e'}| \geq C$ **then**
        $\delta \leftarrow \delta + s_d$
    **else**
      $\delta \leftarrow \delta + s_i$
  **else**
    **if** $i_e = i_{e'}$ **then**
      $\delta \leftarrow \delta + s_b$
    **else**
      $\delta \leftarrow \delta + s_i + s_b$
  **return** $\delta$

**procedure** $\delta_{IRR}(e, e')$
  $\delta \leftarrow 0$
  **if** $|b_e - b_{e'}| \geq C$ **then**
    $\delta \leftarrow \delta + s_b$
  **else**
    **if** $i_e = i_{e'}$ **then**
      **if** $d_e \neq d_{e'}$ **then**
        $\delta \leftarrow \delta + s_d$
    **else**
      **if** $d_e = d_{e'}$ **then**
        $\delta \leftarrow \delta + s_i$
      **else**
        $\delta \leftarrow \delta + s_d + s_i$
  **return** $\delta$

**Figure 3.3.** Algorithms implementing the distance functions for the two x86-64 BID templates `RRI` and `IRR`.

directly build on the scoring system hinted at in Section 3.1.4, using a scoring triplet $\langle s_b, s_i, s_d \rangle$ equal to $\langle s_2, s_3, s_1 \rangle$ for the `RRI` template, and $\langle s_3, s_1, s_1 \rangle$ for the `IRR` one, where $\langle s_1, s_2, s_3 \rangle$ are $\langle 1, 3, 5 \rangle$. These functions also comply with the the requirement that when two BID expressions only differ in their immediate values with a relative difference less than $\kappa$, they are similar.

The EXTRACTION part of STM is based on logging the instrumented accesses to a per-thread buffer, to avoid synchronization costs by the instrumenting code when dealing with multi-threading. To transparently assign each thread with its own private buffer, we exploit a low-level thread-private memory allocation mechanism called *thread-local storage* (TLS) [34]. This model, supported by the majority of architectures and operating systems, allows to define particular data sections in binary object files that are transparently handled by the operating system and the standard library to provide each thread in the application with a private memory area. Memory objects that fall in this area are defined similarly to symbols in other data sections. The only difference with respect to shared memory objects is that an access to a TLS object is thread-private, meaning that there are as many object instances as the number of currently running threads. The implementation also offers the possibility to transparently inject a call to a stub-function that can be used to process each TLS buffer in a custom manner, depending on the final objective of memory tracing—for example, the function could simply flush the logged accesses onto files for post-analysis. This function is invoked prior to leaving a basic block. Currently, the size of this buffer is computed and tuned by SMT depending on the given object file.

To enable SMT in Hijacker, the rules file must be written as suggested in Figure 3.4. First of all, we must create a new version of the executable to which SMT will apply its instrumentation. Recall that each new version must be associated with an entry point and that there must be a way to switch to that version at run-time. Within this version, we must call the `smtracer` preset, specifying a flush function to process TLS buffers via the `function` attribute. The signature for the flush function which will be invoked is:

```
1 void f(unsigned long start, unsigned long size);
```

```
<hijacker:Executable entryPoint="main_instr" suffix="instr">

  <hijacker:Preset name="smtracer" function="myflushfunc" convention="stdcall">
    <hijacker:Param  name="instrfact"   value="1.00" />
    <hijacker:Param  name="chunksize"   value="0" />
    <hijacker:Param  name="tracestack"  value="true" />
  </hijacker:Preset>

</hijacker:Executable>

```

**Figure 3.4.** Example of a `.xml` configuration file to enable `smtracer` in Hijacker

where `start` is the address of the TLS buffer and `size` is the total number of entries in the buffer. Each entry in the buffer can be described by the following structure:

```
struct entry {
  unsigned long address;    // Computed address of this memory access
  unsigned long count;      // Number of times that address has been
                            // found (cluster cardinality)
  unsigned long bbid;       // Basic block ID
}
```

A typical flush function will store the contents of the base TLS buffer into a larger buffer that will be eventually dumped to a file, or consumed at runtime by a dedicated thread or process daemon. To support the task of dumping the contents to a file, a finalization function for the current instrumentation version can be provided. SMT provides an experimental implementation for a flush function that maintains a larger buffer via a lock-free algorithm. The implementation must be linked to the instrumented relocatable-object file prior to producing the final executable. Failing to do so would make the linker complain about missing symbols.

The relevant attributes and values for the `smtracer` preset are described below:

- The `instrfact` param is the instrumentation factor provided to SMT. It has a range between 0 and 1, where 0 means that no memory access is ever instrumented, while 1 means all memory accesses are intercepted.
- The `chunksize` param is the tracing granularity provided to SMT. A value 0 means that accesses are treated at the granularity of a single byte, while a higher value determines a granularity computed as a power of 2 (e.g., 4KB chunks are obtained by passing 12 as granularity).
- The `tracestack` parameter determines whether Hijacker will instrument stack accesses. This is useful when only accesses to the heap or static data sections are of interest.

To deal with the approximation problem described in Section 3.2.1 for the knapsack size $W_b$, SMT implements a simple error propagation algorithm that scans basic blocks in a linear fashion, in the same order as they appear in code. For each basic block, it computes the number of instructions to instrument with both $\zeta_1(\cdot) = \lfloor \cdot \rfloor$ and $\zeta_2(\cdot) = \lceil \cdot \rceil$ and computes the absolute error for both functions, expressed as follows:

$$E_b = \frac{W_b}{m_b} - \omega = \omega_b^{\zeta} - \omega \tag{3.13}$$

To decide which function to use, SMT computes the new absolute error for both $\zeta_1(\cdot)$ and $\zeta_2(\cdot)$, then sums it to a global error accumulator. Since $E_b$ can be positive or negative, SMT choose the function $\zeta(\cdot)$ which produces the least global error. This global error is then propagated to the next basic block in the linear visit of the CFG.

### 3.3.2 Results

In this section we provide an experimental evaluation of the SMT tool against the PARSEC benchmark suite. We provide accuracy and overhead results obtained after running each benchmark with the instrumented code. All the experiments have been executed by running the applications on top of a 32-core HP ProLiant server equipped with 64 GB of RAM, running Linux (kernel 2.6).

#### Overhead and accuracy results

The achieved results are shown in Figure 3.5, where we report both memory access tracing accuracy results and results on the slowdown caused by instrumentation, while varying the instrumentation factor $\omega$ between 0.1 and 1. In this set of experiments we also varied the tracing granularity $\kappa$ determining the granularity of memory access tracing between 16B and 4KB. All the runs have been carried out by relying on 32 threads, and each reported sample (either for accuracy or slowdown) represents the average over three executions.

In these experiments, slowdown is computed by including on the critical path of the application execution only the latency for logging the memory access trace into the TLS buffer. The cost of the processing function at the end of each basic block is not considered as it depends on the final objective of memory tracing. The slowdown value reported in the plots represents the percentage increase in the CPU time used by the benchmark applications with respect to the one observed with no instrumentation at all. We also report results for the case of non-selective instrumentation, where any memory-referencing instruction is instrumented regardless of any notion of equivalence class and granularity. This scenario, marked as 'NS' in the plots, is used as a baseline to assess the slowdown of our selection algorithm. To provide indications of how our algorithm behaves with different user requirements in terms of accuracy, we report for each instrumentation factor and trace granularity two measures for the count accuracy: one is the one described in Section 3.2.3, called $MA$ in the experiments; the other, called $AIA$ is the count accuracy that we would get if we were able to correctly aggregate all accesses to regions at granularity $\kappa$, without making mistakes due to the absence of alignment information in memory address expressions.

As expected, the reduction of the slowdown achieved by our selection algorithm scales (almost) linearly when reducing the value of $\omega$. It must be noted that when there isn't a sufficient number of accesses to honour the instrumentation factor set by the user, the tool can choose to perform a rounding of $W_b$. When the instrumentation

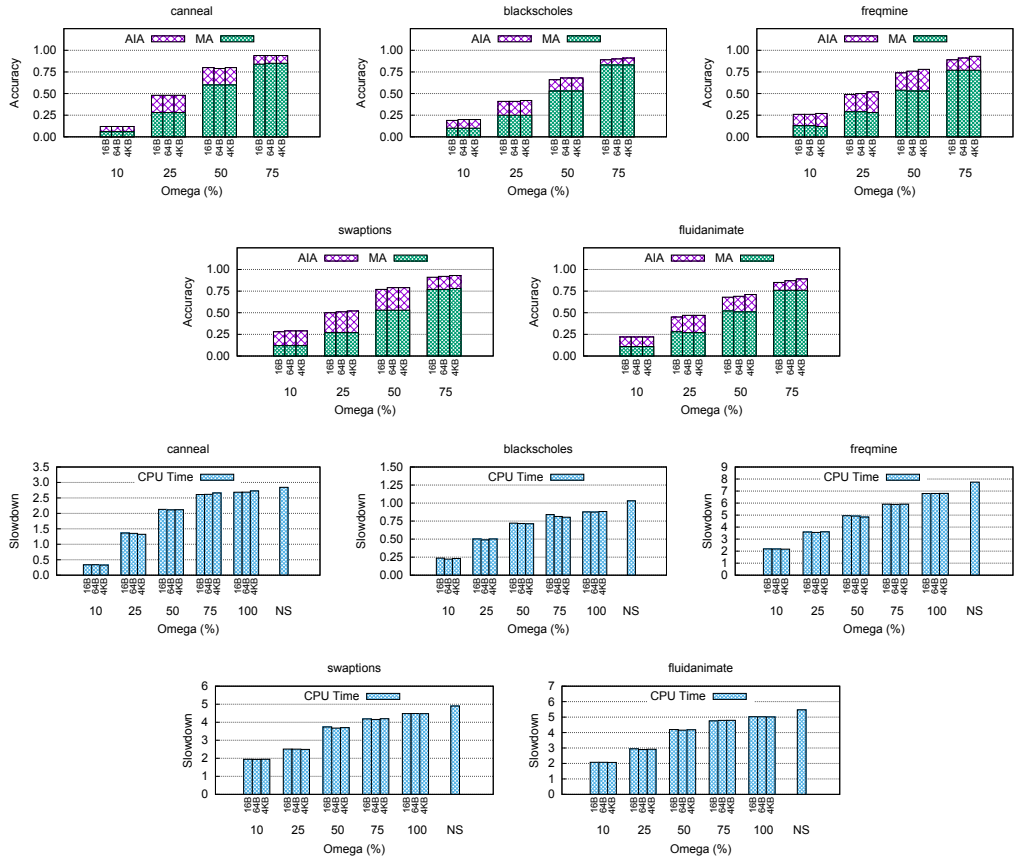



**Figure 3.5.** Results with five representative benchmark applications from PARSEC.

factor is equal to 1, we always beat the NS scheme by a factor that can be as great as an entire baseline (i.e., non-instrumented) run. We generally note a slight reduction of the slowdown with greater values of $\kappa$. This is probably due to the fact that when increasing the tracing granularity, the selecting algorithm performs different choices in terms of accesses to be instrumented, which in turn can alter the code layout of the program and have unexpected micro-architectural effects.

Our selective approach produces accuracy values for $MA$ that scale quite linearly with respect to the instrumentation factor and is always higher than it in literally all cases. As for the case of $AIA$, it is is 15% to 25% higher than $MA$ (as expected) and in many cases it reaches values that are 0.25 higher than $\omega$ in absolute value. For example, it is close (or slightly above) 0.5 and 0.75 when configuring $\omega$ in such a way to only instrument 25% and 50% of the memory-referencing instructions. Except for `canneal`, we achieve values of $AIA$ of the order of 0.25 when instrumenting only 10% of the memory-referencing instructions. Apart from settings where $\omega$ is 10%, $MA$ appears to be (at least) the 60%/70% of $AIA$. The instrumentation factor being equal, this indicates how a conservative selection algorithm does not lead to excessive degradation of the accuracy when memory alignment is important.

We also observe an impact (although limited) by the granularity $\kappa$ on the accuracy. This is likely to be motivated by the fact that PARSEC applications

are generally characterized by basic blocks that are made up by a reduced amount of memory-referencing instructions (on average no more than 7). Therefore, the likelihood of finding similar expressions given any value of $\kappa$ is low. When increasing $\kappa$ the value of $MA$ slightly decreases. This is expected, as the tracing granularity affects the selection algorithm and can potentially lead to select expression with a lower access count. On the other hand, except for `canneal`, we always observe an increase of $AIA$, meaning that when alignment is not important increasing the granularity of tracing is beneficial to the accuracy as it allows to aggregate more access counts.

Overall, our implementation appears to scale quite well with respect to the instrumentation factor in terms of both different accuracy measures and the slowdown of tracing.

## 3.4 Related work

To the best of our knowledge, our selective tracing algorithm has no direct precedent in past literature works. The work in [68] uses static source-code instrumentation to dynamically (at runtime) optimize thread/data placement on heterogeneous NUMA machines. This work is only targeted at tracing the accesses to (large-size) arrays, in order to locate the associated virtual-pages on the same NUMA nodes where the threads mostly using them are hosted. Skeletonization [51] has been presented as a means for emulating the effects of specific inputs on the memory accesses performed by the application. Compared to these approaches we have a different target, i.e., the one of providing a method to statically exchange tracing exactness for tracing efficiency on the one hand, and to dynamically move along the memory tracing accuracy-overhead trade-off curve. Also, our technique copes with accesses to both static and dynamic data structures, thus being of general usage.

A few solutions rely on application-external libraries, like MPI/OpenMP libraries (see, e.g., [19, 50, 88]), and are aimed at determining the level of sharing in the access to library-managed data across concurrent threads/processes. The final target is to optimize the memory layout of the accessed data structures, such as the memory buffers used for data exchange. Variants of these proposals are based on either a-priori knowledge of the source/destination threads for specific data-exchange operations [43] or a knowledge base of the communication pattern built by tracing previous application executions [31, 10]. Our solution does not exploit any specific library (since we rely on x86 binary rewriting), thus being potentially usable in wider application contexts. Also, it does not require any a-priori knowledge base on memory accesses by applications.

Alternative methods to perform memory access tracing build on hardware facilities. In this category we find solutions exploiting the output from memory controllers [7, 17], software-managed TLBs [56, 86] or instruction-based sampling relying on PMUs [22]. Compared to these solutions, we do not require specialized hardware, hence our approach looks highly general. On the other hand, we share with them the idea to provide a customizable memory access tracing support. While PMU-based approaches allow to tune the trade-off between accuracy and overhead simply by means of a customizable sampling period, our instrumentation factor tries

to spend instrumentation coins on the accesses that provide the highest revenue. Additionally, we provide support for the tracing granularity.

As for operating system-based memory access tracing, most of the literature proposals work at the granularity of virtual-pages [30, 44], while in our approach we can select the level of granularity for tracing. Also, these literature approaches are not able to count the number of accesses to individual pages. Rather, they are mostly tailored to the determination of whether a given virtual-page is currently within the locality of the accesses by some thread, which may help to dynamically place it on specific NUMA nodes in parallel machines. Our approach is able to identify the count of accesses that are materialized by the application run. Hence, we can determine both the locality of the accesses and the hotness of the different accessed locations.

## 3.5 Conclusions and future work

We have presented a static instrumentation method to support a flexible memory access tracing scheme. Our method is based on a policy that selects what memory-referencing instructions to instrument depending on the expressions within each basic block of the program. Our method offers a user-configurable parameter that enables the determination of suited trade-offs between the runtime costs caused by the instrumentation of memory accesses and the accuracy of the memory access tracing process. Another user-tunable parameter allows to specify the granularity at which tracing will occur. Our implementation targets the x86 Instruction Set Architecture (ISA) and ELF object files, although the methodology is suited for other architectures and object formats. An experimental study is also provided with benchmark applications taken from the PARSEC suite, where we show the overhead and the accuracy that is possible to obtain in generic memory-intensive applications.

CHAPTER 4

# Tracing transaction-based parallel applications

The diffusion of multi-core machines has raised the need for paradigms to simplify synchronization when accessing shared objects. In this context we find Transaction Processing (TP), a programming paradigm for managing shared-data accesses in concurrent scenarios. TP allows to declare some high-level operations as *transactions* with atomicity and isolation guarantees. Shared-data accesses coming from within such transactions are arbitrated by a dedicated hardware or software TP layer which gives transactions the illusion of running alone and with an all-or-nothing semantics, thus relieving programmers from the task of implementing such isolation and atomicity guarantees by hand in application code. A TP runtime layer takes care of running transactions, coordinating read and/or write accesses to shared objects, committing completed transactions or rolling back those that can't be kept alive without violating isolation.

In the context of transaction-based applications, a major role is played by *concurrency control* mechanisms. A concurrency control mechanism is the protocol in charge of running transactions in the system to guarantee certain correctness criteria (e.g., serializability), while also maximizing the performance of the transactional processing system according to some desired metrics (e.g., throughput). Roughly speaking, we can divide concurrency control mechanisms into two families: optimistic and pessimistic ones. *Optimistic concurrency control protocols* decouple the execution of transactional operations (such as read and write operations) from the validation of such actions. Therefore, a conflict between two transactions can be detected in a deferred fashion. When this happens, at least one transaction is typically aborted and rolled back to execute, possibly from the beginning. These protocols employ *speculation* to assume that transactional actions have a low probability of conflicts. This scheme is well suited for transactional workloads that exhibit a few conflicts over time, since speculation increases the degree of parallelism in the system by letting more transactions performing active work. A drawback of optimistic protocols is that they waste processing time on transactions that are doomed to abort due to a latent conflict that is not timely detected. In contrast, *pessimistic concurrency control*

*protocols* prevent the arousal of conflicts by stopping transactions from performing operations which may conflict with those performed by concurrent transactions, thus reducing the degree of transactional parallelism in the system. Conservative protocols are best employed when concurrent transactions tend to clash quite often. However, this scheme may be too strict and prevent fruitful concurrency from ever happening.

It has been shown that speculative execution has interesting scalability properties in many applications scenarios (such as Discrete Event Simulation, or DES, applications [9]), suggesting that it is a better approach for many workload scenarios than pessimistic models. However, letting transactions execute optimistically poses several challenges. A major research trend in the scientific literature is that of reducing as much as possible the number of transaction aborts. Several approaches have been based on *transaction scheduling* policies [25, 29, 94], which control whether some standing transaction can be admitted to the processing stage or needs to be delayed for a while because of a high likelihood of conflicts with already running transactions. A few of these techniques [32] rely on migrating transactions to queues managed by different threads, so as to increase the likelihood that transactions accessing overlapping data sets are serialized, thus not interfering with each other. An alternative approach to the reduction of the incidence of roll-backs has been the one of adopting *thread scheduling* policies [78, 76, 33, 75, 27]. Unlike transaction scheduling, thread scheduling policies do not delay the processing of standing transactions. Rather, they aim at (dynamically) determining the well-suited level of parallelism to avoid thrashing due to transaction aborts caused by excessive thread-level concurrency. A recent survey of all these techniques can be found in [80].

An important family is that of *hybrid concurrency control protocols*, where some transactional operations are performed according to pessimistic policies, while other are allowed to proceed optimistically. In this context, concurrency control protocols based on *read-validation* schemes are optimistic on read operations while being typically more conservative on write operations. For example, a transaction may read the value of a shared object without locking it, thus unleashing concurrency. Write operations can be implemented by taking a lock on the written object, so as to prevent any concurrent write or read operation from taking place. Depending on the specific concurrency control protocol in use, locks can be taken upon performing the write operation or at commit time. To reveal possible conflicts on the objects previously read by a transaction, a deferred validation operation (or *read-validation*) can be performed. This approach is actually exploited in several families of concurrency control protocols, such as multi-version ones [38]. In the context of distributed transactional systems, like in-memory data management platforms in the Cloud, read-validation concurrency control allows for high scalability even under data replication (see, e.g, [67]).

A major drawback of read-validation protocols is that they allow transactions which are doomed to abort to run until a subsequent validation operation reveals them as invalid. Typically, a transaction is validated at its commit attempt. However, it is also possible to assess the validity of a transaction at earlier points along its lifetime. This can be done either for the sole purpose of performance, or to support correctness criteria stricter than serializability, such as opacity [46]. In any case, a concurrency control protocol doesn't necessarily require to reassess the validity

of a transactional read operation immediately after a conflict has arisen due to updates by a concurrent transaction. This prevents the possibility of early aborting a transaction that is already doomed to abort. The transaction would run until a commit is attempted, or until the concurrency control protocol decides to perform a read-validation. These late aborts do not favor the reduction of wasted computation and can penalize performance and lead to a waste of resources (e.g., CPU).

To counteract this problem, we present a tracing process which intercepts transactional operations to build an analytical model that predicts the abort probability of transactions handled via read-validation schemes. This model can be exploited to determine what are the best moments—along a transaction lifetime—to carry out a validation check which will lead to aborting a transaction with high probability. To this extent, we provide a hint on how to exploit the abort probability predictions returned by the model in combination with a threshold-based scheme to trigger read-validations. We show that our model can precisely estimate the probability of abort of transactions performing a varying fraction of read/write operations, with a maximum absolute error that never exceeds 10% in our implementation. We also show that our on-line treshold-based exploitation scheme can lead to up to 14% better turnaround in experiments carried out with a port of the TPC-C benchmark to Software Transactional Memory (STM).

Overall, the work presented in this chapter involves the Analysis and Exploitation parts of the tracing process illustrated in Chapter 2. The original contributions of this chapter can be summed up as follows:

- We propose a probabilistic model which predicts, at run-time, the probability that a transactional read-set has become invalid.
- We present an off-line experimental assessment of the accuracy on the model, showing that it can estimate the abort probability of transactions with different mixtures of read and write operations with a very reduced maximal error.
- We present a simple on-line model exploitation scheme to validate a transaction as soon as its abort probability hits a threshold value, showing that it can reduce the turnaround of transactions emulating a real-world use case scenario.

## 4.1 Baseline formalisms

We frame our contribution within the scope of read-validation hybrid concurrency protocols (or read-optimistic concurrency protocols). Transactions execute read and write operations on shared objects. They are executed speculatively and can be aborted if something has doomed their execution. The transactional *read-set* is the set of all transactional object read since the beginning of the transaction. Transactions also maintain a *write-set* for the transactionally written objects. Upon an abort, the transaction is roll-backed: this means that the transaction will start from the beginning with empty read and write sets. A running transaction can, at any time, be invalidated by a *conflicting transaction*—i.e., a concurrent transaction performing write operations on the same read and/or written shared objects. When this happens, the concurrency control protocol doesn't necessarily reassess the

validity of the victim transaction in a timely manner. Because of that, it will miss the opportunity to abort the transaction right after that conflict materializes.

