Efficient Pattern Search in Large Traces through Successive Refinement[*]

Felix Wolf[1], Bernd Mohr[2], Jack Dongarra[1], and Shirley Moore[1]

[1] University of Tennessee, ICL
1122 Volunteer Blvd Suite 413
Knoxville, TN 37996-3450, USA
{fwolf,dongarra,shirley}@cs.utk.edu

[2] Forschungszentrum Jülich, ZAM
52425 Jülich, Germany
b.mohr@fz-juelich.de

Abstract. Event tracing is a well-accepted technique for post-mortem performance analysis of parallel applications. The EXPERT tool supports the analysis of large traces by automatically searching them for execution patterns that indicate inefficient behavior. However, the current search algorithm works with independent pattern specifications and ignores the specialization hierarchy existing between them, resulting in a long analysis time caused by repeated matching attempts as well as in replicated code. This article describes an optimized design taking advantage of specialization relationships and leading to a significant runtime improvement as well as to more compact pattern specifications.

1 Introduction

Event tracing is a well-accepted technique for post-mortem performance analysis of parallel applications. Time-stamped events, such as entering a function or sending a message, are recorded at runtime and analyzed afterward with the help of software tools. For example, graphical trace browsers, such as VAMPIR [1], allow the fine-grained investigation of parallel performance behavior using a zoomable time-line display. However, in view of the large amounts of data usually generated, automatic analysis of event traces can provide the user with the desired information more quickly by automatically transforming the data into a more compact representation on a higher level of abstraction.

The EXPERT performance tool [9] supports the performance analysis of MPI and/or OpenMP applications by automatically searching traces for execution patterns that indicate inefficient behavior. The performance problems addressed include inefficient use of the parallel programming model and low CPU and memory performance. EXPERT is implemented in Python and its architecture consists

[*] This work was supported in part by the U.S. Department of Energy under Grants DoE DE-FG02-01ER25510 and DoE DE-FC02-01ER25490 and is embedded in the European IST working group APART under Contract No. IST-2000-28077

of two parts: a set of pattern specifications and an analysis unit that tries to match instances of the specified patterns while it reads the event trace once from the beginning to the end. Each pattern specification represents a different performance problem and consists of a Python class with methods to identify instances in the event stream. Although all pattern classes are organized in a specialization hierarchy, they are specified independently from each other resulting in replicated code and prolonged execution time whenever a pattern implementation reappears as part of another more specialized version.

This article describes an optimized design and search strategy leading to a significant speed improvement and more compact pattern specifications by taking advantage of specialization relationships. The design shares information among different patterns by looking in each step for more general patterns first and then successively propagating successful matches to more specialized patterns for refinement. We evaluate two implementations of the new design, one in Python and one in C++.

The article is outlined as follows: Section 2 describes related work. In Section 3, we outline EXPERT's overall architecture together with the current search strategy in more detail. After that, we explain the successive-refinement strategy in Section 4. Section 5 presents experimental results, followed by our conclusion in Section 6.

2 Related Work

The principle of successive refinement has also been used in the KAPPA-PI [3] post-mortem trace-analysis tool. KAPPA-PI first generates a list of idle times from the raw trace file using a simple metric. Then, based on this list, a recursive inference process continuously deduces new facts on an increasing level of abstraction.

Efficient search along a hierarchy is also a common technique in online performance tools: To increase accuracy and efficiency of its online bottleneck search, Paradyn stepwise refines its instrumentation along resource hierarchies, for example, by climbing down the call graph from callers to callees [2]. Fürlinger et al. [6] propose a strategy for online analysis based on a hierarchy of agents transforming lower-level information stepwise into higher-level information.

Ideas based on successive refinement can also be found in performance-problem specification languages, such as ASL [4] and JavaPSL [5], which is basically a Java version of ASL. Both use the concept of *metaproperties* to describe new performance problems based on existing ones.

3 Overall Architecture

EXPERT is part of the KOJAK trace-analysis environment [8,9] which also includes tools for instrumentation and result presentation (Figure 1). Depending on the platform and the availability of tools, such as built-in profiling interfaces of compilers, the application is automatically instrumented and then executed.

During execution, the program generates a trace file in the EPILOG format to be automatically postprocessed by EXPERT. EPILOG is able to represent region entry and exit events, MPI point-to-point and collective communications, as well as OpenMP fork-join, synchronization, and work-sharing operations.

