

# A Domain Specific Language for Execution Profiling & Regulation

Peter Nguyen

Katrina Falkner

Henry Detmold

David S. Munro

School of Computer Science, University of Adelaide  
Email: {petern, katrina, henry, dave}@cs.adelaide.edu.au

## Abstract

Profiling consists of three stages: the collection of performance data, the processing of that data to infer performance information and the feedback of this performance information into the system. Feedback refers to using the information to change either the runtime system or the sampling parameters for subsequent profiling runs. This paper will concentrate on the latter approach for feedback. The majority of existing profiling tools focus on data collection, requiring manual intervention during processing and feedback. The developer must interpret the results presented to them to identify new profiling strategies.

We introduce the concept of profiling regulation, whereby the processes of collection, processing and feedback are automated. We define a domain specific language, *sampspec*, that provides expressibility and control over the profiling process. The developer provides a declarative specification of the information to collect, the computations to perform and the strategies to employ based on this information. This is in contrast to the manual inspection of results and restarting the profiler. Thus, the profiling process becomes one of specification of strategies for data collection and processing, and how these strategies can adapt over time. In this paper, we describe the system model and illustrate our language through a series of worked examples.

## 1 Introduction

Profiling is the process of gathering information about a runtime system to determine the performance of the system under certain circumstances. This information can be used to reveal areas of the system that are computationally intensive, meaning changes can be made to improve the performance of the relevant parts of the runtime system. The majority of existing profiling systems use a profiling process illustrated in Figure 1, comprised of three stages: collection, processing and feedback. Collection refers to the process of obtaining system state data, processing involves computing over the data to obtain performance information while feedback involves using the performance information to either make changes to the relevant system components or to tune profiling parameters so that different information is collected about the runtime system. The feedback strategies are illustrated in Figures 2a and 2b. In this paper, feedback takes the form of the latter approach.

Having greater control over the execution of the profiling system can be beneficial, especially in the case where the developer has a knowledge of how their system works. In terms of collection, it provides increased flexibility due to the fact that the developer is able to make decisions

Copyright ©2009, Australian Computer Society, Inc. This paper appeared at the 32nd Australasian Computer Science Conference (ACSC 2009), Wellington, New Zealand. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 91, Bernard Mans, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

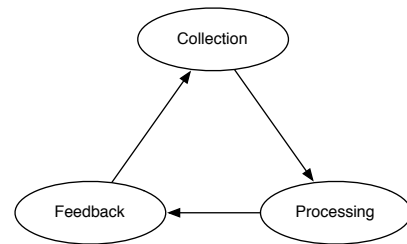


Figure 1: The three stages of profiling.

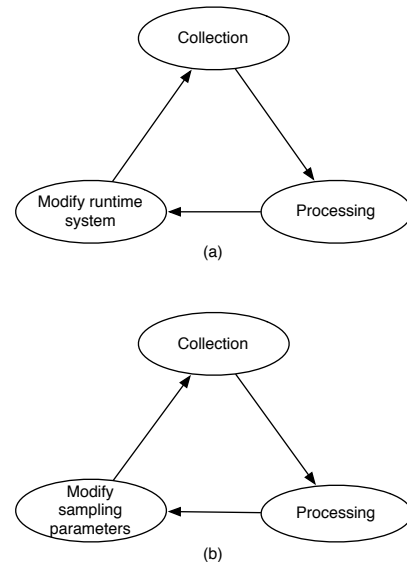


Figure 2: (a) Feedback via modifications to the runtime system and (b) Feedback by fine tuning sampling parameters.

about the information that should be collected. The flexibility also applies to the processing and feedback stages where the developer is able to decide how the data should be processed and whether different aspects of the system should be analyzed as a result of the current profiling run. More importantly, this control can extend to automating the execution of profiling as the developer can specify the required strategies for collection, processing and feedback and how this execution should take place. For this to be possible, a mechanism is required that allows for the collection, processing and feedback stages to be regulated. This *regulator* is the means by which the developer is able to obtain the required control and is illustrated in Figure 3. Via the regulator, the developer decides what information should be collected, how the data should be processed as well as the feedback strategies to use for a given situation.

Most profiling systems offer a broad range of informa-

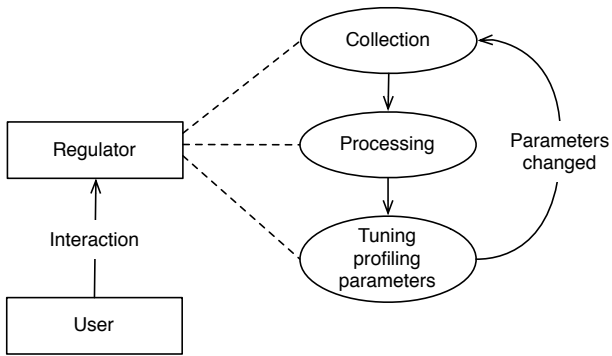


Figure 3: The profiling process with the use of a regulator.

tion to the developer about the performance of the system. However, these systems only provide support for profiling at the data collection level, meaning that the processing and feedback stages require manual intervention by the developer. In this case, the developer determines the appropriate sampling parameters and commences the collection of data. In terms of processing, the results are visualized, resulting in the developer having to manually inspect the data and from this, infer what to measure in the next run. This process involves a restart of the profiling system with modified parameters. This process is illustrated in Figure 4.

