

Proposal for compilation techniques of monitoring tasks to improve applications management performance

Bernard Kaddour
LIRIS – Université Lyon 1
43 boulevard du 11 novembre 1918
F-69622 Villeurbanne cedex
bkaddour@liris.cnrs.fr

Joël Quinqueton
LIRMM
161 rue Ada
F-34392 Montpellier cedex 5
jq@lirmm.fr

Abstract

The emergence of middleware solutions and new services, even on small devices, will need adapted distributed management solutions which address these specificities, both in terms of software design and in terms of performance. We propose a management system where these high-level and low-level management concerns are separated. In particular for JAVA based solutions, we suggest a low-level management agent tightly tied to the JVM/JVMTI and able to receive and to manipulate programmable monitoring expressions. To avoid conflicts with applications already using JVMTI instrumentation techniques, we dynamically compile these expressions into machine code to reduce overload and minimize performance lost.

1. Introduction

Since some years, more and more smart phones or personal digital assistants are widely available and used. These small devices still suffer from some limitations compared with high end fixed terminals, for example lower CPU performance or smaller memory size, but even now days they already have enough processing capabilities to host complete high Operating Systems - either Windows or Linux dedicated versions - and they appear more and more as autonomous embedded systems. Meanwhile, industrially accepted middleware solutions appear [4][13] for new distributed applications. These middleware are currently available for small devices [14] and even Multi-Agents Systems based solutions have been introduced in the research area [2].

Such new distributed applications and services whose components may partially or totally be executed by small devices implies high optimizations and new flexible needs for their management to be effective.

Managing disperse heterogonous entities has already been investigated in the network community and standards defined [16][17] but they not address software management at low-level.

We propose a management system which tries to take into account these particularities and we introduce a system where high-level and low-level management concerns are separated. The high-level management part relies on messages interception mechanisms which, coupled with Aspects Oriented Programming [9] concepts, provides facilities for management activities to dynamically operate, enhance and manage JAVA based applications transparently. The low-level management part introduces and uses programmable monitoring tasks tightly related to the code of the managed JAVA entities and is concerned by performance optimization aspect. Importance of performance for management has already been depicted [12], taking as example a manager in charge of managing hundreds of objects. The massive incoming of network services for small devices imposes to take this point into account. We introduce monitoring tasks that are compiled to minimize management impact from a performance point of view.

The high-level part is not developed here. In this article we present in section 2. the principles of our management system. The section 3. presents the low level management part and the monitoring tasks. Section 3.1. depicts interactions with the underlying JVM/JVMTI and the compilation process whereas section 3.2. presents ideas used for the code generation. We conclude with some future works and perspectives.

2. Overview of our management system

The high-level management part of the management system (named *kernel of management system* or *KMS*) is based upon Aspects Oriented Programming concepts and uses the AspectWertz [1] (*AW*) tool to provide dynamic and flexible management facilities.

Most of these facilities rely on the interception and filtering of messages exchanged by the managed application.

Management entities used by the KMS are *activities*. They are composed of management functions and filter functions.

When received by a KMS, either from a human administrator or from another KMS, an activity can be

deployed: its management functions are executed in a dedicated environment and its embedded filters are dynamically plugged to the managed application. The KMS addresses security issues and provides services to the activities.

Activities are java .jar files, with one entry implementing the `Activity` class known to provide particular methods. In turn, an activity can delegate one or many monitoring tasks towards the low-level part of system. These monitoring tasks are monitoring expressions (represented as resources in an activity .jar file) which can keep a watch on the classes or objects' byte-code at run-time.

3. Monitoring expressions

Monitoring expressions (eMons) and their associated *agent* is the core of our monitoring system, the low-level part of KMS.

Monitoring expressions are mainly composed of (i) expressions based upon fields and variables present in the different classes and instances found in the managed application code and (ii) an event to emit when one of the expressions becomes true.

eMons can be considered near to the DISMAN [10] – and its associated MIBs – spirit. They are its generalization and don't only limit to SNMP variables but also consider JAVA software applications and their JAVA byte-code.

Let's consider the class:

```
class C {
  static public int value =0;
  int field;
  public void m() {
    int local_m0;
    int local_m;
    int sum;
    :
    sum = ...;
  }
}
```

Some management task T could ask to be informed with an *ev_value* event when the expression `C.field +C.value -C.m().local_m` becomes negative. This could be achieved with:

Watch for: `C.field+C.value-C.m().local_m<0`
Event: `ev_value`

To fill up more general needs, we have introduced a small language to describe eMons, their scopes, relationships and definitions.