Our focus is on those concurrent writes by another transaction which conflict with prior reads in the read-set. We refer to these scenarios as *write-after-read* conflicts. Other conflicting operations can result from the interaction between a write and another write—a *write-after-write* conflict—or between a read and a write operation—a *read-after-write* conflict. Write-after-write conflicts can be already handled efficiently by many read-validation schemes, since typically a transactional write operation takes a lock on the transaction object being written. The same holds true for read-after-write ones, as reading an object which is subject to a write lock is disabled when protocols need to enforce full isolation. Two read operations on the same object from two different transaction don't lead to a conflict. However, depending on the protocol in use. a successful read operation in a transaction may conflict with the previous read operations by the same transaction (e.g., in multi-versioning schemes, see Section 4.4). Other conflicts can arise on internal metadata handled by the transactional processing system, hence they depend on the specific implementation. In the rest of this chapter, we ignore all these additional conflicts. Write-after-read conflicts are the only conflicts accounted for when computing the abort probability of a transaction.

A *read-validation* is a transactional operation that is implicitly performed by the transaction to make sure that the objects in its read-set are still valid at that time, i.e., there haven't been any conflicting write to at least one of these objects. For the sake of brevity, and in accordance with our focus on write-after-read conflicts, in the rest of this chapter we will refer to read-validations as simply *validations*. Additionally, we will call *abort probability* the probability of observing any write-after-read conflict on the objects read by a running transaction. Therefore, the abort probability of a transaction is the probability that a transaction has become invalid due to conflicting writes to the objects in its read-set. A *commit validation* is the validation operation which is performed at commit time to check that the read-set of a transaction is still valid at that point. A validation is *proactive* (also termed as *early* validation) if it is performed prior to commit time, in an attempt to anticipate an abort that would only occur at commit time. When a proactive validation produces an abort, we call it an *early abort*.

## 4.2 Analytical model

If we look at the abort probability for a generic transaction $T_x$, we can see that it increases over time due to the conflicting actions performed by other transactions on the objects accessed by $T_x$ itself. When the transaction starts, its probability of abort is zero because it hasn't read or written any shared object yet. As soon as the first object is accessed, the probability starts to increase because other transactions may access the same object concurrently. The more objects are accessed, the faster the probability of abort increases. This is due to the fact that the probability of abort depends on the probability of conflicts on all the accessed objects. Therefore, the more objects are accessed, the more likely it is, at any point, that any of these objects has been concurrently written by some other transaction. Because of this,

longer transactions can be more likely subject to aborts. The rate at which the abort probability increases depends not only on $T_x$, but also on the rate at which concurrent conflicting transaction perform transactional operations. If only a few transactions are executing at any moment, then it is less likely to observe a conflict on any particular shared object. If there are many concurrent transactions running at any time, the abort probability of any transaction tends to grow faster. Some events might, in principle, decrease the abort probability of a transaction. For example, a transaction might decide to release an object from its data-set, or read it again from shared memory so as to make sure to use an up-to-date value. Some concurrency protocols have read-validation operations which can extend the lifetime of a transaction that is doomed to abort because on an invalid read operation, without aborting. The remainder of this section is devoted to providing the details on the analytical model being used to cater for these different situations. It serves as the basis to our Analysis step, as it gives us a way to estimate the abort probability starting from information on the shared-object accesses performed by each transaction.

We denote with $\mathcal{D}$ the repository of data objects available to transactions for read and write operations, with $d_j$ being the $j$-th element of this set. The read-set of a transaction $x$ at time $t$ is denoted with $\mathcal{R}_x(t)$. It is composed of a sequence $r_i$ of reads of elements in $\mathcal{D}$. Each $r_i$ is a tuple $\langle j_i, t_i \rangle$ containing the index of the transactional object being read and the time at which the read occurred. Both $t$ and $t_i$ represents points on a physical-time axis[1]. We also maintain a *global commit clock* which advances with each distinct commit of a transaction. This clock is essentially a global counter of committed transactions and exists on a logical-time axis. To ease the mapping between logical and physical time, as required in our analysis, we conveniently denote with $cc(t)$ the current global commit clock at physical time $t$.

Objects in the read-set are subject to updates by transactions over time. As already anticipated before, the rate at which objects are updated can affect the abort probability of a transaction. Accessing an object that has a very high update rate lead to a faster increase of the abort probability compared to accessing an object that is hardly updated at any time. The number of updates to a given element $d_j$ in $\mathcal{D}$, which have occurred up to time $t$, are denoted as $n_j(t)$. The value $n_j(t)$ is by definition lower than or equal to $cc(t)$, since an object $d_j$ can be updated at most once at each commit. However, at each commit multiple objects can be updated. We call $\lambda_j(t)$ the rate at which the object is accessed at time $t$. It is expressed as the number of updates globally performed by transactions divided by the current global commit clock at time $t$:

$$\lambda_j(t) = \frac{n_j(t)}{cc(t)} \tag{4.1}$$

Observe that the update rate $\lambda_j(t)$ is an average over the time interval $[0, t)$, meaning that it is not necessarily representative of what is happening within that interval, especially if the latter is large. To circumvent this problem, we assume a system with one or more execution phases, such that each phase reaches stationary

[1]Note that such wall-clock time has nothing to do with the version clock (or object timestamp) of time-based and/or multi-version transactional systems.

conditions and such conditions may vary across different phases. As soon as each phase becomes stable, the rate at which updates to each transactional object are carried out by transactions has a characterizing average value $\lambda_j$ that is independent of time. In such a case, $t$ and $t_i$ refer to the elapsed time since the beginning of the current stable execution phase. We then model the system independently in each one of the aforementioned phases.

In our model, the update rate is used to compute the probability that one or more updates affected any object $d_j$ in between two increments of the global commit clock. Suppose $s$ and $t$ are two physical-time instants such that $cc(s) = x$ and $cc(t) = x + 1$. Then, to compute the probability that an object $d_j$ has been concurrently written by another transaction in between $s$ and $t$, i.e., at $cc(t)$, we can use a Bernoulli trial where the success probability $p_j$ is $\lambda_j$ and the failure probability $q_j$ is $1 - p_j$. In this context, we observe a success when the object $d_j$ is updated by any transaction, while a failure means that no update has occurred in between $s$ and $t$:

$$q_j = 1 - \lambda_j \tag{4.2}$$

If we now assume that $s$ is an arbitrary reference time, then the probability of failure in the $(s, t]$ time interval depends on the number of global commit clock increments in the same period. This is because an update can only occur upon a global commit clock increment. Therefore, the failure probability is the probability of observing as many failures as the number of commits performed in the same physical time period, which can be expressed as $cc(t) - cc(s)$. The resulting failure probability $f_j(s; t)$ corresponds to the probability of failure of independent Bernoulli trials, derived using a geometric distribution:

$$f_j(s; t) = q_j^{cc(t)-cc(s)} \tag{4.3}$$

It must be noted that in order to use a geometric distribution we are assuming that the updates to transactional objects represent independent events. This is certainly true for single-write transactions, i.e., transactions that only perform one and only one write operation. It is also true for transactions with a non-deterministic write-set, but only if it still holds that the probability of writing any $d_j$ is independent of which elements were already written. We must additionally guarantee that the probability of success is the same for every trial. This, under the assumption of a stationary system, is clearly true.

When considering a transactional object in $\mathcal{D}$ which is actually read by a transaction $x$, we are only interested in those updates that occur since the object is put into its transactional read-set $\mathcal{R}_x$. In that case, the reference time for a transactional object read by a transaction then coincides with its reading time $t_i$. Anything that happened before this moment is not of interest for the model, as the object didn't belong to the transactional read-set. Given that $s$ equals $t_i$ for each $r_i$ in $\mathcal{R}_x(t)$, the failure probability at time $t$ can be therefore rewritten as:

$$f_{r_i}(t) \equiv f_{j_i}(t_i; t) = q_{j_i}^{cc(t)-cc(t_i)} \tag{4.4}$$

The failure probability for the entire read-set $\mathcal{R}_x(t)$ of transaction $x$ is the probability of failure for any object in the read-set, starting from the time $t_i$ at which

the object was read. It can be considered as the probability of failure of updating a single composite object, such that we observe a success whenever there is at least one update to any of its parts, while we have a failure when no part is updated. Given a read-set $\mathcal{R}_x$ containing objects $r_i = \langle j_i, t_i \rangle$ is *valid* at time $t$ if no object $d_{j_i}$ read at $t_i$ has been updated in $(t_j, t]$ by some concurrent committing transaction. Therefore, for each element in the read-set, we consider the failure probability at time $t$ starting from the reference time $t_j$. According to logical time, this is equivalent to checking that no updates have occurred in $[cc(t_i) + 1, cc(t)]$. The following equation expresses the probability of observing a valid read-set at time $t$:

$$\mathcal{F}_x(t) = \prod_i f_{r_i}(t) = \prod_i f_{j_i}(t_i; t) \tag{4.5}$$

To obtain the abort probability for transaction $x$ at time $t$, we must have at least a success (therefore an update) inside $\mathcal{R}_x(t)$, meaning that the read-set is invalid. In accordance with elementary probability theory, this means that the modelled abort probability of a transaction $x$ having a read-set at time $t$ can be expressed as:

$$\mathcal{A}_x(t) = 1 - \mathcal{F}_x(t) \tag{4.6}$$

A useful property of the geometric distribution that we exploit is the *memory-lessness* property. In accordance with this property, the probability of observing a number of Bernoulli failures greater than $a$, given than a number $b$ of failures has already been observed, is equal to observing a number of failures equal to $a - b$. Therefore, the probability that an update occurs at a time greater than $t$, given that a reference time $s$ already passed without updates, is the probability of observing at least $cc(t) - cc(s)$ increments of the commit clock. To prove this statement, let $T_{r_i}$ be the random variable which represents the first physical time instant at which an update to $r_i$ materializes. Let $F_{r_i}$ be the number of consecutive Bernoulli failures up to $t$. By virtue of the connection between physical and logical time, the probability that $T_{r_i} > t$ can be expressed as:

$$\mathcal{P}[T_{r_i} > t] = \mathcal{P}[F_{r_i} \geq cc(t)] = q_{j_i}^{cc(t)} \tag{4.7}$$

Suppose now that a time $s < t$ has already passed. The probability of waiting a time greater than $t$ given that we already waited a time $s$ can be computed as a conditional probability of observing at least $cc(t)$ failures, given that at least $cc(s)$ were already observed. The resulting quantity is equivalent to the probability of observing at least $cc(t) - cc(s)$ failures, meaning that the geometric distribution doesn't have a memory of the failures already observed along the logical-time axis:

$$\begin{aligned}\mathcal{P}[T_{r_i} > t \,|\, T_{r_i} > s] &= \mathcal{P}[F_{r_i} \geq cc(t) \,|\, F_i \geq cc(s)] \\ &= \frac{\mathcal{P}[F_{r_i} \geq cc(t)\,, F_{r_i} \geq cc(s)]}{\mathcal{P}[F_{r_i} \geq cc(s)]} \\ &= \frac{\mathcal{P}[F_{r_i} \geq cc(t)]}{\mathcal{P}[F_{r_i} \geq cc(s)]} \\ &= q_{j_i}^{cc(t)-cc(s)} \\ &= \mathcal{P}[F_{r_i} \geq cc(t) - cc(s)]\end{aligned} \tag{4.8}$$

We already exploited this property in Equation 4.4. In that case, the reference time $s$ was the time at which we first read the transactional object $r_i$. The object begins to exist in $\mathcal{R}_x(t)$ starting from the physical and logical time instants $t_i$ and $cc(t_i)$. Before reading the object for the first time, any update was not conflicting and could not affect the transaction. To compute the probability that the first conflicting update to $r_i$ is at a time greater than $t$, we must consider that a time $t_i$ already passed without updates:

$$q_{r_i}(t) = \mathcal{P}[T_{r_i} > t \,|\, T_{r_i} > t_i] \tag{4.9}$$

The memorylessness property can also be used upon validating a transaction. Upon a read-set validation, all objects within it are checked to see if there has been any concurrent update by a conflicting transaction. If that is not the case, it means that the transactional object $r_i$ was still valid at time $t_i'$, which can be used as a new reference time. This is equivalent to saying that the object $r_i$ was re-read at $t_i'$, since we know that up to that moment there were no conflicts. If this is true for every object in the read-set, the latter is valid and so is the transaction. Therefore, when a new validation is performed, the reference time for each $r_i$ is considered to be the last reading time, i.e., the time at which the last (re-)read occurred. By refreshing the reading time of all the objects in the read-set, we abide by the memorylessness property and we make sure that future abort probability calculations performed by the model are adjusted.

We have provided an explanation of the analytical model used to compute the abort probability $\mathcal{A}_x(t)$ of a transaction $x$ throughout its lifetime. The model gives us a prediction based on two input values for each read object $r_i$: its average update rate $\lambda_i$ and its reference physical time $t_i$. The model is based on physical time, but internally uses a convenience auxiliary function $cc(t)$ to translate from physical to logical time. Generally speaking, the model can provide an estimation of the abort probability at any time $t$. However, to compute this estimation we need to evaluate the failure probability of each object in the read-set at time $t$ separately. This can give rise to efficiency issues, as a trivial model implementation would need a linear scan of the read-set, thus making the abort probability estimation slower and slower as the transaction reads more and more objects. In the following section, we explain how the model can be computed and used on-line to solicit earlier aborts and improve the execution time of transactions.

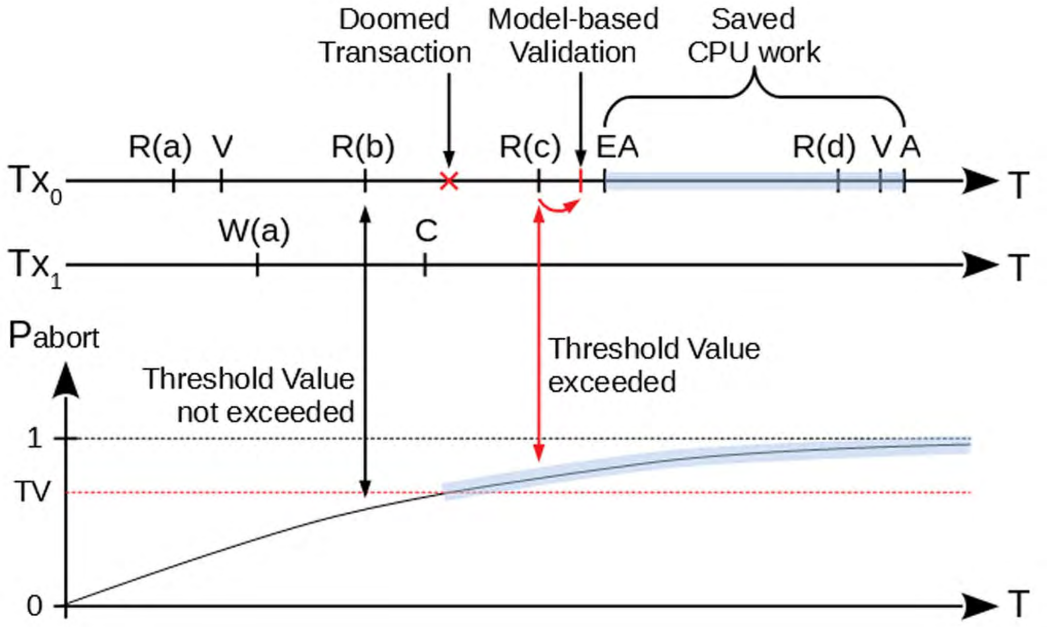



**Figure 4.1.** A visual representation of our threshold-based mechanism for proactive read-validation.

## 4.3 On-line model exploitation

In this section we explain our EXPLOITATION step to install additional proactive validations within the execution of transactions, in an attempt to achieve a higher amount of early aborts and reduce the turnaround of a transaction. These additional early validations are *model-based*, i.e., they are driven by the analytical model explained in Section 5.1, which takes a transactional read-set as input and returns its abort probability as output. When we wish to distinguish model-based validations from other validations already performed by the concurrency control protocol, we refer to the original validations as *spontaneous* validations. Generally speaking, there are many points along the lifetime of a transaction which can be good candidates for proactive validations. For example, a validation can be attempted upon accessing a transactional object for the first time, or when reading an object that may conflict with the pre-existing read-set. Choosing whether and when to perform such an early validation is a choice which depends on correctness and performance aspects. Many concurrency control protocols already perform proactive validations which may lead to early aborts. Figure 4.1 shows two conflicting transactions performing read and write operations on four shared objects—$a$, $b$, $c$, and $d$. If our model-based mechanism is absent, the underlying concurrency control mechanism is still able to perform some early validations (marked as V). The first of such early validations is forced on transaction $\text{Tx}_0$ right after its read access to object $a$, but at this point the read-set of $\text{Tx}_0$ is still valid and so there is no need to abort. In between this access and the next access to object $b$, the read-set of $\text{Tx}_0$ becomes invalid due to a conflicting write access to $a$ performed by another transaction $\text{Tx}_1$, which occurred after the first validation of $\text{Tx}_0$ and becomes visible after $\text{Tx}_1$ commit (marked as

C). Therefore, $Tx_0$ is doomed to abort. However, only after $Tx_0$ has performed a read access to $d$ the underlying concurrency control mechanism decides to trigger another validation and the abort (marked as A) of the transaction $Tx_0$. Hence, the time between the access to object $b$ and that to $d$ is spent doing useless work.

Let's see what happens when we introduce a validation mechanism which is based on our analytical model. A typical usage of the model is to put a threshold on the maximum abort probability that can be tolerated at any time, in order to allow the transaction to run ahead without being (re-)validated. Once this threshold is exceeded, the transaction must undergo a validation of its read-set, which might lead to an early abort. Therefore, a necessary condition for performing a model-based read-validation of transaction $T_x$ is the following:

$$\mathcal{A}_x(t) > TV \tag{4.10}$$

In the figure, we depict the abort probability for transaction $Tx_0$ computed over time. We assume that a threshold value (TV) is used to trigger a model-based validation whenever the abort probability exceeds this value. In the example, this may happen as soon as a read to object $c$ is performed, thus producing an early abort and reducing the time spent on useless computation by an amount which is equal to the highlighted area on the timeline of $Tx_0$. Then, the scheduler used by the transactional system can decide to attempt to re-execute $Tx_0$ immediately, to wait some time (e.g., via an exponential back-off mechanism), or to simply execute another transaction.

It is worth stressing that the condition in Equation 4.10 is only a necessary condition. In fact, the higher the threshold, the higher is the probability of actually experiencing an early abort when the predicted probability exceeds the threshold. In such a case the cost for performing the read-set validation operation can actually pay off. At the same time, for very large threshold values, either the number of threshold violations decreases drastically or the perceived benefit of performing a model-based abort decreases. To motivate the above assertion, let us first consider a transactional system which only validates transactions' read-sets at commit time. Many DBMS implementations based on the read-validation scheme actually adopt this approach. The probability of abort as computed by our model in such a scenario clearly increases as the transaction approaches its end, being it a function of the current time $t$ and of the time at which past read accesses were performed. Unfortunately, there is little gain in validating and aborting a transaction thanks to the model when the transaction itself is about to perform a commit validation anyway. In that case, the cost of performing an additional validation will probably be greater than the cost of waiting till a commit attempt is performed, meaning that further conditions can and should be imposed before triggering a model-based validation in general. On the other hand, lets us consider the case of a transactional system which already performs spontaneous validations in the middle of a transaction to provide some correctness criterion stricter than serializability, such as opacity [46]. In this other scenario, setting too high a threshold value for firing validations means performing a model-based proactive validation only in those rare cases when the aforementioned spontaneous validations have failed to catch any abort. Hence, even if the (predicted and real) probability of abort of the transaction is high at some

point in time, the probability of getting to that point itself—and exploiting the model for triggering validations—is actually very low. In accordance with the above considerations, it would be better to set a threshold value which allows to invoke an early validation when there is more computation time to save (hence a lower threshold value). However, if the threshold is very low, the cost of performing frequent validations can actually be detrimental to the performance of the system. Indeed, the lower the threshold, the higher the fraction of validations which succeed, because a validation is triggered even if the estimated abort probability is low. In that case, running a model-based validation might only be reasonable for a given TV if the estimated time to a spontaneous validation exceeds that spent for the model-based validation. This is especially true for the case of TP concurrency protocols which already perform early validations.

Overall, finding an optimal value for the threshold parameter is a challenging task. Generally speaking, there are at least two main approaches to finding an optimal threshold value. An off-line methodology is to run the application/workload of interest while collecting some high-level profiles and statistics. Then, the best threshold can be found after processing the collected information. For example, one can compute the actual abort probability distribution and set a threshold which is near to the observed mode parameter of that distribution. Another technique can be based on runtime feedback-oriented optimization. The application is run with an initial threshold value—possibly inferred using static off-line techniques such as the one suggested above—and its performance over time is observed. After some time, the threshold is changed and a new observation period starts. A biased exploration, like the hill-climbing approach, can be used to test new values and decide when to stop. In general, a run-time methodology eventually picks the threshold which provides the best performance among all observed values. This process can be repeated over time to optimize multi-phase applications. Additionally, when the workload is comprised of large and small transactions which exhibit quite different memory access patterns, the threshold can be set separately for each transactional profile. The extreme case is that of enabling the model and setting a threshold only for a subset of the profiles. Section 4.4 provides a preliminary experimental evaluation of this threshold-based approach when setting the threshold values manually.