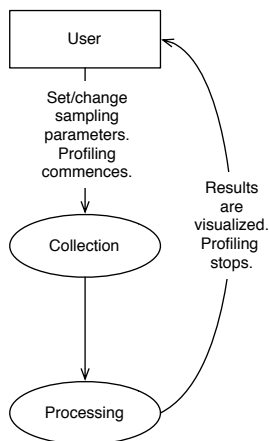


Figure 4: The profiling process in the majority of existing profiling systems.

This paper describes an approach to profiling that makes the regulator mechanism a feasible means for automating the collection, processing and feedback stages. The regulator takes the form of a domain specific language called *sampspec* which allows the developer to programmatically specify the information that should be collected, the computations that should take place on this data as well as the actions to take with respect to subsequent runs. Consequently, *sampspec* provides the requisite expressibility in acting as a regulator.

The structure of this paper is as follows: Section 2 discusses existing profiling systems. Section 3 provides a description of the system model used by *sampspec*. Section 4 provides a description of the language syntax. Section 5 provides language examples. Section 6 provides details about further work with *sampspec* and finally, Section 7 provides conclusions.

## 2 Existing Systems

Existing profiling systems can be divided into groups relating to the extent of instrumentation required. In this case, there are three categories: instrumentation to the application only, instrumentation to both the application and the system and instrumentation to the system only. The remainder of this section will detail the systems via these categories.

### 2.1 Application instrumentation

Profiling systems under this group provide information about the execution of the application by making changes to the application under question to obtain the required performance data. A well known example of a profiling system that uses this approach is *gprof* (Graham et al. 1982). *gprof* provides information about function execution counts for a particular application. In addition, it is able to account for the time spent in the given function when other functions are invoked by the current function. Inserting the code to obtain this information is done via the compiler where additional function calls are inserted at the function entry points of all functions invoked. One function, `mcount`, stores count information about the number of times a given call arc has been encountered while another function, `profil`, maintains a histogram which is used to infer the execution time for the given function.

Obtaining the profiling results is done offline. That is, *gprof* can only process the results once the application has finished execution. The results given consist of two aspects: the flat profile, which gives information about the number of times a function was invoked, and the call graph profile, which provides information about the function's parents and children in addition to the function itself. In terms of feedback, *gprof* does not offer any facilities in terms of being able to change profiling strategies. The same information is provided to the developer for every profiling run.

### 2.2 Application and system instrumentation

In this category, profiling systems modify both the application and the system to obtain the required profiling information. An example of such a system is CrossWalk (Mirgorodskiy & Miller 2003), which is able to identify bottlenecks in not only the application but also the system. It does this by traversing the call graph, examining functions that exceed an execution time threshold and refining the bottleneck to the callees of that function. This continues until a function is reached whereby either the function has no callees or the callees themselves do not exceed the time threshold. In the case where the bottleneck function is a system call, CrossWalk is able to enter the kernel and refine the search in the same way as described above.

CrossWalk is able to achieve an application to kernel link via two instrumentation APIs: *DynInst* for program instrumentation and *KernelInst* for kernel instrumentation (Buck & Hollingworth 2000, Tamches & Miller 1999). The *DynInst* API has facilities that allow the developer to retrieve information about a function's callees, control flow and other information (Mirgorodskiy & Miller 2003). In a similar fashion to *gprof*, there is no choice in terms of alternative profiling strategies to use after a given profiling run.

### 2.3 System instrumentation

The profiling systems in this category only require changes to the system in order to obtain the required information. Moreover, these tools can be classified further into two groups: event tracing and sampling.

### 2.3.1 Event Tracing

Event Tracing involves adding a series of probes at certain points in the system such that upon reaching these execution points, the relevant information is recorded. This approach allows the developer to obtain information about the system with respect to a given system event. The systems that utilize this approach for profiling include *DTrace*, *javana* and RHODOS (Research Oriented Distributed Operating System).

*DTrace* is a dynamic instrumentation tool (Cantrill et al. 2004) that allows developers to instrument both user and kernel level software and obtain information about certain aspects of execution. To achieve this, a series of probes are situated throughout the system such that when the appropriate event takes place, the probe for the event fires and the relevant information is collected. These probes can fire in relation to certain system events such as system calls, function boundaries (ie. entering and exiting kernel functions) and timer based events. The interface to this system is via a language called D (*Solaris Dynamic Tracing Guide* 2005) which allows the developer to specify what information should be collected about the execution of the system by specifying probes that should fire and optionally what circumstances should cause the probe to fire.

*javana* (Buytaert et al. 2006) provides a way for programmers to developer analysis tools for Java applications. It does this by using a dynamic binary instrumentation tool to build vertical maps that are able to provide links between lower level information such as instruction addresses and memory addresses to higher level information such as objects and methods that are being used. From these maps, the system is able to associate machine information to actual Java constructs and provide this information to the developer. Like *DTrace*, *javana* uses a domain specific language to specify what information should be collected upon the relevant events taking place. In the case of *javana*, the language leverages the functionality of a dynamic instrumentation framework called DIOTA (Maebé et al. 2002).

RHODOS (Wickham 1994) is a microkernel-based research operating system which was modified to provide event tracing facilities. To capture information about the execution of the system, probes are placed in three areas of the system: the interrupt interface, the exception interface and microkernel call interface. The structure of the system is such that the placement of the probes allows a temporal profile to be obtained. The profiling is initiated by the developer via an API in which the functions can either be invoked within an existing application or by providing a client process.