Here's an eMon entry:
`eMon_name {`

```
scope =always;
thread=ANY;
active=true;
{
  m is #”C;”m().local_m;
  m0 is #”C;”m().#0; //==local_m0
  C isClass #”C;”;
  float temp_1;
  int temp_2;
}
exp ::= {
  temp_1 =(float)(m0 -m +2);
  {
    #C.field +#C.value -m <0
  }
  temp_2 =@@last();
}
on remove ::= {
  activate another_eMon;
  event callback 12; // ev_value
  #C.m().sum =0;
}
}}
describing an eMon named "eMon_name".
```

First is given the *scope* of the eMon. The scope defines the way the eMon's expressions will be evaluated. It can be "method" for a sole evaluation at entry and exit method points, "thread" if an evaluation is required when a thread appears or disappears, periodical, or "always" if the expressions must be immediately re-evaluated after a variable/field modification.

"eMon_name" is not attached to one particular thread and is defined regardless of the thread which runs it.

By default, eMons inserted in the system are considered inactive until someone or something (e.g. another eMon) explicitly activate them. "eMon_name" is here marked has *active* and must be activate immediately (as long as expressions it is composed of are defined, for example expressions including local variables of a method `m()` are defined only if at least one thread executes `m()`).

Follow some declarations. They can be simple shortcuts for ease, typed or not, in relation with the monitored code, e.g. (`m`, `m0`, `C`), or some local temporary variables used for the eMon own needs.

`m0` is defined as the local variable at index 0 of method `m()` in the class named "C". Indexes in place of full named variables (or fields) are allowed as the monitored code may have been stripped of its variables names.

`temp_1` is an eMon local variable and has not direct relation with the monitored code. These eMon's

local variables must be typed. Their introduction was originally dictated by at least a possibility for the eMons to dynamically obtain and store informations available from the monitoring system (e.g. the JAVA thread number in execution) but also the possibility for expressions to act as counters measuring the difference taken between two evaluations of an expression [8]. They evolved and became general purpose variables.

The expressions body, composed of the main monitoring expression enclosed in brackets. It can be preceded by expressions to evaluate and store *before* the main expression is evaluated whereas following the main monitoring expression are expressions to proceed *after* the main expression evaluation.

To gain in flexibility, some add-ons are allowed:

- built-in functions provided by the system. This set of functions includes usual system-call functions (`kms date()`, `kms_arch()`, `kms_strstr()`, ...). From a design point of view, they can be seen as the public methods provided by the core KMS.

- Special MIB access functions. As the system is designed to interact with the SNMP protocol [16], a group of functions allowing MIB [17] accesses is included [8]. One function per basic SNMP type: `kms int_mib()`, `kms_bool_mib()`, `kms_ostr_mib()`,

- built-in functions in relation with the underlying monitoring system. They can give to an eMon information directly provided by the JVM. A thread, object or class ID can be obtained this way by an eMon.

- `@@last()` is the last evaluated value of the main expression.

The eMon specifies the Time To Live of the expression and the kind of event the expression will send when turning true. The TTL can be expressed by means of a maximum number of activations the expression can reach or by a duration time.

After that, the expression is removed by the system from its table of expressions to monitor and a notification is sent to the activity's owner of the expression (by mean of an `int expressionReached(int expld)` call, a method implemented by the Activity class). SNMP events are also supported [12].

These information are provided to the monitoring system trough the `remove` and `on_remove` statements.

When removed, "eMon_name" generates to the eMon's activity owner a callback with 12 as parameter and it activates the sleeping eMon named "another_eMon". It finally sets to 0 the local variable `sum` defined in the `m()` method.

Modifying such a monitored code variable is only allowed at remove time and not in the expressions body. It is an easy and temporary way to avoid unwanted monitored code behaviors that are known to appear then the main monitoring expression is true.

The expected behavior of the KMS (the whole management system) is such that one can either have to manage a few eMons introduced by a human administrator or have to deal with large sets of eMons automatically introduced by activities. In the last case, we may excepted redundancies between the eMons.

We are led to take care of performance aspect and to introduce an algorithm which addresses immediate re-evaluation of eMon's expressions as soon as even only one of their involved variable has been modified.

The needed performance optimization for re-evaluation of large number of expressions have similarities with the challenge of filtering high throughput network which has led to new fast matching algorithms[18][5].