### 4.3.1 Incremental analytical model computation

As explained in the previous section, the proposed on-line model-based validation mechanism is triggered based on a threshold value TV. At time $t$, the inequality $\mathcal{A}_x(t) > TV$ is checked to verify whether the predicted abort probability exceeds the threshold. If such inequality holds, a model-based validation is triggered. While this scheme can lead to beneficial effects, as explained in Section 4.4, evaluating the abort probability $\mathcal{A}_x(t)$ at any time $t$ can still be inefficient due to the way the failure probability $\mathcal{F}_x(t)$ is computed. In this section, we illustrate an incremental version of our model based on a physical-time decomposition of Equation 4.5 that complements the Analysis step described in Section 5.1. To explain how it works, consider the evolution of the failure probability of a single element $r_i$ in the transactional read-set $\mathcal{R}_x$ over a physical time axis. Whenever a transactional read operation on $d_j$ occurs which adds a new element $r_i$ to the read-set of the transaction, the current time $t$ is

fetched and saved as $t_i$. If $\lambda_j$ is the update rate of $d_j$, then at any later time $t'$ we can compute the probability of failure of $r_i$ as $f_{j_i}(t_i; t')$. If we wait till time $t'' > t'$, the failure probability at that time for $r_i$ becomes $f_{j_i}(t_i; t'')$. This probability can be decomposed in time as the failure probabilities in between $(t_i, t']$ and $(t', t'']$:

$$\begin{aligned} q_{j_i}(t_i; t'') &= \mathcal{P}[T_{r_i} > t'' \,|\, T_{r_i} > t_i] \\ &= \mathcal{P}[T_{r_i} > t' \,|\, T_{r_i} > t_i] \cdot \mathcal{P}[T_{r_i} > t'' \,|\, T_{r_i} > t'] \\ &= q_{j_i}^{cc(t')-cc(t_i)} \cdot q_{j_i}^{cc(t'')-cc(t')} \\ &= f_{j_i}(t_i; t') \cdot f_{j_i}(t'; t'') \end{aligned} \tag{4.11}$$

Suppose we now choose $n$ points along the physical-time axis to evaluate the abort probability. By virtue of Equation 4.11, it is possible to evaluate the failure probability of the element $r_i$ by considering the different time intervals $(\hat{t}_k, \hat{t}_{k+1}]$ separately for each $k = 1, \ldots, n$, starting from time $\hat{t}_0 = t_i$ till time $\hat{t}_n = t$. As a result, we can time-decompose the failure probability $f_{j_i}(t)$ as follows:

$$f_{j_i}(\hat{t}_0; \hat{t}_n) = \prod_{k=1}^{n} f_{j_i}(\hat{t}_{k-1}; \hat{t}_k) \tag{4.12}$$

Equation 4.12 offers us an efficient way to maintain the failure probability of element $r_i$ over time, as the product operation is distributive and allows to rewrite the equation inductively:

$$\begin{aligned} f_{j_i}^{(0)} &= 1 \\ f_{j_i}^{(k)} &= f_{j_i}^{(k-1)} \cdot f_{j_i}(\hat{t}_{k-1}; \hat{t}_k) \end{aligned} \tag{4.13}$$

However, we are interested in the failure probability of the whole read-set $\mathcal{R}_x$, for which we still don't have a physical-time decomposition. Intuitively, we can apply the same reasoning we used for each $q_{j_i}(t)$ to derive a time decomposition for $\mathcal{F}_x(t)$ similar in spirit to the one we achieved in Equation 4.12. Unfortunately, while Equation 4.12 uses $q_{j_i}$ as its base Bernoulli failure probability, to derive a similar decomposition for the whole read-set we must consider the fact that $\mathcal{R}_x$ grows over time. Let $Q_x(t)$ be the *read-set failure probability* at time $t$. It accumulates the Bernoulli failure probabilities for all elements in the read-set at time $t$. If we choose again $n$ points in physical time and define $\mathcal{R}_x(\hat{t}_{k-1}; \hat{t}_k)$ as the subset of elements added to the read-set during the interval $(\hat{t}_{k-1}, \hat{t}_k]$, then the read-set failure probability at time $\hat{t}_n$ can be evaluated inductively as shown:

$$\begin{aligned} Q_x(\hat{t}_0) &= 1 \\ Q_x(\hat{t}_k) &= Q_x(\hat{t}_{k-1}) \cdot \left( \prod_{i \in \mathcal{R}_x(\hat{t}_{k-1}; \hat{t}_k)} q_{j_i} \right) \end{aligned} \tag{4.14}$$

Unfortunately, to evaluate $Q_x(\hat{t}_k)$ given $Q_x(\hat{t}_{k-1})$ we still need to perform a number of computational steps proportional to the number of transactional read

objects added to the read-set in between $\hat{t}_{k-1}$ and $\hat{t}_k$. Not only that, but if we choose $n$ arbitrary points in time and $Q_x(t)$ changes in between any two consecutive physical-time instants, this also affects the kind of time-decomposition we can derive for the failure probability $\mathcal{F}_x(t)$. To explain way, let $\mathcal{F}_x(\hat{t}_{k-1}; \hat{t}_k)$ be the failure probability computed in between two consecutive points in the sequence $k = 1, \ldots, n$. To correctly compute this quantity we must consider, just like we did for $Q_x(\hat{t}_k)$, that $m_k$ elements are added to the read-set in the interval $(\hat{t}_{k-1}, \hat{t}_k]$. Let $\hat{t}_k^{(1)}, \ldots, \hat{t}_k^{(m_k)}$ be the instants at which these reads materialize. If we call $\hat{t}_k = \hat{t}_k^{(0)}$ and $\hat{t}_k = \hat{t}_k^{(m_k+1)}$ then we can compute the failure probability as follows:

$$\mathcal{F}_x(\hat{t}_k^{(0)}; \hat{t}_k^{(m_k+1)}) = \prod_{p=1}^{m_k+1} \left( \prod_{i \in \mathcal{R}_x(\hat{t}_k^{(p-1)})} f_{j_i}(\hat{t}_k^{(p-1)}; \hat{t}_k^{(p)}) \right) = \prod_{p=1}^{m_k+1} Q_x(\hat{t}_k^{(p-1)})^{\,\hat{t}_k^{(p)} - \hat{t}_k^{(p-1)}} \tag{4.15}$$

This formulation of the failure probability allows to compute the failure probability in the interval $(\hat{t}_{k-1}, \hat{t}_k]$ in a number of steps that is equal to $m_k$. Notice that in this case $Q_x(\hat{t}_k^{(p-1)})$ can be computed in a single step as each instant $\hat{t}_k^{(p-1)}$ represents a new object added to the read-set. By virtue of Equation 4.12 we can then rewrite the failure probability at time $\hat{t}_n$, with $k = 1, \ldots, n$, as:

$$\mathcal{F}_x(\hat{t}_0; \hat{t}_n) = \prod_{k=1}^{n} \mathcal{F}_x(\hat{t}_{k-1}; \hat{t}_k) \tag{4.16}$$

Therefore, we can compute an incremental version of this probability as:

$$\begin{aligned} \mathcal{F}_x^{(0)} &= 1 \\ \mathcal{F}_x^{(k)} &= \mathcal{F}_x^{(k-1)} \cdot \mathcal{F}_x(\hat{t}_{k-1}; \hat{t}_k) \end{aligned} \tag{4.17}$$

Equation 4.17 is the equivalent of Equation 4.12 for the case of the whole transactional read-set. However, compared to the latter equation, the former is still inefficient to be computed in the general case when the $n$ points in time are chosen arbitrarily. Indeed, we are forced to add as many other points as the time instants at which new read operations are performed in between them. Therefore, even if we already know the value of $\mathcal{F}_x^{(k-1)}$, computing $\mathcal{F}_x^{(k)}$ can still require an arbitrary number of steps. Equation 4.15 admits a simpler formulation if the $n$ time instants we choose coincide with the times at which new elements are added to the read-set. In that case, we have that $\hat{t}_k = t_i$ and $m = 0$ for each $k$:

$$\mathcal{F}_x(t_i; t_{i+1}) = \prod_{i' \in \mathcal{R}_x(t_i)} f_{j_i'}(t_i; t_{i+1}) = Q_x(t_i)^{\,t_{i+1} - t_i} \tag{4.18}$$

Equation 4.18 can be evaluated in constant time if we compute the read-set failure probability incrementally as shown in Equation 4.14 and instantiate our arbitrary $n$ points to the times at which new objects are added to the read-set. As

a result, we get a simpler time-decomposition for the failure probability $\mathcal{F}_x(t)$ as follows:

$$\begin{aligned} \mathcal{F}_x^{(0)} &= 1 \\ \mathcal{F}_x^{(i)} &= \mathcal{F}_x^{(i-1)} \cdot Q_x(t_i)^{\,t_{i+1}-t_i} \end{aligned} \tag{4.19}$$

In the rest of this chapter, whenever the on-line exploitation scheme is used, we will compute the abort probability as in Equation 4.19.

## 4.4 Experimental evaluation

In this section we provide an experimental analysis of the model accuracy on a micro-benchmark we developed which satisfies the properties of the geometric distribution, as well as the impact of a model-based read-validation on the execution of a real-world TP application.

### 4.4.1 Implementation details

We evaluate our tracing analysis technique within the context of Transactional Memory (TM) systems, a family of transactional-processing systems that cope with general-purpose multi-threading applications. TM allows to mark code blocks in multi-threading applications performing accesses to shared-memory as transactions, while preserving atomicity and isolation guarantees which are typical of TP systems. Programmers' productivity is therefore improved, as it makes concurrent programming almost as simple as sequential programming or coarse-grained locking schemes, while also not sacrificing the advantages—in terms of scalability and high parallelism—provided by fine-grain locking techniques. Nowadays various TM implementations exist, including the ones natively embedded in modern processors via specific hardware support, the so-called Hardware Transactional Memory (HTM) [47]. To achieve atomicity and isolation, off-the-shelf HTM implementations typically augment the capabilities of a cache coherency protocol to maintain transactional cache lines. The granularity of data is therefore the granularity of a cache line, meaning that transactional conflicts can be detected by the cache-coherency protocol when invalidating a transactional cache line. Aborting a hardware transaction means dropping all cache lines that were in transactional mode. Committing a transaction, on the other hand, means flushing to lower cache levels in the memory hierarchy all cache lines temporarily marked as transactional. A major drawback of such HTM implementations is that transactions will abort every time the cache must be invalidated prior to committing the transaction, and even for reasons that don't related to transaction execution. This may occur upon sending an interrupt to the CPU-core running the transactional code block, upon a context or mode switch, when there's a capacity miss or cache-line conflicts between threads running in Simultaneous Multi-Threading (SMT) mode, and so on. The most diffused TM implementations are still based on software support, known as Software Transactional Memory (STM) [82]. These provide the advantage of not requiring any specific hardware technology, thus being much more portable across different machines. Additionally, they can

implement different isolation criteria without being limited by the capabilities of the hardware. Further, STMs have been recently identified as an ideal candidate to simplify the programming of distributed applications deployed in large scale data centers [3] or in cloud computing environments [72].

### LSA and TinySTM

A well-known STM algorithm is called Lazy Snapshot Algorithm (LSA) [39]. It is based on an optimistic execution of transactions. An object value is associated with a *validity range* delimited by the *logical time* at which that value was written inside that object (the lower bound) and the logical time of the next write operation, if any (the upper bound). The logical time of a write operation denotes the *version number* for that object's value. A *global clock* is queried to know the current logical time. It is advanced monotonically by each committing transaction, which also updates the version number of all the shared objects that it writes with the next value of the global clock. A running transaction maintains a *snapshot*, which is an interval of logical times delimiting which values can be read out of an object in order to preserve correctness. In principle, a transaction can read all those values whose versions comply with the snapshot. Upon a read operation on a shared object, the value being read must have a version falling within the current snapshot boundaries. If the read succeeds, the snapshot boundaries are adjusted according to the intersection with the read value's validity range. If such intersection is empty, the transaction is conservatively aborted since it is no longer possible to guarantee correctness. A mechanism that is used to reduce the need for aborting transactions is called *snapshot extension*. It is triggered upon observing an empty snapshot to see if an intersection can be found by tentatively trying to shift the snapshot's upper bound to a logical time higher than the first available value's version number. If that is the case, it means that it was possible to find a valid version number for all objects in the read-set, such that the intersection of their values yield at least one valid instant in logical time—a possible *linearization point*. At commit, a transaction acquires a unique timestamp from the global clock and checks whether this commit time intersects with the current snapshot. In the positive case, the linearization point coincides with the commit time and the transaction can commit. Otherwise, the transaction is aborted. Read-only transactions can commit at every valid logical time in their current snapshots, since they are all valid linearization points.

A classic implementation of the LSA algorithm for the C language is called *TinySTM* and is based on word-sized objects. It is a single-version variant of LSA, meaning that a transaction can only read the latest written version of an object. It is also lock-based, meaning that concurrent accesses are detected by means of revocable locks (hence it is still an optimistic STM). In order to store version numbers and locks for each word location, a lock array is used. It is based on a hash function which maps word addresses to entries in this array, each of which contains (a) a lock bit which is the lock itself, and (b) either the last written version number or (in case the lock bit is set) the identifier of the transaction which is currently writing that word-sized object. Due to its single-version nature, snapshots are single logical time instants and extensions try to advance them to the current global clock timestamp when the culprit read is attempted. We call *transaction timestamp* the transaction's

single-version snapshot.

Read operations on shared objects previously updated by the same transaction are served by reading values from the transaction write set. Instead, read operations performed on shared objects outside the write set lead to sample the version number and the lock bit of the shared object in order to check if (a) the timestamp is less than or equal to the timestamp of the reading transaction, and (b) the object is not currently locked. If both checks succeed, it means that no concurrent transaction has modified the object in the interval between the start of the reading transaction and the actual read operation, hence the read value is valid. Otherwise, either an extension succeeds or the transaction gets aborted.

Upon attempting to commit a writing transaction, all past reads are revalidated in order to check that no object in the read-set has been modified after the transaction started; if so, a timestamp corresponding to the value of the sampled global clock +1 is installed for each written object in the respective entry of the lock array. If the revalidation succeeds, and provided that the thread managed to take the locks for each written object, the write set is committed to memory and the transaction completes. If at least one lock acquisition fails, the transaction is aborted and restarted.

Depending on the chosen locking strategy, lock acquisition for write operations may happen at the time shared-data are transactionally written (ETL, or Encounter-Time Locking) or at commit time (CTL, or Commit-Time Locking). ETL algorithms can be further divided into *write-back* schemes, where the write-set is committed to memory upon commit, and *write-through* ones, where this is done upon write, hence an undo log is needed to revert dirty memory in case of aborts. When relying on CTL for managing the locks associated with the objects written by a transaction, the lock holding time is typically very short, being it bounded by the duration of the transaction validation phase executed at commit-time. Consequently, the probability for a transaction to be aborted due to lock-acquisition failure is typically reduced, compared to the probability to be aborted due to the invalidation of some read object-value, as caused by write operations on that same object by concurrent transactions. This is one significant advantage of CTL when compared to other schemes such as ETL where locks are acquired while transaction processing is still in progress.

### 4.4.2 Results

In this section we report our results in terms of accuracy and transaction turnarounds for respectively a micro-benchmark we developed and a port of the TPC-C benchmark to STM.

#### Off-line accuracy evaluation

To quantitatively assess the accuracy of our model we have carried out an experiment based on a synthetic benchmark for STM. A shared array of elements in memory constitutes the whole transactional repository at disposal of transactions. Each transaction performs accesses to this repository, split between read and write accesses hitting the set of objects randomly, with a varying probability of performing a read

operation. The size of the repository, the number of transactional operations and the probability of performing a read operation as the next transactional operation are all user-defined parameters of the synthetic benchmark. We also varied the number of threads and the number of transactions per thread to explore different levels of parallelism. The experiments have been run on top of a 32-core HP ProLiant server equipped with four two-socket 2GHz AMD Opteron 6128 processors and 64GB of RAM. To obtain accuracy results, we evaluated our model using an off-line methodology. We traced each transaction as follows:

- Upon a begin operation, we save the current commit clock to use it as a reference for the next transactional events. This comes directly from the memorylessness property as explained in Section 5.1.
- Upon a read operation, we save the number of updates to the target transactional object and the current commit clock to compute the update rate for the read object and store its start reference time.
- Upon a spontaneous validation operation, for each element in the read-set we save the new number of updates and the new current commit clock; then, we refresh the update rate for the read object and reset its reference time.
- Upon an abort event, we just record the fact that an abort has occurred.

By re-playing the tracing off-line, i.e., after the benchmark has already terminated, we could compute the abort probability as predicted by the model and compare it with the number of times an actual abort event was generated at run-time. We then gathered the predicted abort probabilities into buckets, storing for each of them the number of samples per bucket and the number of real aborts associated to the abort probability values that fell in the same bucket. By doing this, we could evaluate the actual abort probability observed at run-time for each bucket by dividing the number of aborts by the number of samples.

Figure 4.2 shows how the predicted abort probability compares with the actual abort probability. On the x-axis is the abort probability as computed by the model, while on the y-axis is the actual abort probability, derived as the ratio of aborts over total validations for that specific point on the x-axis. We experimented with two configurations. One sets the number $e$ of elements to $10,000$ and the number $o$ of operations per transaction to 200, thus hitting 5% of the objects in the dataset. The other sets the number $e$ of elements to $1,000$ and the number $o$ of operations per transaction to 50, thus hitting 20% of the objects in the dataset. For each of the previous configurations, we experimented with a percentage $r$ of read operations of 85%, 90%, and 95%, while we also run each experiment with a number $p$ of threads of, 8, 16, and 32. The number $t$ of transactions per thread stayed the same in all experiment, being fixed at half a million. These scenarios represent a realistic situation where contention is fairly spread across all the dataset, with a probability of abort that can still be high due to the relatively high number of read operations compared to the overall dataset size. Additionally, it emulates a real-world application scenario where reads tend to be much more frequent than writes. To obtain a perfect match, the predicted probability must equal the actual probability,

as suggested by the straight, blue line on the plot. The increase in transparency of the black dots used to indicate the actual probability represents the fact that some buckets have a number of samples that is not statistically representative to evaluate the accuracy of the model. This tends to happen when the abort probability (both predicted and actual) is very high, and can be explained by the fact that many transactions abort or commit before reaching that probability. The absolute error committed in any configuration never exceeds 10%, which is only reached when the concurrency is the highest (24 threads, last two rows in the figure). This error can be reduced by decreasing the number of read operations (moving to the right of the figure) or by decreasing the level of parallelism in the system (moving to the top of the figure).

To further evaluate the prediction accuracy of the model, we decided to run another batch of experiments, this time by reducing the percentage of read operations to 50% and reducing accordingly the number of transactions per thread to compensate for the increased execution time to the higher number of aborts and overall concurrency in the system. Figure 4.3 shows the results of this experimentation. As it can be seen, the maximum error still reaches 10% in the case of maximum parallelism with 24 threads, while being completely negligible when the number of threads is set to 8. Overall, these error values suggest that the accuracy of the proposed model might be affected by small-order effects due to numerical stability and possibly by some slight statistical correlations between read and write operations from different transactions that make the geometric distribution assumptions less realistic. Nevertheless, the data also show that our model is reasonably stable regardless of the specific read/write ratio of transactional operations, thus being capable of predicting the abort probability of transactions under the studied settings with negligible or small accuracy errors.

### 4.4.3 On-line model exploitation evaluation

We tested the comparison of the predicted abort probability values to $TV$ for triggering validations by integrating our on-line model exploitation scheme into TinySTM. By default, a validation is only attempted in TinySTM upon explicit read accesses, when such accesses may result into a violation of the opacity correctness criterion. If a thread runs a transaction that does not access shared data for a while (e.g., it manipulates local variables into the stack), or accesses data deemed valid (the object timestamp is within the transaction visibility snapshot), then the underlying STM layer does nothing to detect conflicting accesses. On the other hand, our analytical model can predict the probability that the transaction has become invalid upon performing those accesses and be used to trigger a model-based proactive validation. Our implementation of the abort-probability model for on-line exploitation is based on the CTL variant of TinySTM. We redefine the values of entries in the lock array to also keep track of the number of updates to each transactional location. This number is reset after every epoch, which corresponds to a certain number of commits. Each epoch is associated to a starting time, which is the global commit clock read at the beginning of that epoch. Our tracing technique works as follows:

- Upon commit, for each written transactional word we increment by one its associated total number of updates for the current epoch, or we reset it if in the meantime a new epoch has started. The new number of updates and the new version number are installed into the respective lock array entry. Notice that since we use the CTL version of TinySTM, we don't need to do anything upon transactional write operations.

- Upon read operations, we compute the abort probability and check if it exceeds the given threshold. In the positive case, we trigger a model-based validation. Otherwise, we update the failure probability for the current transaction by taking into account the time since the last update. The operation is computed as described in Equation 4.13 of Section 4.3. This only happens if the read operation is to an element that didn't belong to the transactional read-set or write-set. Then, we read the update rate of the new transactional word and use it to actualize the read-set failure probability as described in Section 4.3.

- On early validations, we use the memorylessness property to rejuventate the read-set of the transaction in its entirety, as explained in Section 5.1. This means that we reset the validity probability and start computing it from scratch incrementally. We compute again the update rate of all entries and re-compute the failure probability for the current transaction based on the new rates. This is done incrementally on each object that is re-read and re-put into the read-set.

To evaluate the effectiveness of our threshold-based mechanism, we have chosen a port of the TPC-C [87] benchmark to STM. TPC-C is representative of OLTP workloads and includes different transaction profiles that simulate a whole-sale supplying items from a set of warehouses to customers within sales districts. In our experiments we instantiated one district, and generated a workload made up by requests spanning four different transaction profiles specified by the benchmark, excluding the "delivery" profile since, according to the TPC-C specification, it is conceived to be run in deferred mode. In our porting to the target STM environment, CPU demands for the different transactional profiles of TPC-C range from tens of microseconds to milliseconds, as shown in Table 4.1, where we also report the percentage mix of the different profiles. It must be noted that of all enabled profiles, "new order" is the only one having a high CPU demand, a relevant share of the whole workload (almost 50%), and a mixture of read and write operations. The "stock level" profile is also long-running, but it has a much smaller share of the workload and it is read-only[2].

