FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Performance Analysis Environment for SVM-Fortran Programs

Michael Gerndt

KFA-ZAM-IB-9417

Juli 1994
(Stand 21.07.94)

Performance Analysis Environment for SVM-Fortran Programs

M. Gerndt

Research Centre Jülich (KFA)
Central Institute for Applied Mathematics
52425 Jülich
Germany

Abstract

This report outlines the design of a performance analysis environment for SVM-Fortran programs. SVM-Fortran is a shared memory parallel programming language developed at KFA for distributed memory multiprocessors. The environment allows to analyze the data locality of a given program via runtime tracing and supports the identification of critical code regions to guide the user or an optimization tool in tuning the program. To reduce the amount of runtime data, the environment combines trace data with static program information and supports an incremental analysis cycle. In addition to the overall design, we present the trace format of runtime events. The trace format includes symbolic information to relate runtime information to the program text which is the essential part of the user interface. The performance analysis environment is part of OPAL, a tool that will combine performance analysis and program optimization.

The work described in this report is being carried out as a part of the Esprit project "Performance-Critical Applications of Parallel Architectures (APPARC) " and of the KFA-Intel collaboration.

Contents

1 Motivation

Massively parallel computers (MPP) offer an immense peak performance. Current architectures consist of hundreds of nodes with physically distributed memory and are either pure distributed memory systems (Paragon), hardware-supported shared memory systems (KSR,Cray T3D), or software-based shared virtual memory machines (Koan/iPSC2, MYOAN/Paragon, ASVM/Paragon).

Shared virtual memory (SVM) machines provide a global address space on top of physically distributed memory via an extension of the virtual memory paging mechanism [1]. Page faults on one processor are served by another processor which currently has the page of the global address space in its private memory. On such systems message passing and shared memory are both available for data exchange among processors.

Different programming models can be provided on top of the basic architectural model. The message passing model is the most important model on distributed memory machines. Recently, the data parallel programming model is promoted for massively parallel systems and was standardized as High Performance Fortran (HPF) [4]. In HPF potential parallelism is identified by the user in form of vector statements and dependence-free loops however the programmer has no explicit control of parallel activities. This is the main feature of the shared memory programming model for MPPs, e.g. the Craft programming model of Cray T3D, KSR Fortran, Fortran-S, and SVM-Fortran.

Our work aims at providing a shared memory parallel programming model on distributed memory systems. The message passing model is too low-level for application programmers. The data parallel model is difficult to implement because it is the compilers task to resolve remote memory accesses via message passing code. In addition, the goal of freeing the user from the burden of actually doing parallel programming will not be reached in the near future. Although, the user does not need to control parallelism in the language he has to understand the generated parallelism to debug and optimize his code.

In the shared memory programming model the user has to control the parallelism explicitly. Additionally, he can access data in the wellknown type of memory references and he can parallelize the application in an incremental manner. Although the basic concepts of shared memory programming are well understood, shared memory programming on massively parallel systems has an important new problem: data locality.

Access to remote memory is much more expensive in current MPPs than access to local memory. This is especially true for SVM systems since remote memory access is implemented via software. The user has to be aware of this difficulty and has to write and optimize his program with respect to this fact. This optimization can be done only if the user is able to understand the runtime behaviour of his code.

Runtime performance information can additionally be used for dynamic optimization. For example, page distribution and page travel information can help to dynamically distribute work to processors. When carefully designing the interface for accessing performance information at runtime it can be used for runtime optimization as well as post-mortem performance analysis.

Two concepts have been developed to allow analysis of the runtime behaviour of programs: trace visualization and performance prediction.

For the message passing model several graphical tools have been implemented that visualize the program behaviour on the basis of runtime traces [5, 6, 7, 8, 9]. KSR provides a tool called GIST [10] that visualizes execution traces from KSR Fortran parallel programs. The traces contain all information that might be of interest in the form of individual events. The tools provide several graphical displays to visualize the information.

The major drawbacks in generating traces are the immense amount of data generated and its influence on program behaviour. These drawbacks will be even more severe when looking

for information about memory references in shared memory parallel programs than for message passing programs. Therefore, it is very important to reduce the amount of information generated at runtime.

Our approach to data reduction is to exploit special language features, to do tracing on different levels of granularity, to allow selective tracing, and to do performance prediction.

- SVM-Fortran supports global work distribution via distribution of templates. The distribution directives are the point to monitor a specific distribution. This information need not be monitored at each loop where scheduling decisions are taken based on a template's distribution.
- Selective tracing allows to request specific information for individual areas of the parallel code. The user can, for example, request only read page faults for a specific array in a specific parallel loop.
- Different granularity levels can be used to incrementally analyze the code, starting from a coarse overview moving on to carefully analyzing individual parts of the code.
- Performance prediction is a new approach in parallel programming and aims at determining performance information from the program text. It is currently pursued to facilitate programming in HPF [11, 12]. Since static analysis is often impossible because of missing runtime parameters, runtime information and static information has to be combined [13]. For example, if loop boundaries that are runtime dependent were traced in a previous run a tool might be able to predict information on data locality based on the references in the loop.

The presented concepts for data reduction are effective only if specific and useful information is requested. A tool might be able to request information automatically from previous, perhaps coarser information. Such an automatic approach requires knowledge about typical performance bottlenecks.

At least as important as the question *"How to generate data?"* is the question *"How to present the data?"*. Visualization tools present information either as an animation of the dynamic behaviour or in form of statistics. Although visualization may give the user insight in the dynamic behaviour, such tools lack source-code information and do not provide an automatic detection of performance bottlenecks.

Our approach to data visualization is twofold. We developed a graphical visualization tool, PARvis [9], that includes state-of-the-art visualization techniques and allows trace analysis on different granularity levels via sophisticated zooming and flexible scrolling techniques. In addition, we plan to develop a source-code-based analysis tool, that will be able to identify performance bottlenecks automatically with the help of information on possible types of performance bottlenecks and by automatically analyzing program behaviour. Determined information will then be presented directly by annotating source code.

2 Design

The basis of our research on SVM is SVM-Fortran [14] and several SVM implementation on Paragon. The overall design is outlined in Figure 1.

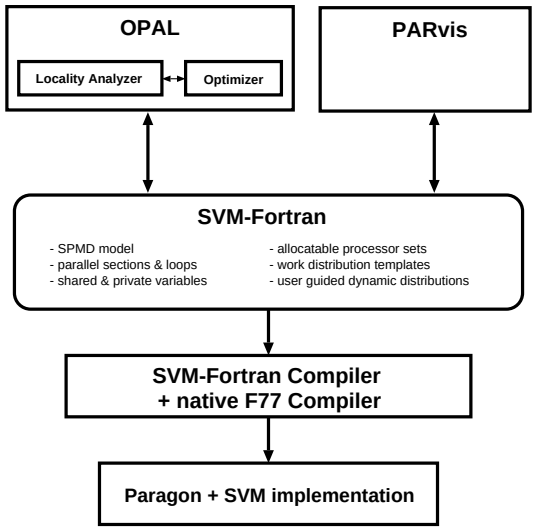

Figure 1: SVM-Fortran programming environment

SVM-Fortran is a shared memory programming extension of FORTRAN 77 aiming at language support for optimizing data locality. Its main features are outlined in the next section. The target system is Intel Paragon for which multiple research groups are developing SVM implementations. We plan to use the external servers MYOAN (IRISA) [2] and MAX (TU München) [3], and ASVM (Intel ESDC), a kernel implementation of SVM.