The common way to speed up expressions processing is to use real CPU native code. We chose the IA32(x86) as it is the most representative and common CPU currently used. It hasn't specificities such as large number of registers or long instructions possibilities. Translation towards most sophisticate architecture would be possible.

Some differences with usual compilation. First, we have over the used variables a much more lighter control than a compiler usually has. Variables are *lend* by the JVM and can at the best be read or modified through get and set functions. An optimization is to avoid these time consumers Java Native Interface /JVM Tool Interface (JVMTI) [7] functions as far as we can and rather use a cache mechanism for the objects, fields and local variables the expressions need. Secondly, eMons' expressions are loosely coupled with the application code they monitor and are evaluated asynchronously with the Java code executed by the JVM. Hence, we can't statically discover from the monitored code some registers optimizations for the eMon's generated code.

3.1. The overall architecture

The used architecture takes advantage of features provided by the JVMTI introduced by Sun. At first available through the JVM Debugger Interface /JVM Profiler Interface couple, this API offers the possibility to attach an agent to the original java byte-code. This agent has facilities to register customized hooks at runtime to the java byte-code, such as e.g. hooks to be informed when a field is modified or when a class is loaded by the JVM.

In particular, it is possible for an agent to modify or inject new java byte-code in the original one, this is known as instrumentation[6][11]. Despite attractive as this injected new byte-code is next automatically compiled by the underlying JIT, we have not chosen a pure instrumentation approach for at least two reasons.

First, instrumentation freezes the monitoring process. When inserted, an eMon is considered as always active and can't afterwards easily be removed, nor we can dynamically add or activate new incoming eMons. Second, only one agent can modify the byte-code this way, and that's the way AspectWerkz [1] already took to provide Aspects used by the high-level management part of the system.

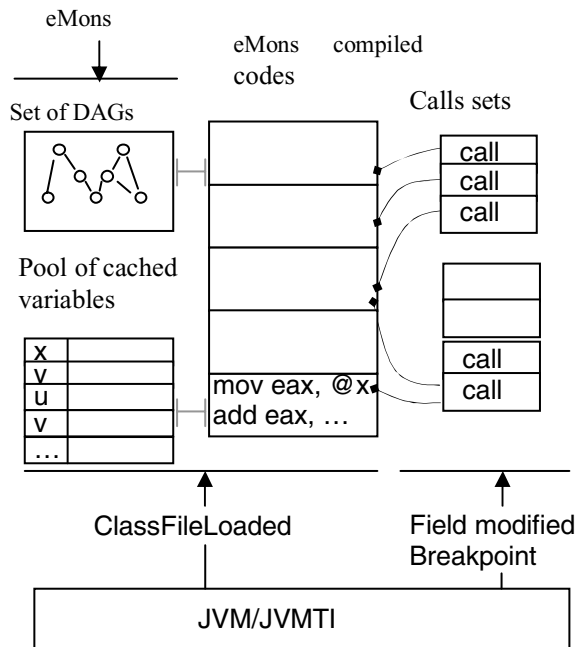


Figure 1. Agent architecture.

We then choose the more drastic approach that is to take in charge the eMons compilation process.

This agent is the monitoring system. First of all the agent places hook functions to be executed when a class is loaded, when a thread enters or exits a method, fields modifications, etc. or when a breakpoint event is generated by the JVMTI. The agent creates its own memory zones, mainly the pool of cached variables and storage areas for eMons compiled code.

When an eMon is inserted in the agent, it is parsed accordingly to its definition, relationships with its siblings eMons are created and finally it's DAG representation is constructed.

The DAG is constructed in a usual manner with the partial and incomplete information available as variables/fields types are generally unknown as this step, neither are their storage zones and as a

consequence nor are the operations types. This is a difference with traditional or JIT compilation.

Associated to the DAG nodes are the classes of variables or fields needed to complete the DAG compilation plus a fast to compute hashkey.

As DAG nodes are unary or binary operators, the hashkey is a 32 bits quantity build from the node left part type, the node right part type and a XOR of the left and right parts node quantities. This key is then used to find DAGs nodes similarities among eMons DAGs. Each time a node appears to be used, it's usage_number is incremented.