To emulate a realistic deploy for modern applications, we have run experiments on a cluster of two 64-bit NUMA HP ProLiant servers. The STM application is deployed on one of these nodes—acting as a back-end data layer—while the other node is used for generating the workload of transactional requests. The server node is equipped with four 2GHz AMD Opteron 6128 processors and 64GB of RAM. The client node has two 2.2GHz AMD Opteron 6174 processors and 32GB of RAM. Both

[2]Read-only transactions don't usually undergo any validation upon their commit. In fact, it is admissible that their read-sets contain overwritten data, provided that the read-set is consistent before reaching commit.

| ID | Profile | Type | CPU demand | % mix |
|---|---|---|---|---|
| 1 | **new order** | RW | $\approx$ 350 $\mu sec$ | 0.49 |
| 2 | **payment** | RW | $<$ 10 $\mu sec$ | 0.43 |
| 3 | **order status** | RO | $\approx$ 10 $\mu sec$ | 0.04 |
| 4 | **stock level** | RO | $\approx$ 650 $\mu sec$ | 0.04 |

**Table 4.1.** Transaction profiles and associated CPU demand.

processor models have eight NUMA nodes. In all the experiments, threads remain pinned to their NUMA nodes so as to minimize the impact of (possible) thread migrations by the operating system. We don't move data across NUMA nodes and don't change the default allocation policy for dynamic memory.

We have run our experiments with continuous injection of transactional requests, using 24 threads for processing the requests at the back-end data management node and 6 threads for managing the socket pool from which the client-generated workload comes. This scenario led to use at most 94% of the CPU computational power at the back-end data management node, thus avoiding hardware resources saturation that would affect the reliability of the experimental analysis. We have run with the highest concurrency, as admitted by 24 threads, and configured TinySTM to rely on the Commit-Time Locking scheme for data-lock acquisition upon write operations. We set the backlog of pending transactional requests to be processed at the server side to 4096, and we experimented with a sustained workload leading the backlog to be close to saturation at any used thread count. Each experiment entails 3 million committed transactions.

In our experiments we varied the parameter $TV$ using the following values: 100%, 85%, 70%, 55%, 40%, 25% and 10%. Upon a read access to a transactional object at time $t$ performed by transaction $x$, the inequality $\mathcal{A}_x(t) > TV$ is checked to verify whether the predicted abort probability exceeds the threshold. If such inequality holds, a new validation task is triggered before completing the access. The configuration with $TV = 100\%$ leads our model-based optimization to never fire any model-based validation, since the estimated abort probability of transactions cannot be greater than one. However, this configuration is important in order to assess what is the actual overhead for managing the model at runtime—especially to access the global commit clock value and to keep information related to the update rate of each distinct transactional word in memory.

In Figure 4.4 we report the variation of the transaction turnaround for profile #1 using the different values of $TV$, relative to the turnaround that can be observed when running the original configuration of TinySTM (not embedding our model-based validation scheme). We will refer to the latter configuration as 'baseline' in our discussion. As already anticipated, we reported data for profile #1 only since profiles #2 and #3 are so short running that no early abort technique allows for actual improvements of their performance. On the other hand, profiles #3 and #4 are read only, and TinySTM already applies to these profiles a set of runtime optimizations that stand aside of the model-based read-validation scheme we present. By the data we see how, although the runtime management of the model introduces about 5% overhead (see the bar for $TV = 100\%$), as soon as we also exploit the model we do

not only recover the loss of performance caused by the runtime model-management overhead; rather, we also achieve a performance gain over the baseline of about 14%. Even more important, we observe that the performance gain provided by the model-based read-validation scheme is stable for large interval of values of $TV$. This is in practice an indication of the effectiveness of the approach even in contexts where no (extremely) fine tuning of the value of $TV$ is adopted.

## 4.5 Related work

In the literature, several analytical models have been presented which cope with very disparate concurrency control schemes for transactional systems (see, e.g., [96, 65, 24]). Most of them have been exploited as off-line tools for performance analysis and prediction, while a few of them have been exploited as runtime decision supports. In the specific context of STM, analytical runtime decision models have been proposed in order to determine suited levels of parallelism for running applications [77, 33, 25] as a way to avoid trashing due to excessive transaction aborts. However, to the best of our knowledge, none of the past literature works provides a model coping with the problem of determining proactively whether to (re)assess the validity of a transaction along its life time in concurrency control mechanisms based on read-validation schemes.

In [28] a model-based proactive approach is exploited in order to determine whether to scale up/down the number of nodes of an in-memory transactional data store depending on workload changes. However, this proposal does not attempt to optimize performance via proactive checks of transactions' validity along their lifetime. The proposed model predicts performance for the scenario where transactions are aborted either when reaching their commit point, or when the concurrency control mechanism detects some inconsistent access to data. Previously issued accesses are not accounted for to proactively predict transaction validity, as instead we do in our proposal. On the other hand, the approach in [28] can be seen as orthogonal to ours. The former is mostly oriented to drive decisions on the amount of resources to be used to sustain a given workload, while our proposal is oriented to optimize the usage of each individual CPU-core by reducing CPU-waste.

In [36] the transaction validity check is triggered periodically, via an ad-hoc operating system support integrated within Linux. Differently from this proposal, our model-based read-validation scheme does not need special support from the operating system, thus being of wider portability. Also, the solution in [36] does not rely on any prediction scheme for choosing at which points to attempt transaction validation. The validation task is triggered independently of the probability that the transaction has become invalid. In our proposal, the validation task is triggered on the basis of predictions carried out by the analytical model.

Several works have targeted the reduction of the incidence of transaction aborts via heuristic based approaches [94, 32, 27]. These solutions either try to sequentialize conflicting transactions on the same thread or control the concurrency degree of the STM-based application by changing the number of threads/transactions that are allowed to run in parallel. A comprehensive survey of the proposed techniques can be found [80]. Other techniques have been oriented to the optimization of the

strategy for managing contention across concurrent transactions [93, 26]. Some of these approaches also enable the runtime adaptation of the contention management strategy to the workload profile [26]. The orthogonal issue of mapping threads to CPU-cores for performance optimization has been addressed in [18]. An approach aimed at reducing the waste of CPU-time and energy caused by transaction aborts, which acts with per-transaction granularity, is the one in [70]. Here the authors propose a solution for enabling a no longer valid transaction to be rolled back partially (rather than totally), which may help saving work otherwise doomed to be unfruitful. Our work is orthogonal to (and ideally combinable with) the above solutions, as our target is to select points along the transaction lifetime where a read-validation operation is likely to produce an early abort of the transaction. On the other hand, similarly to the proposal in [70], we retain the potential to reduce CPU-time waste on per-transaction granularit if our analytical model is exploited to trigger early transaction aborts that would have never be triggered by the underlying transactional layer otherwise.

## 4.6 Conclusions and future work

In this chapter we have presented a model for estimating the abort probability of a transaction maintaining a read-set of all the read-accessed transactional objects. Our proposal has applications to different concurrency control protocols which avoid read-locking data objects, such as optimistic and multi-version ones. Our model can be used to perform proactive validations of transactions, as a means to abort no longer valid transactions as fast as possible. We have shown the accuracy of our model for a synthetic benchmarks under different transactional read/write mixtures. We also evaluated the performance benefits that can be achieved by exploiting the model via a simple threshold-based mechanism to trigger proactive transaction validations whenever the estimated abort probability of a transaction exceeds a threshold. Thanks to this mechanism, we have observed up to 14% performance gain—on a per transaction-profile analysis—for a port of the TPC-C benchmark to an STM environment. As future work we plan to investigate more sophisticated strategies for exploiting the abort probability predictions by the model in combination with other (predicted) costs—such as the expected residual transaction execution time and the real cost of the validation operation as function of the read-set size. We also plan to investigate architectural solutions for reducing the overhead to manage the model at runtime, to further improve the performance of transactional applications.

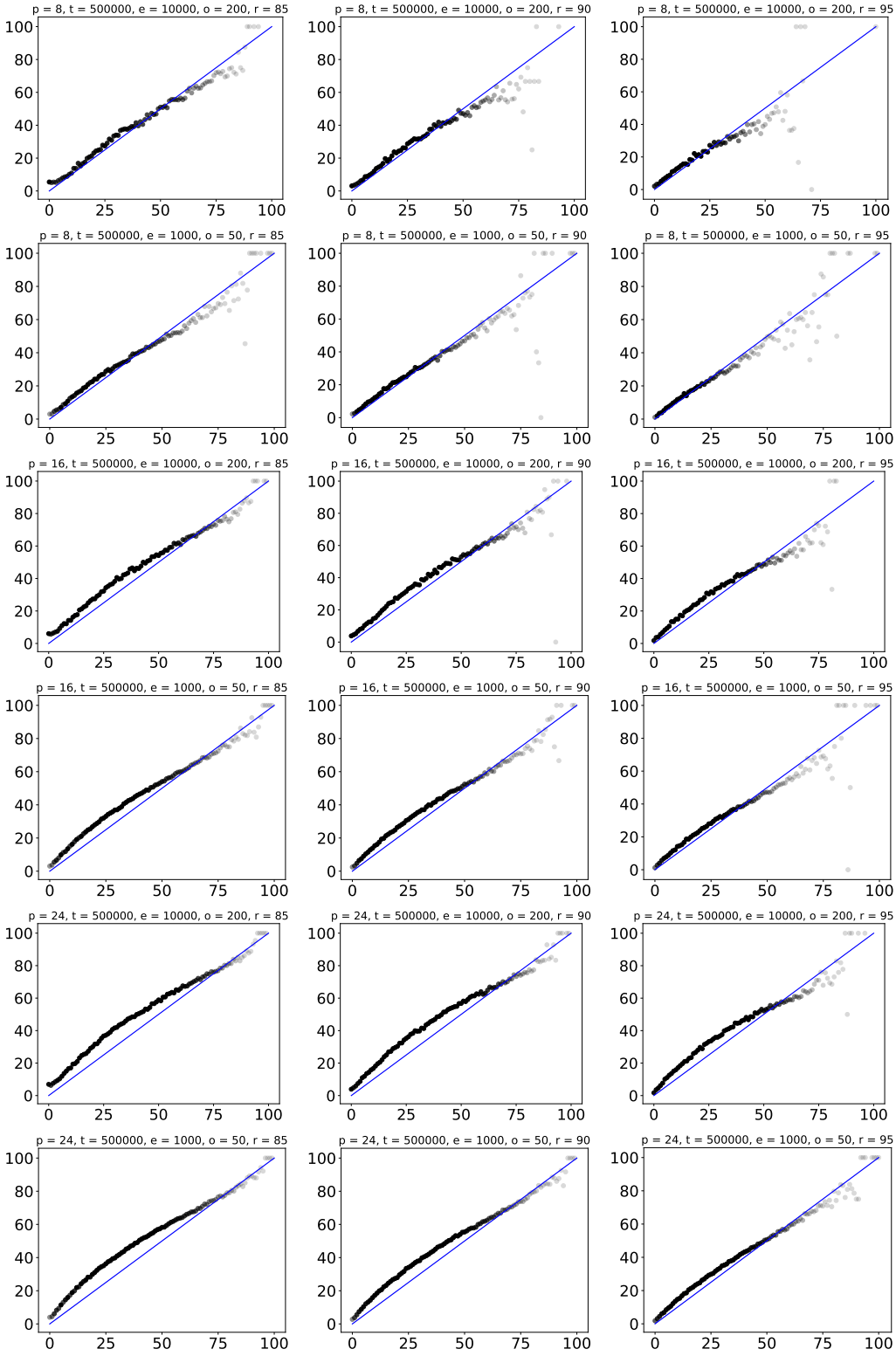



**Figure 4.2.** Model validation plots using a synthetic benchmark. $p$ is the number of threads, $t$ is the number of tranactions per thread, $e$ is the number of transactional objects that can be read or written, $o$ is the number of transactional operations per transaction, $r$ is the probability of performing a read operation at each transactional operation.

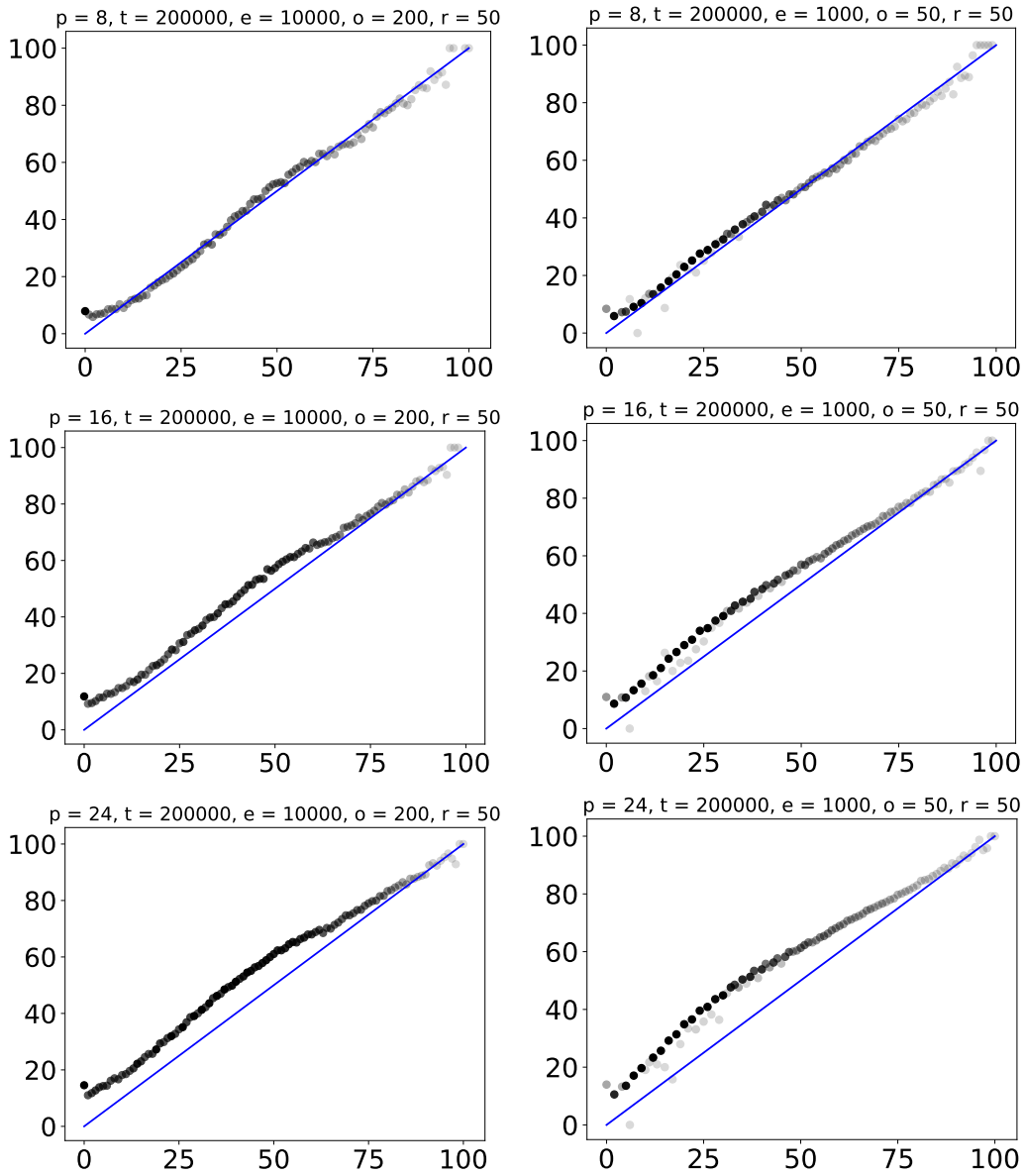



**Figure 4.3.** Model validation plots using a synthetic benchmark when $r = 50\%$. $p$ is the number of threads, $t$ is the number of tranactions per thread, $e$ is the number of transactional objects that can be read or written, $o$ is the number of transactional operations per transaction, $r$ is the probability of performing a read operation at each transactional operation.

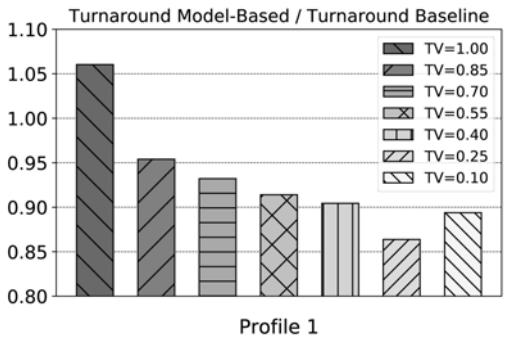



**Figure 4.4.** Turnaround results for profile #1 compared to the baseline.

CHAPTER 5

# Tracing task-based parallel applications

To adapt to the multi-core paradigm shift, several new ideas have been put into software in terms of parallel programming supports, ranging from APIs such as Pthreads [14] or Intel TBB [71] to first-class programming constructs in modern programming languages (e.g., Go [1]). In order to implement parallelization via these supports, applications need to be redesigned or ported to a different programming language with parallelization constructs. In some cases, the user is also responsible for how the parallelism is implemented. A direct consequence of this is that the effort of maintaining the source code increases and tasks like debugging, or testing, become more complex. Parallel programming models based on compiler directives, such as OpenMP, are an alternative to the above-mentioned approaches. These models allow the programmer to disclose both data and task parallelism within programs by means of source-code annotations. These annotations do not have an explicit effect in terms of the semantics of the program. They are instead interpreted by the compiler as hints to perform transformations that parallelize the code, while preserving sequential consistency. The annotation-based approach is very effective because it does not require to rewrite the program, or port it to a different language. In fact, it allows users to parallelize applications incrementally. Starting from the sequential version, additional portions of code can be annotated to specify parallelism. This has an important impact on the productivity that can be achieved thanks to this philosophy, with benefits in terms of programmability, composability, and reusability of code.

A widely-adopted abstraction for parallel programming is the task-based one. A program is represented as an acyclic graph where nodes are tasks and directed edges are constraints between them. A task can only be executed if all its constraints are satisfied, i.e., all the tasks to which it is connected via incoming edges have completed. These constraints establish the partial order of the program. If there are two or more tasks in the graph for which this is true, they can be executed in parallel without affecting the correctness of the program. Compared to data-parallel programming models, where the same operations are applied in parallel to a set of data items

(e.g., MapReduce, SIMD), programming models for task-based parallelization can support a wider set of use cases. Many scientific applications employ algorithms and data structures that are not embarrassingly parallel, hence data parallel models are inadequate. Unfortunately, besides the high potential of task-based programming models, the parallelization process remains manual and prone to errors by the user. This is even more relevant for task-based parallel programming models that use source-code annotations to mark task code and express execution constraints between them. Indeed, the process of annotating code for the sake of parallelization can lead to mistakes that would undermine the potential for parallelism due to performance and correctness problems. Because of this, it cannot be considered a trusted process in general. Wrong annotations can lead to synchronization issues and many hours of bug-haunting, thus forcing developers to debug their programs in conventional (and typically less effective) ways to try to get to the root cause of the problem.

In this chapter, we present a tracing technique to detect parallelization issues coming from wrong or missing annotations in a fictitious programming model which shares many properties with real-world models used in the industry. Our reference parallel programming model uses tasks to mark parallel code regions and dependencies between tasks to synchronize their execution. It also uses task nesting to define multiple levels of parallelism at different scales. An important property of our model is the capability to prove the correctness of applications by means of structural induction: the correctness of each task depends on local task analysis and on the correctness of the tasks it creates. To improve the effectiveness of our tracing technique, we introduce some ad-hoc verification annotations which can be placed manually or automatically to disable code analysis inside well-defined regions of code or when third-party libraries, such as BLAS or MPI, are invoked. An experimental implementation of this technique using dynamic binary instrumentation has been evaluated for the OmpSs-2 programming model against a set of sample kernels and real-world applications. Our results suggest that the synergistic exploitation of run-time analysis and verification annotations can be successfully used to verify complex applications that rely on third-party libraries and task-nesting.

Overall, the work presented in this chapter involves the Selection, Extraction, and Analysis parts of the tracing process illustrated in Chapter 2. The original contributions of this work can be summed up as follows:

- We describe a reference task-based programming model with task dependencies and task nesting for which a definition of program correctness can be derived.
- We show a way to prove program correctness via structural induction and local task analysis, to analyse each task in the program separately.
- We introduce some new verification annotations to improve the accuracy and efficiency of our tracing technique.

## 5.1 Baseline formalisms

In this section we give an overview of the task-based programming model used as a reference in the rest of this chapter. In later sections, we proceed to explain how

structural induction can be used together with local task analysis to leverage the task nesting feature of the model and prove program compliance by inductively proving correctness for each task.

### 5.1.1 The TMP programming model

Our reference task-based programming model, called TMP (Task Multi-Processing) is based on placing annotations in the source code to, starting from a sequential program, produce a parallel version of it. As already mentioned, our model takes inspiration from other existing programming models like OpenMP, an API for shared memory multiprocessing programming in C, C++, and Fortran, and OmpSs-2, a programming model developed at Barcelona Supercomputing Center (BSC) and similar in spirit to OpenMP. In the rest of this section we explain the TMP programming model in further detail. Throughout the section we also introduce some class of errors, denoted with the 'E' prefix, that can occur in TMP programs and which can affect the correctness of the parallelization process.