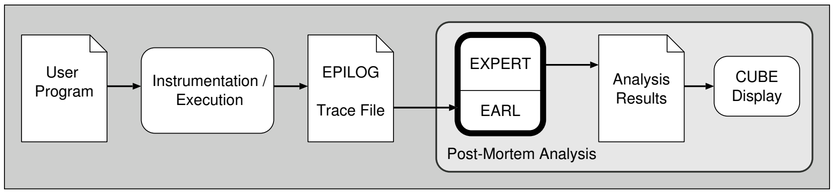

Fig. 1. The KOJAK trace-analysis environment.

To simplify pattern specification, EXPERT uses the EARL library [8] to access individual events. EARL provides a high-level interface to the event trace with random-access capabilities. Events are identified by their relative position and are delivered as a set of attribute values, such as time and location. In addition to providing random access to single events, EARL simplifies analysis by establishing links between related events, such as a link pointing from a region exit event to its corresponding enter event, and identifying event sets that describe an application's execution state at a given moment, such as the set of send events of messages currently in transit. EARL is implemented in C++ and provides both a C++ and a Python interface. It can be used independently of EXPERT for a large variety of trace-analysis tasks.

EXPERT transforms event traces into a compact representation of performance behavior, which is essentially a mapping of tuples (performance problem, call path, location) onto the time spent on a particular performance problem while the program was executing in a particular call path at a particular location. Depending on the programming model, a location can be either a process or a thread. After the analysis has been finished, the mapping is written to a file and can be viewed using CUBE [7], which provides an interactive display of the three-dimensional performance space based on three coupled tree browsers.

3.1 Current Search Strategy

There are two classes of search patterns, those that collect simple profiling information, such as communication time, and those that identify complex inefficiency situations, such as a receiver waiting for a message. The former are usually described by pairs of enter and exit events, whereas the latter are described by more complex compound events usually involving more than two events. All patterns are arranged in a hierarchy. The hierarchy is an inclusion hierarchy with respect to the inclusion of execution-time interval sets exhibiting the performance behavior specified by the pattern.

EXPERT reads the trace file once from the beginning to the end. Whenever an event of a certain type is reached, a callback method of every pattern class that has registered for this event type is invoked. The callback method itself then might access additional events by following links or retrieving state information to identify a *compound event* representing the inefficient behavior. A compound event is a set of events hold together by links and state-set boundaries and satisfying certain constraints.

3.2 Example

The example of a process waiting for a message as a result of accepting messages in the wrong order illustrates the current search strategy. The situation is depicted in Figure 2. Process A waits for a message from process B that is sent much later than the receive operation has been started. Therefore, most of the time consumed by the receive operation of process A is actually idle time that could be used better. This pattern in isolation is called *late-sender* and is enclosed in the spotted rectangle. EXPERT recognizes this pattern by waiting for a receive event to appear in the event stream and then following links computed by EARL (dashed lines in Figure 2) to the enter events of the two communication operations to determine the temporal displacement between these two events (idle time in Figure 2).

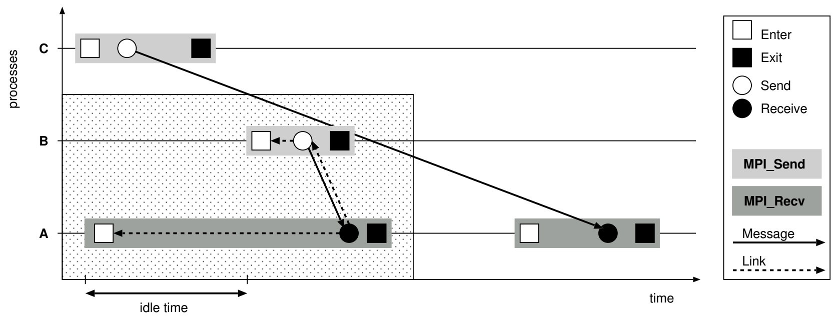

Fig. 2. Idle time as a result of receiving messages in the wrong order.

Looking at this situation in the context of the other message sent from process C to A allows the conclusion that the late-sender pattern could have been avoided or at least alleviated by reversing the acceptance order of these two messages. Because the message from C is sent earlier than that from B, it will in all probability have reached process A earlier. So instead of waiting for the message from B, A could have used the time better by accepting the message from C first. The late-sender pattern in this context is called *late-sender / wrong-order*. EXPERT recognizes this situation by examining the execution state computed by EARL at the moment when A receives the message from B. It inspects the queue of messages (i.e., their send events) sent to A and checks whether there are older messages than the one just received. In the figure, the queue would contain the event of sending the message from C to A.

This combined situation is a specialization of the simple late-sender. However, in spite of this relationship, both of them are currently computed independently. Whenever a receive event is reached, the late-sender callback follows links to see whether there is a receiver waiting unnecessarily. In addition, the callback method responsible for detecting the combined late-sender / wrong-order situation first checks the message queue for older messages and, if positive, tries to identify a late-sender situation as well - resulting in a part of the work done twice.

