UNIVERSITÄT LEIPZIG
Fakultät für Mathematik und Informatik
Institut für Informatik

Operation Graph Oriented Correlation of ASIC Chip Internal Information for Hardware Debug

Diploma thesis

Leipzig, August 2006

vorgelegt von:	Große, Michael
geb. am:	05.05.1981
Studiengang:	Informatik

Zusammenfassung

In dieser Diplomarbeit wird ein neuer Ansatz zum Operations-Tracing für die Hardware-Fehlersuche mit Hilfe einer Analyse von verteilten Traces vorgestellt. Dazu zeichnen Traces Operationen zur Laufzeit auf, deren Einträge mit einem Programm korreliert werden, nachdem ein Problem aufgetreten ist. Das Programm basiert auf einer allgemeinen Methode, die Identifier aus den Operationen verwendet. Da die Identifier sich während der Abarbeitung einer Operation ändern und die Traces unterschiedliche Informationen abspeichern, müssen die Einträge transformiert werden, um die Einträge der selben Operation in anderen Traces zu erkennen. Nach der Korrelation der Einträge rekonstrutiert die Methode die Operationspfade mit Hilfe eines Operationsgraphen, der für jeden Operationstypen die Teiloperationen und deren Reihenfolge beschreibt. Durch die Pfade erhält der Entwickler einen besseren Überblick über die Chip- bzw. Systemaktivität und kann dadurch Probleme schneller eingrenzen, um dann mit Hilfe anderer Informationen deren Ursache zu finden. Der TRACE MATCHER implementiert die beschriebene Methode und wird anhand eines Beispiel-Bridge-Chips evaluiert. Dazu wird der Nutzen für die Hardware-Fehlersuche, die Korrektheit der rekonstruierten Pfade, die Performance der Implementierung und der Konfigurationaufwand bewertet. Außerdem beschreiben Leitlinien für das Trace und System Design, wie das Matching mit sorgfältig gewählten Identifiern in den Operationen verbessert werden kann.

Preface

This thesis presents a novel approach to operation-centric tracing for hardware debug with a retrospective analysis of traces which are distributed across a computer system. Therefore, these traces record entries about the operations at runtime, and a software tool correlates these entries after a problem occurred. This tool is based on a generic method using identifiers saved from operations. Because identifiers are changed along the path of an operation through the system and traces record different information, the entries are transformed to find matching entries in other traces. After the correlation, the method reconstructs the operation paths with help of an operation graph which describes for each type of operation the subtasks and their sequence. With these paths the designer gets a better overview about the chip or system activity, and can isolate the problem cause faster. The TRACE MATCHER implements the described method and it is evaluated with an example bridge chip. Therefore, the benefit for hardware debug, correctness of the reconstructed paths, the performance of the implementation, and the configuration effort are evaluated. At the end guidelines for trace and system design describe how matching can be improved by carefully designed identifiers at operations.

Contents

Chapter 1

Introduction

Computer systems are getting more and more complex. In order to deal with their complexity, chips in these systems are built out of modules which are also used among other chips. Additionally an increasing number of functionality is executed in parallel to improve the performance of the systems. On the other side complex tasks like virtualization and recovery become viable and are implemented in the firmware.

Due to this complexity, hardware debugging of developed systems consisting of chips and firmware is more complicated than before. Some time ago, chips were debugged with logical analyzers attached to their pins. Higher clock frequencies and the resulting mechanical problems forced the designers to move the information for debugging onto chips[Tur04]. In high-availability systems this *chip internal information* is automatically captured to dumps in case of an error on customer machines[KBF$^+$04]. This is necessary, because errors may not be easy to reproduce and those machines cannot be used for testing. Therefore, chips contain traces, which record the executed steps in small memories, and checkers, which are latches indicating the state of a certain condition, and registers which are code-accessible memories. This internal information can be read out over side-band interfaces[IEE01] or an assistant processor aggregates them to a dump if a checker indicates that a severe problem occurred and the recovery failed. The dumps are send to the designers so that the problem can be analyzed.

As a result, chips now contain plenty of debug information, but designers need a fast overview when looking at a problem for the first time. Then a certain part of the chip can be debugged further or the problem is redirected to the responsible person. Therefore, in the high-performance I/O chip MBA[1][PH04] in zSeries[2] systems, operation-centric tracing was introduced[BFG$^+$99] to track operations across the chip. This is accomplished at runtime by a central activity monitor using an operation graph, and additionally a tool called OHRMATCH combines information of the remaining parts of the chip after a problem occurred to reconstruct the operation paths. The activity monitor checks

[1]MBA is the abbreviation for Memory-Bus-Adapter.

[2]Trademark or registered trademark of International Business Machines Corporation.

the paths of operations through the functional units in the chip at runtime for inconsistencies with operation graphs describing for each type of operation the valid subtasks and their sequence. The monitor writes an entry to a central trace either for each inconsistency or for all completed subtasks. In addition OHRMATCH analyzes the trace data in a chip dump in order to reconstruct the operation path from the subtasks in the central trace to the chip interfaces by matching the trace entries of one operation together.

With these information, problems caused by the user, the interaction between the firmware and the hardware, or in the chip logic can be debugged faster. For example, operations might get timeouts while sending them to another chip because the firmware is retiming the link at the wrong point in time. In addition during the power-on of a prototype a user might forget to plug in a cable, so that all operations are dropped at that link. Due to the operation-centric tracing which focuses onto the operations instead of the functional units, the designer notices such problems faster. Hence, using the reconstructed operation paths, a designer can recognize operations where the response is missing and derive the cause of a problem by patterns in operations, or can look then into other chip internal information.

But the information in one chip is not sufficient anymore for debugging. Due to the modularization of chips, the context around functional units and chips becomes more and more important. The implementation of the operation graph as finite-state machine in hardware has significant shortcomings. First, a central trace poses a major problem when many parties try to trace, because the probability of write conflicts increase, so that the written entries can be invalid. The order of the entries might not reflect the order of events in the system if the information has to travel different distances through the system. Additionally, the number of parallel trackable operations is limited by the number of implemented finite-state machines. Because current systems support more outstanding operations, costs in terms of chip area increase significantly for this approach. Consequently, traces are distributed at important places across functional units in chips of a system recoding the passing operations without an activity monitor. Thus, the operation-centric tracing, which reduced the debugging effort, has to be implemented in software by correlating the traces.

OHRMATCH is not sufficient to reconstruct the operation path from the data in the traces distributed across the system, because the tool is highly chip-specific and uses hard coded information about the structure of the chip, the operations, and assumptions about the operations. Therefore, a generic method is required which correlates the trace entries with each other in order to reconstruct the operation paths from chip internal information. Similarly to OHRMATCH, this method must provide a condensed view at executed operations and their paths through several chips in the system. Therefore, the user has to specify the paths for different operation types through the system. The trace entries should be correlated with information from entries like addresses and identifiers created by the system. Additionally the transformation of operations between traces, done by functional units, need to be described. This work should address ambiguities caused by duplicate identifiers as well as the behaviour of I/O operations like resends caused by high system utilization, and splitting of operations with large data blocks into many smaller ones and their later merging.

The method should be implemented in Java and integrated in a new debugging framework called DEBUG TOOL which is accessed by an interactive front-end. Hence, the implementation of the method, which is called TRACE MATCHER, needs to be responsive and take at most 10 seconds for a correlation. Additionally the result of a correlation has to be understandable and the implementation must be robust enough to deal with duplicates. The work should contain an evaluation of the effectiveness and the performance of the method and their implementation. Configurations for the TRACE MATCHER have to be described in a way, that designers can write them for chips and easily reuse them for system-wide analysis. The requirements and limitations for the method should be described, and recommendations for trace design should be given.

1.1 Structure of this work

This thesis is structured as follows. The first chapter presents basics about traces, operations in the system, time and order of events, identifiers, and operation types which are required to understand the following chapters. In the second chapter the state of the art in event correlation is evaluated with respect to a set of criteria derived from the problem description. After that, first the basic ideas of the new approach are described, and then the algorithm is presented in detail including a complexity analysis. The fourth chapter contains the implementation of the new approach called TRACE MATCHER. In the next chapter the results of the work are presented with the necessary steps to create configurations, run the TRACE MATCHER and to interprete the results. The correctness of the results is analyzed and the performance is evaluated at several test cases. The last chapter highlights problems which arose during the work, presents rules for the trace design to reconstruct the operation paths, as well as limitations of the approach and a conclusion.

Chapter 2

Basics

In this chapter basics will be introduced, which are necessary to evaluate existing approaches and to understand the further work. First traces are presented in detail, which are the main data source for reconstructing the operation path with the used method. Therefore, different kinds of traces, their features, and information which is usually traced are outlined. After showing this rather local perspective, we will go on to a chip and system-level analysis of operation paths in the machines. In this analysis sample paths of operations through the system and their representation in the traces are outlined, and structural primitives are derived from the trace placement along these paths.

Then different time concepts are described, which are used in the real world and computer systems to keep track of the order of events. After highlighting their limitations, identifiers will be presented, which are the foundation for the generic method. Therefore, their quality is analysed, scopes are introduced, and different primitives for combining those identifiers are described.

Finally, operation types are presented, which are used to refer to different operation paths through the system. Since many different protocols are used in systems, abstract types are introduced to capture types at a system level so that operation paths can be described.

2.1 Traces

Traces are used in both software and hardware to debug problems. They provide a detailed record of the steps a system has performed during execution. When an error occurs, traces get stopped so that the they contain the steps executed before the error. In contrast,to other chip internal information, the hardware traces do not provide a snapshot of the chip state but information about a period of time. This information is very valuable for debugging, for example to see operations in the interaction between different chips. Thus, one can find operation patterns to be executed for reproducing the problem. The traces must be simple enough to not pose a problem itself during debugging. Additionally the traced information should help the designer in different problem scenarios.

Software traces are created by special trace statements which are spread over the entire code between functional code. These statements trace the position in the control flow, for example the executed function or the chosen branch after a condition. Furthermore these traces contain values of variables which might help to track down the problem. All these traced information is usually printed out to the screen or into a file.

Hardware traces are implemented as cyclic memories which wrap when a certain number of entries has been written. At the rising edge of a trigger signal a new entry is written to the trace, where the signal is usually selecting from several input signals with a multiplexer. Since the hardware traces cannot be changed after chip production, these traces provide different modes which record different information. These modes are carefully designed according to the predicted problem scenarios. The format, for example, for debugging the internal logic of functional unit is different from the information needed to track an operation through the system. The memories used for storing a trace, which are also called trace arrays, are expensive in terms of chip area and power consumption because such arrays usually reaches up to 1000 cells or more. As a consequence traces are used sparingly only at important places in the system and are restricted in size. So the depth of a trace – the number of entries – and the width of each trace entry in bits are limited compared to software traces. Additionally, large traces take a longer time to read out. On the other side a large trace can contain the information one might need to debug a complex problem.

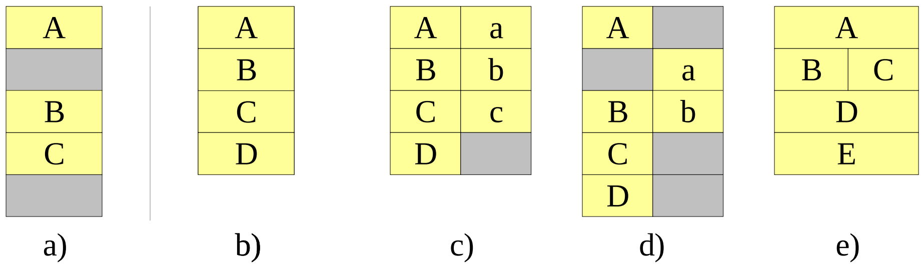


Figure 2.1: One cycle trace and four different event trace formats.

Traces are distinguished in *cycle* and *event* traces shown in the examples a and b in figure 2.1. The first kind triggers in each clock cycle and the second one triggers only if an event occurs, for example a valid signal rises on a bus. Therefore, the cycle trace is restricted to a fixed duration by its depth, for example a 64 entries deep trace will trace 64 ns on a chip running at a clock speed of 1 GHz. Additionally these traces contain equal or unused entries if the state of the traced unit does not change as shown with entry two and five in trace a. In contrary event traces contain a history of events, for example of passing operations, which are also called *operation history register* or short “OHR”. The history of the event traces can reach up to the microsecond range depending on the workload.

Both kinds of traces have their advantages. With a cycle trace the designer can be used to make timing and performance measurements, and to track the utilization of different logic parts. On the other side the trace shows only a short picture of the activities in the chip. Event traces on the other side provide a deeper history of the activities in the chip, but cannot show utilization of different logic parts. That is why mixed forms of these traces have been developed which perform a run-length encoding over the entries in cycle traces or add timestamps from a PLL[1] to an event-trace. Additionally different kinds of traces and setups are used throughout the chip to get a complete picture for debugging.

Another important aspect of traces is the timing. Especially in real-time and high-performance systems, traces should not change the timing of the traced logic. This non-intrusive tracing is possible if the data for the entry is stored in staging latches until all information is present and the entry can be written out. In most cases only the order of these entries is relevant so that this little delay is not a problem.

Traces found in I/O chips are usually operation history registers which write an entry when an operation passes. However, those traces can have different formats shown in examples b to e in figure 2.1. Trace b, for example, is a classic event trace whereas the next two traces are split according to the direction in which the operations pass the trace. Operations heading in different directions are named with uppercase and lowercase letters are triggering the trace independently in example c. In example d an entry is written if an operation in one direction is passing the trace. In the last example the trace stores information about two operations in one entry and similarly one operation across could be traced in two entries. Thus, there are quite different trace setups in the system.

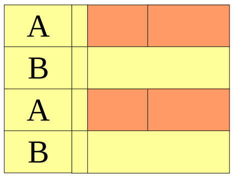


Figure 2.2: Different traced attributes depending on the operation type.

Additionally the logic in a functional unit traces different information depending on the operation type in order to to use the limited memory efficiently as shown in figure 2.2. So beside the type, unique and control information as well as sometimes data from the operation are stored in the trace. Unique information, for example, are identifiers for the sender and recipient of the operation and for this specific operation. In contrary control information are checkers and error bits or a checksum indicating the state of the operation and of their payload. Sometimes the space in the trace is so limited that only a *part of an attribute* is traced. For example the logic might trace only the lower or upper address bits.

[1]PLL is the abbreviation for phase-locked loop which can be used to generate a clock signal from a lower-frequency signal like a reference crystal.

2.2 Operation paths

After looking at single traces, an understanding of trace effects at the system scope is necessary. Therefore, the relation between executed operations, traces, their entries as well as global effects due to the retrospective analysis need to be investigated.

Two example operation types will help to understand the paths of operations through the system. A DMA[2] fetch is an operation which reads data directly from the memory to store them on the devices without utilizing the processor. These DMA fetch requests are issued at the channel and go upbound through the bridge, the hub and the CEC[3] to the memory. If the data is found in the cache, it is sent back the same way downbound as part of a fetch response. In the other case the data is first fetched from the memory and then sent back. In contrary the Sense/Controls are issued by the processor to read or set registers in functional units of the I/O subsystem, for example in the bridge and flow from the CEC downbound through the hub to the bridge. If a response was requested, the response goes upbound to the CEC on the same path.

These two example operations highlight two properties of most operations like out-of-order processing due to caches at the memory and the request and responses structure. Below all properties are described which are relevant for the tracing and the later operation path reconstruction.

1. *consist of a request and in most cases a response*
   The issuer of operations, like a fetches of a memory block, receives a response either with the data or a failure notice. In contrast, unreliable operations like heartbeats, timer pulses and broadcasts require no response.

2. *processed in or out of order*
   The system can process the operations in order or out of order. Unordered processing is used in several applications, for example for control operations passing regular operations, to send two operations over a link. In addition, a newer DMA fetch operation accessing cached data returns earlier than an older fetch accessing uncached data.

3. *resend of operations if recipient is busy or as recovery strategy*
   An operation is resend when the recipient was busy in the first try, for example due to a busy link or a processor with a task running at a higher priority.

4. *operations can be split and merged*
   Protocols are optimized or configured for a special application to reach better latency or throughput. Thus data on the operation might not fit in one operation of another protocol due to the

[2]DMA is the abbreviation for Direct Memory Access.

[3]CEC is the abbreviation for Central Electronic Complex containing multiple processors, chips for accessing the memory and caching data from the memory.

MTU[4]. In this case the data is split and sent out in several operations each smaller than the MTU. The data is then merged on the remote site when all parts are received and sent further as one operation.

These operation properties are reflected in the traces as well, since the entries are constructed from information of passing operations.

As stated in [BFG$^+$99] “operations [...] [are] typically composed of several subtasks which are executed sequentially” at several function units. For debug the tracing captures the execution of these subtask by adding operation history registers in the functional units along the path of the different operation types. Hereby, the functional unit writes an entry to the operation history register during the execution or at the completion of the subtask. So later the trace entries can be used to reconstruct the path of the operation through the system.

After analyzing the traces and their placement in the functional units across systems, the following primitives shown in figure 2.3 have been discovered. These primitives describe a set of source and target traces where entries in the target traces are only caused by operations which have also passed the source traces.

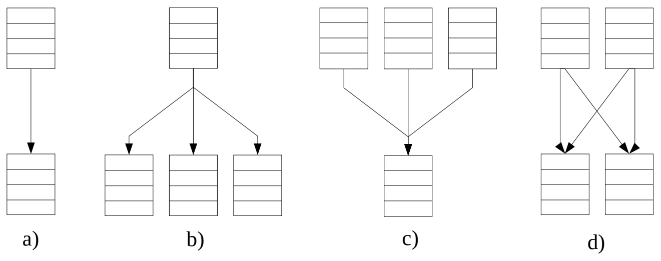


Figure 2.3: Operation path primitives.

The four primitives are called *direct*, *fan-out*, *fan-in* and *fail-over*. The direct primitive connects two traces with each other where the entries in the target trace are caused only by the operations passing the source trace. In constrast the fan-out primitive represents one source trace with many target traces. This primitive is used, for example, at arbiters or demultiplexers. The fan-in primitive is the counterpart for the fan-out primitive with many source traces and one target trace used for multiplexers. The most complex primitive is called fail-over connecting many sources to many targets and is for example used to represent traces at multiple ports of a chip with more than one driving functional unit. With these primitives at hand, all trace connections found in the system can be described.