#### Task creation

TMP allows the expression of parallelism through tasks. Tasks are independent pieces of code that can be executed by the parallel resources at run-time. Whenever the program flow reaches a section of code that has been declared as a task, instead of executing the task code, the program will create an instance of the task and will delegate its execution to the TMP run-time environment. The TMP run-time will eventually execute the task on a parallel resource. Therefore, in order to parallelize a program, the user has to specify what is a task. The TMP runtime system creates a team of threads when starting the user program execution. This team of threads is is composed by a single master thread and several additional workers threads. The *master thread*, also called the initial thread, executes sequentially the user program in the context of an implicit task region called the *initial task* (surrounding the whole program). Meanwhile, all the other additional worker threads will wait until concurrent tasks are available to be executed. Depending on a thread's availability to perform work, the execution of a task is assigned to one of the threads in the team. Whenever a thread encounters a `tmp task` directive, it instantiates a new task and resumes execution after the task-creation construct. The task instance can be executed either by that same thread at some other time or by another thread. Thus, execution of the new task could be immediate or deferred, according to task scheduling constraints and thread availability. Threads are allowed to suspend the current task region at a *task scheduling point* in order to execute a different task. Task scheduling points may occur at the following locations: (a) when creating a task; (b) in a `barrier` directive (see later); (c) just after the completion of a task. Any directive that defines a task or a series of tasks can only appear within code belonging to another task, thus defining multiple levels of parallelism and implementing task nesting (see Listing 5.1). The underlying TMP run-time environment can exploit factors like data or temporal locality between tasks to improve the performance of applications with task nesting. Supporting multi-level parallelism is also required to allow the implementation of recursive algorithms.

```
#pragma tmp task label(T1)
{
  ...

  #pragma tmp task label(T2)
  ...
}
```

**Listing 5.1.** Creating tasks in TMP

### Task dependencies

TMP tasks require data in order to do meaningful computation. Usually a task will use some input data to perform some operations and produce new results that can be later on be used by other tasks or parts of the program. These data references can be declared via the `in`, `out`, or `inout` clause, meaning respectively that a task will use them as an input, an output, or both. The set of all data references constitutes the *dataset* of a task. Each time a new task is created, the dataset is matched against those of previously-created tasks to produce execution-order constraints between them. We call these constraints *dependencies*. This process creates a *task dependency graph* at runtime that guarantees a correct order of execution for the application. There are three possible dependencies: Read-after-Write (RaW), Write-after-Write (WaW) or Write-after-Read(WaR). Tasks are scheduled for execution as soon as all their predecessor in the graph have finished, or at creation if they have no predecessors. Listing 5.2 illustrates an example of tasks with dependencies.

```
#pragma tmp task in(a) out(b) inout(c)
{
  c += a;
  b = c;
}
```

**Listing 5.2.** Creating dependencies in TMP

Note that whether the task really uses that data in the specified way its the programmer responsibility.

**E1** (No matching dataset entry for an access)**.** *It is it is a mistake to access a shared object from within a task if that object is not in the task dataset. In the example, since* ***x*** *is not in the dataset of the task, it won't generate any dependency. Therefore, if there is another task accessing the same variable the two tasks would generate a data race on* ***x****. This would happen regardless of whether the second task has or not a dependency on* ***x****.*

```
int x = 10;

#pragma tmp task
{
  x++;
}
```

By default, every task in TMP can only be connected via dependencies to a limited set of other tasks, which are those in the same *dependency domain*. In

TMP, the dependency domain of a task is the set of sibling tasks. Therefore, the TMP runtime system will establish input/output dependencies between two tasks if their datasets match and they are siblings. To connect two tasks $t$ and $t'$ which are not siblings, but have a common ancestor $t_a$, we can pass the dataset of both $t$ and $t'$ upwards to all their intermediate ancestors until we reach $t_a$. By doing this, the runtime will merge the dependency domain of all tasks from $t$ to $t_a$, and from $t'$ to $t_a$, thus being able to establish a dependency between $t$ and $t'$. This mechanism guarantees synchronization between tasks in different domains and in different hierarchies, However, it is inefficient because augmenting the dataset of a task may defer its execution unnecessarily. The entries that were added to the datasets of tasks at each intermediate nesting level only serve as a mechanism to connect different dependency domains. In this sense, allowing them to defer the execution of the intermediate tasks is unnecessary, since these tasks don't actually perform any conflicting accesses by itself. After all, users are only interested in linking the dependency domains for these tasks, without affecting their execution.

To this extent, the dependency model in TMP supports *weak dependencies*. These are created via the `weakin`, `weakout`, and `weakinout` clauses. Their purpose is to inform the runtime that some descendant of the current task is accessing the data elements specified in the `weak` variant. Weak dependencies do not imply a direct dependency, and thus do not defer the execution of tasks. Their purpose is to serve as linking point between the dependency domains of different tasks in different hierarchies. Listing 5.3 provides an example of weak dependencies. T1.1 and T2 live in different dependency domains. However, the runtime is able to establish a WaR dependency between them because T1 declares the access to $b$ via the `weak` variant. Notice that no synchronization is necessary between T1 and T2 because T1 by itself does not perform any action that requires the enforcement of the dependency. Those actions will be performed by T1.1, or a deeply nested subtask.

```
1 #pragma tmp task in(a) weakout(b) label(T1)
2 {
3   ...
4   #pragma tmp task out(b) label(T1.1)
5   ...
6 }
7
8 #pragma tmp task in(b) weakout(c) label(T2)
9 {
10  ...
11  #pragma tmp task out(b) label(T2.1)
12  ...
13 }
```

**Listing 5.3.** Connecting tasks via weak dependencies in TMP

This mechanism is transparent to the user, but it can't be unintuitive at times.

**E2** (No matching (weak) dataset entry in the parent task)**.** *Failing to specify a (weak) dataset entry in the parent task is a mistake if the current task accesses a shared object or, has itself a weak reference to it. This is an error because the model states that if dependency domains are not properly connected, accesses to the same object in different domains will not be synchronized. In the example, T1.1 is*

*accessing x directly, while T2.1 has a weak reference to it. However, neither T1 nor T2 have a weak reference to x. T1 should declare a* `(weak)in(x)`*, while T2 should declare a* `(weak)out(x)`*. Failing to do so prevents the TMP runtime from creating an execution constrain between T1.1 and the descendants of T2.1 which access x.*

```
int x = 10;

#pragma tmp task label(T1)
{
  #pragma tmp task in(x) label(T1.1)
  ...
}

#pragma tmp task label(T2)
{
  #pragma tmp task weakout(x) label(T2.1)
  ...
}
```

### Task barriers

So far we have illustrated the mechanisms that are used in TMP to transparently synchronize two different tasks that will otherwise conflict on their accesses to a shared object. Strong dependencies can be used if the task $t$ is directly accessing an element, as a way to synchronize the task itself with other tasks that share the same dependency domain. Weak dependencies are used in $t$ if some (deeply nested) subtask $t'$ is accessing that element, to make sure that it will synchronize with tasks that can only be reached through the dependency domains of $t$ or its ancestors.

However, to synchronize accesses between a task $t$ and one of its descendants $t'$ we need to explicitly set synchronization points, as the dependency system cannot create execution-order constraints between $t$ and $t'$ by only looking at their datasets. These synchronization points are defined using the `barrier` directive, and can be specialized by adding a dataset to it. When a dataset is present, the TMP run-time will look at the tasks in the same dependency domain to establish dependencies between the barrier and other (descendant of $t$) tasks. If dependencies are formed, the barrier will wait until its dependencies can be satisfied. Notice that for this mechanism to work as expected, the dependency domains of the descendant tasks need to be properly linked to that of the barrier via at least weak dependencies (see previous section). In the absence of a dataset, the barrier will wait until all descendants of $t$ have completed. In Listing 5.4, the first assert won't be executed until all previous descendant tasks writing on `x` have terminated. The second task might not have terminated at this point. However, by the time the second assert is executed, all previous descendant tasks must have completed, included the second. Using barriers, proper synchronization can be guaranteed between T1 and its three descendant T1.1, T1.1.1, and T1.2.

```
#pragma tmp task label(T1)
{
  int x = 0, y = 2;

  #pragma tmp task weakinout(x) label(T1.1)
```

```
6   {
7     ...
8
9     #pragma tmp task inout(x) label(T1.1.1)
10    x++;
11  }
12
13  #pragma tmp task in(x) inout(y) label(T1.2)
14  y -= x;
15
16  #pragma tmp barrier in(x)
17  assert(x == 1);
18
19  #pragma tmp barrier
20  assert(x == y);
21 }
```

**Listing 5.4.** An example of the different `taskwait` constructs in TMP

Placing `barrier` at appropriate points in code can be very difficult at times and leads to a single class of errors which however can be very difficult to debug by hand.

**E3** (No barrier between an access and previous tasks)**.** *Not protecting an access to an object that is shared with a previously-created child task is a mistake in TMP. Every object that is also accessed by previous child tasks must be protected by a barrier. More precisely, there must be a barrier between the access to the object and the previous child tasks that were created after the last barrier without a dataset, or that were not affected by any barrier with a dataset after their creation. In the example, there a missing* `barrier` *(no dataset) or* `barrier in(x)` *directive before the write to y in task T1. Its absence may lead to a lack of dependencies between it and the descendant task T1.1.1, because the accesses to* `x` *are not synchronized as the two tasks can run in parallel.*

```
1 int x = 10;
2
3 #pragma tmp task inout(x) label(T1)
4 {
5   int y;
6   x = 5;
7
8   #pragma tmp task weakinout(x) label(T1.1)
9   {
10    ...
11    #pragma tmp task inout(x) label(T1.1.1)
12    x++;
13  }
14
15  y = x + 6;
16 }
```

## 5.2 Local task analysis

Table 5.1 provides a compact lists of the errors that were discussed in Section 5.1, grouped according to which elements are checked to detect these errors: accesses

| E | Description |
|---|---|
| *G1. Compare accesses and datasets of the same task* | |
| *E*1 | No matching dataset entry for an access |
| *G2. Compare datasets of two adjacent task in the hierarchy* | |
| *E*2 | No matching weak dataset entry in the parent task |
| *G3. Compare accesses of a task with datasets of child tasks* | |
| *E*3 | No barrier between an access and previous tasks |

**Table 5.1.** Programming errors in TMP

with dependencies (G1 and G2), and dependencies with dependencies (G3). These comparisons are only made within the same task (G1), a task and its child tasks (G2 and G3), or a task and its parent task (G3). It is never necessary to compare accesses from a task with accesses of another task, or access/dependencies from a task to dependencies of a strict ancestor/descendant of that task. To check that a task is free of errors in Table 5.1, we only need to look at what happens within the code of task itself, the annotations of its parent (if any), and the annotations of its children (if any). We call *local task analysis* the kind of processing we carry out within our Analysis step because such analysis doesn't need to reason globally, i.e., at the level of the whole program.

To understand how local task analysis works, let's consider a task $t$ in the program. Let $t_p$ be its parent task, and $t_c$ be a child task. Let $clk_t$ be the time at which $t$ was created by $t_p$. Let $d_{(t,i)}$ be the $i$-th dataset entry of $t$, defined as a tuple $\langle m, weak, I\rangle$, where $m \in \{r, w\}$ is the access mode (read or write), $weak \in \{\top, \bot\}$ tells if the entry is weak or not, and $I$ is a sequence of intervals of type $i = \langle l, h\rangle$, where $l$ is the lowest non-contiguous address since the last interval (if any), and $h$ is the highest contiguous address before the next interval (if any). Let $\mathcal{D}_t$ be the sequence of all dependencies of $t$. Let $a_{(t,j)}$ be the $j$-th memory access performed by $t$, defined as a tuple $\langle m, clk, i\rangle$ with $i = \langle l, h\rangle$ and $clk$ being the time at which the access was performed. Let $\mathcal{A}_t$ be the sequence of all accesses of $t$ (also called the *access-set* of $t$). Finally, let $\mathcal{B}_t$ be the set of barriers of task $t$. Each entry $b_{(t,k)}$ is a tuple $\langle m, clk, I\rangle$, where $clk$ is the time at which the barrier was created.

To detect errors of type E1, we verify if there is at least one access performed by $t$ that doesn't have a corresponding dataset entry.

**Condition 1** (E1 detection)**.** *To detect errors of type E1, we check if, for each $a_{(t,j)} \in \mathcal{A}_t$, there it at least one $d_{(t,i)} \in \mathcal{D}_t$ for which:*

- $a_{(t,j)}.i \subseteq d_{(t,i)}.I$*, and*
- $d_{(t,i)}.m = a_{(t,j)}.m$

*If there is any $a_{(t,j)}$ for which no $d_{(t,i)}$ satisfies the above conditions, then $t$ is affected by E1.*

To detect errors of type E2, we verify that there is at least one dataset entry of $t$ (weak or not) that doesn't have a corresponding dataset entry in its parent (at least weak).

**Condition 2** (E2 detection)**.** *To detect errors of type E2, we check if, for each* $d_{(t,i)} \in \mathcal{D}_t$*, there is at least one* $d_{(t_p,i_p)} \in \mathcal{D}_{t_p}$ *for which:*

- $d_{(t,i)}.I \subseteq d_{(t_p,i_p)}.I$*, and*
- $d_{(t,i)}.m = d_{(t_p,i_p)}.m$

*If there is any* $d_{(t,i)}$ *for which no* $d_{(t_p,i_p)}$ *satisfies the above conditions, then* $t$ *is affected by E2.*

To detect errors of type E3, we verify that there is at least one access performed by $t$ that is not protected by a barrier. The access must have a corresponding dataset entry in one of the previously-created child tasks, such that at least one amongst the access and the dataset entry is a write. The barrier must be missing between the access and the creation of such child task.

**Condition 3** (E3 detection)**.** *To detect errors of type E3, we check if, for each* $a_{(t,j)} \in \mathcal{A}_t$*, there is at least one* $d_{(t_c,i_c)} \in \bigcup_{t_c} \mathcal{D}_{t_c}$ *for which:*

- $a_{(t,j)}.i \subseteq d_{(t_c,i_c)}.I$*, and*
- $a_{(t,i)}.m = d_{(t_c,i_c)}.m$*, and*
- $a_{(t,i)}.clk > clk_{t_c}$*, and*
- *there is no* $b_{(t,k)} \in \mathcal{B}_t$ *for which:*
  - $a_{(t,i)}.m = b_{(t,k)}.m$*, and*
  - $b_{(t,k)}.clk < a_{(t,i)}.clk$*, and*
  - $b_{(t,k)}.clk > clk_{t_c}$*, and*
  - $b_{(t,k)}.I \subseteq d_{(t_c,i_c)}.I$*, and*
  - $b_{(t,k)}.m = w$*, or* $d_{(t_c,i_c)}.m = w$*, or both.*

*If there is any* $a_{(t,j)}$ *for which it is true, then* $t$ *is affected by E3.*

Conditions 1 to 3 give us a way to detect the errors in Table 5.1. However, to translate them into concrete algorithms, we need to convert them into algorithms, and the mathematical objects they use into concrete data structures. Section 5.4 provides an experimental implementation of local task analysis based on binary search trees and operations over them.

### 5.2.1 Verification annotations

As explained in the previous section, detecting parallelization errors requires checking, among other things, the memory accesses of each task with the dataset entries of the same task, or those of other tasks. Notice that this is done for every memory accesses performed within the task, including those accesses coming from third-party or standard libraries. While this might be desirable for some libraries, it may pose accuracy and performance problems in other cases. Tasks might include

code which accesses meta-data that are not interesting from the point of task-based parallelization. For example, some code may use ad-hoc techniques for synchronization that are opaque to TMP (e.g., spinlocks, HTM, RMW instructions, etc...). Every memory access performed within such explicit synchronization code would be detected by our tools as a potential candidate for parallelization errors. Additionally, it might slow down the local task analysis by orders of magnitude. In Listing 5.9, the task is performing an I/O operation which internally may coordinate for the use of a shared I/O buffer via additional memory operations that aren't relevant.

```
#pragma oss task
{
  int x = 5;

  printf("%d", x);
  ...
}
```

Similarly, some code which is relevant for tracing might be quite simple to understand in terms of its interaction with memory, but have an arbitrarily complex structure. Regardless of the concrete number of memory accesses which are performed (and memory addresses which are touched) its semantics could be described very precisely in a succinct way. However, the price to detect such patterns would be function of the complexity of the code, rather than its semantics. In Listing 5.2.1 a task is performing $TS^2$ accesses to the same number of cells of a matrix, but the final effect is that of traversing a well-defined contiguous region for writing.

```
double A[N/TS][M/TS][TS][TS];
...
for (long ii = 0; ii < TS; ii++)
  for (long jj = 0; jj < TS; jj++)
    A[i][j][ii][jj] = value;
```

Tasks might additionally make use of code that internally use synchronization directives and perform accesses to irrelevant meta-data together with relevant accesses, according to some well-defined semantics. In Listing 5.2.1 a task is performing two MPI operations whose visible effects are just those of reading some bytes from a send buffer and writing an equivalent number of bytes to a receive buffer.

```
MPI_Send(sendbuf, size, MPI_BYTE, dst,
  block_id+10, MPI_COMM_WORLD);
MPI_Recv(recvbuf, size, MPI_BYTE, src,
  block_id+10, MPI_COMM_WORLD,
  MPI_STATUS_IGNORE);
```

What is common to all these examples is a given piece of code whose semantics can be confusing for our local task analysis. Fortunately, the quality of our local task analysis in terms of tracing overhead and accuracy can be reduced by user interventions on the source code. In this section, we introduce two new TMP annotations to assist with the task of checking the parallelization of code: (1) the `lint` directive, which accepts three keywords to define data references appearing within the related code: `in`, `out`, and `inout`; and (2) the `verified` clause optional in the `task` construct. Overall, the purpose of both features is the same: pause the analysis for the targeted code, although the accesses declared via the data references

of the `lint` or `task` pragma are registered and taken for granted for the wrapped code. Intercepting these directives while tracing the application therefore affects both the SELECTION and EXTRACTION steps of our tracing methodology, as we can both filter irrelevant accesses performed within the ignored regions, and extract the only relevant information about them—that is, data references information. In the remaining of this section, we proceed to explain the new annotations in more details.

### The `lint` pragma

The `lint` pragma can be put inside task code to avoid analysing the code wrapped by the directive. This is useful to mark code that has no effect on the dataset of a task. For example, we can mask library functions performing I/O operations, or exclude initialization and finalization code in a task that is known to only access private memory. The pragma also allows to specify which memory operations are performed within a marked region of code, and to which memory addresses. The pragma accepts three main keywords: `in`, `out`, `inout`. They are equivalent to those specified for a task dataset and allow the user to state which are the shared object read and/or written within the wrapped code. The local task analysis can use these data-references as a hint of the accesses that were performed within the code, so that it doesn't have to derive them by itself. This brings many benefits in terms of accuracy and performance of the analysis. Consider the four cases below:

```
1  #pragma tmp lint inout(m[0:n])
2  x = malloc(n * sizeof(int));
3
4  #pragma tmp lint out(m[0:n])
5  free(x);
6
7  #pragma tmp lint in(sendbuf[0:size]) out(recvbuf[0:size])
8  {
9    MPI_Send(sendbuf, size, MPI_BYTE, dst,
10     block_id+10, MPI_COMM_WORLD);
11   MPI_Recv(recvbuf, size, MPI_BYTE, src,
12     block_id+10, MPI_COMM_WORLD,
13     MPI_STATUS_IGNORE);
14 }
15
16 double A[N/TS][M/TS][TS][TS];
17
18 #pragma tmp lint out(A[i][j])
19 for (long ii = 0; ii < TS; ii++)
20   for (long jj = 0; jj < TS; jj++)
21     A[i][j][ii][jj] = value;
```

**Listing 5.5.** Four different `tmp lint` use cases

In the first case, we are using the `oss lint` pragma to instruct the analysis that an allocation has been performed. At the same time, we are also disabling the analysis within the `malloc` function, which is probably a good idea given that its implementation can be arbitrarily complex and can involve accesses to shared objects (e.g., locks) defined within the library that implements it. The second case is analogous to the first, but memory is released. Analysis is also disabled, so that whatever happens within the call to `free` cannot be seen. The third example

describes a very common case in which we are using some API functions to read and/or write buffers in memory via I/O operations. It is what happens, for example, with the `MPI_Send` and `MPI_Recv` functions. According to the textual description of these functions, and regardless of their implementation, they are respectively reading/writing $N$ bytes from/to memory. Not only the implementation of these functions can be arbitrary complex so as to slow down the analysis by orders of magnitude, but it can also affect negatively the accuracy of the generated report. Indeed, the synchronization mechanisms used within these functions are independent of the TMP execution model and, as such, may require to access objects that aren't (and shouldn't be) declared as dependencies in the task where these functions are invoked. The fourth case is that of nested loops where the final effect is that of traversing a well-defined portion of an array for reading and/or writing. By marking this code with the pragma and summarizing its behaviour we are saving the cost of analysing a number of accesses that is proportional to the number of iterations. The analysis would still correctly detect all the accesses performed within the loop and there wouldn't be any accuracy concern, but the overall execution time would be much greater due to the cost of analysis.

#### The `verified` attribute

Users are also provided another way to disable analysis at the level of whole tasks, via the `verified` keyword that can be passed to the `oss task` pragma. Conceptually, the effect of the `verified` flag is equivalent to wrapping the entire task code within an `lint` pragma, specifying as dataset the same data-references used for the `task` pragma. Performance wise, the `verified` attribute is equivalent to wrapping the entirety of task code with a `lint` pragma, because analysis is disabled for the whole task. The `verified` task also comes with an optional expression that is evaluated at run-time to decide whether memory tracing for the task will be disabled or not. This expression can be used to conditionally evaluate tasks that are more likely to be subject to programming errors—for example, task that are related to boundary loop iterations. Additionally, it can be used as a way to implement task-level sampling and reduce the overall memory tracing overhead of the application (e.g., instrument one out of $N$ task instances).

```
1 for (int i = 0; i < N; ++i)
2   #pragma tmp task verified(i % M)
3     ...
```

**Listing 5.6.** An example of the `verified` attribute

## 5.3 Proving program-level correctness

Local task analysis and task nesting can be exploited to prove the correctness of the whole program by only looking at what happens within each task individually. To understand why, let's consider a TMP program $p$ and a task $t$ in it. We call $t$ *TMP-compliant* if it is free of errors in Table 5.1 and if, for every $t_c$ who is child of $t$, $t_c$ is *TMP-compliant*. If $t_0$ is the initial task in the program, and it is TMP-compliant, then $p$ is TMP-compliant. The notion of TMP-compliance is therefore a

combination of local task analysis and task nesting: the former is used to check the presence of errors in the task itself; the latter is used to inductively reason about increasing portions of the code in the program, until we reach the initial task which represents the whole program. Local task analysis and task-level TMP-compliance have a direct implication on the correctness of the task. To understand why, let's consider a TMP program $p$ and a task $t$ in it. We call a *conflict* the absence of synchronization between two tasks $t$ and $t''$ which may generate a synchronization problem at run-time (e.g., a data race). If $t$ and $t''$ cannot conflict, they are said to be *conflict-free* (CF). We want to prove that if a task $t$ is TMP-compliant, then no two tasks $t'$ and $t''$ which are descendant of $t$ can conflict. If that is the case, $t$ is *internally conflict-free*. In the rest of this section, we prove that TMP-compliance is a sufficient condition for internal conflict-freedom, meaning that if a task $t$ is TMP-compliant, then its descendant tasks can only race with tasks that are created *outside* of $t$. To prove this fact, we proceed by considering pairs of tasks $t$ and $t'$ in three different cases: (a) when they live in the same dependency domain; (b) when they live in different dependency domains and have a strict common ancestor; (b) when they live in different dependency domains and one is a descendant of the other.