When a new class is loaded the previously registered loadClass hook function attached to the JVMTI takes control over the JVM. Then we create our own representation of this class with fields offsets or methods ID identifiers and the class is marked as "known", considering the build representation accurate even if the class is later unloaded. This mechanism speeds up the parsing process but implies that the monitoring agent doesn't properly work with hacked byte-code differently modified in between two consecutive loads by the JVM.

For classes with variables involved in awaiting eMons' DAGs to complete, fields/variables types are obtained and memory zones are allocated to store static or member fields and member local variables. All these storages take place in the pool of cached variables which are later used by the compiled code.

Current JVMTI doesn't provide for local variables modification the same modified event mechanism it allows for fields. In the case of eMons with method local variables and with a scope sets to always, we have to manually seek where these variables are modified in the java byte-code. So, without any instrumentation modification intents, we walk through the byte-code methods and look for java op-code offsets where local variables are modified. Breakpoints (JVMTI's breakpoints are simple events sent to a *Breakpoint()* event manager) are placed for each one of the encountered offset and the index of the modified variable is registered. Finally two tables are created for the breakpoint event manager.

The first one is an association offset-calls sets where a calls set is a sequence of call cells, each cell contains a call execution code towards the compiled code to execute for the re-evaluation of one expression using the modified local variable. A call cell can be empty and is modified accordingly to the eMon activations/un-activations. Calls sets are modified accordingly to eMons removal or new eMons arrival. This two stage call process – rather than one or two list(s) containing addresses of compiled code to re-evaluate – was preferred as eMons arrival/removal remains rare once the java code is in execution. This

scheme avoids unnecessary C(agent)/Asm(eMons) switches and CPU cycles losts.

The second table is an (offset, next offset) hashtable which tends to improve the breakpoint manager performance for without loops method code. When called, the breakpoint manager firstly gets (GetLocal<Type>()) and updates the modified variable in the pool of cached variables and then checks if it is called with the expected next offset, if this is the case then the calls sets to execute are immediately found, otherwise the offset hashtable is used to obtain the appropriate call-sets.

Emon's DAGs for which the new incoming class was the last expected one to complete information variables are then compiled. The operations types are evaluated and the associated code generated using the addresses in the pool of cached variables, empty call cells waiting for these expressions to be compiled are filled. The generated code tries to minimize the expressions re-evaluation process.

3.2. Underlying compilation ideas

To describe how the most often used parts of expressions are compiled to reduce the overall re-evaluation process, we'll treat the case of the $(x + y) + (u * v)$ expression (E), without object reference for the sake of simplicity.

Nodes of the expression are labeled with binary numbers composed of 0 and 1's. Starting from the root with an empty label, at each step, left node's label is build by concatenating a 0 to its father's one, right node label by concatenating a 1. Such an labeling tree scheme was also used to built fast matching tables in the network filtering area [18].

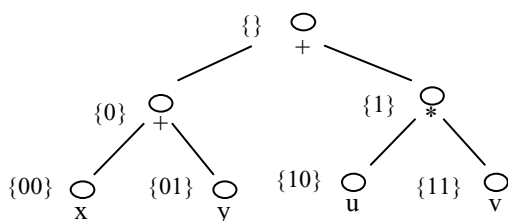


Figure 2. Labeled DAG.

It appears that the label of a variable leaf node describes the path in the DAG from the leaf node towards the root node an gives the list of the nodes witch need to be re-evaluated when the variable is modified. All the other nodes aren't affected by the variable modification and their last known result remains correct.

A *bloc* is a part of the DAG with only one operator and able to memorize its last evaluated result. The

expression E is formed of the bloc $(x + y)$ and the bloc $(u * v)$. The generated code of each bloc can immediately be obtained from code templates that we instantiate according to the types, the operation and an available free register at compile time, tracking of free registers is done with a usual free registers stack.

The generated code for the bloc $(x + y)$ obtained from the (int, int, add) code template is:

```

mem_xy    dw 0
ev_xy     mov reg, @x; // free reg ==reg
          add reg, @y; // op ==add
          mov mem_xy, reg
          ret
  
```

as storage zone of x and y in the cache are known. The same goes for the bloc $(u * v)$, with the next free register.

The whole machine code associated to the expression E is obtained in linear time from the generated codes of $(x + y)$ and $(u * v)$. It is composed of an evaluation code if x or y is modified, an evaluation code if u or v is modified and a complete evaluation for the most general case.

Here is the evaluation code of E when x or y is modified:

```

mem       dw 0
_Entrée_xy call ev_xy
          add reg, mem_el_uv
          mov mem, reg
          ret
  
```

A table (modified variable, entry point) is maintained and later used to select to right evaluation code to execute (via a call cell) when a variable is modified.