In contrary to runtime analysis of information where all data is current, the traces contain information about a period of time where these periods might not be the same across all traces. Therefore, the traces must be stopped at the same time, so that a consistent picture of the activity in the chip or

[4]MTU is the abbreviation for maximum transmission unit describing the maximal size of a packet or an operation sent across a medium.

system emerges during the analysis. These trace stops occur only at nearly the same time because the stop event after a problem must propagate over the chip in a few cycles. Usually this is not a problem because idle cycles occur between the passing operations and the picture exactly at the chip stop is captured by other chip internal information. As a result, the information from each local history shows the events before the error which is later used to reconstruct the running operations. When the traces get stopped, some operations might be on their way between the traces. In contrast,other operations might cause entries after the stop event was sent due to the propagation time of this event.

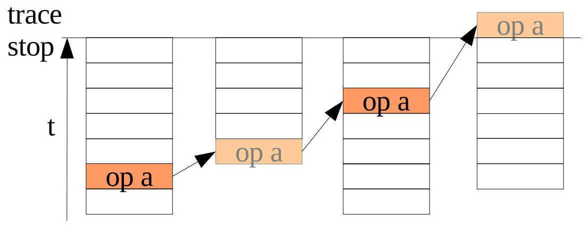


Figure 2.4: Traces along an operation path.

Because traces are cyclic memories, each trace wraps at some point in time and old entries are overwritten with new ones. Thus, no information about old passed operation can be found in a trace anymore. Figure 2.4 shows traces along the path of an operation where the second trace has wrapped, but all the first and the third trace contain an entry for operation a. These differences between traces are caused either by operation paths split in several directions, for example at the fan-in and fan-out primitives, or by different trace depths. The last trace shown in the figure does not contain an entry about the operation because the trace stopped before the entry was written.

2.3 Time and Order of Events

The order of events, describing which event happened before which others, captures the causality between events. This causality between events is widely used in computer science to ensure consistency in replicated databases or liveness and fairness in distributed computations. Because traces distributed across the system contain entries about the passing operations, and each entry can be interpreted as an event, the operation path can be expressed by the order of events. Plenty of research[TvS02] has been conducted about the causal precedence relation which captures causality between events. Different notions of time are used to reconstruct this relation and will be outlined later on.

In daily life people use watches displaying a *global time* to determine which event happens before which others by assigning times to them. As Raynal stated[RS96], the order of events cannot be captured by unprecisely synchronised clocks when the events occur at a high rate. Beside that not all functional units writing to traces have a built-in physical time like a PLL. So global time is not usable to track events in hardware, since the chips easily reach clock frequencies higher than 100 MHz and entries can potentially be written in every cycle to the trace.

Lamport proposed *logical clocks* in 1978[TvS02], because one typically wants to capture the order of the events in a distributed system and not the precise time of these events. So it does not matter if one system is ahead of other systems, as long as all agree on the same time. In contrast,to physical time, logical clocks are only updated when events happen in a part of a distributed system and these changes are propagated in messages with piggybacked information. Depending on the consistency requirements and the knowledge about other clocks in the system, several approaches with different data structures and synchronization protocols have been developed [RS96]. With *scalar time* developed by Lamport, the logical timestamps of two events cannot be compared with each other to determine which event causes another one. Because scalar time can only determine which events timely preceded an event, this notion of time cannot capture the causality precendence among events and cannot be the basis for an operation path reconstruction. The later developed *vector time* fills this gap by keeping the local and global clock information separate, so that by comparing the timestamps the causality can be obtained.

In order to reconstruct the operation path, the order of events using the logical clocks, vector time must be used and the vector timestamps need to be stored in the traces. The functional units with their traces, equivalent to distributed system without a global memory, communicate only by sending signals as messages to other units. Each functional unit would then update the vector timestamps according to the rules described in Raynals work[RS96]. These approach has two significant shortcomings. First, with each sent message the vector timestamp containing the local view about timestamps of all units in the system needs to be transmitted. For example, assuming a system has ten functional units, ten additional values would need to be transmitted to another unit in order to maintain the vector time. The vector timestamps may be reduced with other more sophisticated approaches, but a significant overhead will remain which results in additional logic so that the method would not be applicable soon. And second, these timestamps need to be stored in traces, increasing their size considerable. Additionally their might occur problems with wrapping timestamps due to the size limitations of the traces.

All those approaches which keep the order of events at a system level with time like physical clocks in form of PLLs or logical clocks are not applicable. At a local level between two functional units, that use the same physical clock the situation might be different because synchronization problems decline, but these cases are only “special” cases and are not the foundation for an operation path reconstruction.

2.4 Identifiers

This section will present another potential foundation for operation path reconstruction based on identifiers instead of time. An identifier tries to "establish the identity" based on the "sameness of essential or generic character in different instances"[Mor93, p575]. Therefore, an operation has attributes which establish the identity and are also stored in the traces. Due to the described character, different operations cannot have the same identifier value and so falsely perceived as being identical.

Identifiers can be distinguished in two kinds - artifically-created and user-created identifiers. *Artificial-created* identifiers are created by the system and are designed to be unique, for example sequence numbers in connections, identifiers for connections or tags for outstanding operations. Therefore, these identifiers work for all operation types and at each workload if they are well-designed. Second, *user-created identifiers* are not unique by creation, but have a character of uniqueness due to the distribution of the values. For example, the variance of the values of these identifier is high but some operations might be falsely identified as the same. Addresses in memory accessing operations might contain a high variance if the processes access different parts of their assigned memory areas. But these identifiers have a serious weaknesses. They are not designed to be unique so that they work only in special cases, for example only with certain operation types or at a special workload. As a result, artificial identifiers are the identifiers of choice from now on.

The ideal way to accomplish the goal of unique identifiers for different operations is a *global unique identifiers* also called GUID. GUIDs are identifiers which are unique at all times and at all places in the system. As a result, different operations would be completely distinguishable by their identifier. Because no identifier gets invalid, the identifiers get potential infinite over time. Unfortunately, they are not storable, even not creatable and so not implementable. That is why in real world limited identifiers are used which can be created and stored without problems.

Limited identifiers have a limited range and are only unique in a certain part of the system during a certain period in time which will be called scope from now on. For example, tags at outstanding operations are used to refer to these operations on the response side after sending the request. These tags are only valid on the path between sender and recipient until the response is received or a timeout occurs. The *scope* describes time period and places in which the identifiers are unique, similar to the scope of variables in programming languages.

The scope of identifiers can change over time due to changing requirements in the protocols. In the TCP protocol sequence numbers are used for the sent packets which need to be acknowledged. During the time technology improved and long-distance low-speed connections to space and high-speed backbone connections between large cities or continents became possible[JBB93]. Therefore, the sequence numbers attached to the packets wrap on a 1 GBps connection just after 17 seconds while the timeout for the acknowledgments from the communication partner was recommended to be set to two minutes. On the other side the low-speed traffic legitimates the two minute acknowledgment timeout. As a result, the sequence numbers were extended in the protocol so that the sender must not wait until the packets were acknowledged in order to prevent a sequence number wrap.

The scopes can also change in one system so that the identifier needs to change as well. These changes at the identifier are reflected in the combination of several identifier attributes in an identifier tree shown in figure 2.5. This identifier tree is a simplified descision tree[RN03], which is used to compare the attributes of two trace entries to determine if they were caused by the same operation. In the simple case, for example, the values of attribute A are compared with each other and in the extended case the values of B are used if the values of attribute A are equal. Likewise attribute C is used in the alternative case during the comparison if the attribute B was not found in both compared entries.

The primitives used to build these trees are *extension* and *alternative* which can be combined as shown in the figure 2.5. For example in the second tree, the attributes B is extended with attribute A in order to distinguish operations at a single port from the ones at other ports in an arbiter. Additionally two different paths of operations might join at a part of the system as shown in the third tree and a condition like A is needed to refer to either the identifier of the first or second type. Furthermore the identifier of an operation might get replaced by another identifier or removed because they are invalid at this place in the system This case is already captured by the primitive extension and alternative. By combining these three primitives, identifiers can be built which have a greater scope. Ideally the combined identifers are unique across each entry caused by a specific operation in the system.

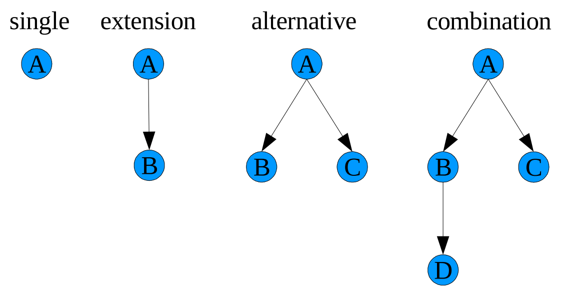


Figure 2.5: Identifier trees combining identifier attributes to compare entries.

The actual comparison of two trace entries starts at the root attribute in the identifier tree. Two entries are equal with respect to the identifier tree if the currently visited identifier attribute is equal and no attribute descibed in the children below is present in both entries. If the attribute has children in the tree, the first of these attributes is used for the comparison. When an attribute in the tree is not present in both entries or the values are not equal in both entries, the next sibling of the attribute is compared. Thus, several attributes can be used to compare two trace entries with each other.

2.5 Operation Types

Operation types distinguish different kinds of operations, for example DMA fetch operations from Sense/Controls operations. On their path through the system an operation is transformed between different protocols. Especially mainframes, with legacy devices and new devices, support a wide range of protocols to connect devices to the system where each of these protocols has grown on its own and has a different representation for each operation type. Additionally these operation types can describe different paths of operations based on the part of the system where they are used.

That is why an abstraction layer above the different representations of each operation type and the usage context is required, to be able to track operation paths through the entire system. These abstract operation types subsume the different protocol-specific operation types and are used to describe the different paths of each operation type around the entire system. In contrast,to the protocol-specific types which are usually based on the notion of sender and receiver, these abstract types are unique at a system level. For example, a DMA fetch operation would have entries in the sending trace of the bridge chip and the receiving trace the connected hub chip. Hence, the different types in both traces must be decoded to the same abstract operation type. Additionally the names of these abstract operation types must be chosen carefully in order to prevent misunderstandings with protocol-specific types.

The protocol-specific operation types must be mapped to abstract operation types. This mapping is distinguished in different complexities from simple over context-independent to context-dependent. In the *simple* mapping, the abstract type can be calculated by a one-to-one mapping from the protocol-specific type. This can be done with a table assigning to each operation-specific type an abstract type. In the *content-independent* mapping, several attributes in the entry are combined to determine the abstract operation type. For example, beside the type attribute in the entry an additional attribute might indicate if a response was requested. The *context-dependent* mapping is the most expressive one because it uses additional context information beside the several attributes to determine the abstract type. Currently this context includes the name of the chip and the trace, so that operation types in a functional unit used in different chips can be distinguished or the responses going either upbound or downbound can be mapped to different abstract types.

Chapter 3

State of the art

In this chapter the state of the art in information correlation will be presented. Correlation refers to the statistical notion where the similarity of two objects is described with a certain probability. In most cases in computer science, the developers want no fuzzy but safe results where two objects are only returned if they are correlated with 100% probability. This type of correlation is called matching. Several ideas or approaches will be analyzed to assess their applicability to the operation path reconstruction based on trace entries. Therefore, event correlation used in network management, string matching as a basic algorithm from introductory computer science courses and the current trace matching tool OHRMATCH are outlined and evaluated.

In order to evaluate the applicability to the reconstruction, criteria are needed to recognize the strengths and weaknesses of the approaches. Based on the problem description and the chapter 2, the following criteria covering non-functional, structural and entry-specific aspects have been chosen for the evaluation. The *generality* demanded by the problem description ensures that the approach is applicable to a broad range of chips and systems. Similarly, the *extensibility* ensures that later changes can incorporated without larger problem. The *understandability* is necessary so that a large range of users with different knowledge can use the method and correctly interpret the results. Additionally the approach must also cover all *structural primitives* like fan-in or fail-over found in the trace structure of I/O subsystems. Because the trace data of customer machines is only accessible after a problem occurred, the approach must support a *retrospective* reconstruction of operation paths based on the trace entries. The approach must deal with incomplete information due to *wrapped traces* in the correlation. The operation properties cause additional criteria like proper handling of *out-of-order* processing and of *one-to-many* relationships between entries. These relationships are necessary to express operation resends as well as the splitting and merging of operations.

In the management of complex networks the event correlation, also called alarm correlation, is used to condense information so that the root cause of a problem can be determined more easily [Tif02][Lew99]. This is particularly important for growing networks with an increasing number of events caused by

Criteria	Event correlation	String matching	OHRMATCH
Generality	ok	ok	fail chip specific, only one operation type
Extensibility	ok	ok	fail not modular enough
Understandability	ok	ok	fail understanding of heuristics needed
Structural primitives	ok	fail fail-over not possible	ok
Retrospective analysis	fail only runtime	ok	ok
Wrapped traces	fail	fail search text not definable	ok
Out-of-order	ok	fail strings are linear	ok
One-to-many/ Many-to-one relations	ok	fail only one-to-one mapping supported	fail

Table 3.1: Comparison of different approaches

attacks or failed components. An event is hereby a piece of information dealing with a problem for example a fault or an intrusion in the system. These events are created at several components of the systems like routers, switches or computers and are stored in log files or are transmitted to the correlator. The event correlators try to determine the causal relationship between the events to see which event caused another one and to condense simple events to more complex aggregated events. Many different approaches have been developed to correlate events based on dependency graphs between functional parts[Gru], rules and reasoning, models and reasoning, codebooks as well as learning approaches[Tif02]. Contrary to these approaches the generic method must support an easy way to deal with incomplete information due to wrapped traces. These approaches rely either on a global time or analyze data at runtime. But time is not an option for the trace matching and a runtime analysis is not possible at customer machines.

String matching tries to find a search text in a larger piece of text and is used for searching and filtering in computer science[Sed93, pp329]. The strings itself consist of a sequence of characters where the same character can occur multiple times. Traces can be interpreted as strings if the contained entries are interpreted as characters with a special equality relation which needs to be defined on the identifiers used in the trace entries. Hence, matching algorithms get applicable to the problem. Therefore, one of the popular string matching algorithms with linear time complexity in the worst-case like Knutt-Morris-Pratt or Boyer-Moore can be used. String matching supports only one structural primitive -

the direct connection - because normally only two strings are compared with each other. The fan-in and fan-out case can be covered by extensions of these algorithms yielding the same worst-case complexity class as standard matching algorithms. In contrast,the fail-over case and out-of-order processing cannot be covered, due to the linearity of strings. In addition the string matching cannot deal with wrapped traces, because the search text cannot be defined easily. So the matching provides just a solution to a part of the problem and not general solution to the operation path reconstruction.

In the introduction the existing tool called OHRMATCH was mentioned. This tool matches the entries of in all traces of one chip and returns as result the corresponding entries in other traces and whether a response was received for the request. The matcher is a script written in ReXX which has grown over time to 22.000 lines of code, however it is chip-specific and is difficult to extend. The tool runs on pre-processed data and can be started in different modes to improve the matching quality. It is usually started at a deep trace which should contain entries about most operations found in other traces. Then the matcher tries to find corresponding entries in all preceding traces by comparing hard-coded identifiers at each entry. Therefore, the order of entries and chip-specific assumptions about the operation order are used to match with relatively high confidence in most cases. As a result, of the assumptions, the corner cases are not covered well. The tool is mainly used by the developer, who knows how to interpret the results of the matcher correctly, and how to tune certain parameters. Because the heuristics and the function must be comprehended first, the understandability is limited.

Hence, no approach fulfills the requirements. Thus, a new generic method must be developed based on some ideas of the approaches in order to satisfy the requirements.

Chapter 4

Approach

After explaining the necessary basics, this chapter presents the developed generic method to reconstruct the operation path based on trace entries. At the beginning the basic ideas behind the approach will be highlighted. These major design decisions help to understand the details in the further sections.

Since the method can be applied to single chips, multiple chips or a entire subsystem, the method need to be configured for the reconstruction. This configuration consists of transformation functions, several logical traces, a transformation graph and an operation graph presented in the next section. After this the matching algorithm will be described in detail. At the end several parameters will be introduced in order to analyse the time complexity of the algorithm for different configurations.

4.1 Basic ideas

The reconstruction of operation paths is based on identifiers in those operations which are part of the entries written to the traces. The identifiers are used to find matching entries of the same operation in adjacent traces. Because these identifiers are only valid during a certain period and region in the system, they must be designed to be unique in the adjacent traces and the traces must be stopped at nearly the same time. Additionally each trace must trigger at the same event when a new operation passes the trace.

Traces have a totally different format so that *logical traces* were introduced as an abstraction. Due to this abstraction several traces can be created from one trace depending on the trace format, entries about the same operation are combined together and the attributes are extracted from data in the traces.

Because traces can wrap and contain different attributes, transformation functions map the attributes of the entries in the source trace to the one of the target trace. After transforming the identifier attributes, identifier trees are used to compare the transformed source entry to the target entry, because different adjacent traces can use different attributes for the comparison. Therefore, the trees describe which attributes of the entries can be used to compare the entries with each other.

Traces usually contain 32 up to 256 entries so that a search starting at each entry is infeasible because an operation passes at average 10 traces in a single chip. So the method uses *no backtracking* and is done *stateless*. Therefore, no information gathered by earlier matches can be used for further matching in the operation path and wrong matching descisions cannot be revidated later on. Additionally the algorithm must be split in two steps, the *matching* part and the *operation path reconstruction*. During the matching, entries of the same operation are linked together so that the paths can be reconstructed later on.

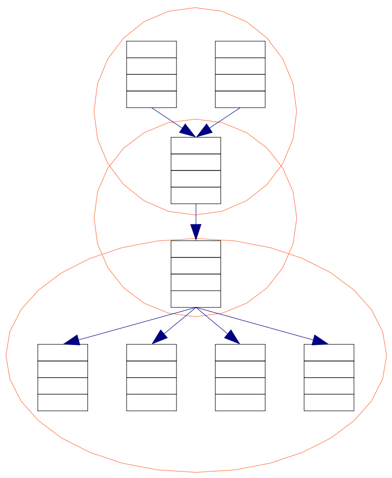


Figure 4.1: Local matching of entries in each locality.