On top of SVM-Fortran we will provide programming tools. OPAL (Optimizer and Locality Analyzer) will combine our performance analysis techniques with source code optimizations. In addition, PARvis will be available to analyze the behaviour of SVM-Fortran programs by visualization techniques. PARvis is another tool which is provided to analyze SVM-Fortran trace files. It gives the user insight in the dynamic behaviour of the application by visualization techniques and by gathering statistics on arbitrary intervals of the total runtime.

In the following we describe the general performance analysis concept for SVM-Fortran programs and the design of OPAL.

Figure 2 gives a detailed overview of the performance analysis cycle. The user can start by specifying own trace regions via directives (A). Trace regions are code sections for which specific runtime information can be traced. Most regions the user might be interested in are trace regions by default, such as subroutines and parallel loops.

If the user does not want to specify own trace regions he can directly start by compiling the code for performance analysis with the SVM-Fortran compiler and the native compiler of the target system (B). The SVM-Fortran compiler inserts specific hooks to trace runtime information.

In the second phase, trace requests are specified for individual trace regions in a trace request file which is input to the execution and guides runtime tracing.

The SVM-Fortran compiler translates the SVM-Fortran program into a FORTRAN 77 program and generates some mapping information in the program information file. For example, all shared variables in a subroutine are numbered consecutively and variables are identified at runtime via these numbers. This mapping is used to implement trace requests for specific variables.

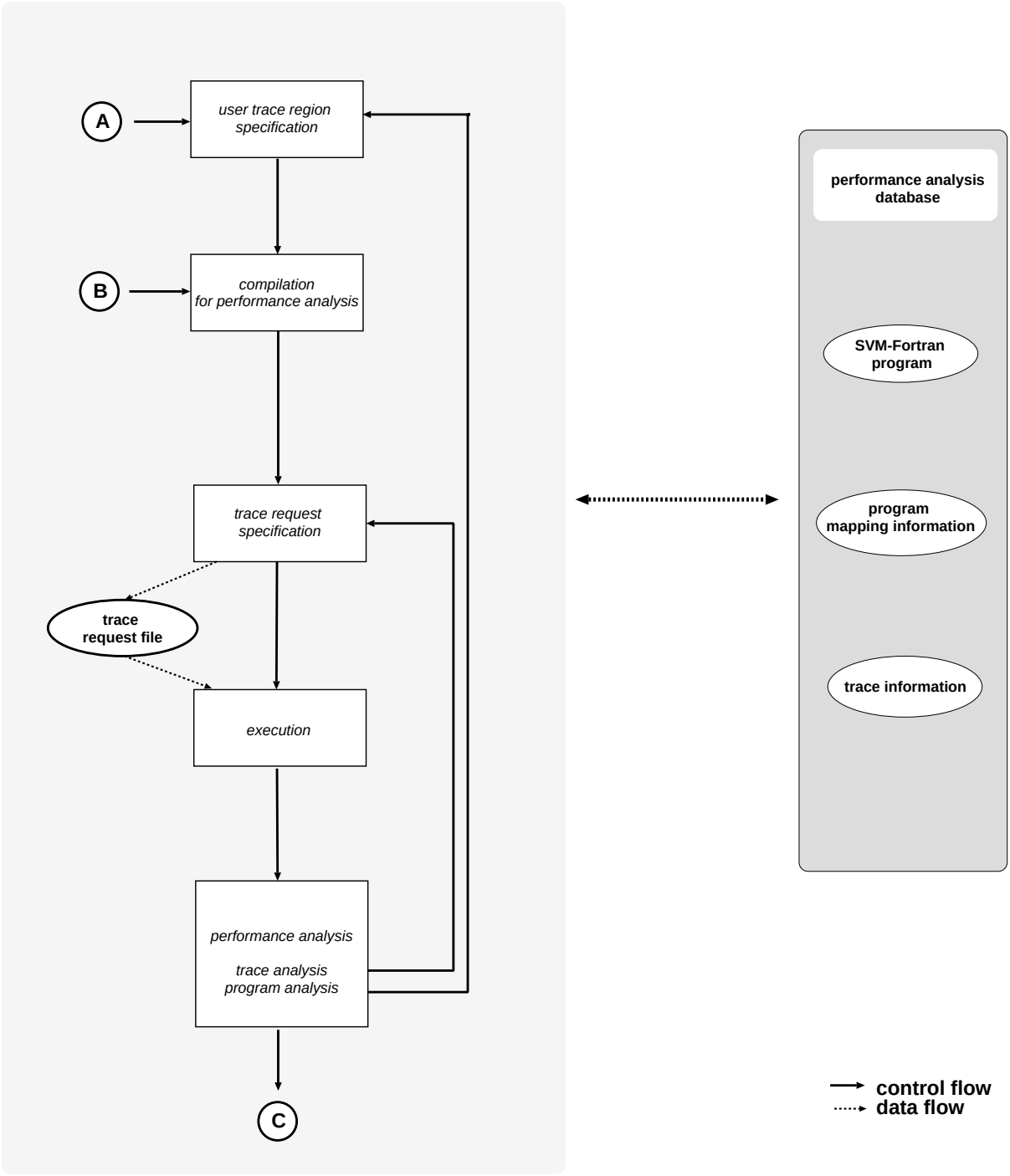

Figure 2: SVM-Fortran performance analysis

During execution, events are written to a trace file according to the trace requests. Events are generated in two different locations: the SVM-kernel and the SVM-Fortran runtime library. The runtime library will read events from a kernel event buffer and a program event buffer and will write the events in the trace format described in this document.

Trace events are used in the analysis step in combination with information gathered from the source code to identify performance bottlenecks. The compiler's mapping information is needed in this step to correctly relate trace information to the source code.

If performance bottlenecks cannot be identified precisely with the available information (exit C) the trace request file can be adapted and more information be gathered without recompilation. All information used in performance analysis is combined in a database.

OPAL will assist the user in all performance analysis phases:

1. user trace region specification

   The tool generates directives for interactively marked code sections.

2. trace request specification

   OPAL will support trace request specification in three modes. It will provide the menu-driven analysis of trace information and the menu-driven specification of trace requests. This mode frees the user from the subtle trace request syntax.

   An advanced interactive mode will guide the user in analyzing the available information and will suggest additional requests. This mainly facilitates handling of big applications.