In terms of profiling feedback, *DTrace* and *javana* do allow the developer to come up with strategies for subsequent profiling runs. However, this is a manual process in that the developer is required to examine the profiling results and from there, stop profiling, implement the relevant strategies and restart the profiling system. RHODOS is more limited in that feedback comes in the form of enabling or disabling probes.

### 2.3.2 Sampling

Sampling involves taking a sample of the system at fixed time intervals in the system. This can be thought of as a particular form of event tracing as information is being recorded at a specific point in the system. Indeed, *DTrace* provides a sampling probe that can be fired upon a timer interrupt. The information obtained from sampling is used to gain a picture of the general execution of system within a time frame. *Shark* and the Digital Continuous Profiling Infrastructure (DCPI) are two examples of sampling systems.

*Shark* (*Shark user guide* 2007) is a MacOSX application that can provide information about the system execu-

tion in a variety of ways, one of which is via sampling. Both kernel and application information is obtained. Furthermore, sampling information can be obtained over a time window in which the developer is able to specify the number of samples to store for a window before they are discarded. This is useful in the case where the developer wishes to capture information about an aspect of the execution that shows visual abnormalities. *Shark* also makes use of the Performance Measurement Counters (PMCs) which are used to keep counts of hardware events in the system. These counters can be used as either a trigger for collecting samples or as part of a timer-based sample where the counter value is recorded at the time of the sample being recorded. Further customisations are allowed with respect to the sampling parameters. However, this process requires a manual restart of the sampling process.

The DCPI (Anderson et al. 1997) is a profiling system that samples using performance counter interrupts. The system is also able to analyze the sampling data in a variety of ways. The analysis ranges from attributing samples to functions, to comparing results from multiple sampling runs. The information collected in the DCPI is fixed, which offers no flexibility to the developer in terms of collecting information. The flexibility of the DCPI lies in the analysis tools which can provide a range of results to the developer.

From this analysis, the existing profiling systems are able to obtain very useful information about the execution of the runtime system. Of the three approaches, the tools based on system instrumentation allow for more information to be obtained. As part of this, these tools are able to obtain information about the execution of applications without having to modify applications to capture this information. However, the common property of all these systems is the requirement of human intervention in the processing and feedback stages. Because of this, automation is a property not provided in these systems. *sampspec* addresses this limitation by allowing the user to specify the information that should be collected, the computations to perform on the collected data as well as the strategies to employ based on the results of the computations.

## 3 System model

An important aspect of profiling regulation via a domain specific language is the design of a system model that highlights the role of the language in providing this regulation. The model used in *sampspec* is based on two notions: that the collection, processing and feedback stages are concurrent computations and that the user has control over the execution of these computations. This model is illustrated in Figure 5. There are three aspects to the model: the initial execution of profiling (①), execution control (②) and the resulting execution of alternative profiling runs (③). The rest of this section will describe the processes that take place in these phases.

### 3.1 Initial execution of profiling

The first requirement for *sampspec* to provide regulation is a mechanism for describing computations to take place during collection, processing and feedback. This is provided via *specifications*, which can be thought of essentially as threads. As a consequence, the standard thread controls can be applied to specifications. Initially, three specifications are created that represents the collection (C), processing (P) and feedback (F) stages.

In the specification C, the information about the parameters for sampling are declared. An example of a sampling parameter is the duration of data collection. These parameters are linked to a particular interrupt which is used as the basis for collecting samples. There are two types of interrupts that the collection of data can be based on: timer and Performance Measurement Counter (PMC).



Figure 5: A flow of execution between three threads.

Timer interrupts are used in thread scheduling to ensure that threads do not exceed their time slice. Hence, samples based on timer interrupts provide a picture about the execution of the runtime system in terms of the percentage of time spent executing certain components. PMCs are used to count a variety of hardware events, such as cache misses, whereby a PMC interrupt is signalled whenever a counter overflows. Because of this, samples based on PMC interrupts give an indication of the processes that are causing the most occurrences of a given hardware event. Furthermore, the developer may be able to choose the overflow value of a counter, meaning that more samples can be collected in relation to the hardware event.

The sampling parameters are then utilized by the runtime system to start the collection of samples. The samples belonging to C need to be accessed by P and to provide this, a mechanism is required that allows specifications to communicate with each other via the passing of data. In this model, the functionality is catered for via streams of which there is an input stream and an output stream for all specifications. Input streams are used to provide data that is required by a specification to carry out computations and maybe linked to the output stream of another specification. Given that some specifications will require data beforehand to carry out computations, the input to a specification will correspond to the output of other specifications. Output streams allow specifications to produce data that may be used by other specifications. In this case, C is a source of information, meaning that inputs are not required. However, C will produce samples that will be used by other specifications for processing. Therefore, C will move its samples to its output stream for other specifications to inspect.

Specifications provide computations that may be applied to data received as input, where the input source corresponds to output data from other specifications. P is one example of a specification that uses input, where the execution of P involves inspecting the samples from C and performing the required analysis. For instance, P may provide a count of the number of occurrences of a particular

sample. The computations finish when there are no more inputs to process. In the event where P has processed all of the available data and there is still data left to process, it will block until more data is available. The results of P's computations are sent to its output for other specifications to process. P illustrates a scenario where a specification requires both an input and an output stream.

The data sent to P's output is used by the specification F for further analysis. The analysis pertains to determining appropriate profiling strategies that should be used for the next profiling run. The specification F is an example of a specification that requires only an input. In this case, F instigates the next stage of profiling by spawning other specifications that contain different computations.