This way minimizes the re-evaluation of common sub-expressions, for example in the case of $0 < (x + y) + (u * v) < 5, (x + y) > 0 \ \&\& \ x + y < 3$; the bloc $x + y$ is effectively evaluated only once if x (or y) is modified and never for a modification of u .

A major drawback of the approach is the extra memory needed compared with a straight forward compilation of expressions. This is an often seen counterpart when using cloning techniques [3][15].

Another disadvantage of this approach appears when considering small expressions with few (e.g. 3) variables where a usual compilation consummates less memory and have a better execution time. But when the number of variables is over 3 or when a common sub-expression is shared by many expressions, using a partial re-evaluation mechanism seems better.

It has finally been chosen to use partial re-evaluation only for DAGs with a `usage_number` upper or equal than 3 (the extra-memory used by the generated code is then acceptable) and if no more than one object instance is needed.

4. Future work and conclusion

The purpose of this work is to investigate the problem of management for distributed applications with performance aspects in mind. We suggest a management system with a low-level management part able to deal with managed software components at runtime. It uses facilities provided by the JVM/JVMTI and compiles monitoring expressions on-the-fly to improve performance of the management system.

Currently both high and low level management remain too tightly coupled. In particular in the case of distributed management, complex monitoring expressions involving instances executed on different systems (for example variables in the client classes code and variables in the server classes code) need to communicate through their associated activities. We plan to definitively separate the monitoring part of the management system and enhance agents with communication abilities for variables values exchanges, leading to an autonomous distributed monitoring system.

It has been pointed out that the case of eMons using only `kms.<type>_mib()` calls, without further JAVA references, are to consider. This leads to the creation of an autonomous program (and not a library) build from the agent library which could then manage devices supporting SNMP interactions.

A second investigating field is the creation of MIBs from which standardized management tools could interact with the proposed system.

5. References

- [1] AspectWerkz, <http://aspectwerkz.codehaus.org/>, BEA Systems., 2004.
- [2] C. Carabelea, O. Boissier, Multi-Agent Platforms on Smart Devices: Dream or Reality ?, *1st Smart Object Conference*, 2003.
- [3] J. Dean, G. Chambers, Towards better inlining decisions using inlining trials, *LISP and Functional Programming*, p. 273-282, 1994.
- [4] EJB, Enterprise JavaBeans Specifications version 2.1, <http://java.sun.com/products/ejb/docs.html>, 2002.
- [5] P. Gupta, N. McKeown, Packet Classification on Multiple Fields, *Proc. ACM SIGCOMM.*, 1999.
- [6] HPROF, A Heap/CPU Profiling Tool in J2SE 5.0, <http://java.sun.com/developer/technicalArticles/Programmin g/HPROF.html>, 2004.
- [7] JVMTI, JVM Tool Interface, v1.0, <http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/>, 2005.
- [8] B. Kaddour, J. Quinqueton, Administration d'applications distribuées par Aspects et expressions programmables, *GRES'05*, 2005.
- [9] G. Kiczales & al, Aspect oriented Programming, *Proc. of ECOOP'97*, p 220-242, 1997.

- [10] R.P.Lopes, J.L. Oliveira, Delegation of expressions for distributed SNMP information processing, p. 395-408, *Proc. Integrated System Network Management 2003*, 2003.
- [11] J. Maebe, K. De Bosschere, Instrumenting self-modifying code, *AADEBUG2003*, 2003.
- [12] J.P. Martin-Flatin, Web-based management of IP networks and systems, PhD Thesis, 2000.
- [13] OMG, <http://www.omg.org/>, O.M.G., 2004.
- [14] OpenCCM, <http://openccm.objectweb.org/>, ObjectWeb, 2005.
- [15] A. Rigo, Representation-based just-in-time specialization and the psyco prototype for python, *PEPM 2004*, p 15-26, 2004.
- [16] RFC1905, Operations for Version 2 of the Simple Network Management Protocol (SNMPv2), <http://www.faqs.org/rfcs/rfc1905.html>, 1996.
- [17] RFC1907, Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2), <http://www.faqs.org/rfcs/rfc1907.html>, 1996.
- [18] V. Srinivasan & al., Packet Classification Using Tuple Space Search, *Proc. ACM SIGCOMM.*, 1999.