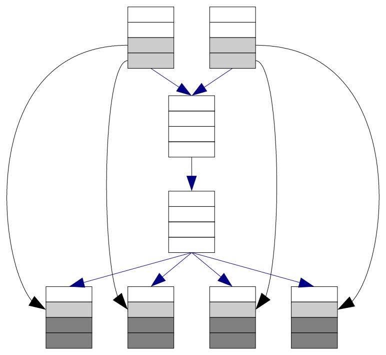


Figure 4.2: Global matching of the unmatched entries across wrapped traces.

Based on the stateless approach, the matching is further split in a *local matching* and a *global matching* step. The local matching shown in figure 4.1, links entries in the localities described by the red circle together. As a result, not wrapped entries are linked together. During the global step in figure 4.2, a search for corresponding entries is done for each remaining grey unmatched entry across already wrapped traces. The number of unmatched entries is usually much smaller than the number of trace entries, so that the branching factor and the time complexity are smaller for a search across all entries.

After the two matching steps the linked entries are checked against the operation graph, and paths are build based on the graph. The remaining dark grey entries stay unmatched because no other trace contains that old entries or the search depth.

4.2 Configuration

The old OHRMATCH use hard-coded information about the system structure and the trace format. In order to create a generic method, a configuration must contain every piece of information used to reconstruct the operation path. Therefore, the developed method requires several input data structures from logical traces over transformation graphs to operation graphs and functions to transform between entry representations in the traces. As a result, the approach is applicable for matching traces in a single chip, a subsystem or the entire system.

4.2.1 Logical traces

Logical traces have been introduced to abstract from the physical traces described in section 2.1. These logical traces decouple the trace matching algorithm from the formats of the physical traces and the structure of the traced data. Additionally the transformation and operation graph are later defined based on the logical traces.

Therefore, the traces contain only entries caused by operations heading in *one direction*. For example, the physical trace d in figure 2.1 is split into two logical traces where one contains entries caused by operations heading to the processor and the other one contains operations in the direction of the devices. So the path of operations heading in certain directions can be later represented by a sequence of logical traces.

Contrary to physical traces found in hardware which contain entries in form of binary data without meaning, logical traces contain entries consisting of *attributes*. These attributes are name-value-pairs where the values can be of any type, but are usually are numbers or strings, for example an address or the name of an operation type.

Traces can start and stop recording passing operations so that an unpredictable period of time can pass between the traced operations. Therefore, the limited identifiers in the physical traces might have the same values because they traced during two different periods where the same identifier can be reused. In contrast,the logical traces contain only the entries since the last trace start event.

4.2.2 Transformation function

A transformation function transforms entries from a source logical trace in a way that they look like entries from the target logical trace. Additionally it describes which identifier tree - as presented in the identifier section 2.4 - must be used to compare two entries. For the matching the transformation of the attributes in this tree need to be described.

The function must abstract from the *logic between the traces* and also from the information written to the trace as shown in figure 4.3. For example, the logic in a functional unit can transform operations between two protocols which have different attributes and also identifiers. This unit transforms the source and target identifier of an operation according to the its translation table. So the traces on the operation paths before and after this unit contain entries, which have different attributes. The transformation function must transform those identifiers found in the source trace to the ones found in the target trace. This is accomplished by implementing a part of the chip logic which changes the identifiers, and uses other chip internal information like the translation table or registers.

The function represents which *information the logic writes to the trace*. Beside the changes at the operations from the logic, traces might record different information depending on the designers needs. For example, one trace can write the complete identifier in each entry and another trace just writes a pointer to an additional register in the entries. Since the matching relies on the information in the traces, these changes at the identifiers need to be transformed in the transformation function as well.

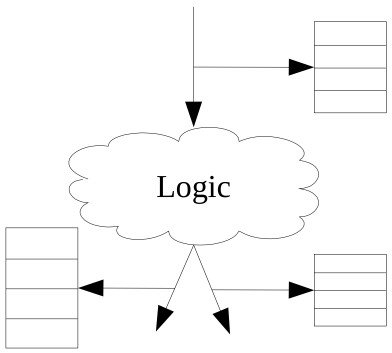


Figure 4.3: Transformation function abstracting the logic between and the format of the traces.

On the path of an operation through the system some traces have wrapped already. In order to combine the path fragments found before and after these wrapped traces, the transformation functions must be *transitive*. So the functions can be applied one after another, acting virtually as one function between the non-wrapped traces. For example in figure 4.4, two functions f and g can be applied one after another to an entry in trace A so that the transformed entry can be compared to the entries in trace C. The functions need to be transitive with respect to the identifier attributes so that no identifier attribute is lost by applying several functions in sequence.

Depending on the represented logic and the traced information, the functions vary in complexity from

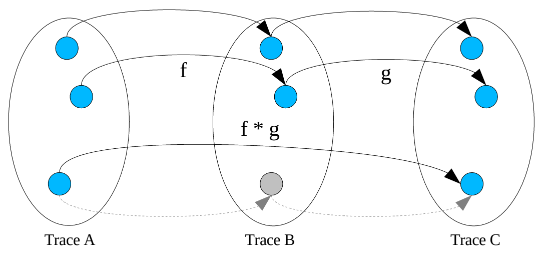


Figure 4.4: Transitive combination of transformation functions.

an identity function, over functions using conditions for the mapping, to functions using chip internal information like registers values. In most cases a identity function is sufficient when the identifier attributes are stored in both traces. Some functions need to transform entries, for example the request type can be mapped to the response type at the border of the considered part of the system. Therefore, conditions are used to deal with the different cases. More complex functions may even use register values or translation tables to transform identifier attributes or to expand some bits of them.

The transformation function between a source and a target trace need to described the *forward* and *backward* transformation of entries. So even not injective functions can used for the entry transformation. Thereby the transformation of entries with the direction of the edge is called forward and the other way around backward.

4.2.3 Transformation graph

The transformation graph describes which traces exist in the system and between which traces the matcher can transform with help of which transformation functions. A transformation graph is a directed graph whose nodes are logical traces which are connected by edges if a transformation function exists between both traces. The edges start at a source trace and end at a target trace, and the transformation functions are attached to them as markings. These graphs are *simple* and *transitive* so that between a pair of logical traces at most one edge can exist, and consecuting transformations must not be expressed with an additional function.

To support the reuse of these graphs, the transformation graph is modular. Therefore, each graph module can be instantiated in each another graph module with a instance name as prefix for the graph module. The instantiations must be cycle free so that a module cannot include another module which includes this module again. Each module has ports which are traces without outgoing edges that are not start or end point of a operation path. Due to these ports already defined transformation graph modules can be combined by connecting outgoing and ingoing ports with each other. So once written modules can be easily combined to get a chip level or system level transformation graph depending on the scope needed for the debug. For example, the graph module `bridge` describing a bridge chip shown in figure 4.5 is instantiated in the graph module `bridge-shell` in figure 4.6 with the prefix

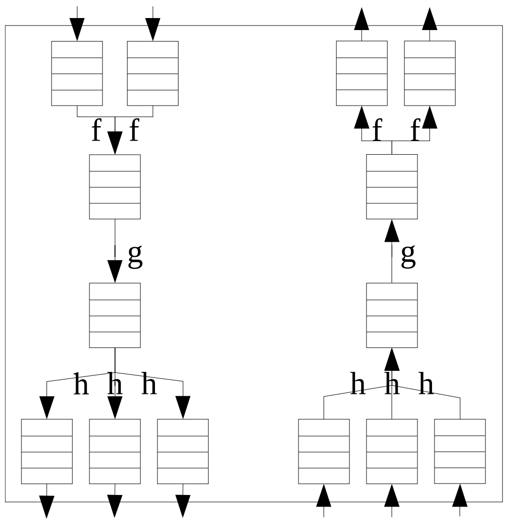


Figure 4.5: Transformation graph module for a chip.

b0. All logical traces defined in `bridge` are accessible with the prefix `b0` in `bridge-shell`. The shells are necessary in order to reuse the definitions. In the second example, two chips are combined for a chip-interaction analysis. Thus, the bridge and the hub module are reused and only a new shell is developed.

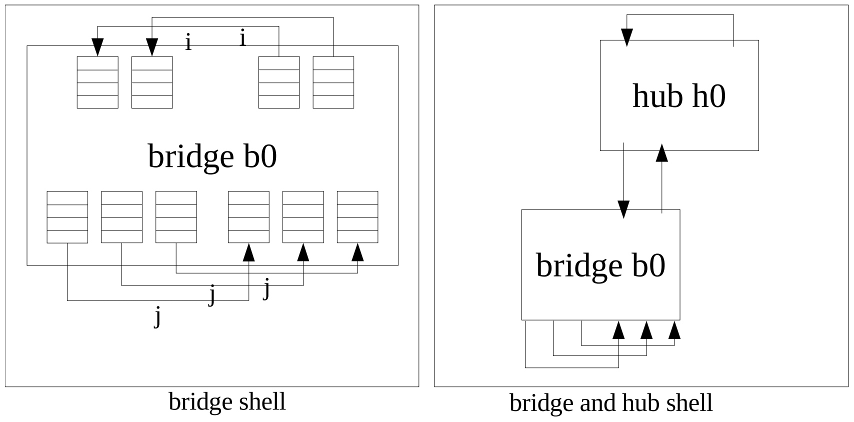


Figure 4.6: Different shells for single chip or interaction analysis between chips.

4.2.4 Operation graph

The operation graph describes for each operation type the subtasks and their sequence. In the old implementation[BFG+99], the subtasks of the operations were described as a sequence of events in a finite-state machine. Those events have been sent out on the completion of a subtask executed in a functional unit. By contrast, the operation graph in the generic method describes the subtasks of the operations as a sequence of logical traces. Since the traces are usually distributed across the functional units in the chips, the logical traces contain an entry about an operation if the subtask was executed. As simplification it is assumed that the subtask is completed when the entry is written to the trace. So each trace contains only entries caused by one kind of event, when a subtask of a passing operation is executed.

The operation graph is a *typed* directed graph of logical traces where the abstract operation types are used as types. An edge starts at a source trace and ends at a target trace and two logical traces can be connected by edges of different operation types. The graph is *simple* and *modular* as the transformation graph described and in addition these graphs are *acyclic* so that for no type exists a logical trace which has a path to itself. Operation resends as well as operation splits and merges are not expressed in the operation graph, but can be handled by special flags at the transformation functions.

The example operation graph in figure 4.7 shows the operation paths for a DMA fetch operation and Sense/Controls operation described in section 2.2.

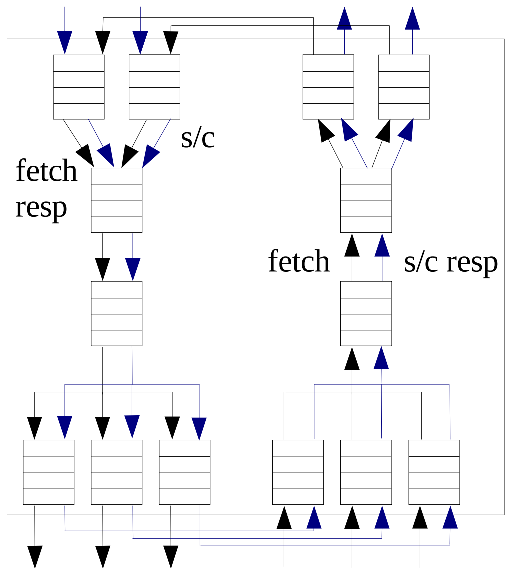


Figure 4.7: Operation graph for the operation types fetch and Sense/Controls with their responses.

In chips usually many request types are mapped to few response types. So one type is not sufficient to express the path of an operation with request and response through the system. Therefore, the graph must support *type conversions* which can connect these two different parts of operation path together. So an edge in the operation graph can have two different types - an outgoing type at the source trace and a incoming type at the target trace. The outgoing and incoming types are usually the same and are only different at the border of the considered part of the system.

The start of the operation path for a certain type is determined by the edges in the operation graph. If no incoming edge to a trace is found for an outgoing edge of a certain type, the operation path can start trace at this trace.

4.3 Algorithm

The matching algorithm uses the described configuration to match the entries, and returns then the reconstructed operation paths and the unmatched entries.

Algorithm 1 Matching algorithm

```
Require: match configuration
Ensure: match result with operation paths and unmatched entries
  match local
  match global
  reconstruct the operation paths
```

The algorithm consists of three steps shown in algorithm 1, two matching steps and a final step where the operation paths are reconstructed from the created links between the entries. The first two steps are called local matching, because only adjacent traces get matched, and global matching where unmatched entries are matched across wrapped traces. In the last step the operation graph and the abstract operation types are used to reconstruct the operation paths from the linked entries. These paths can then be checked for completeness and inconsistencies.

4.3.1 Local matching

The local matching links only not wrapped entries from source and target traces in a locality together. A *locality* contains a source trace and all adjacent target traces plus the source traces for these target traces. For example, each presented primitive is a locality if no other trace is connected from the traces in the primitive. As a result, different trace processing orders can not cause side-effects like different links between entries.

Then all transformation functions which connect a source trace with a target trace in the locality are queried from the transformation graph. Afterwards all entries from the target traces are stored in a single list and a second list is populated with entries from the source traces transformed forward with the functions starting at the respective trace.

Later the identifier trees from each used transformation function get merged so that each entry on the source side can be compared to entries on the target side of the locality.

Algorithm 2 Local matching between adjacent traces

```
Require: logical traces, transformation graph, transformation functions, identifier trees
Ensure: unmatched entries
     while unprocessed source traces left do
 2:    get locality with source traces and target traces
       for all source traces do
 4:      get the entries in the source trace
         apply all transformation functions starting at the source trace to the entries
 6:      add the transformed source entries to the source entries set
       end for
 8:    for all target traces do
         add the entries from this target trace to the target entries set
10:    end for
       merge all identifier trees used between source traces and target traces to merged identifier tree
12:    for all source entries do
         set duplicates to the empty set
14:      while target entries do
           if target entry is unlinked then
16:          if source entry and target entry are equal with respect to the merged identifier tree and
             the masks then
               add target entry to the duplicates sort
18:          end if
           end if
20:      end while
         if duplicates has only one target entry candidate then
22:        link the source entry to the target entry candidate
         else if duplicates has more than one target entry candidate then
24:        mark the entries in duplicates as duplicate
         end if
26:    end for
     end while
```

A *mask* describes which bits in an attribute have been written to the traces, for instance an address is usually not written completely to the trace but only the significant bits representing the memory region. Thus, only these bits can be used for the comparison. Masks are either *locality-specific* when the masks for an attribute are equal across all trace entries in the locality, or *entry-specific* if the masks change as shown on the right side in figure 4.8. In case of the locality-specific masks a linear ordering can be defined on the merged identifier tree with the in-order traversal of this tree. So the mask can be applied to all source and target entries before comparing the entries. As a result, the source and target entries can be sorted in an optimised local matching algorithm, and the comparison is then done in linear time afterwards. Depending on the masks either a standard or an optimised local matching algorithm can be used.

On the other side, the chosen standard algorithm makes no assumptions about the masks and uses

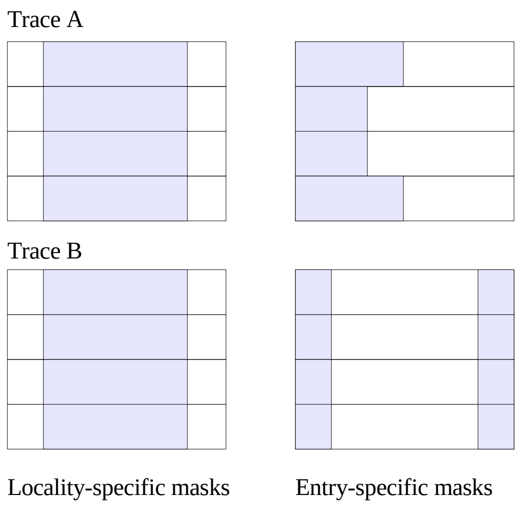


Figure 4.8: Locality-specific and entry-specific masks.

a nested loop to compare each source entry with the target entries. So before each comparison the masks of each identifier attribute must be combined and applied to the source and target entry so that only the bits present in both entries are used for the comparison. The inner loop finishes if all target entries have been compared to the source entry or a duplicate target entry was found. If no duplicate was found, both entries get linked together.

4.3.2 Global matching

After the local matching is finished, several entries are not linked to an entry in the previous or next trace in the operation path. These unlinked entries can be caused by wrapped traces or duplicates. In order to complete these operation paths, the global matching tries to find corresponding entries by searching within the traces reachable from this trace in the transformation graph. Hence, deep traces or traces which contain only a part of the operations act as interpolation points for the matching. The search from the unlinked entry is only done in the directions where the link is missing, for example forward if no next entry is linked from this entry.

As described in the subsection 4.2.2, transformation functions can be applied one after another due to their transitivity property. So a transformed entry which was not found in a target trace, can be transformed again with the another function to the format of the next trace.

Because the format of the traces and of the contained identifier attributes can change along this search, the identifier attributes need to be merged similarly along this transformation. This merged identifier is then used to compare a transformed entry to an entry of the target trace. Therefore, the transformation function needs to define all identifier attributes even if they are not used for local matching because they are only in one of the connected traces. For example `attrB` in figure 4.9 can help to improve the global matching if it is defined in the identifier trees between trace A and B as well as B and C. In addition the transformation functions between the trace A and B as well as B and C must not remove the attribute `attrB`.

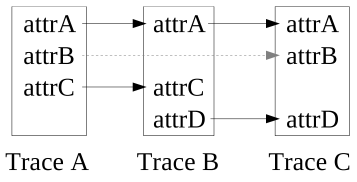


Figure 4.9: Identifier attributes in global matching.

In order to choose the appropriate search algorithm[RN03, pp73], the requirements are used as guideline. All informed or other incomplete algorithms can be ignored because a robust and understandable method is needed for the debug. Since the method needs to be implemented at reasonable cost, space complexity is another important decision factor. Therefore, a depth-limited depth-first search algorithm was chosen to find entries for an operation in traces after a wrapped trace. The algorithm was modified so that it does not return if a match was found but searches further and remembers the best found match.