### 3.2 Specification control

Upon F receiving all data from P, it will, based on the results of analysis, signal the execution of three other threads C', P' and F'. Depending on how the specifications have been executed, F could spawn C', P' and F' or F could resume the execution of C', P' and F' in the case where they have been suspended initially. As well, F may suspend the execution of P and C in the case where the two specifications are still executing. For instance, F may spawn C', P' and F' as soon as the sample counts for a particular thread has met a condition set in F.

### 3.3 Execution of alternative profiling runs

As C', P' and F' are now executing, the same execution process takes place with these specifications as C, P and F. The difference here is that C' may be collecting different information. For example, C may have been collecting information based on L1 cache misses. C' may collect information on L2 cache misses, thus providing an overall picture of the extent of cache misses in the runtime system. Likewise, P' and F' may contain computations that differ from P and F respectively.

## 4 The *sampspec* Language

*sampspec* is a domain specific language for profiling and regulation. In relation to the model described in Section 3, the use of *sampspec* is as follows: A *sampspec* script consists of a series of specification declarations followed by an execute function. The code in Figure 6 illustrates the general structure of a language script. The execute function is used to provide an initial point of execution for creating threads.

```
spec spec_name1(){...}
spec spec_name2(){...}
...
spec spec_namen(){...}
execute(){...}
```

Figure 6: Skeleton code.

The next subsections detail the general features of *sampspec* and the operations that can be performed for a collection, processing and feedback specification.

### 4.1 *sampspec* Features

This section details particular features that are available for use in a specification. These features can be divided into the following categories: data types, iteration, selection, streams and specification control. These concepts are illustrated using a combination of BNF notation and actual language examples.

### 4.1.1 Data Types

The following basic data types are available in this language:

- *int*: 32-bit integers.
- *double*: Double precision floating points.
- *string*: Strings.
- *tid*: Thread ID. 4 bytes are used to store the Thread ID. The representation is a numerical value.
- *instn*: Instruction address. This is used to represent the memory location of an instruction and like the Thread ID, the representation is a numerical value.

In addition to the basic data types above, *sampspec* provides a number of type constructors:

- *tuple\_ref*: Tuples.
- Arrays and hash tables.
- *interrupt\_spec*: Interrupt parameters.
- Streams.

The remainder of this section will provide descriptions of the remaining data types.

### 4.1.2 Tuples

Tuples are a finite sequence of values which consists of a combination of data types. Samples are treated in *sampspec* as a tuple that consists of a thread ID, instruction address and function name combination. Tuples are defined in *sampspec* in the following way:

```
tuple_ref name: '[' [type id,]*type id '];
```

*tuple\_ref* is the name of the tuple type in this language. *name* is the name given for the particular tuple definition. *type* is the data type in the given sequence while *id* is the reference name for the individual piece of data. Referencing individual data elements involves the use of dot notation. For instance, for the following tuple definition:

```
tuple_ref tup:[int value1, int value2,  
              double value3];
```

Assigning the value 100 to the second element of the tuple is done as follows:

```
tup.value2 = 100;
```

### 4.1.3 Arrays and hash tables

*sampspec* provides two data structures types: Arrays and hash tables. Both can store any of the basic types as well as tuples. In terms of array declarations, the format is shown below:

```
type name '[' value '];
```

*type* signifies either one of the basic types or a tuple, *name* is the name of the array while *value* signifies the number of elements the array will have. Array indexing begins at index 0 with the last array element at the position of the array length minus one. In this case, the number of array elements must be specified.

Hash tables are specialized versions of arrays where indexing is based on a hash sequence. The hash sequence can be made up of a number of basic data types rather than just a single type. The hash table declaration format takes the following form:

```
h.table name:  
  '[' [type,]*type ']-> '[' [type,]*type '];
```

*h.table* is the name for the hash table data structure and *name* is the variable name. The type list to the left of the arrow indicates the hash sequence while the type list to the right indicates what the hash table will store. For example, the hash declaration below:

```
h.table table:[int,int]->[int];
```

declares a hash table with the sequence containing two integers that will store integers. Hash tables are mainly used in this language to store aggregations, which is explained in Section 4.2.3.

### 4.1.4 Interrupt parameters

In order to collect samples, the developer must specify a series of parameters that determine what information will be collected and how this information will be stored. Furthermore, these parameters are linked to a particular interrupt which is used as the basis for collecting samples and with this, additional information is specified in relation to the interrupt. To specify these parameters, an interrupt type called *interrupt\_spec* is provided. This is a specific application of a tuple containing the following data:

- *int\_type*: the source of the interrupt. The interrupt sources that can be used include the timer and PMC. The timer interrupt sources that can be used corresponds to four different time units: timer tick, per second, per minute and per hour. The PMC interrupts that can be used corresponds to the number of counters available for use on a given architecture.
- *info\_collect*: the information to collect. This can be a combination of the following: Thread ID, name of executing function or the instruction address.
- *duration\_unit*: the unit of time for the sampling duration. The time units that can be used are the same as the timer interrupt types mentioned earlier (eg. second, minute).
- *rate\_unit*: the unit of time for the sampling rate. The allowable values are the same as sampling duration unit (eg. second, minute).
- *duration*: the sampling duration. This is expressed as an integer.
- *rate*: the sampling rate. This is expressed as an integer.
- *storage\_mode*: the data storage mode. The developer can choose to consolidate the samples or to collect the samples as is. Data consolidation involves summarizing sampling data over a period of time. In this case, the samples are consolidated into a series of samples with sample count information counts attached to each of the samples. In terms of the data types, these windows are arrays whereby the data stored in a single element can be a combination of thread ID, instruction address and function name.
- *win\_length*: the consolidation window length. As part of the consolidation process, the developer must specify the length of the window period.
- *win\_unit*: the unit of time for the consolidating window. The unit of time for a single window period. The units are the same as for the sampling rate.
- *no\_win*: the number of time windows to store. This is expressed as an integer.