   In an automatic mode OPAL will automatically go through the performance analysis steps and determine performance bottlenecks via the incremental analysis outlined in Figure 2.

3. performance analysis

   OPAL will provide a source code based interface to manually analyze trace data. Based on runtime information, e.g. loop bounds and iteration distribution, OPAL may predict performance information by taking into account static program information. In addition, OPAL will have an automatic analysis mode in which it gathers information and will point the user to the specific problems of his code.

3 SVM-Fortran Concepts

Since the performance analysis environment of SVM-Fortran is source-code based, the trace format reflects the language features. In the following we give a very brief overview of SVM-Fortran. More details can be found in [14].

SVM-Fortran is a shared memory parallel Fortran 77 extension targeted mainly towards data parallel applications on shared virtual memory (SVM) systems. The main application area is broader than that of HPF and Vienna Fortran. SVM-Fortran supports coarse grained functional parallelism where the parallel tasks itself can be data parallel.

The execution model of SVM-Fortran is an extension of the Single-Program-Multiple-Data (SPMD) model [15] which is well-known for its low-overhead parallel execution and is best-suited for hierarchical memory machines. Processors are allocated to the parallel application when it is started and are available until program termination.

SVM-Fortran supports nested parallelism. At program start the entire computation forms a single task. A *task* is some computational work. Tasks can be dynamically decomposed into subtasks, e.g. via a parallel section or a parallel loop construct, i.e. each section and each loop iteration is an independent task. A task is either assigned to a single processor or to a set of processors which execute the task cooperatively. The set of processors executing a task is called the task's *active processor set (APS)*. The active processors execute the task either in exclusive mode or in replicated mode. In exclusive mode only one processor, the APS leader, performs the actual computation whereas in replicated mode all processors execute the same code. The default execution mode is exclusive mode.

SVM-Fortran provides the standard features of shared memory parallel Fortran languages, i.e. shared and private data, multi-dimensional parallel loops and parallel sections, classical synchronization operations as well as SVM-specific synchronization such as variable locking and atomic update.

SVM-Fortran provides specific features to determine the distribution of tasks onto processors. Similar to Fortran-S and KSR Fortran loop annotations can be used to determine a static or dynamic work distribution scheme. Examples for loop-level scheduling are BLOCK and CYCLIC as well as self scheduling and affinity scheduling [16].

Data locality is not a problem to be solved on the level of individual do-loops but is a global problem. Therefore, SVM-Fortran borrowed the concepts of processors arrangements and templates from HPF as tools to specify scheduling decisions globally via template distributions.

Parallel loops can use predefined scheduling or semi-dynamic scheduling. In predefined scheduling loop iterations are assigned to processors according to the distribution of the appropriate template element. In semi-dynamic scheduling templates are distributed according to dynamic scheduling decisions. Thus, a dynamic scheduling decision can be applied to subsequent loops to ensure data locality.

Templates can be handled very flexible. They can be created dynamically, distributed and redistributed at any point in the program and passed via the subroutine interface. SVM-Fortran supports standard distributions like BLOCK, CYCLIC, and GENERAL_BLOCK, indirect distributions and linked distributions. Indirect distributions allow the user to use arbitrary integer expressions in the specification of the target processor. Linked distribution is a form of alignment where a distribution is described via the distribution of another template.

Example 3.1:

```
        SUBROUTINE G(....,T,N,...)
CSVM$   TEMPLATE:: T(10,10)             ! fixed size template
CSVM$   TEMPLATE:: T1(N,N)              ! automatic template
CSVM$   TEMPLATE:: T2(:,:)              ! creatable template
CSVM$   PROCESSORS:: P(2,2)

CSVM$   DISTRIBUTE (BLOCK,BLOCK) ONTO P :: T1

        M=...
CSVM$   PDO(LOOPS(I,J),STRATEGY(ON_HOME(T1(I,J))))
        DO I=1,M
           DO J=1,M
              ...
           ENDDO
        ENDDO

CSVM$   CREATE:: T2(M,M)
CSVM$   REDISTRIBUTE (BLOCK,BLOCK) ONTO P :: T2

CSVM$   PDO(LOOPS(I,J),STRATEGY(ON_HOME(T2(I,J))))
        DO I=1,M
           DO J=1,M
              ...
           ENDDO
        ENDDO

CSVM$   PDO(LOOPS(I,J),PROCESSORS(P),STRATEGY(BLOCK,BLOCK))
        DO I=1,10
           DO J=1,10
              ...
           ENDDO
        ENDDO
```

This example illustrates templates as well as the scheduling features of SVM-Fortran. Template `T` is a fixed size template that is an argument of the subroutine. The shape of the local template `T1` is determined at runtime and depends on the argument `N`. The shape of the creatable template `T2` is adapted to the iteration space of the first two loops.

The first two loops use predefined scheduling according to a template's distribution. Since `T` is not adapted to the iteration space the resulting load balance may not be optimal. When adapting the size of template `T2` to the iteration space the template's distribution leads to a much better load balance for the second loop.

The third loop uses static loop-level scheduling. This feature is useful if the programmer does not want to declare and use templates for scheduling.

■

4 Trace format

The trace format was defined within the APPARC project. It consists of kernel events and language events. The format of the kernel events is common to MYOAN, MAX, and ASVM, whereas the language events are specific to SVM-Fortran.

The events are defined in the standard SDDF format [17]. This format was chosen since the structure of the individual records can easily be changed without complex changes to the I/O-components of the tools. Furthermore, differences in the data representation between the parallel system and the platform of the analysis tool are handled automatically by the SDDF library.

4.1 Kernel Events

The following table specifies the events generated by the SVM implementation and the reason or location where the event is generated.

Event	Location
read-page-fault-sum	trace region exit
write-page-fault-sum	trace region exit
read-page-fault-serviced	read page fault
write-page-fault-serviced	write page fault
invalidate-copy	receipt of invalidation
reduce-access-permission	receipt of read access request
page-travel	trace region exit
o-page-dist	start/end of trace region
c-page-dist	start/end of trace region
page-out	page is paged out

In the following description of the events *node id* is meant to be a global node number related to the program, thus a number between 0 and `NUMPROCS()`-1. Some events include the pair *data start address* and *data end address* which specifies a memory area for which the data were gathered. This is used when information is traced only for specific variables.

The events **o-page-dist** and **c-page-dist** capture the current distribution of pages. This global information is needed when, for example, more detailed tracing is required for a parallel loop, e.g. individual page faults, or when static program analysis is applied to predict performance characteristics.

The events provide data of different granularity. While **o-page-dist** and **c-page-dist** capture the current page distribution only, **read-page-fault-sum**, **page-travel**, and **read-page-fault-serviced** give more and more detailed information.

The following list describes the individual events:

read-page-fault-sum event record.

Read page faults are counted in a trace region for a specific data address range.