This algorithm is complete if the search depth is large enough. In addition, it has a space complexity of $O(maximalDepth)$ and a time complexity of $O(|N| + |E|)$ like other complete search algorithms.

Algorithm 3 Global matching of unmatched entries

```
Require: logical traces, transformation graph, transformation functions, identifier trees
    for all traces do
        for all unmatched entries in trace do
            if entry is not linked to a next entry then
                search forward in the transformation graph from this trace for a match with the entry
                if match was found then
                    link the found unmatched entry to the match
                end if
            end if
            if entry is not linked to a previous entry then
                search backward in the transformation graph from this trace for a match with the entry
                if match was found then
                    link the found match to the unmatched entry
                end if
            end if
        end for
    end for
```

During the search multiple potential matches can be found because entries about the same operation are stored in several traces and duplicate entries are caused by operations resends or merges.

$$rank(m) = \frac{1}{depth(m)} \tag{4.1}$$

These potential matches must be ranked with the ranking function $rank(m)$ in a way that entries in near traces are preferred, and duplicates are detected. Equation 4.1 shows the used ranking function. If a match has a rank better than the ones of all other matches, these two entries are linked together. In contrast,candidate entries with the same rank stay unlinked. Due to this behaviour the global matching might not link any entries together if only duplicates are left during global matching because these duplicates get the same ranking.

Algorithm 4 Forward search in global matching

```
Require: entry to search for starting at a trace
Ensure: best match
    set best match to none
2:  for all outgoing traces in from the trace do
        get the transformation function between trace and outgoing trace
4:      merge the passed identifier tree with the tree of the function to merged identifier tree
        get the transformed entry by applying the transformation function to the entry
6:      for all target entries in outgoing trace do
            if transformed entry and target entry are equal with respect to the merged identifier tree and
            the masks then
8:              get the rank for the match
                if rank is better than best rank then
10:                 set the best match and best rank
                else if rank equals the best rank then
12:                 mark the best match as invalid
                end if
14:         end if
        end for
16:     if depth is less than max depth then
            search forward in the transformation graph from the outgoing trace for a match with the
            merged identifier tree and the transformed entry
18:         if match is better than best match then
                set the best match to match
20:         end if
        end if
22: end for
```

4.3.3 Operation path reconstruction

After both matching steps the matching entries are linked together. These links do not reflect the operation paths because no knowledge about the operation type was used during the matching. For example, the last response entries are linked to the first request entries. During the reconstructing of the operation path, which is final step of the generic method, these links must be removed. The figure 4.10 contains the basic cases for the link structures created by the matching steps. The operation path for one type is shown in black edges and the links between the green found entries are shown in grey. In the first case, all traces contain entries about the operation and these have been linked together. In contrary, some traces wrapped already in the second case. For example, the matcher found no entry in the last trace indicating either an incomplete operation or a wrapped last trace. The last two cases c and d show traces with an incomplete request and a response with already wrapped traces on the request side. The operation in the case d is complete even though four traces wrapped already.

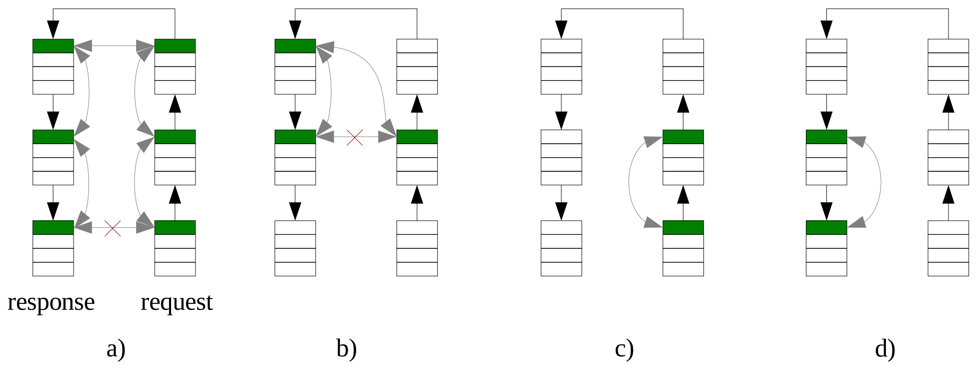


Figure 4.10: Operation path reconstruction from the linked entries.

In order to calculate the operation paths, the oldest entry about operation must be determined. So all abstract operation types *types* are first determined from the linked entries, which might contain not all types as specified in the operation graph shown in case c and d. After that, the algorithm chooses the first type *startType* from the determined *types* according to the operation graph. Therefore, the found types are checked whether the request for a response is present in the linked entries later on. Then the entry *startEntry* with the shortest path to the operation start for this type is calculated.

Now the operation is reconstructed by adding all entries to the path which are linked with the next pointer. Additionally all those entries are marked as done to prevent duplicate entries in two paths and to recognise unmatched entries later on.

Algorithm 5 Operation path reconstruction from linked entries

```
    for all entry in entries do
2:      if entry is matched or not yet in a path then
            get type set $types$ from linked entries
4:          find first type $startType$ in $types$ based on operation graph
            find entry $startEntry$ in entries of type $startType$ with shortest path in operation graph to an
            operation path start
6:          build operation path $path$ starting from $startEntry$ and mark entries
        end if
8:  end for
```

4.4 Complexity analysis

The generic method is configuration so that the complexity calculation must include parameters describing the configuration. Thus, the impact of the configuration at the matching process can be analysed. These parameters are listed below:

$traces$:	The number of logical traces in the configuration.
$entries$:	The average number of entries in a logical trace.
$funPerTrace$:	The average number of transformation functions starting at the logical traces.
$branching$:	The average branching factor in the transformation graph.
T_f:	The average complexity of all transformation functions used in the configuration.
$unlinked$:	The ratio of entries with a missing link to all entries after the local matching. This parameter can not be derived explicitly from the configuration, but describes implicitly the identifier quality and the wrapping of the traces. For example, certain workloads might fit better to the structure and the depth of the traces.
$unmatched$:	The percentage of unmatched entries after local and global matching. This parameter also describes implicitly the identifier quality. For example, old entries from very deep traces cannot be found in any other traces and stay unmatched, or duplicate entries remain unmatched as well.
$maxDepth$:	The maximal search depth describes how deep the depth-limited search will search for matching entries during global matching.

Additionally to these parameters the complexity of the approach is also dependent on the implementation of certain functions which are used in the algorithm. For instance, the comparison of the identifier trees has a time complexity of T_{cmp}. The list below shows the time complexity of all other functions used during matching.

$T_{adjacentTrace}$:	Query for the adjacent traces of a trace in the transformation graph.
T_{merge}:	Merging of the identifier trees.
T_{cmp}:	Comparing two entries with each other.
T_{rank}:	Ranking of match candidates in the global matching.
$T_{opgraph}$:	Query to the operation graph.
$T_{shortestPath}$:	Shortest path calculation from a trace to another trace in the operation graph.

In order to determine the time complexity of the approach, the complexity of the three steps are analysed and added together.

$$
\begin{aligned}
O(locality) &= 2 \cdot traces \cdot T_{adjacentTrace} && (4.2)\\
O(transform) &= traces \cdot funPerTrace \cdot entries \cdot T_f && (4.3)\\
O(merge) &= traces \cdot funPerTrace \cdot T_{merge} && (4.4)\\
O(cmp) &= traces \cdot funPerTrace \cdot entries \cdot branching \cdot entries \cdot T_f) && (4.5)\\
O(local) &= O(locality) + O(transform) + O(merge) + O(cmp) && (4.6)\\
&= 2 \cdot traces \cdot T_{adjacentTrace} + traces \cdot funPerTrace \cdot\\
&\quad (T_{merge} + entries \cdot (T_f + unlinked \cdot branching \cdot entries \cdot T_{cmp}))\\
&\approx traces \cdot funPerTrace \cdot entries \cdot (T_f + branching \cdot entries \cdot T_{cmp}) && (4.7)
\end{aligned}
$$

The complexity of the local matching algorithm consists of the locality calculation in line 2, the transformation of the entries in the source traces in line 5, the merging of the identifier trees in line 11 and the actual comparison of transformed source and target entries in line 16. During the locality calculation the adjacent target and source traces are queried for each trace from the transformation graph as shown in equation 4.2 where each query costs $T_{adjacentTrace}$. The transformation part transforms each entry in all traces with the transformation functions starting at the trace. So each call to a transformation function with average complexity T_f effects the time complexity of the entire algorithm. In the next step of the local matching the identifier trees of all transformation functions in the locality are merged together as shown in equation 4.4. At the end the transformed source entries are compared with the target entries as main part of the local matching described in equation 4.5.

Altogether the complexity of the local matching in equation 4.6 is mainly caused by the transformation of the entries and the comparison loop. The identifier quality has a large impact on this comparison because unlinked entries must be compared again to other entries. Additionally many branches in the transformation graph and many functions increase the number of entries which must be compared and as a result the complexity. A large number of functions, for example, can be caused by many different identifiers through the system or incomplete information in the traces, so that additional information from registers must be used.

$$\begin{aligned} O(global) &= 2 \cdot unlinked \cdot traces \cdot entries \cdot \\ &\quad \left(T_{adjacentTrace} + T_f + T_{merge} + unlinked \cdot entries \cdot (T_{cmp} + T_{rank})\right) \cdot \\ &\quad \left(branching^{maxDepth}\right) \end{aligned} \tag{4.8}$$

The next step of the algorithm is the global matching. This step is based on a depth-limited depth-first search which does not stop if a candidate match entry was found, but searches further in the remaining traces so that the behaviour is similar to a breadth-first search with less consumed memory. The search is starting at the missing links of entries so that the complexity of this algorithm part depends on the quality of the local matching expressed by the ratio *unlinked* of the unlinked entries to all entries. Additionally the branching factor *branching* of the transformation graph has an important impact on the search complexity described by equation 4.8.

$$\begin{aligned} O(type) &= (1 - unmatched) \cdot traces \cdot entries && (4.9) \\ O(startType) &= paths \cdot T_{opgraph} && (4.10) \\ O(startEntry) &= paths \cdot (1 + entryPoints \cdot T_{shortestPath}) && (4.11) \\ O(build) &= (1 - unmatched) \cdot traces \cdot entries && (4.12) \\ O(reconstruct) &= O(type) + O(startType) + O(startEntry) + O(build) \\ &= 2 \cdot (1 - unmatched) \cdot traces \cdot entries \\ &\quad + paths \cdot (T_{opgraph} + 1 + entryPoints \cdot T_{shortestPath}) \\ &\approx traces \cdot entries + paths \cdot entryPoints \cdot T_{shortestPath} && (4.13) \end{aligned}$$

At the end of the algorithm the operation paths gets reconstructed. Therefore, the type of each matched entry is determined. After that, the first operation type *startType* for the linked entries is determined by accessing the operation graph for each operation path as shown in equation 4.9. The first trace entry for the *startType* is determined, and then the nearest start of the operation path is calculated using the shortest path in the operation graph like in equation 4.11. In the final step the path is built starting from the *startEntry* and all matched entries are marked. The complexity of $T_{shortestPath}$ is higher than the one of the operation graph access $T_{opgraph}$ and the constant coefficients in $O(type)$ and $O(build)$ have no impact at the worst-case complexity. Thus, the complexity can be simplified to the last equation 4.13.

$$\begin{aligned} O(algorithm) &= O(local) + O(global) + O(reconstruct) && (4.14) \\ &\approx traces \cdot entries \cdot (funPerTrace \cdot (T_f + branching \cdot entries \cdot T_{cmp}) + \end{aligned}$$

$$unlinked \cdot (T_f + T_{merge} + unlinked \cdot entries \cdot (T_{cmp} + T_{rank})) \cdot$$
$$branching^{(maxDepth)}) + traces \cdot entries + paths \cdot entryPoints \cdot T_{shortestPath}$$

The complexity of the entire algorithm is shown in equation 4.14.

$$\begin{aligned} O(algorithm) &\approx traces \cdot entries \cdot (1.5 \cdot (T_f + 1.5 \cdot entries \cdot T_{cmp} + 1) + \\ &\quad 4 \cdot unlinked \cdot (T_f + T_{merge} + unlinked \cdot entries \cdot (T_{cmp} + T_{rank}))) + \\ &\quad paths \cdot entryPoints \cdot T_{shortestPath} \end{aligned} \tag{4.15}$$

$$\begin{aligned} &\approx traces \cdot entries \cdot T_f \cdot (1.5 + 4 \cdot unlinked) + \\ &\quad traces \cdot entries^2 \cdot (2.25 \cdot T_{cmp} + 4 \cdot unlinked \cdot (T_{cmp} + T_{rank})) + \\ &\quad paths \cdot entryPoints \cdot T_{shortestPath} \end{aligned} \tag{4.16}$$

Since equation 4.14 provides no direct insight, it is further simplified with approximations in equation 4.15. The average number of functions per trace $funPerTrace$ is usually relative small with about 1.3 to 1.5 in comparison to $traces \cdot entries$, so that it is approximated with a 1. In addition, *branching* takes also a value around 1.3 to 1.5 due to the combination of the structural primitives. When assuming a $maxDepth$ of 3, the term $branching^{(maxDepth)}$ evaluates to about 4.

Hence, the complexity is further approximated in order to assess the number of calls to the transformation functions, the data structures, and the comparison function. In addition, critical parameters are be derived. As shown in equation 4.16, the algorithm complexity depends heavily on the number of unlinked entries after the local matching expressed by the ratio $unlinked$. An expensive search is started for each of those entries during the global matching and the transformation function is applied multiple times for each entry. Similarly, entries which are not matched during the global matching are compared over an over during searches for other entries resulting in the quadratic term $entries^2$. Additionally this global search depends on the structure of the considered part of the system expressed by the simplified number 4. With a larger fan-out or fan-in combined with a deeper search, the complexity can further surge. The operation path reconstruction has usually a smaller influence because it is called just for the found paths. By contrast, operation paths consisting of just one entry increase the complexity here as well.

Chapter 5

Implementation

In this chapter the TRACE MATCHER as implementation of the described generic method will be presented. The matcher is integrated into the DEBUG TOOL which will be described first. Therefore, the architecture of the modular tool is outlined and the steps to process the input trace data from simulation or the real hardware are described. After that, the added components trace analyzer and decoder are addressed. These two components are used to create the logical traces with help of a trace definition, and to decode the operation types of the protocols before determining the abstract operation types harnessed in the matching algorithm.

Then the input data structure for the algorithm called configuration is outlined. This configuration contains the logical traces from the system, the transformation functions, and the OTGraph which contains both the transformation and the operation graph.

After that the local matching, global matching and operation path reconstruction of the matcher are explained.

5.1 Debug tool

Debug tools are used to analyze and display all internal information of a system for debugging. Such tools are developed and tested in simulation and later used during the power-on of the first system prototypes and during the whole chip lifetime until maintainence ends.

In contrast,to previous chip-specific debug tools the newly developed DEBUG TOOL provides a general architecture for analyzing and displaying of the internal information. This is necessary to deal with systems consisting of many chips with reused units. So the general infrastructure to read dumps, traces and so forth have to be developed only once and can be reused with a specific configuration in other parts of the tool.

5.1.1 Architecture

The new tool uses a component-based architecture with a small central core which loads and registers plug-ins, performs caching and maintains configurations. A basic set of general configurable plug-ins is provided by the tool developers, so that the designers later just need to contribute configurations. Additionally specific plug-ins must be written to analyze special features of a functional unit or a chip.

The entire architecture evolves around data processing where data is read into the system from data sources and manipulated by several functions until it is presented to the user in the front-end. All those data sources and functions are written as plug-ins, registered in the core and pass data around in *data elements*. Data sources are called *data providers* and read for example data from chip dumps, determine the system configuration or provide chip meta-data from a special database. The traces, for example, are read in by those data providers either from machine dumps or from simulation runs. On the other side the functions processing the created data are called *analyzers* because they analyze and combine the data from different data providers, filter them or provide readable representations. At the end *view descriptions* contain calls to a list of analyzers in an easily maintainable form and provide so problem-specific views for the designers. For example, one description might contain an overview about the chip activity and another one specific information about certain checkers.

One aim of the tool is to calculate necessary information on demand from the data sources instead of providing a general static expansion of all data in the chip. So disk space is saved and designers are not overwhelmed by the sheer amount of information.

The DEBUG TOOL is implemented in the programming language Java. This tool runs as central server where the designers start clients communicating over the network with this server. Therefore, the client triggers the processing with a view description calling certain analyzers at the server which call them self other analyzers and at the end data providers. Then the calculated information is returned over the network and displayed in the client. So other clients can be implemented later on as well.

The TRACE MATCHER, implementing the generic method, is integrated in this architecture as analyzer shown in figure 5.1. Thereby the artefacts are shown in grey, and the functionality is split into several data providers highlighted in orange, and analyzers shown in yellow. The transformation graph and the operation graph are created by the graph data provider. In order to create the logical traces from the traces found in hardware, the trace data is read in either from simulation or chip dumps and then examined by the trace analyzer with a trace definition describing the trace format. After that, the operation types are decoded from the attributes in the logical trace the entries, so that the abstract operation types can be decoded afterwards from them. The transformation functions are independently registered as analyzers in the debug tool. They are resolved with the function resolver during the matching.