- *pmc\_info*: the PMC event to measure. In the case where the developer decides to base samples on PMC interrupts, the developer must specify the event number to base the collection on.

Assigning values to an *interrupt\_spec* variable utilises a dot notation shown below:

```
id.id = values;
```

where the first *id* production is the name of the interrupt variable, the second *id* production corresponds to one of the variables defined above while *values* represents the assignment value. For example, to provide information about the interrupt source in the interrupt variable called *timer*, the following code is used:

```
timer.int_type = value;
```

#### 4.1.5 Streams

The format for declaring an input or output for a specification is shown below:

```
source: [element,]* element;
```

*source* is either the declaration of ‘input’ or ‘output’ while *element* refers to the elements of the input or output. All data types as well as data structures can be specified in the element list. However, streams are typed, meaning that in the case where multiple sources are specified, they must all be of the same type where the first element in the list specification defines the type of the stream. Additionally, the developer can specify inputs or outputs of other specifications in the element list. The format for referring to the input or output of another specification is shown below:

```
specification_name.source;
```

Iteration over a stream takes place via a **foreach** declaration. Furthermore, in the case where the stream contains tuple data, a mapping primitive can be used that will establish an association with the iterator and the data inside the iterator. The mapping format is shown below:

```
iterator<-tuple.decln;
```

*iterator* refers to the iterator while *type.decln* refers to the tuple sequence. The expression states that the iterator will map to the tuple declaration on the right hand side of the arrow. In this case, type checking will be used to ensure that the mapping is valid.

## 4.2 API functions

In addition to the language constructs, *sampspec* also provides an API that allows for the developer to control the following functionality:

- specification control,
- sample collection control, and
- aggregations

The remainder of this section will describe these functions in relation to these three areas.

### 4.2.1 Specification control

As specifications are threads, functionality exists for the developer to have control over the execution of these specifications. For a specification to begin execution, the developer must explicitly start the thread by invoking the start function for the specification. While specifications are executing, the developer can choose to suspend or resume the execution of certain specifications by invoking the suspend and resume functions respectively. Moreover, the developer can choose to kill specifications by invoking the kill function. A series of functions are used which take as an argument the specification to affect. The information about these functions is summarized in Table 1.

Function name	Purpose	Parameters
start	To start the execution of a specification	The specification to start executing
suspend	Suspending the execution of a specification	The specification to suspend
resume	Resuming the execution of a suspended specification	The specification to resume execution
kill	Killing a specification	The specification to kill

Table 1: List of specification control functions

### 4.2.2 Sample collection control

A series of functions are available for controlling the collection of samples. The developer is able to start, stop, pause and resume collection. To commence the collection of samples, a function called *sample* is invoked and takes an interrupt variable as an argument. This is invoked as part of an output for a specification to signify that the samples generated can be used by other specifications.

To stop the collection of samples, a function called *coll\_stop* is invoked. This function takes two arguments: a constant that corresponds to the interrupt to stop and another constant that determines what is done with the sampling data. The developer is able to either keep or discard the samples that have been collected. Stopping the collection of samples means that the only way to collect more samples is to restart the collection by invoking the *sample* function. These functions, along with pause and resume, are summarized in Table 2.

Function name	Purpose	Parameters
sample	Commence sample collection for a given interrupt	<i>interrupt_spec</i> variable
coll_stop	Stopping the collection of samples	Interrupt source to stop, constant for further action to take place
coll_pause	Pausing the collection of samples	Interrupt source to pause
coll_resume	Resuming the collection of samples	Interrupt source to resume

Table 2: List of sample collection control functions

### 4.2.3 Aggregations

To assist with the data processing phase, *sampspec* provides a series of aggregation operations. These operations allow the developer to obtain certain statistical information as a specification receives the available data. Before any aggregations can take place, a hash table is required and it must be linked to an aggregation operation. The linking is established via a series of functions which, when invoked, returns a hash table that has been linked to the given aggregation operations. These functions are defined in Table 3 in which the argument for each function is a list of types that defines the hash sequence the aggregation operation is based on.

Function name	Purpose
aggn_create_count	Creation of a hash table linked to the count aggregation operation
aggn_create_sum	Creation of a hash table to specifically store sum aggregations
aggn_create_avg	Creation of a hash table to store arithmetic mean aggregations
aggn_create_min	Creation of a hash table to store minimum aggregation data
aggn_create_max	Creation of a hash table to store maximum aggregation data
aggn_create_stdev	Creation of hash table to store standard deviation aggregation data

Table 3: List of aggregation link functions

These aggregations can either be maintained per data sequence or, if the developer has consolidated data, over a time window to produce a single value. For per data sequence aggregations, the addition, arithmetic mean, minimum, maximum and standard deviation functions take as arguments the hash table that has been linked to the particular operation, the hash sequence to base the aggregation on as well as the actual value to analyze next. The count aggregation only takes the hash table and hash sequence as arguments. For window aggregations, the argument to these functions is the window for analysis. This information is summarized in Table 4. The hash sequence is represented as a tuple.