3.3 Profiling Patterns

A similar problem occurs with patterns collecting profiling information. Here, every pattern filters out matching pairs of enter and exit events that form region instances satisfying certain criteria, such as invocations of synchronization routines, to accumulate the instances' durations. Here also, a large fraction of the work is done more than once, when for example, one pattern accumulates communication time while a specialization of it accumulates collective-communication time.

4 Successive Refinement

The new search strategy is based on the principle of successive refinement. The basic idea is to pass a compound-event instance, once it has been detected by a more general pattern, on to a more specialized pattern, where it can be reused, refined, and/or prepared for further reuse. In the previous version, patterns have only been able to register for primitive events, that is, events as they appear in the event stream, as opposed to compound events consisting of multiple primitive events. The new design allows patterns also to publish compound events that they have detected as well as to register for compound events detected by others.

Figure 3 shows three pattern classes in EXPERT's pattern hierarchy. The hierarchical relationships between the three classes are expressed by their `parent()` methods. The `register()` methods register a callback method with the analysis unit to be invoked either upon a primitive event or a compound event. Descriptions of valid compound-event types for which a callback can be registered are stored in a developer map and can be looked up there manually. At runtime, the callback methods may return either a valid compound-event instance or `None`. If a valid instance is returned, the analysis unit invokes all callbacks subscribing for this type of compound event and a data structure holding the instance is supplied as an argument to theses callbacks.

The class `P2P` registers the method `recv()` as callback for a primitive receive event. `recv()` returns a compound event representing the entire receive operation (`recv_op`) including the receive event and the events of entering and leaving the MPI call. This corresponds to the left gray bar on the time line of process A in Figure 2. The `LateSender` class registers the method `recv_op()` for the compound event returned by `recv()` and tries there to identify the remaining

```
class P2P(Pattern):
    [...]
    def register(self, analyzer):
        analyzer.subscribe('RECV', self.recv)
    def recv(self, recv):
        [...]
        return recv_op

class LateSender(Pattern):
    [...]
    def parent(self):
        return "P2P"
    def register(self, analyzer):
        analyzer.subscribe('RECV_OP', self.recv_op)
    def recv_op(self, recv_op):
        if [...]
            return ls
        else:
           return None

class LateSendWrOrd(Pattern):
    [...]
    def parent(self):
        return "LateSender"
    def register(self, analyzer):
        analyzer.subscribe('LATE_SENDER', self.late_sender)
    def late_sender(self, ls):
        pos     = ls['RECV']['pos']
        dest_id = ls['RECV']['loc_id']
        queue   = self._trace.queue(pos, -1, dest_id)
        if queue and queue[0] < ls['SEND']['pos']:
            loc_id   = ls['ENTER_RECV']['loc_id']
            cnode_id = ls['ENTER_RECV']['cnodeptr']
            self._severity.add(cnode_id, loc_id, ls['IDLE_TIME'])
        return None
```

Fig. 3. Late-sender / wrong-order pattern based on successive refinement.

parts of the late-sender situation, as depicted in the spotted rectangle in Figure 2. In the case of a positive result, the successfully matched instance is returned and passed on to the `LateSendWrOrd` class. It is supplied as the `ls` argument to the `late_sender()` method, which has previously been registered for this type of compound event. `ls` is a Python dictionary containing the various constituents of the compound event, such as the receive event and the enter event of the receive operation, plus calculated values, such as the idle time lost by the receiver. The method examines the queue to see whether there are older messages that could have been received before the late message. If positive, the idle time is accumulated.

The difference from the old version (not shown here) is that the results are shared among different patterns so that situations that appear again as part of others are not computed more than once. The sharing works in two ways. First, if a compound event is successfully matched, it is passed along a path in the pattern hierarchy and is refined from a common to a more specialized situation by adding new constituents and constraints. Second, since subscribers on a deeper level of the hierarchy are not invoked if the match was already unsuccessful on a higher level, negative results are shared as well, which increases the search efficiency even further by suppressing matching attempts predicted to be unsuccessful. In contrast, the old version might, for example, try to match the late-sender situation twice: the first time as part of the late-sender pattern and the second time as part of the late-sender / wrong-order pattern even if the simple late-sender pattern was already unsuccessful. In addition to being more efficient, the new design is also more compact since replicated code has been eliminated.

The profiling patterns mentioned in Section 3.3 offered a similar opportunity for optimization. Previously, every profiling pattern had to do both: accumulating time and hardware-counter values and then filtering based on call-path properties. The new design performs accumulation centrally by calculating a (call path, location) matrix for the execution time and every hardware counter recorded in the trace. After the last event has been reached, the matrices are distributed to all profiling-pattern classes where the filtering takes place. Because now the accumulation is done once for all patterns and because filtering is done only once per call path as opposed to once per call-path instance, the new version is again much more efficient.