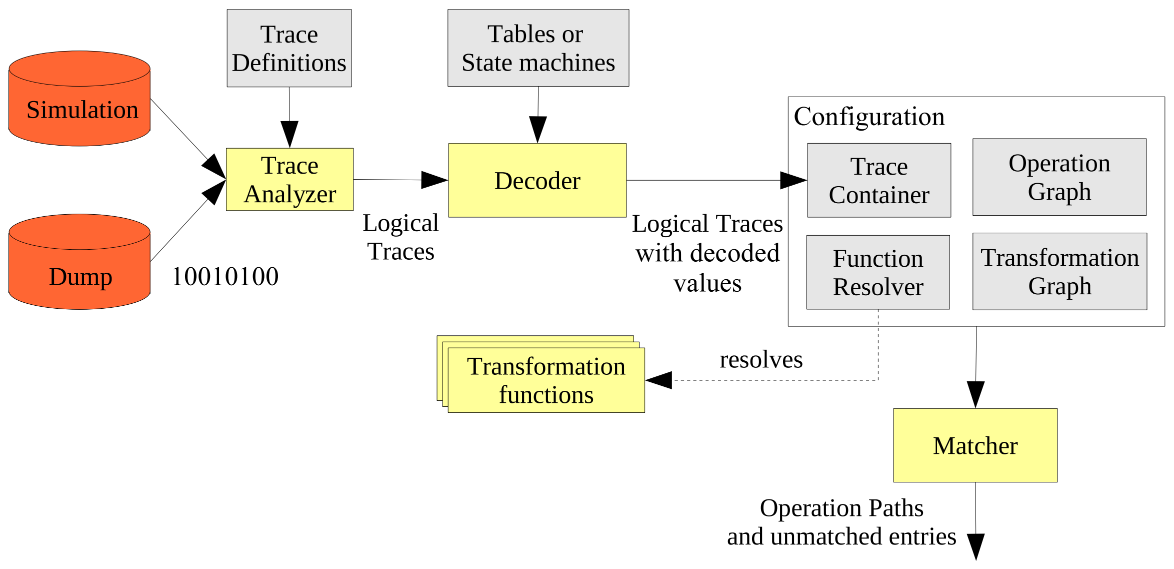


Figure 5.1: Overview about the components.

5.1.2 Trace analyzer

As already described, hardware traces have usually a totally different format and contain binary data which has by itself no meaning. That is why the trace analyzer uses a trace definition for each trace in order to extract entries containing attributes from the trace data. These definitions are stored in plain-text files and can be reused for other traces with the same format.

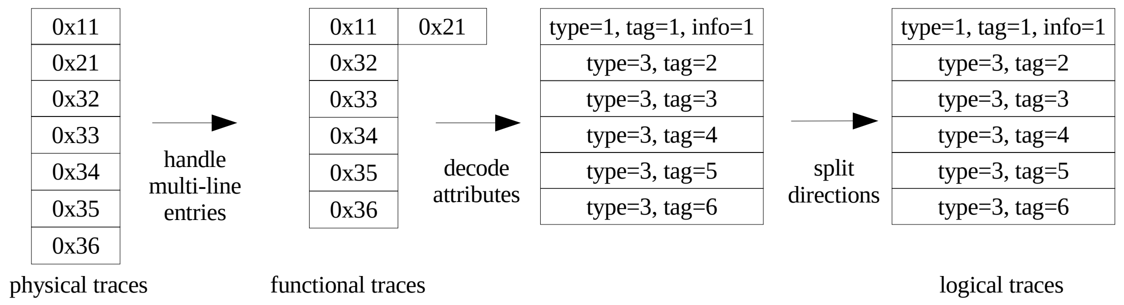


Figure 5.2: Stages from physical arrays to logical traces.

A trace can be composed out of several arrays which are described in trace definitions. Beside the traced information from the logic, an array might contain maintainence information like checksums which must be removed to get a consistent view at all traces. After that, those pre-processed arrays are combined together to *physical traces* which contains lines with data.

Since the traced information about an operation can be written to multiple entries in the physical trace, those entries must be combined to one *functional trace entry* in the next step. The figure 5.2 shows such a combination of the physical entries `0x11` and `0x21`. As a result, each of those entries contains information about at least one operation. Finally attributes are extracted from the data in the functional trace entries according to the trace definition. These attributes are name-value-pairs where the value is a combination of the found bits.

In order to create *logical traces* usable for the matching algorithm, the names of the attributes in trace definition must follow a naming scheme. So the extracted attributes about operations heading in different directions can be split in several traces by their name. The direction and the actual attribute name are divided by a dot as shown in the figure 5.3. If a logical trace contains an entry with a `tracestart` attribute with value 1 which indicates that the tracing was started, entries before this entry are removed from the trace. Additionally logical traces contain only entries whose `valid` attribute has the value 1 so that even cycle traces can be used for the trace matching.

The parser for the trace definition files was written using parser generator called JavaCC[jav05]. Thus, the grammar can be changed later on with little effort.

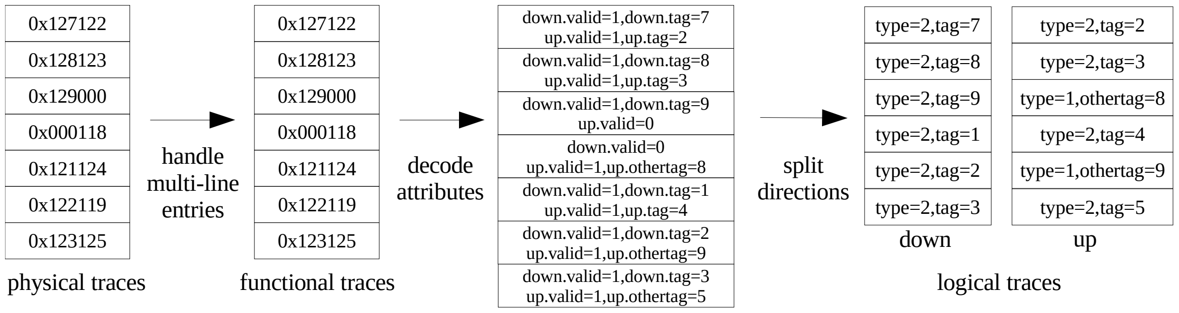


Figure 5.3: Trace analyzer extracts two logical traces from one physical trace.

Trace definition files, for example in listing 5.1, consist of sections with different purposes. The *specifications section* contains a description of all arrays used for the trace and the parts of them containing traced information and not checksums. The globals section contains *globals* which are trace-dependent and entry-independent constants used during the condition evaluation in the entry analysis. Those constants are fetched from registers in the chip which are referenced in the *registers section* with logical names. Logical names are used to refer to registers in simulation runs in the same way like to ones in . *Literals* which are constants and are defined in the literals section, as shown in line 11. They can be used in comparisons and assignments and provide a way to write easy maintainable and meaningful trace definition files. Attributes are described in the *attributes section* grouped in *attribute blocks* with conditions which must hold in order to extract the attribute.

Since the trace contains different attributes which can depend on other attributes or register values, the trace analyzer supports extracting independent and dependent attributes. For example in the listing 5.1, the attribute `down.tag` in the trace entry depends on the operation type `down.type` which is dependent on the valid attribute `down.valid` and on the selected trace mode `tracemode` as shown in line 17, 20 and 23. The valid bit `down.valid` is an independent attribute which is extracted for each entry in line 15.

Attributes can also be set independently of entries and traces to a value. For example, the attribute `valid` can be set to 1 with an assignment statement `valid = 1` in an attribute block.

As shown in line 20 of listing 5.1, the globals and attributes can be compared against literals and values. These primitive logical expressions can be combined with the logical relations “and” and “or” expressed by `&&` and `||` Additionally parentheses can be used to group those expressions and to force a certain evaluation order. In the current implementation the conditions and the attribute blocks below cannot be nested in each other so that the conditions have to be repeated in other attribute blocks.

Listing 5.1: Trace definition file for the described sample trace

```
 1 Specifications {
     arrays = ARRAY_0(0:63);
 3 }
   Registers {
 5   LogicalName.To.The.Trace.Register;
   }
 7 Globals {
     tracemode = LogicalName.To.The.Trace.Register(0:3);
 9 }
   Literals {
11   FETCH = 0x1; DEFAULT = 0x0;
   }
13 Attributes {
     UNCONDITIONALS: {
15     down.valid,      0,  0;
     }
17   down.tracemode==DEFAULT && down.valid==0b1: {
       down.type,      4,  7;
19   }
     down.tracemode==DEFAULT && down.valid==0b1 && down.type==FETCH: {
21     down.tag        8, 11;
     }
23   down.tracemode==DEFAULT && down.valid==0b1 && down.type!=FETCH: {
       down.othertag, 8, 11;
25   }
   }
```

During evaluation the conditions are checked one after another and if a condition evaluates to true, the attributes in the block are extracted from the functional trace entry. So attributes used in the conditions need to be described in the blocks before this condition, for instance `down.type` is described before `down.valid`. Beside extracting the attributes, extracted bits are stored in masks. Thus incomplete attribute values where some bits are missing can be recognized later and displayed accordingly as well as used for the trace matching.

5.1.3 Decoder

The decoder is an analyzer in the debug tool which decodes numerical values to strings according to a translation table or a state machine. These tables and machines are also described in special files.

The translation table files have a simple format where a numerical value is assigned to a decoded string in each line with a equal sign. The filename describes the name of the attribute to decode. If one of the described attribute values is found in the entry, the value is replaced by the string decoding this value.

In contrast,the state machine files assign the value returned by a state machine to the attribute described by the filename. The value is returned if one sequence of transitions from a list for this value occurred. This transition sequence contains a list of name-value pairs expression the condition causing the value. The state machine starts at the start state and looks for the first attribute in those sequences, compares the value to the described value or chooses another first attribute if the attribute is not present. If the comparison succeeds, the machines changes either the state and compares the next attribute in the sequence or returns the value described by the sequence.

5.2 Configuration

The TRACE MATCHER uses a configuration interface which provides access to the operation graph, the transformation graph, a resolver for the transformation functions and a container with the logical traces. The function resolver returns the transformation function referenced in the transformation graph. The trace container contains logical traces for the trace names.

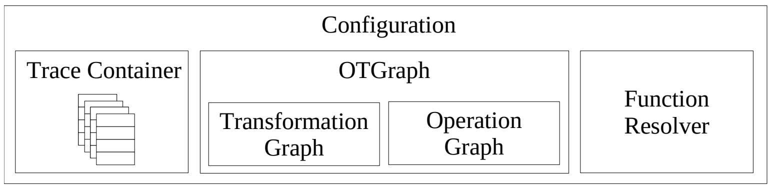


Figure 5.4: Default configuration used in the matcher.

The configuration interface can be implemented in different ways. Currently the default configuration in figure 5.4 contains a function resolver returning analyzers from the debug tool core, the OTGraph expressing both the operation graph as well as the transformation graph and a trace container fetching all logical traces at the beginning of the matching.

5.2.1 Naming

Naming is essential in order to address data and to resolve this data in the tool. The TRACE MATCHER must accomplish two things with the naming. First, the physical traces must be addressed when accessing the data providers. Therefore, a hierarchical naming scheme is used, so that a trace in a chip is referenced by composing names like `bridge0.tracename` or `io.bridge0.tracename`.

Additionally the naming must deal with logical traces, which were introduced in this work. Because the physical traces can contain multiple logical traces with operations heading in different directions, referencing just the physical trace is not sufficient. As a result, another identifier was added so that physical traces can be split in the logical traces as described in the trace analyzer subsection. This identifier is optionally appended after the name of the physical trace with a dollar sign like this `io.bridge0.tracename$up`. Thus the trace analyzer fetches the physical trace with name before the dollar sign and the calculated attributes with a name prefix `up` are extracted afterwards from the analyzed traces and are added with the remaining suffix as name to the logical trace. In addition a hash sign `#` can be used for unidirectional traces to add context information for the decoder. So the decoder has sufficient information to calculate the abstract operation type. For example, the identifier `up` in the logical trace name `io.bridge0.tracename#up` can be used for this purpose.

5.2.2 OTGraph

The configuration needed by the matcher is stored in files. One major design goal during the file format design was to reduce the maintainance effort and inconsistency possibilities. Therefore, a plain-text format is used which is easily editable in every editor.

The OTGraph, as abbreviation for operation and transformation graph, file format contains a description for both the operation and transformation graph so that the designer has to maintain just one file instead of two. On the other side the untyped transformation graph and a typed operation graph are put in one format, so that the untyped transformation graph is derived from the typed operation graph and some additional restrictions are necessary for two valid graphs.

The format is similar to the graphs used in the graphviz visualization tool[GN00] with some restrictions, for example no enclosing graph is needed and no clusters are supported. Each line is either empty, a comment, a node statement with attributes, an edge statement or a graph module instantiation. Edge statements are written in the form `A -> B` and module instantiations look like this: `instanceName:moduleName`. Additionally nodes and edges can have attributes separated by a semicolon attached in brackets after the statement, for example `A -> B [attribute1=foo; attribute2=bar]`.

The nodes in the OTGraph are logical trace names and are connected by typed edges. Those abstract operation types are specified in attributes to edge statements which are called `typeout` and `typein` in order to track type changes from request type to response type, for example `[typeout=fetch; typein=fetch_resp]`. If the types do not change from request to response, the `type` attribute can be used which is internally expanded to a `typein` and a `typeout` attribute. Additionally some abbreviations where added to the graph to support operation paths which are equal for different types. So the designer can add edges with an enumeration of types in the type attribute, for example `type=fetch,store_resp`, which are expanded to an edge with type fetch and one with type store_resp.

The transformation graph is derived from the OTGraph. Therefore, the function names in the `fun` attribute must be the same across all typed edges between each two traces because the transformation graph is typeless.

Listing 5.2: Sample OTGraph for a store request and a fetch request with response

```
           -> IntDev#up [type=fetch,snsctrl]
IntDev#up -> ArbSMB$up [type=fetch,snsctrl;fun=int2smb]
ArbSMB$up -> PCU$up    [type=fetch,snsctrl;fun=smb2pcu]
PCU$up    -> IntHub#up [type=fetch,snsctrl;fun=pcu2int]
IntHub#up ->           [type=fetch,snsctrl]

               -> IntHub#down   [type=fetch_resp,store_resp]
IntHub#down    -> PCU$down      [type=fetch_resp,store_resp;fun=int2pcu]
PCU$down       -> ArbSMB$down   [type=fetch_resp,store_resp;fun=pcu2smb]
ArbSMB$down    -> IntDev#dn_rcv [type=fetch_resp,store_resp;fun=smb2int]
IntDev#dn_rcv -> IntDev#dn_snd [type=fetch_resp,store_resp;fun=smb2int]
IntDev#dn_snd ->               [type=fetch_resp,store_resp]
```

A sample OTGraph file is shown in listing 5.2. This graph contains the operation paths for the operation types fetch and store and a transformation graph with transformation functions `int2smb`, `smb2pcu` and `pcu2int`.

In the current implementation the OTGraph files are analyzed with regular expressions, so that each statement has to be written in a single line.

5.2.3 Identifier tree

Identifier trees are currently not used in the current implementation due to a lack of time. So the earlier concept of identifier list was used, which relies on some constraints and has the same expressiveness. An identifier list consists also of a set of identifier attributes.

In contrast,to the comparison based on the tree, the comparison based on the list visits all identifier attributes in the list and compares only the ones present in both compared entries. So the dependencies between the identifier attributes described in the tree are captured in the list if the not compared attributes from the tree are not in both entries. For example, if the attributes A, B and D are in both entries, the entry C must not be in both entries so that the identifier list returns the same results as the identifier tree.

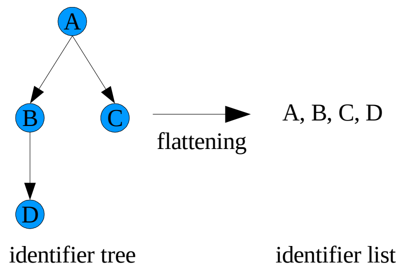


Figure 5.5: Flattening identifier trees to identifier lists.

In order to determine the list from the tree, the identifier trees are flattened by the designer with an level-order traversal. A level-order traversal visits the attributes at one level in the tree first before visiting the children on the next level until the leafs are reached. The figure 5.5 shows an example flattening of an identifier tree.

5.2.4 Transformation function

The transformation functions described in 4.2.2 are implemented as a class for each function which implements the interface `TransformationFunction` shown in listing 5.3. When a new OTGraph is added to the DEBUG TOOL, the referenced functions must be added to the tool as analyzers.

A transformation function contains both, the code to transform an entry and the identifier to compare the transformed entry to the entries in the target trace. The identifier is described by an identifier list which is created by flattening the identifier tree.

Listing 5.3: Transformation function interface

```
public interface TransformationFunction {
        public MatchTraceEntry transform(MatchTraceEntry entry,
                        Context context, boolean forward);

        public String[] getIdentifiers();
}
```

The transformation of the entries is done with the `transform` method which needs additional context information. This context contains, for example, the name of the logical trace where the function was applied on the entry. Thus the direction can be determined by extracting the suffixes after the `#` and `$` from the logical trace name. As a result, the function can be reused for similar transformations using other context information. Additionally the context provides a class to access registers in the chip so that configuration registers or lookup tables can be read out and used for the transformation.

The forward and backward transformation of entries is described in one transformation function because the code of them is usually similar. The `forward` flag in the function determines if either the entry in the source trace should be transformed to the target format or an entry from the target trace should be transformed in the other direction. So functions which are not injective can return `null` for one direction.

5.2.5 Logical traces

The logical traces, described in section 5.1.2, are calculated with the trace analyzer and contain logical trace entries of the interface `MatchTraceEntry` as shown in listing 5.4.

This interface provides access to attributes in the trace entry and the masks of numerical attributes extracted by the trace analyzer. Special attributes are prefixed with an underscore and used during the trace matching process. Two of them build the linked structure with the `_prev` and `_next` attribute and the accessor methods `next` and `prev`. The `isUnlinked` method checks if a link to the next or previous entry is missing in this entry. Similarly the `isUnmatched` method is used after matching to determine if the entry is part of an operation path.

Listing 5.4: Representation of entries in the matcher

```
public interface MatchTraceEntry {
  public String PREV = "_prev";
  public String NEXT = "_next";
  public String FLOW = "_flow";
  ...
  public void put(String key, Object o);
  public Object get(String key);
  ...
  public void putMask(String attributeName, long mask);
  public Long getMask(String attributeName);
  ...
  public String type();
  public MatchTraceEntry next();
  public MatchTraceEntry prev();
  public boolean isUnlinked();
  public boolean isUnmatched();
```

5.3 Matcher

The TRACE MATCHER implements the presented generic method using the described configuration and returning the match result. This match result contains the reconstructed operation paths and the unmatched trace entries.

Since the matcher is dependent on many parts in the debug tool and needs a configuration in order to run, interfaces are used at important places to encapsulate the dependencies to other system parts. For example, the described OTGraph provider analyzing the file format returns an object which implements both the operation graph and the transformation graph interface. So another implementation could use two files to describe a module or another format and provide the graphs in objects implementing these interfaces. The same holds for the logical traces, the configuration and the trace entries. As a result, parts of the system can be later exchanged without breaking other parts.