Aggregation operation	Function name (per time window)	Function name (per data sequence)	Arguments (per data sequence)
Addition	sum_w	sum_s	Hash table, hash sequence, value for analysis
Arithmetic mean	avg_w	avg_s	Hash table, hash sequence, value for analysis
Count	count_w	count_s	Hash table, hash sequence
Minimum	min_w	min_s	Hash table, hash sequence, value for analysis
Maximum	max_w	max_s	Hash table, hash sequence, value for analysis
Standard deviation	stdev_w	stdev_s	Hash table, hash sequence, value for analysis

Table 4: List of per data sequence and per window aggregation functions.

### 4.3 Specification execution

As part of executing specifications, another feature of *sampspec* is the ability for the developer to signal that a series of specifications should execute in parallel. This is achieved by using a ‘|||’ notation in the following format:

```
[thread_action |||]* thread_action;
```

*thread\_action* refers to one of the four specification control operations where the specifications to execute in parallel is essentially a list separated by the ‘|||’ sequence.

## 5 Language examples

This section will provide some language examples that demonstrates the use of these language constructs.

### 5.1 Basic collection and processing

The first example illustrates a very basic collection and processing specification with one specification collecting data and the other specification using the output of the collection specification to process data. To specify the data to collect for a profiling run, the developer creates a specification and inside it, provides the sampling parameters by declaring and initializing the relevant information via an interrupt variable. From there, the developer commences the collection of samples by invoking the *sample* function. The code for this is shown in Figure 7. Starting from line 5, the sampling parameters for the specification called *coll1* are declared. In this case, the thread ID and instruction address information is collected (line 6), the timer precision is at the timer tick level (line 8) and samples will be collected at every timer tick for a duration of 30 seconds (lines 10 and 11). In addition, the samples for this specification are being collected as is without any consolidation taking place (line 9).

```
1 spec coll1()
2 {
3   intrpt timer;
4
5   timer.int_type = TIMER;
6   timer.info_collect = INFO_TID | INFO_INST;
7   timer.duration_unit = TIME_SEC;
8   timer.rate_unit = TIME_TICK;
9   timer.storage_mode = NO_CONSOLIDATE;
10  timer.duration = 30;
11  timer.rate = 1;
12
13  output: sample(timer);
14 }
```

Figure 7: Collection specification code.

The code in Figure 8 illustrates a specification called *proc1* that will use data from *coll1* to perform computations. Firstly, for *proc1* to retrieve *coll1*’s data, *proc1* must state that the output of *coll1* will be an input and is shown on line 3. From there, the data being processed will be stored in a hash table storing integers where the hash sequence for this particular table is made up of a thread ID and instruction address tuple (line 5). The hash table is created via the invocation of the count aggregation creation function. The iteration block states that the variable *i* will iterate over each element of the stream (line 8) and will aggregate over the data via the count operation (line 10). The foreach statements work due to the typed nature of the streams. After all sample data has been obtained, *proc1* proceeds to print out the results by printing out each element of the hash table (line 21). In order to print out the results, a mapping is provided on line 20 to allow the developer to access the individual data elements for a given data set.

For the collection and processing to be carried out, the developer must provide a starting point for execution. This is done via the *execute* function and is illustrated in Figure 9. In this case, *coll1* and *proc1* begin execution in parallel as indicated by the use of the ||| (line 3).

### 5.2 Profiling and regulation

The next example demonstrates the use of specification control as a means of regulating execution as described in

```

1 spec proc1()
2 {
3   input: coll1.output;
4
5   h_table values = aggn_create_count(tid, instn);
6   tuple_ref tup: [tid tid_value, instn inst_value];
7
8   foreach i in input
9   {
10    count_s(values, i);
11  }
12
13  tuple_ref aggn: [tid tid_val, instn inst_value,
14                 int agg_value];
15
16  sys_print("<TID, instn addr>\t\tValue:\n");
17
18  foreach i in values
19  {
20    i <- aggn;
21    sys_print("<%d, %d>\t\t%d\n", i.tid_val,
22            i.inst_value,
23            i.agg_value);
24  }
25 }

```

Figure 8: Processing specification code.

```

1 execute()
2 {
3   start(coll1) ||| start(proc1);
4 }

```

Figure 9: *execute* function.

the system model. In this example, six specifications are used where one half is used for a collection, processing and feedback run and the remaining half is used for another collection, processing and feedback run. The code for the first collection, processing and feedback run is shown in Figures 10, 11 and 12 respectively. The code in Figure 10 is a declaration for sample collection based on the first PMC as stated (line 5). The collection of samples is based on the event type defined on line 9.

```

1 spec collect1()
2 {
3   intrpt pmc1;
4
5   pmc1.int_type = PMC1;
6   pmc1.info_collect = INFO_TID;
7   pmc1.duration_unit = TIME_SEC;
8   pmc1.storage_mode = NO_CONSOLIDATE;
9   pmc1.pmc_info = PMC1_EVENT_1;
10
11  pmc1.duration = 30;
12  pmc1.rate = 1000;
13
14  output: sample(pmc1);
15 }

```

Figure 10: A collection specification using a PMC interrupt.

The code in Figure 11 is similar to the code in Figure 8. However, *process1* sends the contents of the hash table containing aggregation data to its output (line 12) rather than printing to standard output.