5 Evaluation

We evaluated our new strategy using five traces from realistic applications, three from pure MPI codes and two from hybrid OpenMP/MPI codes (Table 1). For one of the applications, the SWEEP3D ASCI benchmark, we ran the original MPI version monitoring also cache-miss values and a hybrid version without hardware counter measurements. The second and the third rows of the table contain the number of CPUs used to generate the traces as well as the trace-file sizes in millions of events. Please refer to [8] for further details about the codes.

Table 1. Execution times of the old and the new implementation.

		TRACE	CX3D	SWEEP3D-HW	SWEEP3D	REMO
Type		MPI	MPI	MPI	hybrid	hybrid
No. of CPUs		16	8	16	16	16
Events	[10^6]	19.7	1.9	0.4	4.7	11.1
Old Python	[min]	332.0	30.9	4.8	60.3	285.1
New Python	[min]	42.0	12.5	1.7	14.6	22.0
New C++	[min]	44.0	13.9	2.3	6.8	2.5
New Python	[speedup]	7.9	2.5	2.8	4.1	13.0
New C++	[speedup]	7.6	2.2	2.1	8.9	114.0

We compared two implementations of our new strategy, one in Python and one in C++, against the old Python implementation. The test platform was an IBM AIX system with a Power4+ 1.7 GHz processor. The first three rows below the event numbers list the plain execution times in minutes, the last two rows give speedup factors in relation to the old version for a more convenient comparison.

In all examples the new strategy implemented in Python achieved a speedup of at least a factor of 2.5 and in many cases the speedup was even much higher (13.0 maximum). Although the C++ version obtained significant additional speedup for hybrid traces (e.g., 114.0 vs. 13.0 for REMO), it was surprisingly unable to deliver any additional performance in the analysis of pure MPI traces, which we hopefully can improve in the near future.

6 Conclusion

The benefit of our new design is twofold: a significant runtime improvement by avoiding repetition of detection work on the one hand and less redundant and therefore more compact pattern specifications on the other hand. In particular the latter achievement will allow us to extend the set of patterns more easily in the future. An integration of the underlying concepts into the ASL [4] specification language might help share these results with a broader community. The KOJAK software is available at http://www.fz-juelich.de/zam/kojak/.

References

1. A. Arnold, U. Detert, and W. E. Nagel. Performance Optimization of Parallel Programs: Tracing, Zooming, Understanding. In R. Winget and K. Winget, editors, *Proc. of Cray User Group Meeting*, pages 252–258, Denver, CO, March 1995.
2. H. W. Cain, B. P. Miller, and B. J. N. Wylie. A Callgraph-Based Search Strategy for Automated Performance Diagnosis. In *Proc. of the 6th International Euro-Par Conference*, volume 1999 of *Lecture Notes in Computer Science*, Munich, Germany, August/September 2000. Springer.
3. A. Espinosa. *Automatic Performance Analysis of Parallel Programs.* PhD thesis, Universitat Autonoma de Barcelona, September 2000.
4. T. Fahringer, M. Gerndt, B. Mohr, G. Riley, J. L. Träff, and F. Wolf. Knowledge Specification for Automatic Performance Analysis. Technical Report FZJ-ZAM-IB-2001-08, ESPRIT IV Working Group APART, Forschungszentrum Jülich, August 2001. Revised version.
5. T. Fahringer and C. Seragiotto Júnior. Modelling and Detecting Performance Problems for Distributed and Parallel Programs with JavaPSL. In *Proc. of the Conference on Supercomputers (SC2001)*, Denver, Colorado, November 2001.
6. K. Fürlinger and M. Gerndt. Distributed Application Monitoring for Clustered SMP Architectures. In *Proc. of the 9th International Euro-Par Conference*, Klagenfurt, Austria, August 2003.
7. F. Song and F. Wolf. CUBE User Manual. Technical Report ICL-UT-04-01, University of Tennessee, Innovative Computing Laboratory, Knoxville, TN, 2004.
8. F. Wolf. *Automatic Performance Analysis on Parallel Computers with SMP Nodes.* PhD thesis, RWTH Aachen, Forschungszentrum Jülich, February 2003. ISBN 3-00-010003-2.
9. F. Wolf and B. Mohr. Automatic performance analysis of hybrid MPI/OpenMP applications. *Journal of Systems Architecture*, 49(10-11):421–439, 2003. Special Issue "Evolutions in parallel distributed and network-based processing".