`node id`	Processor identification
`clock microseconds`	Time stamp
`segment id`	Data segment identification
`data start address`	Start address to count read page faults for
`data end address`	End address to count read page faults for
`number of page faults`	Number of read page faults in the address range

write-page-fault-sum event record.

Write page faults are counted in a trace region for a specific data address range.

`node id`	Processor identification
`clock microseconds`	Time stamp
`segment id`	Data segment identification
`data start address`	Start address to count write page faults for
`data end address`	End address to count write page faults for
`number of page faults`	Number of write page faults in the address range

read-page-fault-serviced event record.

A read page fault has occurred and is serviced:

`node id`	Processor identification
`fault clock microseconds`	Time when the read page fault occurred
`segment id`	Data segment identification
`faulting address`	Exact faulting address
`faulting page number`	Number of the faulting page
`instruction address`	Instruction that caused the read page fault
`node id that sent the page`	Processor identification of the sending node
`serviced clock microseconds`	Time when the page was received on the faulting node

write-page-fault-serviced event record.

A write page fault has occurred and is serviced:

`node id`	Processor identification
`fault clock microseconds`	Time when the write page fault occurred
`segment id`	Data segment identification
`faulting address`	Exact faulting address
`faulting page number`	Number of the faulting page
`instruction address`	Instruction that caused the write page fault
`node id that sent the page`	Processor identification of the sending node
`serviced clock microseconds`	Time when the page was received on the faulting node

invalidate-copy event record.

A write page fault has occurred and caused the invalidation of a copy of the page.

`node id`	Processor identification
`clock microseconds`	Time when the page was invalidated
`segment id`	Data segment identification
`faulting address`	Exact faulting address (remote task)
`faulting page number`	Number of the faulting page
`requesting node`	Processor identification of the sending node

reduce-access-permission event record.

A read page fault has occurred and caused the change of access permission of the page from write to read-only.

`node id`	Processor identification
`clock microseconds`	Time when the page was invalidated
`segment id`	Data segment identification
`faulting address`	Exact faulting address (remote task)
`faulting page number`	Number of the faulting page
`requesting node`	Processor identification of the sending node

page-travel event record:

This event record reflects the traveling of pages covering a specific address range between the processors in a trace region. Every processor records how many pages it has received from the other processors.

`node id`	Processor identification
`clock microseconds`	Time stamp
`segment id`	Data segment identification
`data start address`	Start address to count page transfers for
`data end address`	End address to count page transfers for
`number of received pages`	The i–th element of this array reflects the number of pages received from processor i.

o-page-dist event record:

The o-page-dist event states the current distribution of writable pages of a specific data address range. Every processor records the number of owned pages and the page numbers of these pages.

`node id`	Processor identification
`clock microseconds`	Time stamp
`segment id`	Data segment identification
`data start address`	Start address to count the distribution for
`data end address`	End address to count the distribution for
`number of owned pages`	Number of owned pages
`owned pages`	List of owned pages

c-page-dist event record:

The c-page-dist event states the current distribution of read-only pages of a specific data address range. Every processor records the number of copies and the page numbers of these copied pages.

`node id`	Processor identification
`clock microseconds`	Time stamp
`segment id`	Data segment identification
`data start address`	Start address to count the distribution for
`data end address`	End address to count the distribution for
`number of copied pages`	Number of copied pages
`copied pages`	List of copied pages

page-out event record:

A page is paged out to the backing store or another node due to lack of memory space.

`node id`	Processor identification
`clock microseconds`	Time stamp
`segment id`	Data segment identification
`page number`	Number of the page
`target node`	node id of target node

4.2 SVM-Fortran Language Events

The events include the following program text information, generated in the program information file by the SVM-Fortran compiler:

- subprogram number

  Each user subprogram can be identified by this number.

- directive number

  The SVM-Fortran directives of each subprogram are numbered consecutively according to their textual appearance in the subprogram.

- variable number

  For each subprogram all shared variables can be identified by this unique number.

Some events specifying information global to the active processor set (APS) are written only by its leader. Whether an event is such a global event is specified within the following event specifications.

Table 1 gives an overview of the possible events generated for SVM-Fortran language constructs. The following abbreviations are used:

all region start events:	o-page-dist
	c-page-dist
all region stop events:	page-travel
	read-page-fault-sum
	write-page-fault-sum

The trace requests in the trace request file determine which of these events are generated during execution on the basis of individual trace regions.

4.2.1 Global Scheduling

Event	Location
trace-region-start	trace region entry
trace-region-stop	trace region exit
template-info	TEMPLATE, CREATE directives
template-dist	DISTRIBUTE, REDISTRIBUTE directives
undef-template	UNDEF directive
destroy-template	DESTROY directive
proc-arrangement	PROCESSORS directive
pdo-info	PDO directive
pdo-iteration-space	end of PDO
section-info	SECTION of a PSECTION
aps-info	PDO, SECTION directives
variable-mapping	shared variable declaration

The event records **template-info**, **template-dist** and **proc-arrangement** are traced through the whole program regardless of the currently defined trace regions because it is possible to pass templates and processor arrangements through the subprogram interface.

Directive/Location	Events
user subroutine start	trace-region-start all region start events
user subroutine stop	all region stop events trace-region-stop
SHARED	mapping-event
PDO	trace-region-start pdo-info aps-info all region start events
end of PDO	template-dist all region stop events wait-barrier-start wait-barrier-stop pdo-iteration-space trace-region-stop
PSECTION	trace-region-start all region start events
SECTION	section-info aps-info all region start events
PSECTION_END	all region stop events wait-barrier-start wait-barrier-stop trace-region-stop
BARRIER	wait-barrier-start wait-barrier-stop
BARRIER_CHECKIN	barrier-checkin
BARRIER_CHECKOUT	wait-bcheckout-start wait-bcheckout-stop
CRITICAL_SECTION	wait-crit-sec-start wait-crit-sec-stop
CRITICAL_SECTION_END	exit-crit-sec
LOCK	wait-lock-start wait-lock-stop
LOCK_END	release-lock
ATOMIC UPDATE	wait-lock-start wait-lock-stop
PROCESSORS	proc-arrangement
TEMPLATE	template-info
UNDEF	undef-template
CREATE	template-info
DESTROY	destroy-temlate
DISTRIBUTE	template-dist
REDISTRIBUTE	template-dist
REPLICATED_REGION	trace-region-start all region start events
REPLICATED_REGION_END	all region stop events wait-barrier-start wait-barrier-stop trace-region-stop
EXCLUSIVE_REGION	trace-region-start all region start events
EXCLUSIVE_REGION_END	all region stop events wait-barrier-start wait-barrier-stop trace-region-stop

Table 1: SVM-Fortran directives and events

trace-region-start event record:

A process begins to work in a trace region.

`node id`	Processor identification
`clock microseconds`	Time when the trace region is exited
`subprogram number`	Trace region identification
`directive number`	
`trace region type`	subroutine, psection, pdo, replicated region or exclusive region

trace-region-stop event record:

A process stops working in a trace region.