The code in Figure 12 performs analysis on the aggregations results generated by *process1* and checks to see whether there is a thread that has exceeded a threshold. If so, it goes about using the specification controls to switch execution to other specifications. In this case, *feedback1* checks to see whether a thread has exceeded 1000 sample counts (line 10). If this is true for a thread ID, the specification starts the execution of *coll2*, *proc2* and *feedback2* (lines 15 and 16). As well, it suspends the execution of *process1* and itself (lines 16 and 17). This demonstrates that a specification does not necessarily have to begin execution via the *execute* function.

```

1 spec process1()
2 {
3   input: coll1.output;
4
5   h_table results = aggn_create_count(tid);
6
7   foreach sample in input
8   {
9     count_s(results, sample);
10  }
11
12  output: results;
13 }

```

Figure 11: A processing specification using data from *coll1*

```

1 spec feedback1()
2 {
3   input: process1.output;
4
5   tuple_ref agg_tuple:[tid tid_val, int int_val];
6
7   foreach res in input
8   {
9     res <- agg_tuple;
10    if(res.int_val > 1000)
11    {
12      sys_print("Thread ID %d has caused" > 1000 "
13              "occurrences of the given event\n",
14              res.tid_val);
15      start(coll2) ||| start(proc2) |||
16      start(feedback2) ||| suspend(process1) |||
17      suspend(feedback1);
18    }
19  }
20 }

```

Figure 12: A feedback specification using data from *process1*

The code for *coll2* is shown in Figure 13. The collection of information is based on the second PMC (line 5) and both thread ID and instruction address information is being collected for a sample (line 6). Furthermore, information is being collected based on a defined event for the second PMC (line 9).

```

1 spec coll2()
2 {
3   intrpt pmc;
4
5   pmc.int_type = PMC2;
6   pmc.info_collect = INFO_TID | INFO_INST;
7   pmc.duration_unit = TIME_SEC;
8   pmc.storage_mode = NO_CONSOLIDATE;
9   pmc.pmc_info = PMC2_EVENT_3;
10
11  pmc.duration = 30;
12  pmc.rate = 1000;
13
14  output: sample(pmc2);
15 }

```

Figure 13: Specification code for *coll2*

The code in Figure 14 corresponds to the processing for the second collection phase. The code is similar to *process1* in that it will aggregate over all samples in input and send the results to its output stream.

The specification in Figure 15 takes as input the output of *process2*. In this case, if the threshold has been exceeded (line 10), a message is displayed about the thread ID and the instruction (line 12). When *feedback2* has finished all computations, it will resume the execution of *process1* and *feedback1* (line 19).

Finally, the code in Figure 16 provides the starting point of execution for profiling. In this case, *coll1*, *proc1* and *feedback1* begin execution which will lead to *coll2*, *proc2* and *feedback2* executing via *proc1*. Whilst the application of the system model in this example is very sim-

```

1 spec proc2()
2 {
3   input: coll2.output;
4
5   h_table results = aggn_create_count(tid, instn);
6
7   tuple_ref input_iter: [tid thrID, instn addr];
8
9   foreach currSample in input
10  {
11    currSample <- input_iter;
12    count_s(results, [input_iter.thrID,
13                    input_iter.addr]);
14  }
15
16  output: results;
17 }

```

Figure 14: Specification code for *proc2* using the output of *coll2* as input.

```

1 spec feedback2()
2 {
3   input: proc2.output;
4   tuple_ref aggn_iter: [tid tid_v, instn addr,
5                       int agg_val];
6
7   foreach aggns in results
8   {
9     aggns <- aggn_iter;
10    if(aggns.agg_val > 1000)
11    {
12      sys_print("The instruction "
13              "with address %d for the thread "
14              "with ID %d is intensive\n",
15              aggns.addr, aggns.tid_v);
16    }
17  }
18
19  resume(process1) ||| resume(feedback1);
20 }

```

Figure 15: Code for *feedback2*.

ple, it does demonstrate the ability for regulation in that the developer is able to execute different profiling strategies when appropriate.

```

1 execute()
2 {
3   start(coll1) ||| start(proc1) ||| start(feedback1);
4 }

```

Figure 16: Execution code for the second example

## 6 Further work

The current implementation of *sampspec* is for the Intel build of version 1.4.1.1 of the OKL4 microkernel (Liedtke 1993), where code has only been added to the microkernel as required to generate samples. All other functionality i.e. the modeling of specifications, copying the samples from the microkernel to developer space etc. is implemented in user space. The code added to the microkernel is required as the timing involved in collecting samples would not be achievable otherwise. Furthermore, the use of the PMC facilities requires privileged access, which is only possible while executing in the microkernel.

*sampspec* forms one part of a project aimed towards providing a software evolution framework, where software evolution is the ability for a system to change in response to the requirements of an application. In the case where there is a single application running on a standard operating system, performance will not necessarily be optimized for that single application due to the default operating system policies. By providing a system that allows the developer to specify their own policies, it results in a separation between mechanism and policy. Hence, the

system executes according to the needs of the application rather than the reverse situation due to the fact that the underlying mechanisms execute in relation to the developer-defined policies. Extensive research has gone into providing systems that can comply to the needs of the application (Morrison et al. 2000, Falkner et al. 2007).