**Theorem 1** (CF1)**.** *Let $t$ and $t'$ be in the same dependency domain. If they are not affected by E1, then they are conflict-free.*

*Proof.* If the two tasks are not affected by E1, it means that every access in the access-set of $t$ and $t'$ is contained in the respective datasets. If the access-sets of $t$ and $t'$ don't intersect, the datasets of $t$ and $t'$ won't, and there won't be any conflict. If they do intersect, there will be an intersection in the datasets of $t$ and $t'$ which will generate a dependency, because the two tasks are in the same dependency domain. Therefore, there won't be any conflict. □

Condition CF1 is sufficient to prevent conflicts between the two tasks, if $t'$ and $t$ are in the same dependency domain, because the model guarantees that they will be able to form dependencies, if necessary. Notice that we don't need to compare the accesses of $t$ and $t'$ directly because the local task analyses on $t$ and $t'$ compose to imply conflict-freedom between the two tasks. The opposite is not true: conflict-freedom between the two tasks doesn't imply that $t$ and $t'$ are exempt of those errors E1. The absence of these errors is a sufficient but not necessary condition. In Listing 5.7, the second task won't be created until the first has completed. T1 and T2 are both affected by E1, but the program is conflict-free.

```
1 #pragma tmp task label(T1)
2 {
3   x++;
4 }
5
6 #pragma tmp barrier
7
8 #pragma tmp task label(T2)
9 {
10  x--;
11 }
```

**Listing 5.7.** CF1 is not a necessary condition for conflict-freedom

**Theorem 2** (CF2)**.** *Let $t$ and $t'$ be two tasks in different dependency domains with a common strict ancestor $t_a$. Let $\alpha(t)$ and $\alpha(t')$ be the sequences of ancestors of $t$ and $t'$ up to $t_a$ (excluded). Let $\widehat{t}$ and $\widehat{t'}$ be respectively the ancestors in $\alpha(t)$ and $\alpha(t')$ which are direct children of $t_a$. If $t$ and $t'$ are not affected by E1, and $t \cup \alpha(t) \setminus \widehat{t}$ and $t' \cup \alpha(t') \setminus \widehat{t'}$ are not affected by E2, then $t$ and $t'$ are conflict-free.*

*Proof.* If the tasks $t \cup \alpha(t) \setminus \widehat{t}$ and $t' \cup \alpha(t') \setminus \widehat{t'}$ are not affected by E2, then it means that the tasks $\alpha(t)$ and $\alpha(t')$ declare (at least weak) dataset entries for every entry respectively in $t$ and $t'$. This means that the tasks $\widehat{t}$ and $\widehat{t'}$, which are in the same dependency domain, will be able to form (at least weak) dependencies between them, thus affecting the execution of their descendent tasks up to $t$ and $t'$. If the two tasks $t$ and $t'$ are not affected by E1, it means that every access in the access-set of $t$ and $t'$ is contained in their respective datasets. If the access-sets of $t$ and $t'$ don't intersect, the datasets of $t$ and $t'$ won't, and there won't be any conflict. If they do intersect, there will be an intersection in the datasets of $t$ and $t'$ which will generate an execution-order constraint at the level of tasks $\widehat{t}$ and $\widehat{t'}$, which are in the same dependency domain. Therefore, there won't be any conflict. □

To evaluate CF2 we don't need to check the accesses of $t$ and $t'$ directly, because we are composing the local task analyses carried out on the two tasks and their ancestors separately. Once again, conflict-freedom between the two tasks doesn't imply that $t$ and $t'$ are free of errors E1, nor that they and their ancestors are not affected by E2. In Listing 5.8, T1 is affected by E2 and T2 is affected by E1, but the overall program is conflict-free.

```
1 #pragma tmp task in(a) label(T1)
2 {
3   ...
4   #pragma tmp task out(b) label(T1.1)
5   ...
6
7   #pragma tmp taskwait
8 }
9
10 #pragma tmp task label(T2){
11   b += 1
12 }
```

**Listing 5.8.** CF2 is not a necessary condition for conflict-freedom

**Theorem 3.** *Let $t$ and $t'$ be two tasks such that $t'$ is a descendant of $t$ and $\alpha(t')$ are the ancestors of $t'$ until $t$ excluded. Let $\widehat{t'}$ be the ancestor in $\alpha(t')$ which is a direct child of $t$. If $t$ is not affected by E3, $t' \cup \alpha(t') \setminus \widehat{t'}$ are not affected by E2, and $t'$ is not affected by E1, then $t$ and $t'$ are conflict-free.*

*Proof.* If $t$ is not affected by E3, it means that every access from $t$ which intersects with the dataset of $\widehat{t'}$ is guarded by a barrier, whose dataset contains the conflicting accesses. If tasks in $t' \cup \alpha(t') \setminus \widehat{t'}$ are not affected by E2, then it means that the tasks $\alpha(t')$ declare (at least weak) dataset entries for every dataset entry in $t'$. This means that the task $\widehat{t'}$ and the barrier, which are in the same dependency domain, will be able to form (at least weak) dependencies between them, thus affecting the

execution of their descendent tasks up to $t'$. If $t'$ is not affected by E1, it means that every access in the access-set of $t'$ is contained in its dataset. If the accesses in $t$ and the access-sets of $t'$ intersect, there will be an intersection in the datasets of the barrier and $t'$ which will generate an execution-order constraint at the level of the barrier and $\widehat{t'}$, which are in the same dependency domain. Therefore, there won't be any conflict. This means that $t$ can't be affected by E3. □

CF3 doesn't require to check the accesses of $t$ and $t'$ directly because the local task analyses can be composed. However, conflict-freedom between the two tasks doesn't imply that $t$ is free of error E3, nor that $t' \cup \alpha(t') \setminus \widehat{t'}$ are free of errors E2, nor that $t'$ is free of errors E1. In Listing 5.9, T1.1 is affected by E2 as there is no (weak) reference to $x$. T1 is affected by E3 because it relies on an external synchronization mechanism (e.g., a monitor) to coordinate with T1.1.1. Nevertheless, T1 and T1.1.1 are conflict-free (excluding the meta-data used to achieve proper synchronization).

```
1  synchr_variable_t sv;
2
3  #pragma tmp task label(T1)
4  {
5    #pragma tmp task label(T1.1)
6    {
7      #pragma tmp task label(T1.1.1) out(x)
8      {
9        x--;
10       sv_release(sv);
11     }
12   }
13
14   sv_wait(sv);
15   x++;
16 }
```

**Listing 5.9.** CF3 is not a necessary condition for conflict-freedom

Let's now consider a task $t$ for which we claim TMP-compliance. We can prove that $t$ is internally conflict-free.

**Theorem 4.** *Let $t$ be $t$ be a TMP-compliant task. Then, there cannot be two tasks $t'$ and $t''$ in $t$ that conflict, meaning that $t$ is internally conflict-free.*

*Proof.* If $t$ is TMP-compliant, it means that it is free of errors E1, E2, and E3, and that every descendant $t'$ of $t$ is TMP-compliant. We can use conditions CF1, CF2, and CF3 to prove the theorem in an incremental manner:

1. The fact that each $t'$ is free of errors E1, by CF1, excludes conflicts with any tasks $t''$ in the same dependency domain.

2. The fact that each $t'$ is also free of errors E2, by CF2, excludes conflicts with any task $t''$ that is not in the same dependency domain, but for which there is a common ancestor $t_a$ that is in $t$ or coincides with $t$.

3. The fact that each $t'$ is also free of errors E3, by CF3, excludes conflicts with any task $t''$ that is a descendant of $t'$.

Additionally, the fact that $t$ is TMP-compliant implies that $t$ is free of errors E3, meaning that it can't conflict with any of its descendants $t'$. This is because by CF3, if such tasks $t'$ are TMP-compliant, they are free of errors E1, E2, and 3, hence there can't be conflicts.

□

We can now prove that a program $p$ which is TMP-compliant is also internally conflict-free, hence no task $t$ in it can conflict with any other task $t'$.

**Theorem 5.** *Let $p$ be a TMP-compliant program. Then, $p$ is conflict-free.*

*Proof.* If $p$ is TMP-compliant, then the initial task $t_0$ is TMP-compliant, meaning that it is also internally conflict-free. Since there are no other tasks outside of $t_0$, internal conflict-freedom is sufficient to prove that $p$ is conflict-free. □

In general, a program $p$ doesn't need to be TMP-compliant to be conflict-free. In fact, the examples in Listings 5.7 to 5.9 can be used as building block to provide a working counter-example.

## 5.4 Experimental assessment

In this section we discuss an experimental implementation of our analysis. Our objective is to demonstrate that it can be effectively used to evaluate scientific applications that benefit from task-based parallelization. To do this, we implemented our analysis into a dynamic binary instrumentation tool for the OmpSs-2 programming model. We then evaluated the tool on a set of nine benchmarks representative of different execution kernels and scientific problems.

### 5.4.1 Implementations details

In this section we provide further details on the OmpSs-2 programming model and the tool we developed.

#### The OmpSs-2 programming model

OmpSs-2 is a programming model developed at Barcelona Supercomputing Center (BSC) and similar in spirit to OpenMP, an API for shared memory multiprocessing programming in C, C++, and Fortran. Compared to OpenMP, OmpSs-2 has a more advanced execution model to synchronize the execution of different tasks via the specification of weak dependencies, just like TMP. It also has a different way of sycnhronizing a task with its descendant, which however can be easily emulated with OmpSs-2 `taskwait` functionality (analogous to TMP's `barrier`). TMP is a strict subset of OmpSs-2. Compared to TMP, OmpSs-2 use the `oss` prefix for compiler directives and supports additional functionalities:

- It defines a mechanism to release the dependencies that do not need to be enforced anymore. For instance, a task may use certain data only at the beginning and then perform other lengthy operations that unnecessarily delay the release of the dependencies associated to that data.

- It supports `commutative` and `concurrent` dataset entries. They are similar to `inout` dataset entries, but dependencies between two commutative entries don't imply any particular order (although guranteeing sequential execution), while dependencies between two concurrent entries are enforced (meaning that synchronization is left to the user).
- Tasks creation can be blocking, using the conditioned `if` attribute. When the condition is true, the parent tasks won't resume execution until the child task has terminated. This functonality can be used to implement barriers with datasets.
- Tasks creation can be disabled using the conditioned `final` attribute. When the condition is true, all child tasks that are created will be executed within the context of the parent task, hence all weak dependencies become strong for the task. This funcionality can be exploited to avoid the overhead of creating child tasks when this overhead is bigger than the time to execute the task themselves.

OmpSs-2 also supports other parallelization directives, such as task reductions, task loops, and critical regions. For a in-depth explanation of these features, consult the official OmpSs-2 specification [13]. We introduced some changes to the OmpSs-2 software stack at BSC to make it possible to analyse OmpSs-2 applications in the way described in this chapter. More precisely, we introduced the following changes:

- We added a new instrumentation variant to Nanos6, BSC's OmpSs-2 runtime, to allow our run-time tool to subscribe to some task-level events of interests (more details on this in the next section).
- We introduce a `pragma oss lint` directive and a new `verified` flag to the `pragma oss task` directive, whose meanings are the same as the ones described in Section 5.2.1. These changes have been introduced into the Mercurium source-to-source compiler [40].
- We also enriched the Mercurium built-in infrastructure for static analysis with a new algorithm that analyzes the source code of an OmpSs-2 program and annotates it with the directives proposed in Section 5.2.1. This static-time analyser is not part of the work illustrated in this chapter, but is used in the experimental evaluation to demonstrate the effectiveness of placing verification annotation in source code for the sake of tracing accurancy and tracing overhead. More details on the techniques employed in the static-time analyser can be found in [73].

#### OmpSs-2 Linter

Our run-time tracing process is based on a dynamic binary instrumentation tool built on top of Intel's PIN [69]. It takes as input an application parallelized using OmpSs-2 and provides a report of all parallelization errors that were encountered by tracing the application. When an OmpSs-2 application is run through this tool, memory accesses issued by tasks are recorded and temporarily saved to a storage area.

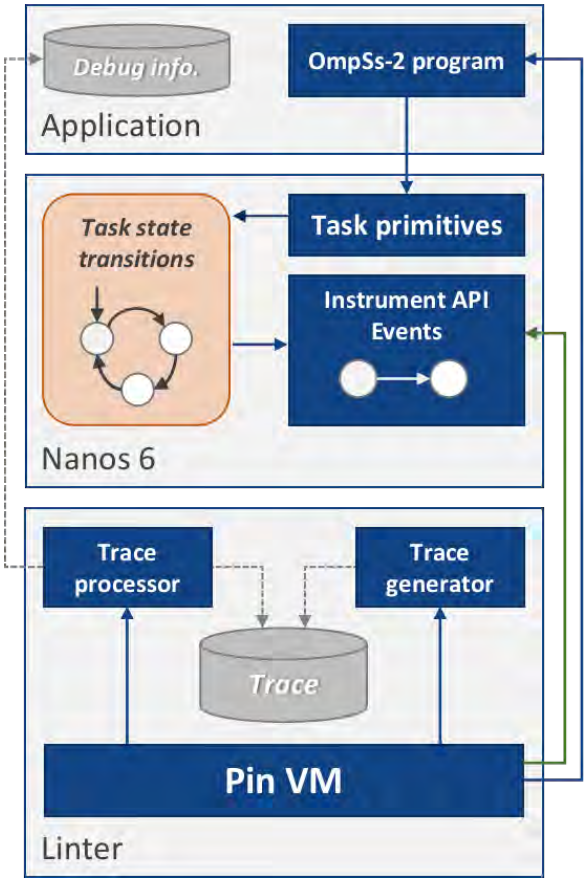



**Figure 5.1.** Interaction of the run-time tool with other software components

For each task, the recorded memory accesses are later processed at task completion time and compared with task information (e.g., dependencies) to check for potential parallelization errors. For each of such errors a warning is generated to report to the user about the problem. The report comes with additional contextual information such as: address and size of the mismatching memory access (if any) along with its access mode (i.e., read or write); name of the involved dependency (if any) with the expected directionality (`in`, `out`, `inout`); variable name (if found); task invocation point in the source code (i.e., line in the respective file).

The tool operates at two different levels of abstraction: (1) the abstraction provided by the OmpSs-2 programming model to deal with task and dependencies, which also forms the basis of TMP (see Section 5.1); (2) the abstraction provided by the target Instruction Set Architecture (ISA) to recognize accesses to memory, which in our case is AMD64. As illustrated in Figure 5.1, the target program is composed of the actual OmpSs-2 application and, if available, debugging information (i.e. symbol table and DWARF sections). The target program interacts with Nanos6 to execute in parallel on the available cores. When `oss` pragmas are translated by Mercurium into calls to Nanos6, these calls invoke task-based primitives which update the internal execution state of each task. Internally, Nanos6 maintains a state machine for each task to keep track of its execution state over time. Nanos6

provides an Instrument API to subscribe to state transitions in the task state machine and perform custom actions. Our instrumentation tool is composed of three main components: the PIN virtual machine (VM) that performs dynamic binary instrumentation and two modules that perform memory access tracing of the binary executable. The frontend module is dedicated to intercepting the accesses performed by the application at run-time and generating the actual traces, while the backend module is responsible for the processing of traces and generates the final user report. Our tool interacts with the rest of the software as follows. It executes the original application via the Pin VM. Events of interests at the ISA abstraction level (i.e., memory reads and writes) are intercepted via the PIN VM itself, which gives control to the trace generator module. Out tool also intercepts events of interests at the OmpSs-2 programming model level via Nanos6's Instrument API; these events are: when a task is created or destroyed, begins or ends, is put to wait via a full or partial synchronization, when dependencies are available or released. When one of these events occur, the PIN VM once again gives control to the frontend module. The backend module is invoked when a task completes execution. It loads traces from the storage area and combines them with information coming from Nanos6 via the Instrument API to detect parallelization errors. To provide contextual information, each entry of this report is further combines with information coming from the debug metadata available, if any, in the executable file. To date, the tool support all core functionalities of OmpSs-2 shared with TMP, plus the manual release of dependencies, the `if` and `final` task attributes, `commutative` and `concurrent` dataset entries. Other advanced parallelization constructs are not supported, meaning that all errors resulting from the use of these directives can't be detected. Despite that, the resulting restricted OmpSs-2 model is nevertheless general enough to support a wide-range of real-word applications and use cases, as shown in the next section.

Internally, local task analysis is implemented using binary search tree operating on intervals of contiguous memory addresses. These intervals can be easily derived from dataset entries, due to the way OmpSs-2 enables the user to specify as dataset entries entire C, C++, Fortran objects, or portions of it (included arrays). As for task accesses, our tool implements an aggregation operation that merges together all the accesses coming from the same instruction that touch consecutive memory accesses. While this aggregation has a detrimental effect on performance when it comes to build the access-set of a task, it speeds up the subsequent local task analysis. Indeed, checking for the different errors becomes a matter of intersecting different binary search trees, looking for overlapping intervals and checking for the conditions 1 to 3. For all intervals for which no match can be found, that interval is reported as an error. The case of E3 errors is lightly more complex as it involves barriers. Our approach to implement condition 3 was to maintain an expiry clock for all dataset entries of all child task of a task. The expiry clock is set as soon as we exit from a barrier. If it is a barrier with no dataset, all the child dataset entries are set as expired. Otherwise, we only set as expired the dataset entries that match with the entries in the barrier. Accesses are then compared with the child dataset entries that are not exprired at the time each access is performed.

### 5.4.2 Benchmarks

Our testbed is composed of a set of six typical kernels (`matmul`, `dot-product`, `multisaxpy`, `mergesort`, `cholesky`, and `nqueens`) and three proxy application (`nbody`, `heat`, and `HPCCG`). For each of these benchmarks we provide a small description of their inner workings and the accepted input parameters.

#### matmul

This benchmark runs a matrix multiplication operation $C = A \cdot B$, where $A$ has size $N \times M$, $B$ has size $M \times P$, and the resulting matrix $C$ has size $N \times P$. Parallelization is achived using tiling which relies on block partitioning with block size $TS$. The problem size for this benchmark is given by $M \times N \times P$. The degree of parallelization for this benchmark is given by $TS$. In our experiments we use $M = N = P = 128$ and $TS = 8$.

#### dot-product

The dot-product takes two equal-length $N$vectors and returns a single scalar. Parallelization is achieved using tiling which relies on block partitioning with block size $TS$. The problem size for this benchmark is given by $N$. The degree of parallelization for this benchmark is given by $TS$. In our experiments we use $N = 8192$ and $TS = 64$

#### multisaxpy

This benchmark runs several SAXPY operations. SAXPY stands for "Single-Precision A · X Plus Y". It is a Level 1 operation in the Basic Linear Algebra Subprograms (BLAS) package, and is a common operation in computations with vector processors. Each SAXPY operation solves the equation $Y = A \cdot X + Y$, where $X$ and $Y$ are two vectors of size $N$, and $A$ is a scalar value. Parallelization is achieved using tiling which relies on block partitioning with block size $TS$. The problem size for this benchmark is given by $N$. The degree of parallelization for this benchmark is given by $TS$. A further parameter $I$ specifies the number of times the SAXPY problem will be solved. In our experiments we use $N = 4194304$, $TS = 1024$, and $I = 1$.

#### mergesort

This benchmark runs a Merge Sort operation. It recursively halves an unsorted vector $X$ and sorts these chunk, until it gets to a sorted vector $X'$. Parallelization is achieved via a divide-and-conquer approach which relies on a maximum chunk size $TS$. This means that all chunks will a size lower than $TS$ won't create new tasks. The problem size for this benchmark is given by $N$. The degree of parallelization for this benchmark is given by $TS$. In our experiments we use $N = 65536$ and $TS = 512$.

### cholesky

This benchmark runs a Cholesky decomposition over a square matrix $A$ of side $N$. The code uses the CBLAS and LAPACKE interfaces to both BLAS and LAPACK. Parallelization is achieved using tiling which relies on block partitioning with block size $TS$. The problem size for this benchmark is given by $N$. The degree of parallelization for this benchmark is given by $TS$. In our experiments we use $N = 16384$ and $TS = 128$.

### nqueens

This benchmark computes, for a $N \times N$ chessboard, the number of configurations of placing $N$ chess queens in the chessboard, such that none of them is able to attack any other. It is implemented using a Branch-and-Bound algorithm. A sub-problem consists of checking if the current queen, placed at coordinates $(i, j)$ on the chessboard, can be attacked by any of the existing $M < N$ of queens already placed. Each sub-problem is truncated as soon as an attack from one of the existing $M$ queens is found. Otherwise, the current queen is added to the solution. If it was the $N$-th queen, the current solution is a valid solution to the N-queens problem. Placements are evaluated from left to right, top to bottom. Parallelization is achieved by spawning an independent task for each sub-problem, until we reach the $j$-th column. The rest $N - j$ columns won't generate any tasks and will be executed serially. The input to the problem are the chessboard size $N$ and the column threshold $TS$. Therefore, the problem size for this benchmark is given by $N$. The degree of parallelization for this benchmark is given by $TS$. In our experiments we use $N = 10$ and $TS = 4$.

### nbody

An N-body simulation numerically approximates the evolution of a system of bodies which interact with each other. It has applications in many scientific fields: astrophysical simulation, protein folding, and turbulent fluid flow simulation, to name a few. This application makes use of MPI for coarse-grained parallelization. Both computation and communication phases are taskified. However, communication tasks are serialized to prevent deadlocks between processes, since communication tasks perform blocking MPI calls. A familiar example is an astrophysical simulation in which each body represents a galaxy or an individual star, and the bodies attract each other through the gravitational force. N-body simulation arises in many other computational science problems as well. For example, protein folding is studied using N-body simulation to calculate electrostatic and Van der Waals forces. Turbulent fluid flow simulation and global illumination computation in computer graphics are other examples of problems that use N-body simulation. The input to the problem are the number of interacting particles $N$ and the number of iterations $I$ to compute the interactions between them. The problem size for this benchmark is given by $N$. The degree of parallelization for this benchmark is given by $TS$, which is the number of particles that are handled in parallel by each thread of each process, and $R$, which is the number of processes. In our experiments we use $N = 4096$, $TS = 64$, and $I = 1$.