In order to support locality-specific and entry-specific masks, the values of the attributes in the identifier list are extracted before the entry comparison to *ids* which is just an array with the values. Those ids can be compared easily without an access to a hashtable. Additionally in a later implementation the masks can be applied before handling each locality if the masks are locality-specific. In the current implementation the masks are applied before each comparison for entry-specific masks.

As described earlier a context is needed in order to transform entries. This context contains a register provider so that the transformation functions can access registers like a translation table or some a configuration. Additionally the name of the chip and the direction of the logical trace are passed to a transformation function so that the function can contain chip- and direction-dependent logic.

5.3.1 Local matching

While iterating over all traces, the local matching creates localities. For each locality the implementation creates a data structure called `LocalMatchContext`. This context contains all data which is relevant for the matching in this locality like the entries in the source and target traces to transformation functions. Additionally it transforms the source entries and generates the ids for the entries during matching. Furthermore a list of entries is kept which contains the candidates which should be linked after the matching.

The local matching algorithm can be easily exchanged because the TRACE MATCHER calls the local matcher by an interface, and the `LocalMatchContext` and an `EntryLinker` are passed to the local matcher. Hence, one can easily switch between the default implementation which works with entry-specific masks and an alternative one based on locality-specific masks using the linear ordering to reach a smaller time complexity.

5.3.2 Global matching

After the local matching the unlinked entries are determined by selecting each entry with at least one missing link. For each of those missing links at these entries either a forward or a backward search is started depending on the missing link.

Before each search a GlobalMatchContext is created which ranks the candidate match entries and keeps track of them to compare a match to the previous best one. Additionally duplicates are here detected by comparing the ranks of two candidates matches. After the search the best candidate is linked to the entry with the missing link if an unambiguous match was found.

Each search is limited to a fixed maximum depth of three which was determined with a test case in the evaluation of the results section.

$$rank(m) = \frac{1}{depth(m)} \cdot quality(m) \tag{5.1}$$

$$quality(m) = \frac{usedIdentifiers(m)}{|identifiers|} \tag{5.2}$$

The ranking function $rank(m)$ was adapted to the identifier lists so that a second term $quality(m)$ described in equation 5.1 and 5.2 was needed. This term expresses, how much attributes were used for the comparison.

During the global matching an id cache is used, which contains the created ids of trace entries for a combination of a trace and identifier list as key. Thus, comparing an entry to the same trace with the same identifiers requires not an additional creation of the ids if the cache still contains them.

5.3.3 Operation path reconstruction

During the operation path reconstruction the start entry is determined by choosing the nearest entry to an entry point of the operation path. This calculation is done for each operation path and entry point, so that additional optimization makes sense.

Therefore, the distance calculation is implemented using the Floyd-Warshall algorithm[Sed93, pp541] returning the distance of a trace to each other trace. Thus the algorithm is applied for each operation type. Additionally the minimal distances of response types are taken into account for distances between traces using request types. As a result, querying the data structure with a start trace at the request side of an operation path to a trace on the response side returns the correct result. So the matrix is created once at the beginning of the reconstruction for the operation graph in $O(types \cdot traces^3)$ and the later accesses are constant-time queries to the typed distance matrix.

The class Checker uses this distance matrix and the operation graph in the OTGraph, in order to create the operation paths and to detect errors caused by wrong matches. For example, it recognizes types which are not described in the operation graph as next type for a specific request operation type.

Chapter 6

Results

In this chapter the implemented approach is evaluated with respect to the benefit for the hardware debug. Therefore, a bridge chip of the zSeries[1] systems is used as representative example chip throughout the chapter. First, the steps to develop a configuration for a chip are presented. Then the usage of the TRACE MATCHER with this configuration is sketched, and different visualizations of the match results are presented and compared with respect to their benefit for the debug.

After that, the TRACE MATCHER is evaluated using several test cases which focus on different parts of the matcher under different assumptions. Thus, the match results based on insufficient identifiers are discussed. Based on this analysis, recommendations are given in order to interpret match results correctly. Later the performance is evaluated based on the test cases, and discussed afterwards. Additionally, the performance is compared to the calculated complexity of the method.

6.1 Example chip

The configuration and measurements in the chapter focus on a bridge chip in a zSeries server. The I/O subsystem, called channel subsystem and shown in figure 6.1, of these servers can address over 65.535 devices[PH04][CBB$^+$04]. The bridge chip is part of the path from the CEC to the devices which consists of a hub, a bridge, a channel chip executing the channel program, the control unit, and the actual device. Thus, the hub and bridge chips are used to fan-out I/O operations to the appropriate channel. The channel program is loaded from the memory, and sends commands to the control units using an instruction set which is common across different control units. This program fetches data from the memory using DMA fetch operations, and tells the control unit with the commands to write those data to the devices attached to the control unit. In the example we try to track the operation paths of operations issued at the channel to the CEC, and the other way around. The bridge chip is

[1]Trademark or registered trademark of International Business Machines Corporation.

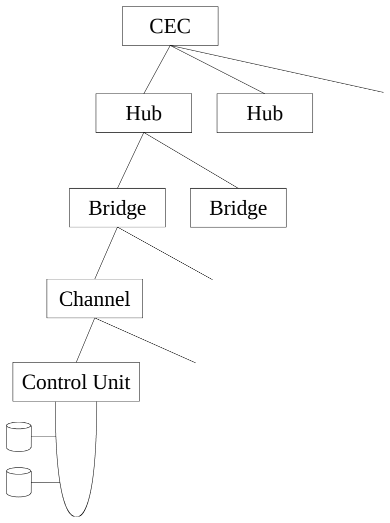


Figure 6.1: I/O subsystem in the zSeries.

shown in figure 6.2 and connects hub chips with channel chips. These interfaces are connected over an arbiter with a speed-matching buffer (ArbSMB) which adapts the speed of both kinds of interfaces to each other. A protocol conversion unit (PCU) translates between the different protocols used at both interfaces. Additionally, the chip contains a maintainence interface (Maint) which configures the functional units. This chip contains 15 physical traces which are split in 18 logical traces. These

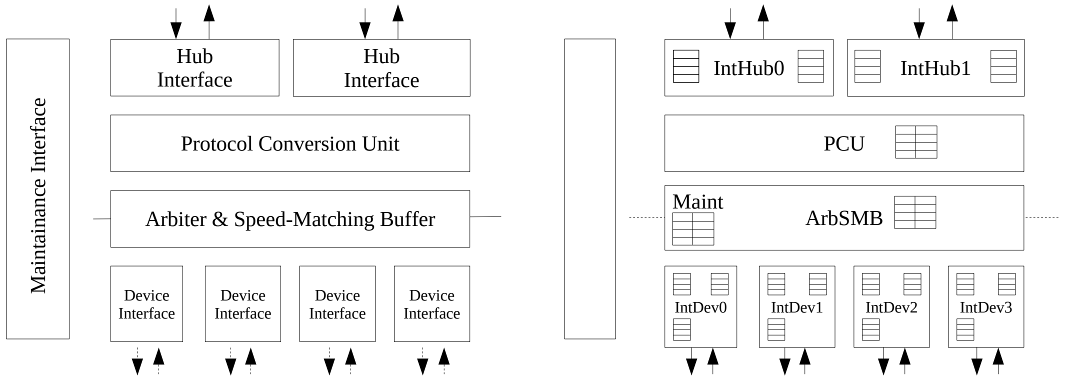


Figure 6.2: Example bridge chip with its functional units and their logical traces.

logical traces are shown in the figure figure 6.2 which contains also the names used for this traces. The traces are usually split in a part for operations heading downbound and upbound which are named with the suffix `down` and `up`. At the device interface the downbound traces are split in a receive trace `dn_rcv` and a send trace `dn_snd`. The chip supports about 20 different operation types from DMA fetch over DMA store to Sense/Controls with 10 different operation paths through this chip.

6.2 Configuration workflow

In order to reconstruct the operation paths a configuration if is necessary for the analyzed part of the system. Hence, for each new chip a new OTGraph module, and transformation functions must be developed. Additionally new tables to decode the abstract operation types must be written. Therefore, the steps are which are necessary to create a usable configuration are presented in this section. Potential pitfalls will also be highlighted during the presentation.

In order to debug chips and to create a configuration, an understanding of the chip and system structure is needed. Hence, the developer of the configuration must make himself familiar with the types of operations executed in the system. This includes the issuing and target functional units for each type of operation as well as the subtasks of the operation and their properties, for example if the operation is responseless or not.

In an early development phase, a designer with this system knowledge can add traces at the important points in the operation paths, for example operation issuing points, chip borders, protocol conversion units or generally at each functional unit. If the development is already in a later stage, the developer must analyze the traces which are present in the system. Therefore, he must first find all traces in the system and evaluate if the assumptions for the logical traces are fulfilled. This is accomplished by examining all possible modes of each trace for the traced directions, for example to the processor or to the devices. In addition the trace must be able trigger only and only if an operation passes or the traced data must contain an operation valid bit. The logical traces should fit together in a way that a dense net of traces emerges which record the executed subtasks of the operations in different functional units. As a result, the operations can be tracked in detail down to the subtasks instead of just knowing that the operation has entered and not yet left the chip. After that, the developer must describe the physical traces with trace definition files shown in listing 5.1, which must contain at least the identifier attributes, and the attributes to decode the operation type for the chosen trace mode.

Afterwards the traced attributes in the chosen modes are evaluated for attributes with identifier character. Therefore, the allocation scheme or the variance of the values of these identifier attributes must be investigated. Thus, the developer makes sure that the operations can be identified across the considered part of the system. For example, an attribute might identify only a part of the operation types, or cannot be found in adjacent traces. Furthermore one need to make sure that the dependencies of the conditional identifier attributes are tracked correctly, for instance for different operation types. When the analysis reveals that information is missing in the trace which is present in the operation, these information should either be added to the trace mode, or the design should be changed if possible.

When both, the identifier attributes in all logical traces and the operation paths for each type are well-known, the OTGraph can be defined. Therefore, a unique name with respect to the graph module has to be assigned to each logical trace, and an edge has to be added to the graph module for each connection between the logical traces as shown in listing 5.2. The unique names are later used to address the physical traces in the system, so the names must fulfill this requirement. In the example in listing 6.1, a shell around a chip is used for single chip reconstruction, so that entries from the upbound trace `IntHub#up` are linked to the downbound trace `IntHub#down` at the hub interface with the transformation function shown in listing 6.2. With this shell the traces in this single chip can be analyzed without data from other chips. The transformation function describes the used identifier attributes in an identifier list returned in the `getIdentifiers` method.

Listing 6.1: Graph module bridge-shell for single-chip matching

```
bridge0:bridge
bridge0.IntHub#up -> bridge0.IntHub#down \
  [typeout=fetch;typein=fetch_resp;fun=inthublink]

bridge0.IntDev0#dn_snd -> bridge0.IntDev0#up \
  [typeout=snsctrl;typein=store_resp;fun=intdevlink]
bridge0.IntDev1#dn_snd -> bridge0.IntDev1#up \
  [typeout=snsctrl;typein=store_resp;fun=intdevlink]
bridge0.IntDev2#dn_snd -> bridge0.IntDev2#up \
  [typeout=snsctrl;typein=store_resp;fun=intdevlink]
bridge0.IntDev3#dn_snd -> bridge0.IntDev3#up \
  [typeout=snsctrl;typein=store_resp;fun=intdevlink]
```

The transformation function at the hub interface which is called `IntHubLink` is registered as analyzer with the name `inthublink` at the DEBUG TOOL and is used by the matcher to transform entries. Similarly the developer must define the remaining transformation functions using this rather simple form of type transformation or by additionally accesses to a translation table or other registers. These functions should stay as simple as possible in order to keep the performance of the trace matching at a reasonable level.

Listing 6.2: Transformation function used at the interface to the hub

```
...
public IntHubLink extends TransformationFunction {
    private static final String[] IDENTIFIER_ATTRIBUTES =
            { "_type", "nid", "tag", "intr_id"};

    public MatchTraceEntry transform(MatchTraceEntry entry,
                                     Context context, boolean forward) {
        MatchTraceEntry tr = super.transform(entry, context, forward);
        String type = entry.type();
        if (forward && "fetch".equals(type)) {
            tr.put(MatchTraceEntry.TYPE, "fetch_resp");
        } else {
            // remove type for backward transformation
            // and other abstract operation types
            tr.remove(MatchTraceEntry.TYPE);
        }
        return tr;
    }

    public String[] getIdentifiers() {
        return IDENTIFIER_ATTRIBUTES;
    }
}
```

Then the system-wide usable abstract operation types need to be defined based on the decoded protocol-specific operation types, which are valid at a certain part of the system. Therefore, tables or state machines have to be defined for the abstract operation types so that the decoder can assign the types to the entries based on numerical representation of those types. After that, the operation paths of the different types can be described in the OTGraph.

At the end all created files must be uploaded to the debug tool. This includes the graph module describing the chip, the files for the decoder to decode the operation types and Java archives containing the transformation functions. After registering the files and the transformation functions, the trace matcher can reconstruct the operation path based on data from simulation runs or dumps from chips.

6.3 Debug workflow

After describing the chips in a configuration, the TRACE MATCHER can be applied to debug data from a simulation run, or a dump written after a problem occured at the real hardware. Therefore, the designer must specify which OTGraph module should be used depending on the available data and the part of the system which needs to be debugged. For example, the designer might have data from all parts of the system but only wants to analyze the first bridge chip `b0` so that the example configuration created in the last section can be used. Hence, the developer uses the `bridge-shell` described in listing 6.1 as graph module for the matcher.

At the moment the trace matcher is called from the command line with a chip simulation result as parameter. The OTGraph module is automatically selected for the user from the configuration of the simulation. Additionally, the directory with trace definition files as well as the tables and state machines for the decoder are passed with a wrapper script to the trace matcher. In the near future, the TRACE MATCHER will be called from the DEBUG TOOL after selecting the parts of the system which should be analyzed.

After matching the trace entries and reconstructing the operation paths, the operation paths and unmatched entries are stored in the match result. This result can be visualized in different ways with a post-processing step depending on the preference of the designer and the specific problem. Two major visualizations for chip or subsystem debug are presented here in detail and other ones will be outlined as well.

The obvious visualization for the created data structures is a *list of the reconstructed operation paths* together with the attributes in the entries along the path for a condensed view at the operation as shown in listing 6.3. Additionally the unmatched entries are displayed below the paths. The example contains two operation paths of a DMA fetch operation through the bridge chip with the accumulated attributes from each entry. Thus, the developer can see the identifiers like the `tag`, identifiers from the Infiniband[inf04] header `ib_lrh_DLID`, `ib_lrh_SLID`, and other information used along the path of the operation like the fetched address `addr`, and control bits in the operation like an error flag `err` of an used protocol. The `_pos` attribute in the unmatched entry represents the position in the logical trace where smaller numbers are used for newer entries.

Listing 6.3: Visualization of the operation paths and attributes in the entries along the path

```
0: 100, true: (bridge0.IntHub1#down,9)/snsctrl,(bridge0.PCU$toarb,9)/ \
snsctrl,(bridge0.ArbSMB$down,9)/snsctrl,(bridge0.IntDev3#dn_rcv,0)/ \
snsctrl,(bridge0.IntDev3#dn_snd,0)/-,(bridge0.IntDev3#up,0)/snsctrl_resp, \
(bridge0.ArbSMB$up,9)/snsctrl_resp,(bridge0.PCU$toint,9)/snsctrl_resp, \
(bridge0.IntHub1#up,9)/snsctrl_resp
ib_lrh_SLID=0x30, addr=0x676191e26910, tag=0x15, ib_lrh_DLID=0x203, \
nid=0x9, _type=snsctrl

1: 100, true: (bridge0.IntHub1#down,15)/snsctrl,(bridge0.PCU$toarb,15)/ \
snsctrl,(bridge0.ArbSMB$down,15)/snsctrl,(bridge0.IntDev3#dn_rcv,1)/snsctrl, \
(bridge0.IntDev3#dn_snd,1)/-,(bridge0.IntDev3#up,1)/snsctrl_resp, \
(bridge0.ArbSMB$up,13)/snsctrl_resp,(bridge0.PCU$toint,13)/snsctrl_resp, \
(bridge0.IntHub1#up,13)/snsctrl_resp
ib_lrh_SLID=0x30, addr=0xc75fc9039000, tag=0x14, ib_lrh_DLID=0x203, \
nid=0x9, _type=snsctrl

2: 100, true: (bridge0.IntHub1#down,23)/snsctrl,(bridge0.PCU$toarb,23)/ \
snsctrl,(bridge0.ArbSMB$down,23)/snsctrl,(bridge0.IntDev3#dn_rcv,3)/snsctrl, \
(bridge0.IntDev3#dn_snd,3)/-,(bridge0.IntDev3#up,2)/snsctrl_resp, \
(bridge0.ArbSMB$up,22)/snsctrl_resp,(bridge0.PCU$toint,22)/snsctrl_resp, \
(bridge0.IntHub1#up,22)/snsctrl_resp
ib_lrh_SLID=0x30, addr=0x57504ef0dba0, tag=0x13, ib_lrh_DLID=0x203, \
nid=0x9, _type=snsctrl

Unmatched entries
bridge0.IntDev1#dn_rcv
_trace=bridge0.IntDev1#dn_rcv, tag=0x2, _pos=0x2f, _type=snsctrl
```

This list of operation paths shows just the paths of each single operation, but not the relation between these paths. Hence, errors caused through the interaction of operations like deadlocks, are not recognizable. sequencing problems in the firmware, recovery problems or deadlocks are not easily recognizable.

Thus, the paths can be sorted according to different criteria like their first trace, the number of traces where entries were found, or the start trace of an operation type according to the operation graph. The ordering according to the start trace displays the most useful information, since the sequencing of the operations starting at specific functional unit can be analyzed. Another sorting of the paths is based on a deep trace where almost all operations can be found. Hence, the time relationship between the paths is clarified with this ordering, or a timestamp in a trace in the path of the operations.