`node id`	Processor identification
`clock microseconds`	Time when the trace region is exited
`subprogram number`	Trace region identification
`directive number`	
`trace region type`	subroutine, psection, pdo, replicated region or exclusive region

template-info event record:

Information about a template is written by the APS leader.

`node id`	Processor identification
`clock microseconds`	Time stamp
`template id`	Template identification
`subprogram number`	
`directive number`	
`number of dimensions`	Number of dimensions
`template shape`	Template shape

template-dist event record:

The template-dist event holds the distribution of a template. Instead of writing the template directly, i. e. for every template element write the processor number that owns this element, we suggest writing the inverse function of the template distribution: every processor writes his own local elements of the template.

`node id`	Processor identification
`clock microseconds`	Time stamp
`template id`	Template identification
`owned indices`	Template indices that the processor owns

undef-template event record:

The undef-template event occurs if a template is set undefined.

`node id`	Processor identification
`clock microseconds`	Time stamp
`template id`	Template identification

destroy-template event record:

The destroy-template event occurs if a creatable template is no longer needed and destroyed with a DESTROY directive.

`node id`	Processor identification
`clock microseconds`	Time stamp
`template id`	Template identification

proc-arrangement event record:

Information about a processor arrangement is written by the leader of the active processor set.

`node id`	Processor identification
`clock microseconds`	Time stamp
`proc arrangement id`	Processor arrangement identification
`subprogram number`	
`directive number`	
`number of dimensions`	Number of dimensions
`shape of dimensions`	For every dimension start and end index
`pids`	Processor identifications of the processors in column–major order

pdo-info event record:

A parallel loop is encountered by the active processor set and the leader writes the pdo-info event record.

`node id`	Processor identification
`clock microseconds`	Time stamp
`subprogram number`	PDO identification
`directive number`	
`proc arrangement id`	Processor arrangement identification
`proc arrangement shape`	user-specified shape
`block cyclic length`	user-specified block length (loop-level scheduling)
`const template indices`	values of constant indices (predefined scheduling, dynamic scheduling)

pdo-iteration-space event record:

At the end of a parallel loop the APS leader writes the pdo-iteration-space event record.

`node id`	Processor identification
`clock microseconds`	Time stamp
`subprogram number`	PDO identification
`directive number`	
`iteration space`	for each loop 9 values min (lower bound, upper bound, stride) max (lower bound, upper bound, stride) increment (lower bound, upper bound, stride) 0 - no increment or variable increment value - constant increment

section-info event record:

One section of a parallel section is processed. The leader of the new formed APS writes the section-info event record. The `proc arrangement id` and `shape of dimensions` are optional.

`node id`	Processor identification
`clock microseconds`	Time stamp
`subprogram number`	SECTION identification
`directive number`	
`proc arrangement id`	Processor arrangement identification that works on this section
`shape of dimensions`	For every dimension start and end index

aps-info event record:

A new active processor set is formed and the leader of the APS writes the numbers of the involved processors.

`node id`	Processor identification
`clock microseconds`	Time when the trace region is exited
`pids`	Processor identifications of the processors

variable-mapping event record:

VM-address of shared variables.

`node id`	Processor identification
`clock microseconds`	Time stamp
`subprogram number`	
`variables/address`	List of variable number/VM-address pairs

4.2.2 Synchronization

SVM–Fortran offers several operations (barrier, lock, critical section, barrier checkin and checkout) to synchronize cooperating processes. In order to detect load imbalances or just to figure out how much time the processes spend waiting for each other we introduce the following event records.

Event	Location
wait-barrier-start	explicit or implicit BARRIER
wait-barrier-stop	explicit or implicit BARRIER
barrier-checkin	BARRIER_CHECKIN directive
wait-bcheckout-start	BARRIER_CHECKOUT directive
wait-bcheckout-stop	BARRIER_CHECKOUT directive
wait-lock-start	LOCK directive
wait-lock-stop	LOCK directive
release-lock	LOCK directive
wait-crit-sec-start	CRITICAL_SECTION directive
wait-crit-sec-stop	CRITICAL_SECTION directive
exit-crit-sec	CRITICAL_SECTION directive

wait-barrier-start event record.

A process starts waiting at a barrier.

`node id`	Processor identification
`clock microseconds`	Time stamp
`subprogram number`	BARRIER identification
`directive number`	

wait-barrier-stop event record.

A process stops waiting at a barrier and proceeds with its work.

`node id`	Processor identification
`clock microseconds`	Time stamp
`subprogram number`	BARRIER identification
`directive number`	

barrier-ckeckin event record.

A process enters a barrier via a `BARRIER CHECKIN` directive.

`node id`	Processor identification
`clock microseconds`	Time stamp
`subprogram number`	BARRIER identification
`directive number`	

wait-bcheckout-start event record:

A process starts to wait at a `BARRIER CHECKOUT` directive until all other cooperating processes have arrived at the corresponding `BARRIER_CHECKIN` directive.

`node id`	Processor identification
`clock microseconds`	Time stamp
`subprogram number`	BARRIER_CHECKOUT identification
`directive number`	

wait-bcheckout-stop event record:

A process stops waiting at a `BARRIER_CHECKOUT` directive.

`node id`	Processor identification
`clock microseconds`	Time stamp
`subprogram number`	BARRIER_CHECKOUT identification
`directive number`	

wait-lock-start event record:

A process wants to access a variable that is guarded by a lock and starts waiting on the lock to become free.

`node id`	Processor identification
`clock microseconds`	Time stamp
`subprogram number`	LOCK identification
`directive number`	
`locking address`	Address pairs of locked memory locations

wait-lock-stop event record:
A process gets access to a variable that is guarded by a lock.

`node id`	Processor identification
`clock microseconds`	Time stamp
`subprogram number`	LOCK identification
`directive number`	

release-lock event record:
A process releases a lock.

`node id`	Processor identification
`clock microseconds`	Time stamp
`subprogram number`	LOCK identification
`directive number`	

wait-crit-sec-start event record:
A process wants to enter a critical section and starts to wait.

`node id`	Processor identification
`clock microseconds`	Time stamp
`subprogram number`	CRITICAL_SECTION identification
`directive number`	

wait-crit-sec-stop event record:
A process is allowed to enter a critical section and stops waiting.

`node id`	Processor identification
`clock microseconds`	Time stamp
`subprogram number`	CRITICAL_SECTION identification
`directive number`	

exit-crit-sec event record:
A process is exits a critical section.

`node id`	Processor identification
`clock microseconds`	Time stamp
`subprogram number`	CRITICAL_SECTION identification
`directive number`	

4.2.3 SVM Run-Time Support

The mapping of shared segments into the virtual address space of the executing task is monitored to map page faults to variables.

To overcome the problems introduced by false–sharing it is possible to change the coherence strategy for a certain address space with the `COHERENCE` directive. The `PREFETCH` directive enables to hide network transfer latencies by fetching data before it is needed.