For this particular framework, an alternative approach is being considered that uses a microkernel system. The key to the microkernel approach lies in the fact that unlike a system running on a monolithic kernel, where all the operating system functionality is situated in the kernel, only the functionality that is essential for a system to execute is placed in the kernel. The remaining functionality is moved into user space, meaning that it is left up to the developer to implement this functionality. This has the potential to facilitate the evolution of software components given that the implementation takes place in user space. L4 provides a suitable basis for experimentation as it provides high performance IPC, and is not limited by poor performance as were earlier microkernels (Liedtke 1993, Liedtke et al. 1997).

*sampspec* is able to provide some adaptation facilities with respect to the profiling system which allows the developer to gain more insight into the execution of the system. Further work is required to extend *sampspec*, and its system model, to support not only feedback into the profiling system but also feedback to modify the runtime system. As part of this, a constraint system would be required to determine whether any change should take place in the system at some point in time. There are some systems that are able to provide code optimizations via sampling (Zhang et al. 1997). However, these systems do not give the developer sufficient control over the process of profiling as well as the optimizations to perform.

Another avenue for further research into *sampspec* is leveraging the language into existing profiling systems. This has the potential for the developer to be able to introduce regulation that was previously not present in these tools.

For space reasons, the description of abstract regulation strategy is fairly limited. However, since *sampspec* is a Turing complete programming language, the specifications of the strategy are computationally complete. The programming of these specifications is not always trivial and involves considerable skill in developing them. Further work would include the expression of more complex strategies, such as processing samples that belong to a particular branch of a function.

## 7 Conclusions

This paper has described a domain specific language called *sampspec* that allows the developer to profile a runtime system in a way that allows the profiling process to be regulated. Via *sampspec*, the developer is able to specify the information that should be collected, the computations that should be performed on this data and the strategies that should be used based on the results of these computations. Existing profiling systems do not provide the ability for regulation as profiling is stopped and restarted for any feedback into the profiler to take effect. Collecting profiling data can of course be done in many ways. For example, different languages could be used for the three stages of profiling. However, here we want to capture the system model within one language while retaining the ability to cross call to other languages.

## Acknowledgements

We would like to thank the reviewers for providing constructive comments about the paper. We would also like to thank Professor Ron Morrison for his suggestions and comments.

## References

- Anderson, J. M., Berc, L. M., Dean, J., Ghemawat, S., Henzinger, M. R., Leung, S.-T. A., Sites, R. L., Vandevoorde, M. T., Waldspurger, C. A. & Weihl, W. E. (1997), 'Continuous profiling: where have all the cycles gone?', *ACM Trans. Comput. Syst.* **15**(4), 357–390.
- Buck, B. & Hollingworth, J. K. (2000), 'An API for runtime code patching', *The International Journal of High Performance Computing Applications* **14**(4), 317–329.
- Buytaert, D., Maebe, J., Eeckhout, L. & Bosschere, K. D. (2006), Building Java program analysis tools using Javanna, in 'OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications', ACM, New York, NY, USA, pp. 653–654.
- Cantrill, B. M., Shapiro, M. W. & Leventhal, A. H. (2004), Dynamic instrumentation of production systems, in 'ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference', USENIX Association, Berkeley, CA, USA, pp. 2–2.
- Falkner, K., Balasubramaniam, D., Detmold, H. & Munro, D. (2007), Informed Evolution, in '1st European Conference on Software Architecture, ECSA 2007, Aranjuez, Spain. Lecture Notes in Computer Science Volume 4758', pp. 288–291.
- Graham, S. L., Kessler, P. B. & Mckusick, M. K. (1982), 'gprof: A call graph execution profiler', *SIGPLAN Not.* **17**(6), 120–126.
- Liedtke, J. (1993), Improving IPC by kernel design, in 'SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles', ACM, New York, NY, USA, pp. 175–188.
- Liedtke, J., Elphinstone, K., Schiinberg, S., Hartig, H., Heiser, G., Islam, N. & Jaeger, T. (1997), Achieved IPC Performance, in 'HOTOS '97: Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)', IEEE Computer Society, Washington, DC, USA, p. 28.
- Maebe, J., Ronsse, M. & Bosschere, K. D. (2002), DIOTA: Dynamic instrumentation, optimization and transformation of applications, in 'In Proc. 4th Workshop on Binary Translation (WBT02) held in conjunction with the International Conference on Parallel Architectures and Compilation Techniques (PACT)'.
- Mirgorodskiy, E. V. & Miller, B. P. (2003), CrossWalk: A tool for performance profiling across the user-kernel boundary, in 'In International Conference on Parallel Computing (ParCo)', pp. 745–752.
- Morrison, R., Balasubramaniam, D., Greenwood, R., Kirby, G., Mayes, K., Munro, D. & Warboys, B. (2000), 'A Compliant Persistent Architecture', *Software - Practice and Experience, Special Issue on Persistent Object Systems* **30**(4), 363–386.
- Shark user guide* (2007), Apple Computer Inc.
- Solaris Dynamic Tracing Guide* (2005), Sun Microsystems Inc.
- Tamches, A. & Miller, B. P. (1999), Fine-grained dynamic instrumentation of commodity operating system kernels, in 'Proceedings of the 3rd Symposium on Operating Systems Design and Implementation', pp. 117–130.
- Wickham, G. (1994), Monitoring the performance of a microkernel based operating system and supported applications, PhD thesis, School of Computing and Mathematics, Deakin University.
- Zhang, X., Wang, Z., Gloy, N., Chen, J. B. & Smith, M. D. (1997), System support for automatic profiling and optimization, in 'SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles', ACM, New York, NY, USA, pp. 15–26.