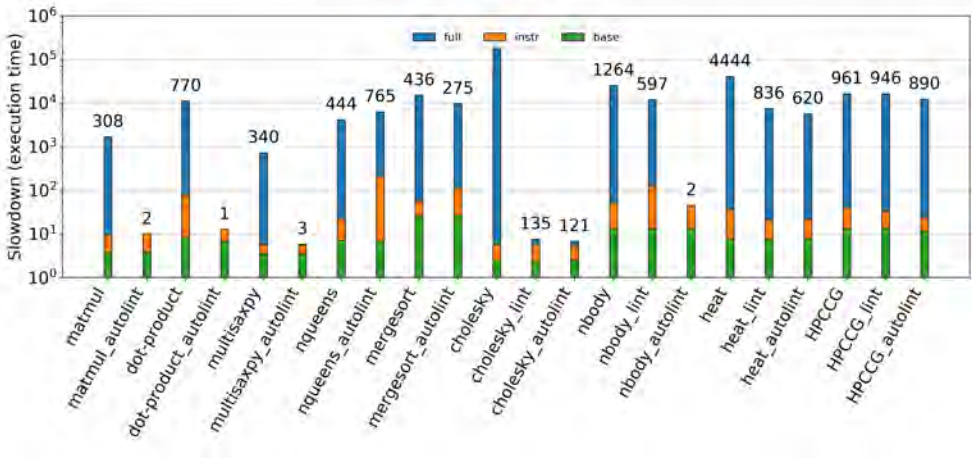



**Figure 5.2.** Execution time slowdown for the selected benchmarks.

### heat

The Heat simulation uses an iterative Gauss-Seidel method to solve the heat equation, which is a parabolic Partial Differential Equation (PDE) that describes the distribution of heat (or variation in temperature) in a given region over time. Just like N-body, the application use MPI for coarse-grained parallelization. In mathematics, it is the parabolic partial differential equation par excellence. In statistics, it is related to the study of the Brownian motion. Additionally, the diffusion equation is a generic version of the heat equation, and it is related to the study of chemical diffusion processes. The input to the problem is the size along one dimension of the grid used to compute heat $N$ and the number of iterations $I$. The problem size for this benchmark is given by $N$. The degree of parallelization for this benchmark is given by $TS$, which is the block size along one dimension, and $R$, which is the number of processes. In our experiments we use $N = 4096$, $TS = 64$, and $I = 1$.

### HPCCG

Solves the equation $A \cdot X = b$, where $A$ is a large sparse matrix, and $B$ and $X$ are vectors such that $X$ is unknown. The problem is discretized with a finite-difference scheme on a 3D rectangular grid domain and solved via a preconditioned conjugate gradient method. The input to the problem are $N_x$, $N_y$, and $N_z$, which are the sub-grid dimensions in the 3D space assigned to the different processes. Parallelization is achieved by assigning each sub-grid to a different process and by using block partitioning based on the number of threads for each process. Therefore, the problem size for this benchmark is given by $N_x \times N_y \times N_z \times R$, where $R$ is the number of processes. The degree of parallelization for this benchmark is given by $R$ and $T$, which is the number of threads per process. In our experiments we use $N_x = 50, N_y = 150, N_z = 50$.

### 5.4.3 Results

In this section we show that our tool can be used to evaluate the parallelization of applications even in presence of small inputs. We believe that this doesn't limit the effectiveness of our tool, as for many scientific applications (which are well-represented by the benchmarks we use) changing the input size doesn't affect *what* kind of tasks are executed but only *how many* of them. It must also be noted that even if an application is sensitive to changes in the inputs (e.g., changes in the internal control-flow of a task), these changes can usually be stimulated through small variations in the input size. Therefore, in general, it is not necessary to use big input sizes to achieve a complete coverage of code. This property is not a property of our tool, but rather that of applications, and is also leveraged by other debugging and testing tools. Nevertheless, for all those cases in which it is necessary to test an application with large or productive-level inputs, we can exploit the `lint` pragma and the `verified` attribute to focus the analysis only on the tasks and the region of code whose activation depends on those inputs. This approach makes our tool more effective, because it allows to spare the tracing overhead on the remaining code that could be tested with smaller input size.

Our experimentation proceeds as follows. For each benchmark, we report the slowdown and the absolute execution time to get a report of all the detected parallelization errors. All the experiments have been conducted on the MareNostrum4 supercomputer. Each compute node is equipped with two Intel Xeon Platinum 8160 CPUs with 24 cores each, thus totalling 48 cores per node, and 96 GB of main memory. The interconnection network is based on 100 Gbit/s Intel OmniPath HFI technology. The MPI benchmarks are run on 4 nodes, while the other benchmarks are run on a single node.

Figure 5.2 shows the slowdown and the absolute execution time (in seconds) for the benchmarks described above. Each bar represent a different benchmark and a different case. The `lint` suffix represents the case of running the benchmark using the `lint` directives to annotate third-party libraries. Of all benchmarks, only *cholesky* and the MPI benchmarks use the `pragma oss lint`, respectively to annotate the Intel MKL and Intel MPI third-party libraries. The `autolint` suffix represents the case of running the benchmark with the aid of a static analysis tool (mentioned in Section 5.4.1) which in turn exploits the existing `pragma oss lint` (if any) to extend the scope of these pragmas to bigger regions of code. This case is evaluated to demonstrate the effectiveness of verification annotations as a means to simplify local task analysis. The absence of any suffix means that the benchmark is run without the aid of the static analysis tool and without using the `pragma oss lint` directives. For each bar, we also report the execution time cost split into: (a) the minimum instrumentation cost to run the application using PIN (the `base` case in the legend); (b) the minimum instrumentation cost to instrument memory instructions, even without processing them (the `instr` case); (c) the full analysis cost (the `full` case in the legend).

Overall, the mechanism works in three phases. Initially, users can annotate the portions of code that they know for sure are correct with the proposed directives and clauses (further explained in Section 5.2.1). Then, a *compile-time tool* analyzes the code with the objective of verifying parts of code. At this stage, the compiler

can take different actions: (a) any detected error is reported to the user prior to execution, (b) the parts of the code that cannot be analysed statically are left for instrumentation, and (c) the parts of code that are analyzed and decided to be correct, are verified using the same directives and clauses offered to the user with such a purpose. Finally, a *run-time tool* executes the code and instruments only those parts that have been verified neither by the user nor by the compiler.

As we can see from the figure, the slowdown for the pure runtime instrumentation case (no suffix) can be quite high for some benchmarks (e.g., *dot-product* or *mergesort*). In the case of *cholesky*, the overhead is quite high due to the heavy use it makes of Intel's MKL library. However, we note that even so, the absolute execution time is in the order of minutes, thus not undermining the usability of the toolchain. We run a breakdown analysis of these cases and detected the major source of overhead to be the insertion of accesses in our binary search tree, which is used to aggregate contiguous accesses coming from the same instruction over time, and to compare them with task dependencies. In the case of *nqueens*, another major source of overhead is that of saving the errors encountered in the application, which still uses a binary search tree. Although we intend to develop a more efficient implementation for this data structure using interval trees **??**, we are still bound to pay the instrumentation cost depicted in the `base` and `instr` cases. This is not only due to the way PIN internally works, but also to the nature of each application. In fact, we already disable instrumentation whenever the application performs calls to the OmpSs-2 runtime system or the standard C/C++ libraries. Additionally, we disable instrumentation of private-memory instructions (such as stack instructions) in regions where there can't be conflicts due to the absence of a barrier.

When using external libraries protected by the `pragma oss lint` directive, the improvements in terms of slowdown can be critical. By appropriately marking those calls with the new pragmas, the runtime instrumentation tool only needs to instrument the `pragma oss lint` directive itself, and store a number of accesses that is proportional to the intervals specified in the `in`, `out`, and/or `inout` parameters of the pragma itself. This is critical for the case of *cholesky*, as each task only performs a single call to a different function in the MKL library, but there calls internally hide a huge number of accesses to memory that are the main source of overhead. Performance improvements can also be observed for the case of MPI benchmarks, which use the Intel MPI library, although the impact is smaller. For example, while *heat* is communication-intensive and so protecting calls to MPI is highly effective, *nbody* and *HPCCG* are computation-intensive, hence the use of pragma doesn't improve the execution overhead by much.

Using `pragma oss lint` directives and leveraging the compile-time analysis tool brings the most evident benefits, as it can be seen for *matmul*, *dot-product*, and *multisaxpy*. In this case, the compile-time tool can automatically wrap whole for-loop cycles into pragmas, or even put the `verified` flag to tasks within loops. In all these cases, the performance improvements are drastic because the instrumentation tool can disable tracing for most of the execution time of the application. We note that these improvements are not uncommon for real-world scenarios, as many kernels have a regular loop structure which can be easily analysed by means of techniques like those mentioned in [73]. As for *nqueens*, we observe that the static tool is unable to infer useful information due to the nature of the code and can only wrap in

pragmas simple memory-accessing statements. This has an effect in our runtime tool that is actually worse than not placing them, thus suggesting that additional work can be put in the static tool to avoid wrapping simple statements, and in the runtime tool to reduce the overhead of processing `pragma oss lint` directives. In *mergesort*, the tool can only partially simplify the handling of accesses in the merge phase of the algorithm, because the number of iterations which are performed in each part depends on the contents of the two sub-arrays to merge. For this reason, the overhead improvements are minimal. Similar considerations can be made for the MPI benchmarks and especially for *HPCCG*, where the main kernel performing a MKL-like *dgemm* operation couldn't be annotated at all due to the fact that a sparse matrix representation is used. As for *cholesky*, we observe that each task only performs a single call to a MKL library function, hence the compile-time tool can successfully promote the existing `oss lint` pragmas to the `verified` clause at the level of tasks, but this brings little additional benefits compared to the `lint` case.

Overall, experimental evaluation suggests that the absolute execution cost of running the selected applications against the runtime tool is affordable, and that a synergistic exploitation of static analysis and pragmas can drastically reduce this cost.

### 5.4.4 Related works

The strategies for correctness checking can be classified in: (1) static tools, which analyze the code at compile-time, and (2) dynamic tools, which analyze the code at run-time. Static tools need to be over-conservative because at compile-time they have to assume that any path is possible, and so these tools struggle to obtain no false negatives and minimal false positives. On the other hand, dynamic tools have the benefit of knowing all variables values, but are limited to the paths taken in a given execution and the input data-set, and also they introduce changes in the runtime behavior, possibly masking errors that may appear in a regular execution. The following paragraphs introduce different relevant works that have been developed in the last years to enhance the correctness of parallel programming models, and also highlight the need for further tools that overcome the limitations of the existing ones. Most of the static approaches for correctness detection focus on a specific programming model, or even a subset of it. As for OpenMP, there are solutions focused in subsets of the language such as *ompVerify* [11], a tool based on the polyhedral model that is able to detect several errors in OpenMP parallel loops, and *OpenMP Analysis Toolkit* (OAT) [55], a tool that uses Satisfiability Modulo Theories (SMT) solver based symbolic analysis to detect data races and deadlocks. A more general solution is provided by Lin [54], who described a control flow graph and a region tree to statically detect non-concurrent blocks of code and race conditions in OpenMP2.5 programs with the Sun Studio 9 Fortran compiler.

For task-based models such as OmpSs and the OpenMP tasking subset, there are static solutions that allow catching errors in the synchronization of tasks and variables that may lead to non-deterministic results (due to data races) and runtime failures (due to accessing dead variables) [73]. These techniques adopt a conservative approach, in the sense that performance is secondary when correctness is on the line (e.g. privatize a variable in order to avoid a race-condition). Considering concurrent

models such as Ada, there have also been efforts to introduce correctness checking techniques at compile time. These include GNATprove [2], a formal verification tool for Ada based on the GNAT compiler and Meyer's *design-by-contract* paradigm. However, although model checking based techniques are very mature, their usefulness depends on contracts that are also written by programmers, hence are liable to have errors. For that reason, more general approaches have appeared including the analysis of Ada and OpenMP mixed programs [74]. However these only consider the Ada Ravenscar profile and are not yet implemented.

Finally, other static solutions that can be applied in different environments are the following: RacerX [37], a flow-sensitive, interprocedural analysis to detect race conditions and deadlocks; Chord [61], a static analysis tool able to detect races, deadlocks and atomicity violations, among others, in the Java bytecode; Warlock [84], a tool for finding races in C programs, or Extended Static Checking (ESC) [23] and ESC/Java [41], tools for Modula-3 and Java respectively, which use a theorem-proving techniques to find errors like null dereferences, array bounds errors and typecast errors among others.

For the dynamic detection of programming errors coming from parallelization, most of the literature is focused on algorithms and tools to check for data and determinacy races [8, 81, 49, 42]. Archer [5] adapts ThreadSanitizer, which can detects data races in unstructured parallel programs, to the case of OpenMP 3.1 (i.e., basic tasking with no dependencies). Similarly to our tool-chain, it employs a hybrid approach and section-based analysis [21], where a static phase is used to discard all sequential code and a dynamic phase is used to check for data races in the remaining concurrent parts. It is sound but not complete, as it is sensitive to the particular scheduling of events at run-time and cannot detect data races coming from concurrent tasks executed by the same thread. Sword [6] is a tool that is capable of detecting all and only data races in OpenMP programs comprised of only nested fork-join parallelism (i.e., only parallel constructs). DFinspec [57] is a tool for detecting determinacy races in task-parallel programs where tasks have dependencies but are atomic (i.e., there are no task-switching points inside task code). TaskSanitizer [58] solves the same problems for OpenMP program comprised of only tasks with dependencies and internal scheduling points. ROMP is another tool targeting at OpenMP with tasking [45]. It uses an approach close to Sword to built the HB relation for nested fork-join parallelism parts and one similar to TaskSanitizer for the HB relation of tasks with dependencies. A more general approach for async/finish programs that can include futures—a parallel programming paradigm which is more general than the one offered by OpenMP—is provided in [85]. Our approach differs from the ones adopted by the above tools in two ways. We don't explicitly check for data or determinacy races as our tools check for every kind of programming error in the code, including performance errors. Further, we exploit the structural induction of OmpSs-2 to only check the correctness for each task at each nesting level separately.

Other tools aim at checking the correctness of parallel programs in a way akin to that of our tool-chain. StarSscheck [15] is a correctness checker for applications written in the StarSs programming model. Our tool-chain is similar in spirit to StarSscheck, but supports a wider range of errors due to the wider flexibility of the OmpSs programming model (task nesting being the most evident). Additionally,

we exploit the structural induction of OmpSs to deal with task nesting efficiently. Lastly, differently from StarSscheck, we also complement our dynamic analysis with a compile-time tool to prove the correctness of OmpSs-2 programs independently of a specific input. Other tools verify the correctness of MPI applications [95, 91, 48, 79]. Our tool-chain cannot check the MPI correctness of MPI-pure or hybrid applications, although it can check the OmpSs-2 parallelization of applications that make use of both MPI and OmpSs-2 for respectively coarse-grained and fine-grained parallelism.

## 5.5 Conclusions and future work

We have presented an analysis technique to detect parallelization errors in applications which use task-based parallel programming model with task nesting and whose correctness can be proved by structural induction. Our technique only requires local analysis of code, i.e., independently for each task. Because of the way our reference task-based programming model is defined, by locally analyzing each task we can infer programming errors at a global scale, i.e., for the whole program. We note that this model has its roots in well-known programming models such as OpenMP and OmpSs-2. More importantly, our approach can be easily used with other parallel programming abstractions based on tasks and structural induction, thus making it applicable to a wider range of scenarios, thus making our contribution of wide applicability in real-world settings. We also introduced two new annotations that use the same syntax employed to describe a task dataset. Application developers can use these two annotations to inform the analysis about the set of data references performed in a specific regions of code. We can therefore safely ignore what happens inside these regions, thus improving both performance and accuracy. We implemented our technique into a run-time tool that analyzes binary code and informs the user about possible parallelization errors in the applications. Experiments run on a series of benchmarks varying from simple execution kernels to real-world applications with a suggest that our tool is capable of analyzing a wide range of applications within an acceptable execution time limit. Future works entail extending our analysis to other OmpSs-2 and OpenMP task-based construct. We also plan to improve our analysis to detect inefficient parallelization constructs and suggest the use of more efficient ones.

CHAPTER 6

# Conclusions

Since the beginning, the objective of the research included in this thesis has been that of investigating on transparent techniques and tools to support application development and execution, taking into account the increasing complexity of hardware and the increasing demands of scientific and large-scale applications. In this thesis, we discuss ways to bridge the gap between the two worlds using a framework that involves tracing the application, analysing the traces that were produced, and exploiting the produced information for different purposes, with a focus on applications that rely on parallel programming. Our scientific contributions, in terms of tracing and analysis, exist within this well-known framework. A pivotal point of this thesis is also that the complexity of hardware should have the least possible impact on the complexity of applications, especially in terms of structure and development. For this reason, the techniques and tools we present (1) find their natural placement within the scope of environmental software, and (2) support the existence of abstract and intuitive programming paradigms and models.

The experience we gained during this journey tells us that the methodology we followed is promising, but has some limitations that prevent us from taking it to its full potential. There are so many intertesting aspects of applications that are worth observing, and information is king to bring any improvement in terms of development and/or execution. However, a fundamental overhead-accuracy-precision trade-off exists, at least empirically: in order to obtain the desired information, the cost is high; lowering the cost means lowering the quality of information, hence the effectiveness of any exploitation scheme. Sometimes, this cost is too high and can outweight the benefits. Not only that, but when tracing is performed at run-time, it can essentially change the properties of the applications being executed (a phenomenon described as *tracing perturbation*), thus leading us to observe the effects of our own observation.

Nowadays, some interesting technologies are raising, with the intent of pushing the trade-off curve toward low-cost, low-perturbation, and high-quality tracing. For example, hardware monitoring units in modern processors promise to be able to trace low-level hardware events quite efficiently. The vast range of events which can be catched makes this technology extremely valuable in many different contexts. Our expectation is that it will be the gold mine of the next decade, and that many

researchers will join this gold rush. However, despite its potential, it can only be used to trace applications at their lowest possible level—that of machine-instructions, or even micro-operations, running into a processor. To trace higher-level events software instrumentation remains the only option. One might argue whether it makes more sense to provide instrumentation internally, within that portion of the software stack that we are interested in tracing (e.g., calls to a run-time library which implements task-based parallelism primitives). Another option is to instrument these events externally, by means of ad-hoc instrumentation tools. Our empirical experience in this sense is that the first option is more promising, as the tracing logic can share code and data with other pre-existing logic in that portion of the software stack, without having to re-invent the wheel.

Overall, in the foreseeable future, we expect a better integration between the different components and tools that make up environmental software, as well as between environmental software itself and the underlying hardware. It is clear to us that the abstractions offered at the level of application software must remain generic and unopinionated with respect to the hardware, in order to be portale and comprehensible to programmers. We also claim that tracing should not occur within the application, so as not to perturb it excessively, but rather in middle-level software or in hardware. This is not only important in terms of the cost-benefit relationship we mentioned before. In fact, in our opinion, it also allows developers to still have a quite good idea of what is the implicit cost of using certain language constructs in their programs. We also demand better frameworks and tools for the analysis of programs at any level, and debugging/linting/profiling tools that are aware of the programming abstractions used by the application. To support these tools, it is also important to enrich programs with annotations and directives which can serve as documentation for other developers, and be exploited by compilers to generate or export richer meta-data for later use by these analysis tools.

As future work, we wish to continue working on the methodology presented in this thesis to study other relevant problems in parallel programming. Our focus won't simply be theoretical, but also on implementations that are efficient enough to be used in production environments, for on-line or off-line exploitation. Our intent is to use new tracing technology, such as the hardware monitors previously described, to reduce the effects of tracing in terms of overhead and perturbation. We also wish to invest more effort on the analysis and exploitation parts, to increase the positive gap between the benefits of tracing and its cost. Our intuition is that a stricter, closer cooperation between kernel-space and user-space software can maximize the effects of trace exploitation in on-line schemes. Also, we believe that a tighter cooperation between analysis tools and the run-time libraries providing abstract programming paradigms and models can be the key to improve the effectiveness of off-line exploitation schemes.

# Bibliography

[1] Go: Concurrent programming. https://web.archive.org/web/20190815110617/https://programming.guide/go/go-concurrency-tutorial.html. Accessed: 2019-08-15.

[2] AdaCore, Altran, Astrium Space Transportation, CEA-LIST, ProVal at INRIA and Thales Communications. Project Hi-Lite: GNATprove. http://www.open-do.org/projects/hi-lite/gnatprove (2017).

[3] Aguilera, M. K., Merchant, A., Shah, M., Veitch, A., and Karamanolis, C. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, vol. 41, pp. 159–174. ACM (2007). ISBN 978-1-59593-591-5. doi:10.1145/1294261.1294278.

[4] Allen, F. E. In *Proceedings of a Symposium on Compiler Optimization, pages 1–19, New York, NY, USA*. ACM (1970).

[5] Atzeni, S., Gopalakrishnan, G., Rakamaric, Z., Ahn, D. H., Laguna, I., Schulz, M., Lee, G. L., Protze, J., and Müller, M. S. Archer: Effectively spotting data races in large openmp applications. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 53–62 (2016). doi:10.1109/IPDPS.2016.68.

[6] Atzeni, S., Gopalakrishnan, G., Rakamaric, Z., Laguna, I., Lee, G. L., and Ahn, D. H. Sword: A bounded memory-overhead detector of openmp data races in production runs. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 845–854 (2018). doi:10.1109/IPDPS.2018.00094.

[7] Awasthi, M., Nellans, D. W., Sudan, K., Balasubramonian, R., and Davis, A. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *19th International Conference on Parallel Architecture and Compilation Techniques (PACT 2010), Vienna, Austria, September 11-15, 2010*, pp. 319–330 (2010). Available from: http://doi.acm.org/10.1145/1854273.1854314, doi:10.1145/1854273.1854314.

[8] Banerjee, U., Bliss, B., Ma, Z., and Petersen, P. Unraveling data race detection in the intel thread checker. In *In Proceedings of STMCS '06* (2006).