Event	Location
shared-segment-address	allocation of shared segment
coherence-switch	COHERENCE directive
prefetch	PREFETCH directive

shared-segment-address event record:

The address in the virtual address space for a shared segment. is changed.

`node id`	Processor identification
`clock microseconds`	Time stamp
`segment id`	Shared segment identification
`address`	Start address

coherence-switch event record:

The coherence strategy for a shared or partially shared variable is changed.

`node id`	Processor identification
`clock microseconds`	Time when the coherence is changed
`subprogram number`	COHERENCE identification
`directive number`	
`segment id`	Data segment identification
`data start address`	Start address to change the coherence for
`data end address`	End address to change the coherence for
`coherence strategy`	New coherence strategy
`time microseconds`	Time needed to change the coherence

prefetch event record:

Some shared or partially data is transferred to a processor on demand of a `PREFETCH` directive.

`node id`	Processor identification
`clock microseconds`	Time when the prefetch is initiated
`subprogram number`	PREFETCH identification
`directive number`	
`segment id`	Data segment identification
`data start address`	Start address of the data to be prefetched
`data end address`	End address of the data to be prefetched

4.2.4 Overhead Information

Since the collection of trace information produces some overhead on the system it is necessary to measure the time spent for trace actions. An equally important topic is the collection of SVM overhead data.

Event	Location
trace-overhead	trace overhead
svm-overhead	SVM runtime overhead
kernel-trace-buffer-overflow	trace buffer overflow

trace-overhead event record:

Time of specific trace overhead.

`node id`	Processor identification
`clock microseconds`	Time stamp
`duration microseconds`	overhead
`overhead type`	type of overhead

svm-overhead event record:

Time of SVM overhead.

`node id`	Processor identification
`clock microseconds`	Time stamp
`duration microseconds`	overhead
`overhead type`	type of overhead

kernel-trace-buffer-overflow event record:

The kernel trace buffer that the event records are written to has overflown.

`node id`	Processor identification
`clock microseconds`	Time stamp

5 User Interface

We already introduced the concept of *trace regions* that cover the interesting parts of the program. Trace regions can be nested, i.e. within a deeper nested trace region more specific information can be collected. In SVM-Fortran we define the following program parts as *implicit* trace regions:

- PROGRAM
- SUBROUTINE
- PDO
- PSECTION
- SECTION
- REPLICATED_REGION
- EXCLUSIVE_REGION

Stage 0:			All traces are switched off
Stage 1:			trace-region-start
		+	trace-region-stop
		+	trace-overhead
		+	trace-overflow
Stage 2:	Stage 1	+	pdo-info
		+	section-info
		+	aps-info
		+	template-info
		+	template-dist
		+	undef-template
		+	destroy-template
		+	proc-arrangement
Stage 3:	Stage 2	+	Synchronization events
Stage 4:	Stage 3	+	read-page-fault-sum
		+	write-page-fault-sum
Stage 5:	Stage 4	+	page-travel
		+	o-page-dist
		+	c-page-dist
		+	coherence-switch
		+	prefetch
Stage 6:	Stage 5	+	read-page-fault
		+	write-page-fault
		+	invalidate-copy

Table 2: Performance analysis stages

In addition to the implicit trace regions, the user can define trace regions with the directives `TRACE_USER_REGION` and `TRACE_USER_REGION_END`. User–defined trace regions shall fit into the structure of predefined trace regions (PROGRAM, SUBROUTINE, PDO, PSECTION, SECTION, REPLICATED_REGION, and EXCLUSIVE_REGION). That means that a user–defined

trace region can only appear inside of a subprogram. For example, if a `TRACE_USER_REGION` directive appears inside of a replicated region the corresponding `TRACE_USER_REGION_END` directive shall appear there, too.

To remove some burden of work from the user, SVM–Fortran's performance monitoring provides some standard information stages that help to get trace data from a coarse to a fine level. The stages are defined in Table 2.

The individual stages will be selectable via the trace request file. The request file is a sequence of local and global trace requests. Local trace requests are related to individual trace regions, e.g. loop information request, whereas global requests are related to the entire code.

All request can select special nodes, special trace regions, and individual variables. Variables and subroutines are identified by their names, and trace regions can be identified either by the do-loop label, a region number [1], or a user specified label.

The following syntax rules outline the structure of a trace request file. OPAL will provide a user-friendly interface for the specification of requests and will generate the appropriate entries automatically.

[1] Regions in a subroutine are numbered according to their textual appearance.

Syntax:

TraceRequestFile **is** *<FileEntryList>*

FileEntry **is** *<Comment>*
or `CONFIG` *<ConfigType>*
| *specification of configuration parameters*
or `INCLUDE` *<TR_Filename>*
| *reads another trace file*
or `REQUEST` *<RequestEntry>*
| *individual requests*

Comment **is** `%`

ConfigType **is** `SVM_TraceBuf_Size:` *<Size>*
or `SVM_Kernel_TraceBuf_Size:` *<Size>*
or `SVM_TraceStage:` *<StageNo>*
| *specification of a tracing stage*

RequestEntry **is** `[(`*<NodeSpec>*`)]` *<GlobalReq>*
or `[(`*<NodeSpec>*`)]` *<LocalReq>*
| *NodeSpec restricts this request to the specified nodes.*

NodeSpec **is** *<NodeList>*
or `*`
NodeList **is** `NodeNo[,`*<NodeList>*`]`
or *<NodeRange>*`[,`*<NodeList>*`]`

NodeRange **is** `NodeNo-NodeNo`

GlobalReq **is** `GLOBAL` *<GlobalReqList>*

LocalReq **is** `LOCAL` *<LocalReqEntry>*

⇒

Syntax:

GlobalReqList **is** <*GlobalReqType*>[,<*GlobalReqList*>]

GlobalReqType **is** `MappingInfo`
| *variable mapping*
| *template mapping*
| *processor arrangement mapping*

or `TrOverhead`
or `KernBufOverflow`
or `*`

LocalReqEntry **is** <*SubName*>.<*RegIdent*>:<*LocalReqList*>

SubName **is** *Name of Subroutine*
or `@MAIN`
| *main program*

or `*`
| *all user subprograms*

RegIdent **is** `PDO_LABEL (`<*LabelSpec*>`)`
or `REG_LABEL (`<*LabelSpec*>`)`
| *user-specified label*

or `REG_NO (`<*LabelSpec*>`)`
or `*`
| *all trace regions*

LocalReqList **is** (<*LocalReqType*>)[,<*LocalReqList*>]

LocalReqType **is** <*SVMInfo*>
or <*SyncInfo*>
or <*LangInfo*>

⇒

Syntax:

SVMInfo **is** `RPFSum[`<*Varlist*>`]`
or `WPFSum[`<*Varlist*>`]`
| *read and write page fault sums*
or `RPF[`<*Varlist*>`]`
or `WPF[`<*Varlist*>`]`
| *individual page faults*
or `InvalCopy[`<*Varlist*>`]`
| *invalidation information*
or `PageTravel[`<*Varlist*>`]`
| *page travel information*
or `OwnPageDist[`<*Varlist*>`]`
or `CurPageDist[`<*Varlist*>`]`
| *page distribution information*
or `PageOut[`<*Varlist*>`]`
| *paging to other nodes or disks*
or `SVMOverhead`
| *SVM runtime overhead*
or `*.SVMInfo[`<*Varlist*>`]`
| *all previous requests*

SyncInfo **is** `BARRIER`
or `LOCK`
or `WaitCritical`
or `*.SyncInfo`

LangInfo **is** `REGION`
| *start and stop time of regions*
or `PDOInfo`
| *scheduling and iteration space information*
or `SECTIONInfo`
| *scheduling information*
or `APSInfo`
| *changes in the APS for parallel loops and parallel sections*
or `CoSwitch`
or `Prefetch`
or `*.LangInfo`
| *all previous requests*

Varlist **is** <*VariableName*>`[,`<*Varlist*>`]`

◇

6 Status

SVM-Fortran is becoming more and more a stable language. There will be a reference manual in a few months. The compiler is currently working as a prototype implementing a single level of parallelism, global work distribution constructs, and synchronization features.

PARvis is a full functioning tool that will be used on the traces as soon as the SVM implementation will be able to generate the trace information. Currently an ASVM prototype is used on KFA's Paragon. This prototype does not include performance analysis support. This will be added in the next months.

We are currently designing the performance analysis part of OPAL and will have a first implementation of the menu-driven mode by the end of this year.

Thanks

I would like to thank Thierry Priol for his cooperation in designing the kernel trace format and my colleaques Andreas Krumme and Selçuk Özmen for providing me with the grammar of the trace request file and for their helpful comments in preparing this document.

References

[1] K. Li, *Shared Virtual Memory on Loosely Coupled Multiprocessors*, Ph.D. Dissertation, Yale University 1986, Technical Report YALEU/DCS/RR-492

[2] G. Cabillic, Th. Priol, I. Puaut, *MYOAN: an Implementation of the KOAN Shared Virtual Memory on the Intel Paragon*, IRISA, Technical Report No. 812, April 1994

[3] R.G. Hackenberg, *MaX - Investigating Shared Virtual Memory*, High-Performance Computing and Networking, Lecture Notes in Computer Science, Springer Verlag, Munich, Germany, April 94

[4] HPFF, *High Performance Fortran Language Specification*, High Performance Fortran Forum, May 1993, Version 1.0, Rice University Houston Texas

[5] Th. Bemmerl, O. Hansen, Th. Ludwig, *PATOP for Performance Tuning of Parallel Programs*, H.Burkhart (Ed.), Proceedings CONPAR 90 - VAPP IV, LNCS 457, 756-765, Zürich, 1990

[6] M.T. Heath, J.E. Finger, *ParaGraph: A tool for visualizing performance of parallel programs*, ParaGraph User Guide, Oak Ridge National Laboratory, December 1993

[7] B. Thomas, K. Peinze, *Suprenum comfort of parallel programming*, Supercomputer, Vol.6, No. 2, pp. 31-43, 1989

[8] D.A. Reed, R.A. Aydt, T.M. Madhyastha, R.J. Noe, K.A. Shields, B.W. Schwartz, *An Overview of the Pablo performance Analysis Environment*, Technical Report, University of Illinois, Department of Computer Science, 1992

[9] W.E. Nagel, A. Arnold, *Performance Visualization of Parallel Programs - The PARvis Environment* - , Caltech Technical Report, CCSF-47, May 1994

[10] Kendall Square Research Corp., *KSR/Series Performance Analysis*, KSR/Series Manuals, Waltham, 1993

[11] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, A. Schwald, *Vienna Fortran - A language Specification Version 1.1*, University of Vienna, ACPC-TR 92-4, March 1992

[12] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.W. Tseng, *An Overview of the Fortran D Programming System*, Rice University, Rice COMP TR91-154, March, 1991

[13] T. Fahringer, H. Zima, *A Static Parameter based Performance Prediction Tool for Parallel Programs*, Proceedings ICS'93, Tokyo, pp.207-219, 1993

[14] R. Berrendorf, M. Gerndt, W. Nagel, J. Prümmer, *SVM-Fortran*, Interner Bericht KFA-ZAM-IB-9322, Zentralinstitut für Angewandte Mathematik, Forschungszentrum Jülich, 1993

[15] F. Darema-Rogers, V.A. Norton, G.F. Pfister, *Using a Single-Program-Multiple-Data Computational Model for parallel Execution of Scientific Applications*, Research Report RC 11552 (#51726) 11/19/85, IBM Watson Research Center Yorktown Heights, 1985

[16] M.A. Linn, *Untersuchungen zur Ablaufplanung bei Parallelrechnern mit virutell gemeinsamem Speicher*, Dissertation, to be published, 1994

[17] Ruth A. Aydt, *The Pablo Self-Defining Data Format*, Department of Computer Science, University of Illinois, Urbana, Illinois 61801, 1993

A SVM–Kernel SDDF Trace Format

```
#1001:
// "event" "sum of read page faults"
"read-page-fault-sum" {
    int     "node id";
    double  "clock microseconds";
    int     "segment id";
    int     "data start address";
    int     "data end address";
    int     "number of page faults";
};;

#1002:
// "event" "sum of write page faults"
"write-page-fault-sum" {
    int     "node id";
    double  "clock microseconds";
    int     "segment id";
    int     "data start address";
    int     "data end address";
    int     "number of page faults";
};;

#1003:
// "event" "read page fault occur and service"
"read-page-fault-serviced" {
    int     "node id";
    double  "fault clock microseconds";
    int     "segment id";
    int     "faulting address";
    int     "faulting page number";
    int     "instruction address";
    int     "node id that sent the page";
    double  "serviced clock microseconds";
};;

#1004:
// "event" "write page fault occur and service"
"write-page-fault-serviced" {
    int     "node id";
    double  "fault clock microseconds";
    int     "segment id";
    int     "faulting address";
    int     "faulting page number";
    int     "instruction address";
    int     "node id that sent the page";
    double  "serviced clock microseconds";
};;
```

```
#1005:
// "event" "invalidation message received"
"invalidate-copy" {
    int     "node id";
    double  "clock microseconds";
    int     "segment id";
    int     "faulting address";
    int     "faulting page number";
    int     "requesting node id";
};;


#1006:
// "event" "reduce access permission"
"reduce-access-permission" {
    int     "node id";
    double  "clock microseconds";
    int     "segment id";
    int     "faulting address";
    int     "faulting page number";
    int     "requesting node id";
};;


#1007:
// "event" "page travel for a specific data region"
"page-travel" {
    int     "node id";
    double  "clock microseconds";
    int     "segment id";
    int     "data start address";
    int     "data end address";
    int     "received pages"[];
};;
#1008:
// "event" "distribution of writable pages"
"o-page-dist" {
    int     "node id";
    double  "clock microseconds";
    int     "segment id";
    int     "data start address";
    int     "data end address";
    int     "number of owned pages";
    int     "owned pages"[];
};;
```

```
#1009:
// "event" "distribution of read-only pages"
"c-page-dist" {
    int     "node id";
    double  "clock microseconds";
    int     "segment id";
    int     "data start address";
    int     "data end address";
    int     "number of copied pages";
    int     "copied pages"[];
};;

#1010:
// "event" "page out a page"
"page-out" {
    int     "node id";
    double  "clock microseconds";
    int     "segment id";
    int     "page number";
};;
```