Additionally, the information from trace entries along the paths should initially be restricted, to permit a faster overview. For example, this overview can just show, whether a response was received for a request. This information can be calculated by an additional post-processing which uses the match result and further information about the chip. Therefore, the deep trace must be chosen on the request side of the operations. Then the developer can use this condensed view to get a first impression about the chip or subsystem activity.

The attributes accumulated along the operation path shown in line 5 and 12 help the developer to recognize patterns in the operations. For example, an operation accessing a certain address range, flowing over a certain port, using a specific identifier or operation type can show expected or unexpected behavior. Based on that information the developer can dig deeper into other internal information of the chip.

In case that an error was noticed during the operation path reconstruction, an operation path is marked as invalid. Thus, these invalid paths are printed as well. In the example, another fetch operation is linked to the fetch response of the first fetch operation. This problem can be resolved with a more sophisticated checking code in the operation path reconstruction.

Listing 6.4: Visualization of the linked entries in the traces

```
1  bridge0.PCU$down
   0: IntHub1#down,0  ArbSMB$down,0   tag=0x41, nid=0x9, _type=fetch_resp
3  1: IntHub1#down,1  ArbSMB$down,1   tag=0x41, nid=0x7, _type=fetch_resp
   2: IntHub1#down,2  ArbSMB$down,2   tag=0x41, nid=0x8, _type=fetch_resp
5  3: IntHub1#down,3  ArbSMB$down,3   tag=0x41, nid=0x6, _type=fetch_resp

7  bridge0.ArbSMB$down
   0: PCU$down,0      IntDev3#down,0  tag=0x41, nid=0x9, _type=fetch_resp
9  1: PCU$down,1      IntDev1#down,0  tag=0x41, nid=0x7, _type=fetch_resp
   2: PCU$down,2      IntDev2#down,0  tag=0x41, nid=0x8, _type=fetch_resp
11 3: PCU$down,3      IntDev0#down,0  tag=0x41, nid=0x6, _type=fetch_resp

13 bridge0.IntDev0#dn_rcv
```

```
0: ArbSMB$down,3                  tag=0x41

bridge0.IntDev1#dn_rcv
0: ArbSMB$down,1                  tag=0x41

bridge0.IntDev2#dn_rcv
0: ArbSMB$down,2                  tag=0x41

bridge0.IntDev3#dn_rcv
0: ArbSMB$down,0                  tag=0x41
```

The second visualization displays in a trace view beside the entries *references to the previous and next entry* in the operation path as shown in figure 6.4. These references can be links similar to the ones on web sites and the links can be browsed in the same way. In the example four fetch responses can be seen in six traces whose entries are printed line-by-line. These lines show first the position of the entry in the trace where smaller numbers are newer entries, the previous and next entry in the operation path, and attributes extracted in this trace. Usually the output contains more attributes, but the space is limited on this page so that only the identifiers are shown. The entries in the last four traces have no link to the next entry because the operation path ends in the devices below this chip. The developer would start debugging at a certain trace and browse in the operation paths and traces to gain new insight about the problem.

6.4 Evaluation

After presenting the configuration and debug workflow, the implementation is evaluated with respect to the correctness of the returned operation paths and the performance for relevant test cases. These test cases should be simple enough to be understandable, and on the other side test an implementation in the general and corner cases. That is why only the two operation types DMA fetch and Sense/Controls have been used because the number of types has a low impact on the reconstruction. In contrary, problems arise from ambiguous identifiers in operations. Such identifiers occur in current chips since the identifiers were not designed for a retrospective analysis.

In order to evaluate the method for unique identifiers across the traces in the system, test cases in one group get restricted in simulation time and the number of issued operations. This group is called `noidwrap`, fulfills the requirements for the abstract method, and is split into two cases. First, the `notracewrap` where all traces contain each operation. This is an ideal case which occurs seldom in reality because the entries wrap in most cases before the traces are stopped. However, it shows the complexity of the local matching since no unlinked entries will remain for the global matching. In contrast, the second test case `tracewrap` shows the impact of trace wrapping at the algorithm and the complexity of the global matching.

In the second group of test cases, the simulation and the issued operations are not restricted, and the simulation is stopped before all operations are completed. Thus, the traces contain entries similar to debug problems at the real machine. Additionally, the used identifiers are not unique in the tests and so the group is called `idwrap`. The effects of these identifiers will be analyzed in the correctness subsection.

6.4.1 Correctness

Debug relies on drawing right conclusions about the activities in a system based on data from this system. Hence, debug tools must correctly analyze this data in order to support and not hamper the designer with the results during debug. In order to analyze the match results for correctness, the TRACE MATCHER and the configuration for the matcher must be analyzed.

The generic method relies on identifiers in operations which uniquely identify their entries in a retrospective analysis. Thus, no two entries can be linked together which do not belong to the same operation. Consequently, only the special cases operation resend as well as operation splits and merges cause identifier duplicates in the entries. Since these entries belong to the same operation path, they can be linked together. In figure 6.3 the only problem is shown, which occurs during global matching of the special cases. Only one of the entries in the trace A, where the search starts, is linked to the target trace C. The second entry stays unmatched because the entry in the target trace is already marked as matched and not considered anymore. This problem can be solved by adding an additional flag to each link to indicate that they have been created in global matching and may be considered for further matches.

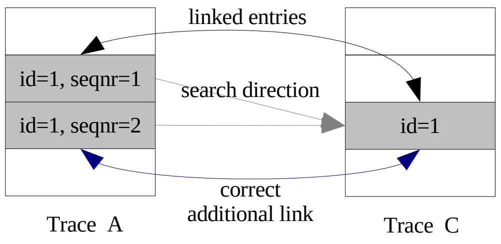


Figure 6.3: Search in global matching can in some cases not detect duplicates.

In contrast, operations in current systems contain identifiers which do not definitely identify an operation in the traces across the considered part of the system over the period shown in the trace entries. The problems resulting from this insufficiency can be distinguished in different cases which are shown in figure 6.4. The figure shows only entries with the same identifier and the entries caused by the same operation are connected by arrows. An entry with a `new` identifier refers to an incomplete operation so that no entry can be found in the next traces on the path. Similarly, an entry marked with `old` refers to an operation which can only be found either in the source trace A or the target trace B. In reality, any number of entries can occur before the *new* and after the *old entries* shown by the dots, as long as the corresponding entry of this operation is in the target trace. In addition there might be several old or new entries instead of the single shown one. Case a can occur for example, if all but some identifiers are used in the system by long-lasting operations so that short-running operations can only use the remaining identifier multiple times. When the identifiers are designed for the retrospective analysis, only case a occurs. Hence, the source and the target trace contain only one entry so that the paths can be reconstructed without ambiguities. In contrary either *duplicates* appear in the traces as in case c, d, f or one *trace does not contain the entry* as in case b. The worst case for the trace matching is case e because the algorithm cannot notice that the wrong entries get linked together in this special case.

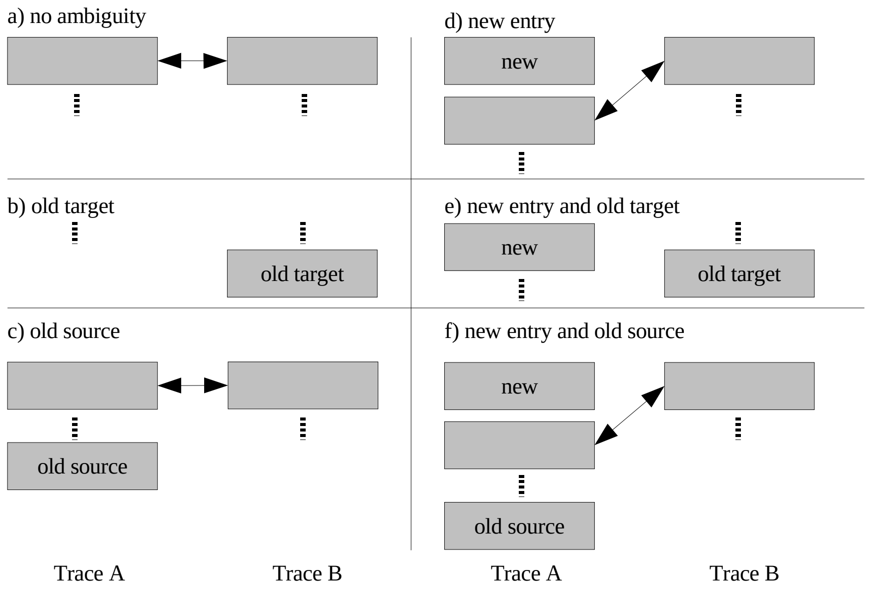


Figure 6.4: Problematic cases for insufficient identifiers.

In order to reduce the amount of incorrect matches due to insufficient identifiers, the duplicate handling was added to the TRACE MATCHER which can detect duplicates in many cases. The duplicates are detected during the local matching because the entries are not linked in the first place but only marked. Since the same merged identifier list is used for comparisons in the entire locality, the duplicates can be flagged. The markings and the flags are then considered during the entry linking so that only entries without duplicates are linked. Thus, only case e is not covered. For example, an entry of a single new operation is found in a trace at the beginning of the operation path, and another trace at the end of the path contains an entry of another old operation with the same identifier. As a result, the matcher would link them both and then put them in the same operation path.

In contrast, the global matching has a less effective duplicate handling. Because the identifier list can change during each search and other searches might apply other masks, found duplicates noticed in one search might not be duplicates in other searches. Therefore, marking duplicates and linking the entries later is not an option for the global matching. Additionally a search from trace A to trace B detects the duplicates shown in figure 6.3, but a search in the other direction starting at one of the duplicate entries cannot notice the duplicates. Thus, the wrong entries might be linked together resulting in a incorrect reconstruction of the operation paths. Since even not the standard cases can be linked together, entries of the special cases like resends as well as split and merge of operations cannot be linked together by keeping the correctness. Consequently entries with duplicates are not linked together.

As a result, the designer must interpret the results of the TRACE MATCHER if the identifiers are not unique. Therefore, different techniques can be used:

1. Usually some *operations are processed between operations using the same identifier*. Thus, the operations with ambiguous identifiers can be distinguished with additional assumptions about the number of passed or passing operations. For example, operations can at a point of the system only be passed by a control operation, and one entry between entries with a duplicate identifier in the source trace which can be found in the target trace indicates how the entries belong together. This approach is not generally applicable because all but one identifier can be in use for long-lasting operations. Additionally, different structural primitives pose a problem in this kind of resolution. As a result, ambiguous operations cannot in general be distinguished using operations between them.

2. Other attributes, registers or *other chip internal information* might contain additional hints about the entries which belong to the same operation. So the designer can use them to distinguish entries with the same identifiers. Hence, those information can maybe added to the transformation functions.

6.4.2 Performance

The performance was evaluated on a laptop system with a Intel Pentium M processor under Linux, which executed the TRACE MATCHER using the Sun JDK 1.4.2. In order to reduce side-effects the workload on the machine stayed nearly the same across all test cases. Additionally, the average runtime was determined after calling the matcher multiple times. All times do not include the time to create the configuration.

In all cases the matcher was applied to trace data from simulations of the example bridge chip. Thus, the configuration created in this chapter with $funPerTrace$ of 1.227 and *branching* of 1.227 was used. Additionally, only the two described operation types DMA fetch and Sense/Controls were issued in the simulations.

At first, the parameter *maxDepth* for the global matching was determined using the `noidwrap` test case number 3. The matcher was applied to data from this simulation run with different values for the maximal search depth in the global matching. The average runtime was determined after 25 trace matcher runs. As the table 6.1 and the figure 6.5 shows, the optimal *maxDepth* for this configuration is 2 because the number of unmatched entries is minimal for the first time. The remaining 13 unmatched entries are in traces which contain old operations which have already been overwritten in the other traces. Since other configurations have a different structure and the runtime gets longer for deeper searches, the depth is set to 3 for safety. All following TRACE MATCHER calls use this search depth for the global matching.

depth	0	1	2	3	4	5	6	7
entries per trace	47.091	47.091	47.091	47.091	47.091	47.091	47.091	47.091
entries	1036	1036	1036	1036	1036	1036	1036	1036
matched entries	834	837	1033	1033	1033	1033	1033	1034
unmatched entries	202	199	3	3	3	3	3	2
average runtime in ms	52.48	61.96	95.28	119.52	142.48	183.96	238.2	325.6

Table 6.1: Determining the search depth maxDepth for the global matching

The average runtime of the algorithm for the different depths in figure 6.5 was approximated by the function $f(x) = a \cdot b^x + c$. Since the same test case is used for all matcher runs, the local matching complexity stays constant. Therefore, only the global matching and the operation path reconstruction have influence on the runtime increase with higher search depths. Thus, the `fit` command of gnuplot[TW$^+$06] was used to approximate the data with the function $a \cdot b^x + c$. This fit function uses a nonlinear least-squares (NLLS) Marquardt-Levenberg algorithm. After applying the fit command, the variables had the following values: a was 41.16, b was 1.33, and c was 14.01. In contrary the worst-case complexity of the global matching calculated in chapter 4 was determined for this test case using the parameter values. With each deeper search the number of entries rises by factor *branching* which has the value 1.227 for this test case. Thus, the calculated rise by 1.227 while the experimental ones rise by 1.33. Consequently, they are nearly the same but in the test case the global matched searched in more traces with higher branching.

After that, the TRACE MATCHER was applied to all test cases in the `noidwrap` group, to evaluate the performance of the matcher for unique identifiers at the operations. The average runtime was determined after 1500 trace matcher runs for each `notracewrap` test case and 200 runs for each `tracewrap` test case and is shown in figure 6.6. First, the matcher was applied to a series of test cases called `notracewrap` where all traces contain entries about all operations. The results for the first group are shown in table 6.2. This is an ideal case which does seldom occur in reality, but it highlights the complexity of the local matching because no global matching is necessary. On the other side in the `tracewrap` test cases in table 6.3, a global matching is needed so that a search is done for the unlinked entries. Since the identifiers do not wrap, no ambiguous identifiers occur but only the identifiers which have no corresponding entry in another trace. Hence, the global matching reduces the number of unmatched entries with each search where at least one match was found.

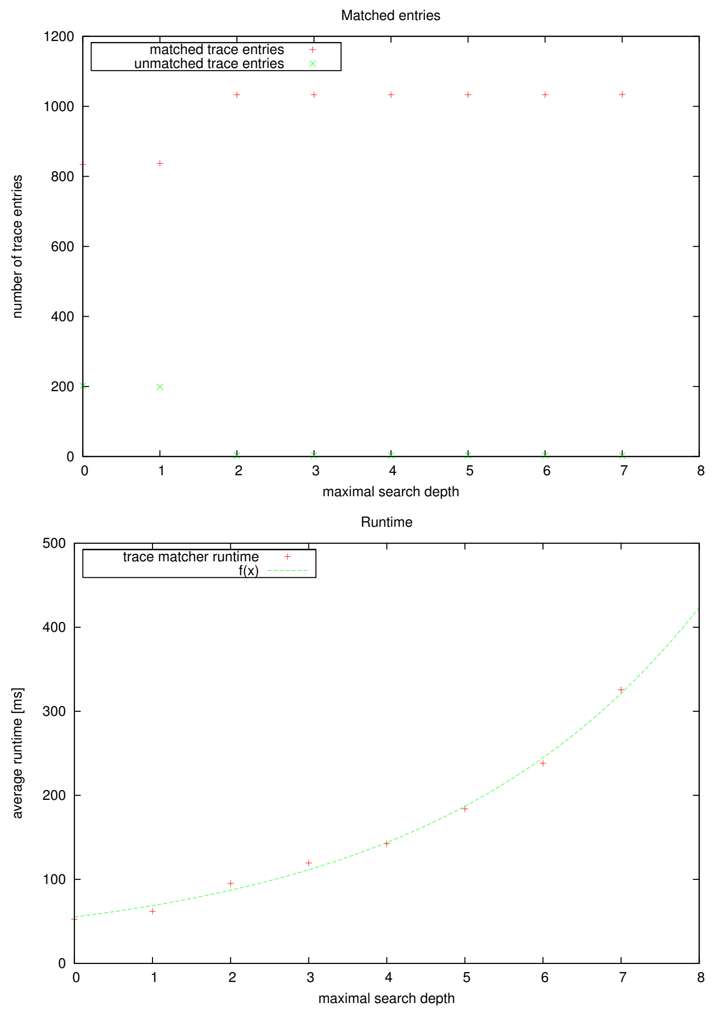


Figure 6.5: Determining the search depth maxDepth with noidwrap test case.

At the end the TRACE MATCHER was applied to simulation runs of the example chip which were not restricted. Hence, the identifiers wrapped in the test cases called `idwrap`. Additionally, the traces were stopped similar to the real machine in the simulation in order get realistic trace data. The average runtime was determined after 50 trace matcher runs. The results of these runs are shown in table 6.4 and in figure 6.7.

The number of issued operations were increased with each run so that at the end even the deepest trace was filled completely with entries. The large number of unmatched entries is caused by deep traces and identifier duplicates. Thus, few unmatched entries remained in the first test case, and the number of them increased because later only the deepest traces contained the entries. Due to the different trace depths, the number of unlinked entries after the local matching increased with the issued operations as well. Consequently, the runtime rose caused by more searches in the global matching.

The deep traces at the device interface are four times deeper than most traces and record only a fourth

	notracewrap 1	notracewrap 2	notracewrap 3
entries per trace	1.636	3.273	6.545
entries	36	72	144
matched entries	36	72	144
unmatched entries	0	0	0
average runtime in ms	10.15	16.16	24.18

Table 6.2: Matcher runtime and quality for the noidwrap-notracewrap test cases.

	tracewrap 1	tracewrap 2	tracewrap 3
entries per trace	22.091	37.818	47.091
entries	486	832	1036
matched entries	478	828	1033
unmatched entries	8	4	3
average runtime in ms	62.56	132.31	183.77

Table 6.3: Matcher runtime and quality for the noidwrap-tracewrap test cases.

of another equal deep trace. For example, in the device traces of run idwrap 4 at average 68 entries out of 81 unambiguous entries have been matched, and 160 stay unmatched due to duplicates. In consequence most of their entries, 2078 out of 2895, stayed unmatched. The other traces contained the remaining 27 unmatched entries which were also caused by duplicates and wrapped entries.