[9] Barnes, P. D., Jr., Carothers, C. D., Jefferson, D. R., and LaPre, J. M. Warp speed: Executing time warp on 1,966,080 cores. In *Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '13, pp. 327–336. ACM, New York, NY, USA (2013). ISBN 978-1-4503-1920-1. Available from: `http://doi.acm.org/10.1145/2486092.2486134`, `doi:10.1145/2486092.2486134`.

[10] Barrow-Williams, N., Fensch, C., and Moore, S. W. A communication characterisation of splash-2 and parsec. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009, October 4-6, 2009, Austin, TX, USA*, pp. 86–97 (2009). Available from: `http://doi.ieeecomputersociety.org/10.1109/IISWC.2009.5306792`, `doi:10.1109/IISWC.2009.5306792`.

[11] Basupalli, V., Yuki, T., Rajopadhye, S., Morvan, A., Derrien, S., Quinton, P., and Wonnacott, D. ompVerify: polyhedral analysis for the OpenMP programmer. In *International Workshop on OpenMP*, pp. 37–53. Springer (2011).

[12] Bienia, C. *Benchmarking Modern Multiprocessors*. Ph.D. thesis, Princeton University (2011).

[13] BSC. Ompss-2 specification (2019). Available from: `https://pm.bsc.es/ftp/ompss-2/doc/spec/`.

[14] Butenhof, D. R. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1997). ISBN 0-201-63392-2.

[15] Carpenter, P. M., Ramirez, A., and Ayguade, E. Starsscheck: A tool to find errors in task-based parallel programs. In *Euro-Par 2010 - Parallel Processing* (edited by P. D'Ambra, M. Guarracino, and D. Talia), pp. 2–13. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). ISBN 978-3-642-15277-1.

[16] Carrington, L., Snavely, A., Gao, X., and Wolter, N. A performance prediction framework for scientific applications. In *ICCS Workshop on Performance Modeling and Analysis (PMA03*, pp. 926–935 (2003).

[17] Casanova, H., Desprez, F., and Suter, F. On cluster resource allocation for multiple parallel task graphs. *J. Parallel Distrib. Comput.*, **70** (2010), 1193. Available from: `http://dx.doi.org/10.1016/j.jpdc.2010.08.017`, `doi:10.1016/j.jpdc.2010.08.017`.

[18] Castro, M., Goes, L. F. W., Ribeiro, C. P., Cole, M., Cintra, M., and Mehaut, J.-F. A machine learning-based approach for thread mapping on transactional memory applications. In *Proceedings of the 2011 18th International Conference on High Performance Computing*, pp. 1–10. IEEE Computer Society, Washington, DC, USA (2011).

[19] Chen, H., Chen, W., Huang, J., Robert, B., and Kuhn, H. MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters

and multiclusters. In *Proceedings of the 20th Annual International Conference on Supercomputing, ICS 2006, Cairns, Queensland, Australia, June 28 - July 01, 2006*, pp. 353–360 (2006). Available from: http://doi.acm.org/10.1145/1183401.1183451, doi:10.1145/1183401.1183451.

[20] Daigle, R., Xia, C., and Torrellas, J. Low perturbation address trace collection for operating system, multiprogrammed, and parallel workloads in multiprocessors. Tech. rep., Multiprogrammed, and Parallel Workloads in Multiprocessors,º technical report, Center for Supercomputing Research and Development, Univ. of Illinois at Urbana-Champaign (1996).

[21] Das, M., Southern, G., and Renau, J. Section-based program analysis to reduce overhead of detecting unsynchronized thread communication. *ACM Trans. Archit. Code Optim.*, **12** (2015), 23:23:1. Available from: http://doi.acm.org/10.1145/2766451, doi:10.1145/2766451.

[22] Dashti, M., Fedorova, A., Funston, J. R., Gaud, F., Lachaize, R., Lepers, B., Quéma, V., and Roth, M. Traffic management: a holistic approach to memory placement on NUMA systems. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pp. 381–394 (2013). Available from: http://doi.acm.org/10.1145/2451116.2451157, doi:10.1145/2451116.2451157.

[23] Detlefs, D. L., Leino, K. R. M., Nelson, G., and Saxe, J. B. Extended static checking. (1998).

[24] di Sanzo, P., Ciciani, B., Palmieri, R., Quaglia, F., and Romano, P. On the analytical modeling of concurrency control algorithms for software transactional memories: The case of commit-time-locking. *Perform. Eval.*, **69** (2012), 187.

[25] di Sanzo, P., Sannicandro, M., Ciciani, B., and Quaglia, F. Markov chain-based adaptive scheduling in software transactional memory. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, pp. 373–382 (2016). Available from: http://dx.doi.org/10.1109/IPDPS.2016.104, doi:10.1109/IPDPS.2016.104.

[26] Didona, D., Diegues, N., Kermarrec, A., Guerraoui, R., Neves, R., and Romano, P. Proteustm: Abstraction meets performance in transactional memory. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, pp. 757–771 (2016). Available from: http://doi.acm.org/10.1145/2872362.2872385, doi:10.1145/2872362.2872385.

[27] Didona, D., Felber, P., Harmanci, D., Romano, P., and Schenker, J. Identifying the optimal level of parallelism in transactional memory applications. *Computing*, **97** (2015), 939. Available from: http://dx.doi.org/10.1007/s00607-013-0376-3, doi:10.1007/s00607-013-0376-3.

[28] Didona, D., Romano, P., Peluso, S., and Quaglia, F. Transactional auto scaler: Elastic scaling of replicated in-memory transactional data grids. *TAAS*, **9** (2014), 11:1.

[29] Diegues, N., Romano, P., and Garbatov, S. Seer: Probabilistic scheduling for hardware transactional memory. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2015, Portland, OR, USA, June 13-15, 2015*, pp. 224–233 (2015). Available from: `http://doi.acm.org/10.1145/2755573.2755578`, `doi:10.1145/2755573.2755578`.

[30] Diener, M., da Cruz, E. H. M., Navaux, P. O. A., Busse, A., and Heiss, H. kmaf: automatic kernel-level management of thread and data affinity. In *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, pp. 277–288 (2014). Available from: `http://doi.acm.org/10.1145/2628071.2628085`, `doi:10.1145/2628071.2628085`.

[31] Diener, M., Madruga, F. L., Rodrigues, E. R., Alves, M. A. Z., Schneider, J., Navaux, P. O. A., and Heiss, H. Evaluating thread placement based on memory access patterns for multi-core processors. In *12th IEEE International Conference on High Performance Computing and Communications, HPCC 2010, 1-3 September 2010, Melbourne, Australia*, pp. 491–496 (2010). `doi:10.1109/HPCC.2010.114`.

[32] Dolev, S., Hendler, D., and Suissa, A. Car-stm: scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pp. 125–134. ACM, New York, NY, USA (2008).

[33] Dragojević, A. and Guerraoui, R. Predicting the scalability of an stm: A pragmatic approach. In *Presented at: 5th ACM SIGPLAN Workshop on Transactional Computing* (2010).

[34] Drepper, U. Elf handling for thread-local storage. Tech. rep. (2013). Available from: `https://www.akkadia.org/drepper/tls.pdf`.

[35] DynamoRIO. http://www.dynamorio.org/.

[36] Economo, S., Silvestri, E., di Sanzo, P., Pellegrini, A., and Quaglia, F. Prompt application-transparent transaction revalidation in software transactional memory. In *16th IEEE International Symposium on Network Computing and Applications, NCA 2017, Cambridge, MA, USA, October 30 - November 1, 2017*, pp. 157–162 (2017).

[37] Engler, D. and Ashcraft, K. Racerx: effective, static detection of race conditions and deadlocks. In *ACM SIGOPS Operating Systems Review*, vol. 37, pp. 237–252. ACM (2003).

[38] Felber, P., Fetzer, C., Marlier, P., and Riegel, T. Time-based software transactional memory. *IEEE Trans. Parallel Distrib. Syst.*, **21** (2010), 1793.

[39] Felber, P., Fetzer, C., Marlier, P., and Riegel, T. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, **21** (2010), 1793. doi:10.1109/TPDS.2010.49.

[40] Ferrer, R., Royuela, S., Caballero, D., Duran, A., Martorell, X., and Ayguadé, E. Mercurium: Design decisions for a s2s compiler. In *Cetus Users and Compiler Infastructure Workshop in conjunction with PACT* (2011).

[41] Flanagan, C., Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. Extended static checking for java. In *ACM Sigplan Notices*, vol. 37, pp. 234–245. ACM (2002).

[42] Flanagan, C. and Freund, S. N. The roadrunner dynamic analysis framework for concurrent programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, pp. 1–8. ACM, New York, NY, USA (2010). ISBN 978-1-4503-0082-7. Available from: http://doi.acm.org/10.1145/1806672.1806674, doi:10.1145/1806672.1806674.

[43] Fujimoto, R. M., Panesar, K. S., and Panesar, K. S. Buffer management in shared-memory time warp systems. In *PADS*, pp. 149–156 (1995).

[44] Gennaro, I. D., Pellegrini, A., and Quaglia, F. Os-based numa optimization: Tackling the case of truly multi-thread applications with non-partitioned virtual page accesses. In *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2011, May 16-19, 2016 – Cartagena, Colombia* (2016).

[45] Gu, Y. and Mellor-Crummey, J. Dynamic data race detection for openmp programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pp. 61:1–61:12. IEEE Press, Piscataway, NJ, USA (2018). Available from: https://doi.org/10.1109/SC.2018.00064, doi:10.1109/SC.2018.00064.

[46] Guerraoui, R. and Kapalka, M. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, February 20-23, 2008*, pp. 175–184 (2008).

[47] Herlihy, M. and Moss, J. E. B. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, **21** (1993), 289. Available from: http://doi.acm.org/10.1145/173682.165164, doi:10.1145/173682.165164.

[48] Hilbrich, T., de Supinski, B. R., Hänsel, F., Müller, M. S., Schulz, M., and Nagel, W. E. Runtime mpi collective checking with tree-based overlay networks. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pp. 129–134. ACM, New York, NY, USA (2013). ISBN 978-1-4503-1903-4. Available from: http://doi.acm.org/10.1145/2488551.2488570, doi:10.1145/2488551.2488570.

[49] Jannesari, A., Kaibin Bao, Pankratius, V., and Tichy, W. F. Helgrind+: An efficient dynamic race detector. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pp. 1–13 (2009). doi:10.1109/IPDPS.2009.5160998.

[50] Karlsson, C., Davies, T., and Chen, Z. Optimizing process-to-core mappings for application level multi-dimensional MPI communications. In *2012 IEEE International Conference on Cluster Computing, CLUSTER 2012, Beijing, China, September 24-28, 2012*, pp. 486–494 (2012). Available from: http://dx.doi.org/10.1109/CLUSTER.2012.47, doi:10.1109/CLUSTER.2012.47.

[51] Ketterlin, A. and Clauss, P. Efficient memory tracing by program skeletonization. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2011, 10-12 April, 2011, Austin, TX, USA*, pp. 97–106 (2011). Available from: http://dx.doi.org/10.1109/ISPASS.2011.5762719, doi:10.1109/ISPASS.2011.5762719.

[52] Laurenzano, M., Simon, B., Snavely, A., and Gunn, M. Low cost trace-driven memory simulation using simpoint. *SIGARCH Computer Architecture News*, **33** (2005), 81. Available from: http://doi.acm.org/10.1145/1127577.1127593, doi:10.1145/1127577.1127593.

[53] Laurenzano, M., Tikir, M. M., Carrington, L., and Snavely, A. PEBIL: efficient static binary instrumentation for linux. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2010, 28-30 March 2010, White Plains, NY, USA*, pp. 175–183 (2010). Available from: http://dx.doi.org/10.1109/ISPASS.2010.5452024, doi:10.1109/ISPASS.2010.5452024.

[54] Lin, Y. Static nonconcurrency analysis of openmp programs. In *IWOMP*, pp. 36–50. Springer (2008).

[55] Ma, H., Diersen, S. R., Wang, L., Liao, C., Quinlan, D., and Yang, Z. Symbolic analysis of concurrency errors in openmp programs. In *ICPP*, pp. 510–516. IEEE (2013).

[56] Marathe, J. and Mueller, F. Hardware profile-guided automatic page placement for ccnuma systems. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2006, New York, New York, USA, March 29-31, 2006*, pp. 90–99 (2006). Available from: http://doi.acm.org/10.1145/1122971.1122987, doi:10.1145/1122971.1122987.

[57] Matar, H. S., Mutlu, E., Tasiran, S., and Unat, D. Output nondeterminism detection for programming models combining dataflow with shared memory. *Parallel Computing*, **71** (2018), 42 . Available from: http://www.sciencedirect.com/science/article/pii/S016781911730193X, doi:https://doi.org/10.1016/j.parco.2017.11.008.

[58] Matar, H. S. and Unat, D. Runtime determinacy race detection for openmp tasks. In *Euro-Par 2018: Parallel Processing* (edited by M. Aldinucci,

L. Padovani, and M. Torquati), pp. 31–45. Springer International Publishing, Cham (2018). ISBN 978-3-319-96983-1.

[59] Moore, S. V. A comparison of counting and sampling modes of using performance monitoring hardware. In *Proceedings of the International Conference on Computational Science-Part II*, ICCS '02, pp. 904–912. Springer-Verlag, London, UK, UK (2002). ISBN 3-540-43593-X. Available from: http://dl.acm.org/citation.cfm?id=645458.653464.

[60] Mytkowicz, T., Diwan, A., Hauswirth, M., and Sweeney, P. F. Understanding measurement perturbation in trace-based data. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pp. 1–6 (2007). doi:10.1109/IPDPS.2007.370515.

[61] Naik, M., Park, C.-S., Sen, K., and Gay, D. Effective static deadlock detection. In *Proceedings of the 31st International Conference on Software Engineering*, pp. 386–396. IEEE Computer Society (2009).

[62] Nethercote, N. and Seward, J. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments, VEE 2007, San Diego, California, USA, June 13-15, 2007*, pp. 65–74 (2007).

[63] Nilakantan, S., Lerner, S., Hempstead, M., and Taskin, B. Can you trust your memory trace? a comparison of memory traces from binary instrumentation and simulation. In *2015 28th International Conference on VLSI Design*, pp. 135–140 (2015). doi:10.1109/VLSID.2015.28.

[64] Nowak, A. and Bitzes, G. The overhead of profiling using PMU hardware counters (2014). Available from: http://dx.doi.org/10.5281/zenodo.10800, doi:10.5281/zenodo.10800.

[65] Osman, R., Coulden, D., and Knottenbelt, W. J. Performance modelling of concurrency control schemes for relational databases. In *Analytical and Stochastic Modelling Techniques and Applications - 20th International Conference, ASMTA 2013, Ghent, Belgium, July 8-10, 2013. Proceedings*, pp. 337–351 (2013).

[66] Pellegrini, A. Hijacker: Efficient static software instrumentation with applications in high performance computing (poster paper). In *Proceedings of the 2013 International Conference on High Performance Computing & Simulation*, HPCS, pp. 650–655. IEEE Computer Society (2013).

[67] Peluso, S., Ruivo, P., Romano, P., Quaglia, F., and Rodrigues, L. E. T. GMU: genuine multiversion update-serializable partial data replication. *IEEE Trans. Parallel Distrib. Syst.*, **27** (2016), 2911.

[68] Piccoli, G., Santos, H. N., Rodrigues, R. E., Pousa, C., Borin, E., and Quintão Pereira, F. M. Compiler support for selective page migration in numa architectures. In *Proceedings of the 23rd International Conference on*

*Parallel Architectures and Compilation*, PACT '14, pp. 369–380. ACM, New York, NY, USA (2014). ISBN 978-1-4503-2809-8.

[69] Pin. http://www.pintool.org/.

[70] Porfirio, A., Pellegrini, A., di Sanzo, P., and Quaglia, F. Transparent support for partial rollback in software transactional memories. In *Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*, pp. 583–594 (2013).

[71] Reinders, J. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism* (2007). ISBN 978-0-596-51480-8.

[72] Romano, P., Palmieri, R., Quaglia, F., Carvalho, N. M. R., and Rodrigues, L. An optimal speculative transactional replication protocol. In *Proc. of ISPA*, vol. 0, pp. 449–457 (2010). ISBN 978-0-7695-4190-7. doi:http://doi.ieeecomputersociety.org/10.1109/ISPA.2010.94.

[73] Royuela, S., Ferrer, R., Caballero, D., and Martorell, X. Compiler analysis for OpenMP tasks correctness. In *Computing Frontiers*, p. 7. ACM (2015).

[74] Royuela, S., Martorell, X., Quiñones, E., and Pinho, L. M. Safe parallelism: compiler analysis techniques for ada and openmp. In *Ada-Europe International Conference on Reliable Software Technologies*, pp. 141–157. Springer (2018).

[75] Rughetti, D., Di Sanzo, P., Ciciani, B., and Quaglia, F. Machine learning-based self-adjusting concurrency in software transactional memory systems. In *Proceedings of the 2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 278–285. IEEE Computer Society, Washington, DC, USA (2012).

[76] Rughetti, D., di Sanzo, P., Ciciani, B., and Quaglia, F. Analytical/ml mixed approach for concurrency regulation in software transactional memory. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2014, Chicago, IL, USA, May 26-29, 2014*, pp. 81–91 (2014). Available from: http://dx.doi.org/10.1109/CCGrid.2014.118, doi:10.1109/CCGrid.2014.118.

[77] Rughetti, D., di Sanzo, P., Ciciani, B., and Quaglia, F. Analytical/ml mixed approach for concurrency regulation in software transactional memory. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2014, Chicago, IL, USA, May 26-29, 2014*, pp. 81–91 (2014). Available from: http://dx.doi.org/10.1109/CCGrid.2014.118, doi:10.1109/CCGrid.2014.118.

[78] Rughetti, D., Romano, P., Quaglia, F., and Ciciani, B. Automatic tuning of the parallelism degree in hardware transactional memory. In *Euro-Par 2014 Parallel Processing - 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*, pp. 475–486 (2014).

[79] Saillard, E., Carribault, P., and Barthou, D. Combining static and dynamic validation of mpi collective communications. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pp. 117–122. ACM, New York, NY, USA (2013). ISBN 978-1-4503-1903-4. Available from: `http://doi.acm.org/10.1145/2488551.2488555`, `doi:10.1145/2488551.2488555`.

[80] Sanzo, P. D. Analysis, classification and comparison of scheduling techniques for software transactional memories. *IEEE Transactions on Parallel and Distributed Systems*, **PP** (2017), 1. `doi:10.1109/TPDS.2017.2740285`.

[81] Serebryany, K. and Iskhodzhanov, T. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pp. 62–71. ACM, New York, NY, USA (2009). ISBN 978-1-60558-793-6. Available from: `http://doi.acm.org/10.1145/1791194.1791203`, `doi:10.1145/1791194.1791203`.

[82] Shavit, N. and Touitou, D. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pp. 204–213. ACM, New York, NY, USA (1995). ISBN 0-89791-710-3. Available from: `http://doi.acm.org/10.1145/224964.224987`, `doi:10.1145/224964.224987`.

[83] Song, D., et al. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, pp. 1–25. Springer-Verlag, Berlin, Heidelberg (2008). ISBN 978-3-540-89861-0. Available from: `http://dx.doi.org/10.1007/978-3-540-89862-7_1`, `doi:10.1007/978-3-540-89862-7_1`.

[84] Sterling, N. Warlock-a static data race analysis tool. In *USENIx Winter*, pp. 97–106 (1993).

[85] Surendran, R. and Sarkar, V. Dynamic determinacy race detection for task parallelism with futures. In *Runtime Verification* (edited by Y. Falcone and C. Sánchez), pp. 368–385. Springer International Publishing, Cham (2016). ISBN 978-3-319-46982-9.

[86] Tikir, M. M. and Hollingsworth, J. K. Hardware monitors for dynamic page migration. *J. Parallel Distrib. Comput.*, **68** (2008), 1186. Available from: `http://dx.doi.org/10.1016/j.jpdc.2008.05.006`, `doi:10.1016/j.jpdc.2008.05.006`.

[87] TPC Council. TPC-C Benchmark, Revision 5.11. (2010).

[88] Trahay, F., Rué, F., Faverge, M., Ishikawa, Y., Namyst, R., and Dongarra, J. Eztrace: A generic framework for performance analysis. In *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2011, Newport Beach, CA, USA, May 23-26, 2011*, pp. 618–619 (2011). Available from: `http://dx.doi.org/10.1109/CCGrid.2011.83`, `doi:10.1109/CCGrid.2011.83`.

[89] Uhlig, R. A. and Mudge, T. N. Trace-driven memory simulation: A survey. *ACM Comput. Surv.*, **29** (1997), 128. Available from: http://doi.acm.org/10.1145/254180.254184, doi:10.1145/254180.254184.

[90] van Riel, R. and Chegu, V. Automatic NUMA Balancing. Tech. Rep. 1.0 (2014).

[91] Vo, A., Aananthakrishnan, S., Gopalakrishnan, G., d. Supinski, B. R., Schulz, M., and Bronevetsky, G. A scalable and distributed dynamic formal verifier for mpi programs. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–10 (2010). doi:10.1109/SC.2010.7.

[92] Wagner, M., Doleschal, J., and Knüpfer, A. Tracing long running applications: A case study using gromacs. In *High Performance Computing Simulation (HPCS), 2015 International Conference on*, pp. 129–136 (2015). doi:10.1109/HPCSim.2015.7237031.

[93] Wang, Q., Kulkarni, S., Cavazos, J. V., and Spear, M. Towards applying machine learning to adaptive transactional memory. In *Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing* (2011).

[94] Yoo, R. M. and Lee, H.-H. S. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the 20th annual Symposium on Parallelism in Algorithms and Architectures*, pp. 169–178. ACM, New York, NY, USA (2008).

[95] Yu, H. Combining symbolic execution and model checking to verify mpi programs. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, pp. 527–530. ACM, New York, NY, USA (2018). ISBN 978-1-4503-5663-3. Available from: http://doi.acm.org/10.1145/3183440.3190336, doi:10.1145/3183440.3190336.

[96] Yu, P. S., Dias, D. M., and Lavenberg, S. S. On the analytical modeling of database concurrency control. *J. ACM*, **40** (1993), 831.