B SVM–Fortran SDDF Trace Format

B.1 Global Scheduling

```
#1301:
// "event" "start a trace region"
"trace-region-start" {
    int     "node id";
    double  "clock microseconds";
    int     "subprogram number";
    int     "directive number";
    int     "trace region type";
};;

#1302:
// "event" "stop a trace region"
"trace-region-stop" {
    int     "node id";
    double  "clock microseconds";
    int     "subprogram number";
    int     "directive number";
    int     "trace region type";
};;

#1303:
// "event" "information about a template"
"template-info" {
    int     "node id";
    double  "clock microseconds";
    int     "template id";
    int     "subprogram number";
    int     "directive number";
    int     "number of dimensions";
    int     "template shape"[][];
};;

#1304:
// "event" "distribution of a template"
"template-dist" {
    int     "node id";
    double  "clock microseconds";
    int     "template id";
    int     "owned indices"[][];
};;
```

```
#1305:
// "event" "undefine a template"
"undef-template" {
    int     "node id";
    double  "clock microseconds";
    int     "template id";
};;

#1306:
// "event" "destroy a template"
"destroy-template" {
    int     "node id";
    double  "clock microseconds";
    int     "template id";
};;


#1307:
// "event" "arrangement of a processor set"
"proc-arrangement" {
    int     "node id";
    double  "clock microseconds";
    int     "proc arrangement id";
    int     "subprogram number";
    int     "directive number";
    int     "number of dimensions";
    int     "shape of dimensions"[][];
    int     "pids"[];
};;


#1308:
// "event" "start a parallel loop"
"pdo-info" {
    int     "node id";
    double  "clock microseconds";
    int     "subprogram number";
    int     "directive number";
    int     "proc arrangement id";
    int     "proc arrangement shape"[][];
    int     "block cyclic length"[];
    int     "const template indices"[];
    double  "startup time microseconds";
};;
```

```
#1309:
// "event" "parallel loop iteration space"
"pdo-iteration-space" {
    int     "node id";
    double  "clock microseconds";
    int     "subprogram number";
    int     "directive number";
    int     "iteration space"[][];
};;


#1310:
// "event" "start a section of a parallel section"
"section-info" {
    int     "node id";
    double  "clock microseconds";
    int     "subprogram number";
    int     "directive number";
    int     "proc arrangement id";
    int     "shape of dimensions"[][];
};;

#1311:
// "event" "new active processor set"
"aps-info" {
    int     "node id";
    double  "clock microseconds";
    int     "pids"[];
};;

#1312:
// "event" "shared variable mapping"
"variable-mapping" {
    int     "node id";
    double  "clock microseconds";
    int     "subprogram number";
    int     "variable address" [][]; // variable number, address
};;
```

B.2 Synchronization

```
#1101:
// "event" "start waiting at a barrier"
"wait-barrier-start" {
    int     "node id";
    double  "clock microseconds";
    int     "subprogram number";
    int     "directive number";
};;

#1102:
// "event" "stop waiting at a barrier"
"wait-barrier-stop" {
    int     "node id";
    double  "clock microseconds";
    int     "subprogram number";
    int     "directive number";
};;

#1103:
// "event" "checking in for a barrier"
"barrier-checkin" {
    int     "node id";
    double  "clock microseconds";
    int     "subprogram number";
    int     "directive number";
};;

#1104:
// "event" "start waiting at a barrier checkout"
"wait-bcheckout-start" {
    int     "node id";
    double  "clock microseconds";
    int     "subprogram number";
    int     "directive number";
};;

#1105:
// "event" "stop waiting at a barrier checkout"
"wait-bcheckout-stop" {
    int     "node id";
    double  "clock microseconds";
    int     "subprogram number";
    int     "directive number";
};;
```

```
#1106:
// "event" "start waiting at a lock"
"wait-lock-start" {
    int     "node id";
    double  "clock microseconds";
    int     "subprogram number";
    int     "directive number";
    int     "locking address"[][];
};;

#1107:
// "event" "stop waiting at a lock"
"wait-lock-stop" {
    int     "node id";
    double  "clock microseconds";
    int     "subprogram number";
    int     "directive number";
};;

#1108:
// "event" "release a lock"
"release-lock" {
    int     "node id";
    double  "clock microseconds";
    int     "subprogram number";
    int     "directive number";
};;

#1109:
// "event" "start waiting at a critical section"
"wait-crit-sec-start" {
    int     "node id";
    double  "clock microseconds";
    int     "subprogram number";
    int     "directive number";
};;


#1110:
// "event" "stop waiting at a critical section"
"wait-crit-sec-stop" {
    int     "node id";
    double  "clock microseconds";
    int     "subprogram number";
    int     "directive number";
};;
```

```
#1111:
// "event" "exit a critical section"
"exit-crit-sec" {
    int     "node id";
    double  "clock microseconds";
    int     "subprogram number";
    int     "directive number";
};;
```

B.3 Run-Time Support

```
#1201:
// "event" "shared segment address"
"shared-segment-address" {
    int     "node id";
    double  "clock microseconds";
    int     "segment id";
    int     "start address";
};;

#1202:
// "event" "switch coherence strategy"
"coherence-switch" {
    int     "node id";
    double  "clock microseconds";
    int     "subprogram number";
    int     "directive number";
    int     "segment id";
    int     "data start address";
    int     "data end address";
    int     "coherence strategy";
    double  "time microseconds";
};;

#1203:
// "event" "prefetch some data"
"prefetch" {
    int     "node id";
    double  "clock microseconds";
    int     "subprogram number";
    int     "directive number";
    int     "segment id";
    int     "data start address";
    int     "data end address";
};;
```

B.4 Trace Overhead

```
#1401:
// "event" "time for trace actions"
"trace-overhead" {
    int     "node id";
    double  "clock microseconds";
    double  "duration microseconds";
    int     "overhead type";
};;


#1402:
// "event" "time for svm actions"
"svm-overhead-start" {
    int     "node id";
    double  "clock microseconds";
    double  "duration microseconds";
    int     "overhead type";
};;

#1403:
// "event" "overflow of kernel trace buffer"
"kernel-trace-buffer-overflow" {
    int     "node id";
    double  "clock microseconds";
};;
```