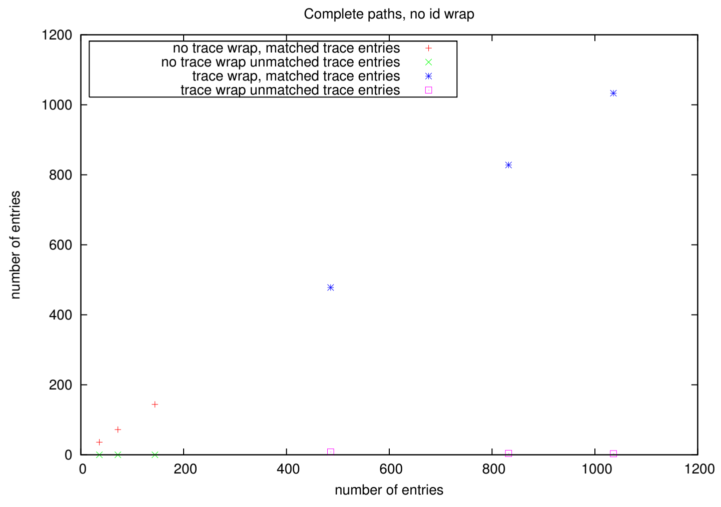
	idwrap 1	idwrap 2	idwrap 3	idwrap 4
entries per trace	50.364	91.5	133.682	148.955
entries	1108	2013	2941	3277
matched entries	996	1355	1174	1231
unmatched entries	112	658	1767	2046
average runtime in ms	212.36	489	932.1	900.7

Table 6.4: Matcher runtime and quality for the idwrap test cases.

6.4.3 Configuration effort

Configurations are the key to use the TRACE MATCHER. The configurations are described in several files with human editable plain-text formats. Each of those format describes only one or two aspects of the system, for example the traces are described independently from the transformation, or the operation type decoding. That is why the decoding descriptions can be reused for the same protocol at another part of the system. Additionally, the modularity of the OTGraph supports reuse of chip descriptions in system descriptions.

The transformation functions are not hard to write as well. The designer needs no deep knowledge about Java but must only make small changes at the attributes, or access registers with a simple interface using the chip and register name. Additionally, just a concise array needs to be returned as



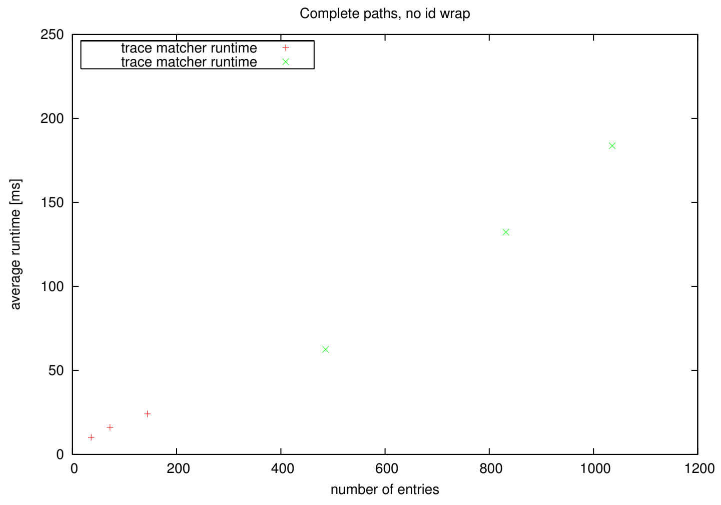


Figure 6.6: Matcher runtime and quality for the noidwrap test cases.

identifier list. Hence, these functions can be written by everybody with an understanding of the chip function. In contrast to the simple single chip view, the functions must be reviewed for multiple chip configurations in order that they address registers correctly.

All descriptions must be reviewed for new chip releases, because changes might result in a different behavior either in the functional logic, the tracing logic, or the trace configuration. Hence, the TRACE MATCHER must get the configuration files from a versioned repository.

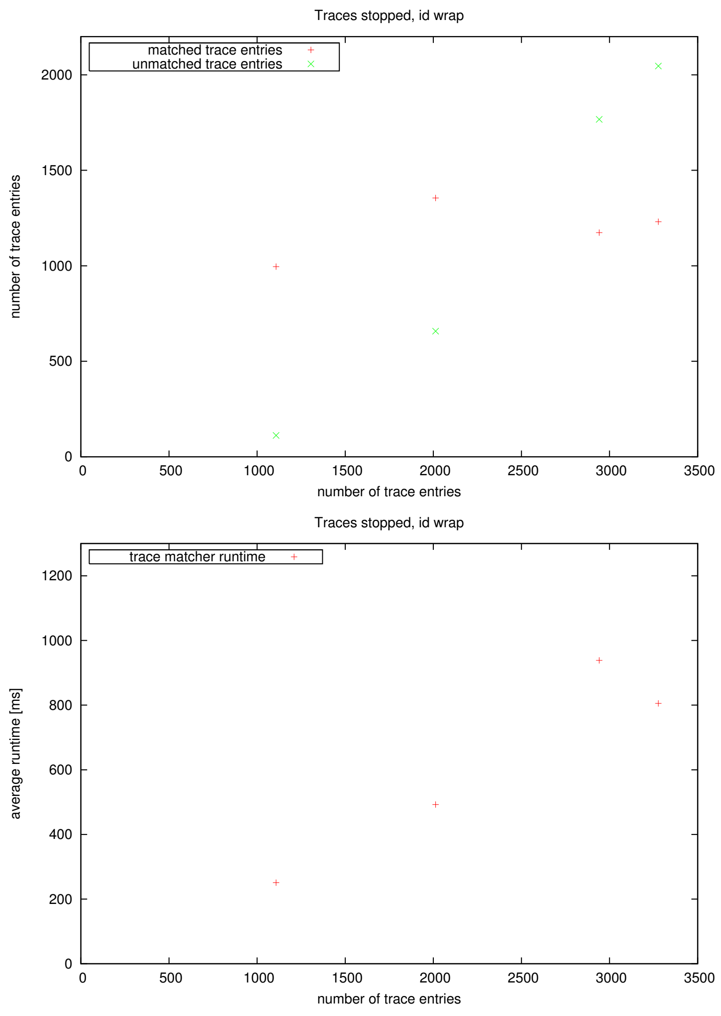


Figure 6.7: Matcher runtime and quality for the idwrap test case

Chapter 7

Consequences

In this chapter the key results of the work will be highlighted. Additionally, problems occurred during the development of the generic method, of the TRACE MATCHER, and of the example configuration which are described. A set of guidelines for trace and system design will be presented which describe how identifiers and traces can be improved for a retrospective analysis. After that, a summary subsumes the previous chapters. Finally, an outlook is given which points to further possible work and potential extension of the matcher.

The developed method is generic since no hard-coded information but only information from a configuration is used to reconstruct operation paths. Additionally it is extensible because the transformation functions and graphs in configurations can be written in a modular way so that multiple chip and subsystem configurations can be assembled out of reused chip configurations. With these configurations, operations paths can be reconstructed either in single chips, across multiple chips, or even in the complete subsystem. Reordered operations and all structural primitives are supported as well. The TRACE MATCHER returns the correct results if the identifiers are unique in a retrospective analysis. Because this is not the case for current chips, the matcher is robust enough in order to be applied to these chips at the cost of a higher time complexity. As a result, of the duplicate handling, operation paths containing regular duplicate identifiers due to operation resends or splits of large data blocks in operations, cannot be reconstructed. However, the matcher can return incorrect results in corner cases for ambiguous identifiers. Some hints were given in the previous chapter to recognise those cases. Additionally, the performance is acceptable with less than a second for a correlation. Therefore the generic method and their implementation are a sustainable foundation for reconstructing operation paths from chip internal information.

7.1 Problems

The major problem during the development was to gather information which was necessary to reconstruct operation paths. In order to develop the generic method and to write a configuration for the TRACE MATCHER, an understanding of the chip and system architecture, the paths of operations through the system, and the attributes of those operations were necessary. Designers and the documentation referred to the same kind of operations in different parts of the system with different terms, because the used protocols and the associated terminology have changed over time. Moreover, information was spread across many documents where only small parts of them were necessary in order to write a configuration. Similarly, designers usually knew only their functional units or nearby parts of the chip, but not all parts of the system. That is why gathering information for the operation path reconstruction was a hard task.

Additionally triggers, modes, and formats of the different traces were reviewed in order to be able to apply the generic method. As a result, some trace configurations have been modified. For example, a default trace mode was introduced for all traces in the system, which records identifiers of operations and writes only an entry if an operation passes the trace.

It turned out that identifiers of operations, which were thought to be unique, are not unique in a retrospective analysis but only at chip runtime. Hence, a duplicate handling was added to the TRACE MATCHER and some hints about the interpretation of the match results were given in the last chapter. Additionally the next section contains guidelines for trace and system design in order to create unique identifiers by design.

The method is not applicable to complex operation paths where no operation graph can be described due to a missing general type for all subtasks. Similarly, entries from operations without identifiers like broadcasts are not reconstructed. Additionally, the method relies on identifiers in traces, and their transformation between these traces. Thus, transformations which rely on data which is not accessible after a trace stop or in a chip dump cannot be expressed. For example, transformations using data from the memory might require a complete infeasible memory dump, or result in badly maintainable complex transformation functions.

7.2 Guidelines for trace and system design

First of all traces used for the trace matching should be *operation history registers* which add an entry to the trace when an operation passes the trace. Hence, traces record operations heading in different directions by triggering independently for those directions. Thus, no space is wasted and no entries are written to the trace which would be ignored by the TRACE MATCHER anyway.

Currently the design of identifiers of operations is *driven by functionality*. So identifiers used in the operations are only unique at chip runtime and not in a retrospective analysis, as the evaluation of a sample bridge chip showed. During the design of protocols and new systems the size of those identifiers is chosen based on the number of outstanding operations at an operation source and the number of those sources in system. While designing those identifiers, the system architects should keep in mind a later analysis during debug. Thus, the trace depths must be aligned to the width of the identifiers. For example, an operation has a tag which has 16 possible values for the same number of outstanding operations. Additionally, a trace in the system is 32 entries deep so that the identifier must have at least 47 possible values because all tags but one can be used by long-lasting operations during the system runtime. Consequently there are enough values left to fill the trace with different identifiers. Hence, for an easier implementation the identifier is aligned with the power of two so that 64 values are used instead of 16 values. This example shows the basic problem of the identifier design and presents a principal solution. Similarly the different structural primitives and identifier combinations need to be investigated with respect to the effect on the identifier width.

Additionally the allocation scheme of those identifiers can support or prevent trace matching. A *priority-based* scheme for example, will assign always the first free identifier from the identifier pool so that many entries contain the same identifier. Thus no useful matching is possible. In contrast,a *round-robin* scheme repeats identifiers not that often by design. For instance, same identifier values are only used when all values have been used one time or the other identifiers are currently in use by long-lasting operations. Consequently operations represented by entries in the traces, can be uniquely identified when the identifiers are carefully designed and a round-robin assignment is used.

Because identifiers are limited in their width, traces which should be analysed must be *stopped at nearly the same time*. As a result, designed identifiers in the entries refer to the same operations.

Traces should also contain all necessary information to determine the *abstract operation type* because all but a single trace might wrap. In this case the TRACE MATCHER is able to calculate the type from this entry, and can reconstruct the operation path with help of the operation graph.

7.3 Summary

In this work, chip internal information which is distributed across the system has been correlated in order to retrospectively track operations. Therefore, identifiers from operations, which are written to traces during execution, have been used to match the entries from distributed traces together. Identifiers have been transformed between different trace formats. Then the operation paths were reconstructed from these matched entries. The work stated explicitly the prerequisites for trace matching, discussed corner cases as well as problems, and gave recommendations for trace and system design.

The generic method was implemented in a modular manner in the TRACE MATCHER. In contrast to OHRMATCH, the matcher is generic and extensible so that it matches not only information of one specific chip but can be applied to arbitrary chips or group of chips in the system. In addition, the results presented by the matcher are understandable because either the reconstructed operation paths are shown, or a custom visualisation can be written for the debug.

The TRACE MATCHER returns the correct results if the identifiers are unique in a retrospective analysis. Because this is not the case for current chips, the matcher is robust enough in order to be applied to these chips at the cost of a higher time complexity. Therefore, a duplicate handling was implemented, and operation paths with regular duplicate identifiers caused by operation resends or operation splits cannot be reconstructed. However, the matcher can return incorrect results in corner cases like global matching for ambiguous identifiers but a possible solution was presented. Moreover, some hints were given in the chapter 6 in order to recognise the corresponding unmatched entries if duplicates occurred. The matcher was applied to an example bridge chip, and reconstructed the operation paths These reconstruction was finished in less than one second, which is reasonable enough for interactive work.

The TRACE MATCHER integrates well in the modular DEBUG TOOL, and adds the components trace analyser and decoder which are also used for other purposes like displaying trace entries. The implementation provides a modular framework for trace abstraction and trace matching which can be reused for later improvements or other approaches.

7.4 Outlook

The implementation of the generic method is a solid foundation for later developments. Until better identifiers are used in chips, the TRACE MATCHER could be extended with additional processing steps. These steps can be added after the local and global matching respectively, and can use the context data structures from the matching. Chip specific assumptions can be used in order to resolve identifier duplicates and increase the number of matched entries. Therefore, the quality would decrease, nevertheless the additional number of matched entries may be beneficial for the hardware debug. The resends as well as splits and merges of operations could be implemented if identifiers are unique by design in a retrospective analysis.

The configuration could be dynamically created with a special analyser using knowledge about the system configuration in a machine and operations in the system. Thus, operation paths could be analysed without choosing the OTGraph module after assembling the system configuration by hand.

In addition traces from firmware execution could also be used in the TRACE MATCHER, as long as they fulfill the requirements of logical traces and provide identifiers which could be found in hardware traces.

Bibliography

[BFG+99] T. Buechner, R. Fritz, P. Guenther, M. Helms, K. D. Lamb, M. Loew, T. Schlipf, and M. H. Walz. Event monitoring in highly complex hardware systems. *IBM Journal of Research and Development*, 43(5/6), 1999.

[CBB+04] E. W. Chencinski, M. J. Becht, T. E. Bubb, C. G. Burwick, J. Haess, M. M. Helms, J. M. Hoke, T. Schlipf, J. M. Turner, H. Ulland, M. H. Walz, C. H. Whitehead, and G. Zilles. The structure of chips and links comprising the ibm eserver z990 i/o subsystem. *IBM Journal of Research and Development*, 48(3-4):449–459, 2004.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.

[Gru] B. Gruschke. Integrated event management: Event correlation using dependency graphs. `http://citeseer.ifi.unizh.ch/gruschke98integrated.html`.

[IEE01] IEEE. Ieee 1149.1 standard test access port and boundaryscan architecture. `http://standards.ieee.org`, 2001.

[inf04] *InfiniBand Architecture Specification Volume 1, Release 1.2*. InfiniBand Trace Association, 2004.

[jav05] Javacc is a parser/scanner generator for java. `http://javacc.dev.java.net/`, 1996-2005.

[JBB93] V. Jacobson, R. Braden, and D. Borman. Tcp extensions for high performance. `http://citeseer.ist.psu.edu/jacobson92tcp.html`, 1993.

[KBF+04] S. Koemer, R. Bawidamann, W. Fischer, U. Helmich, D. Klodt, B. K. Tolan, and P. Wojciak. The z990 first error data capture concept. *IBM Journal of Research and Development*, 48(3-4):557–567, 2004.

[Lew99] L. Lewis. Event correlation in spectrum and other commercial products. `http://www.aprisma.com/literature/white-papers/wp0551.pdf`, 1999.

[Mor93] John M. Morse, editor. *Merriam Webster's Collegiate Dictionary*. Merriam-Webster, Incorporated, 1993.

[PH04] W.G. Spruth P. Hermann, U. Kebschull, editor. *Einführung in z/OS und OS/390. Web-Services und Internet-Anwendungen für Mainframes*. Oldenbourg Wissenschaftsverlag GmbH, München, 2004.

[RN03] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

[RS96] Michel Raynal and Mukesh Singhal. Logical time: Capturing causality in distributed systems. *Computer*, 29(2):49–56, 1996.

[Sed93] Robert Sedgewick. *Algorithmen in C*. Addison-Wesley Publishing Company, 1993.

[Tif02] Michael Tiffany. A survey of event correlation techniques and related topics. `http://citeseer.ist.psu.edu/tiffany02survey.html`, 2002.

[Tur04] Jim Turley. Nexus standard brings order to microprocessor debugging. `http://nexus5001.org/nexus-wp-200408.pdf`, 2004.

[TvS02] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice-Hall, Upper Saddle River, NJ, 2002.

[TW$^+$06] Colin Kelley Thomas Williams et al. Gnuplot - an interactive plotting program. version 4.0, july 2006. `http://www.gnuplot.info/docs/gnuplot.html`, 1986-2006.

List of Figures

List of Tables

List of Algorithms

List of Listings

Appendix A

Acknowledgements

Many people have helped me during the development of the thesis. Firstly, I would like to thank my supervisor C. S. Smith for such an interesting topic. He provided an inspiring and open atmosphere during the work and the many discussions we had. It was a pleasure to work with him.

I would like to thank Professor Udo Kebschull for supervising me, and providing so many constructive comments in such a short time. In addition, I want to appreciate the several trips he did to the development labroratory in Böblingen for presentations and discussions.

I want to thank all people from the I/O Hub department for all their help, support and interest in my work. Additionally, the people from the simulation department supported me during work by providing example data from simulation runs even after plenty of adjustments.

I want to thank Georg Martius for proofreading the final version of this thesis. He read the document on short notice, made corrections and offered valuable suggestions for improvement.

Especially, I would like to give my special thanks to my sister, my parents, and the flat-mates in Stuttgart who enabled me to complete this work.

Appendix B

Source code

The source code of the developed TRACE MATCHER is only stored at the CD-ROM attached to the version of the supervisors. Source code which was written by other developers at the IBM corporation was not added to the medium due to legal reasons. That is why the code is not functional by itself.

Eigenständigkeitserklärung

Ich versichere, daß ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Leipzig, 08.08.2006

Michael Große