External Retrofitting of Autonomic Capabilities onto an Enterprise Application

Johan Riesen

KTH Computer Science and Communication

Master of Science Thesis
Stockholm, Sweden 2007
External Retrofitting of Autonomic Capabilities onto an Enterprise Application

Johan Risén

Master's Thesis in Computer Science (30 ECTS credits) at the School of Computer Science and Engineering
Royal Institute of Technology year 2007
Supervisor at CSC was Örjan Ekeberg
Examiner was Anders Lansner

TRITA-CSC-E 2007:133
ISRN-KTH/CSC/E--07/133--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication
KTH CSC
SE-100 44 Stockholm, Sweden
URL: www.csc.kth.se
External retrofitting of autonomic capabilities onto an enterprise application

Abstract
I have done my master thesis project on autonomic computing at Amadeus, a major provider of online travel booking systems. Autonomic computing is an effort to decrease the cost of owning an IT infrastructure through self-management of systems. The approach is to develop an infrastructure for an external autonomic manager which can interact with legacy systems.

This infrastructure is implemented and applied to a database cache in an effort to decrease the need of manual configuration and optimization. External adaptation has several advantages over integrated adaptation on the cache, mainly due to the clustered deployment of the cache.

Using several different approaches the need for human administration was decreased slightly, mainly through recommendation of queries to cache. Performance was insignificantly improved on current traffic. However, if traffic changes due to localization or application updates the autonomic manager can be used to provide additional dependability and perform immediate optimization without the need of human interaction.
Extern applicering av autonoma egenskaper på en företagsapplikation

Sammanfattning
Jag har gjort mitt examensarbete på företaget Amadeus, en stor distributor av system för resebokningssystem på Internet. ”Autonomic computing” är ett försök att reducera kostnaden av att äga en IT-infrastruktur genom att göra system självunderhållande.

Ansatsen i arbetet är att utveckla en infrastruktur för att kunna bygga en applikation som kan koppla upp sig mot äldre system och underhålla dessa.

Denna infrastruktur implementeras och appliceras på en databascache i ett försök att minska behovet av manuell konfiguration och optimering. Extern adaptation har flera fördelar gentemot adaption utförd inuti cachen, mycket beroende på att cachen körs på ett kluster.

Ett flertal ansatser att externt påverka cachen har prövats. Behovet av manuell konfiguration har minskats en bit, i huvudsak genom rekommendation av vad som kan cachas. Prestanda på cachen på nuvarande trafik är i stort sett oändrad oavsett vilken av mina idéer som testas. Vid trafikförändringar så kan dock ett flertal olika externt implementerade regler användas för att öka pålitlighet och utföra omedelbar optimering utan mänskligt ingripande.
5.5 Replacing cached procedures with others .......................................................... 23
5.5.1 Results ............................................................................................................. 23
5.6 Recommending stored procedures for caching .................................................... 24
5.6.1 Result ............................................................................................................. 26
5.6.2 Usage with external manager ........................................................................... 26
5.7 Removing stored procedures from caching .......................................................... 27
5.7.1 Result ............................................................................................................. 27
5.8 Setting the size of the cache .............................................................................. 29
5.9 Removing stored procedures from caching with memory ................................... 32
5.10 Limiting the cache in memory size .................................................................... 34
5.10.1 Results ........................................................................................................ 34
6 Analysis ................................................................................................................. 35
6.1 External adaptation applied to the solutions ...................................................... 35
6.2 Utility of the solutions ....................................................................................... 35
6.2.1 Finding cacheable stored procedures ............................................................... 35
6.2.2 Determining what to cache ........................................................................... 36
6.2.3 Setting the size of the cache ........................................................................ 37
6.3 Conclusion ........................................................................................................ 38
6.4 Future Work ..................................................................................................... 38
6.4.1 Cache .......................................................................................................... 38
7 References ........................................................................................................... 39
8 Appendix I ............................................................................................................ 42
8.1.1 Memory calculations ..................................................................................... 42
8.1.2 Measuring the size ....................................................................................... 44
1 Introduction

1.1 Amadeus
Amadeus is one of the leading companies in travel solutions. They provide a Global Distribution System (GDS), a computer system that travel agents use to book airline seats, rental cars, hotel rooms, and other travel reservations and services. Amadeus mainly handles airline bookings and more than 500 airlines distribute their flights through Amadeus.

Few end users use the GDS directly, but use one of the more user friendly interfaces for it that Amadeus provides. Amadeus has different products to serve the needs of travel agencies, airlines, big corporations and single customers.

My work was done in the e-Travel department which develops two main products. The e-Retail Engine/PlanitGo which is a booking engine running on Amadeus servers. Other companies, like airlines and travel agencies, who wish to provide booking functionality on their websites may use the Amadeus e-Retail Engine and customize it to their needs by over a 1000 customizable parameters. The end users of e-Retail are single customers that wish to do their own bookings online.

The e-Retail Engine handles about 150 million pages per month (Jan 2006) on a set of 30 servers. This amounts to roughly 15 page requests per second and 4,500 user sessions at any given moment on a single server.

Amadeus e-Travel Management/Aergo is the other big product and includes the booking engine as well as a ready made user interface and is intended for corporations who wish to book their business trips online.

Both products are web applications built on the Java 2 enterprise edition technology and a lot of source code and functionality are shared between the two systems.

1.2 Autonomic computing, introduction
With the increasing capabilities of today’s computers and the introduction of distributed computing [1], systems are getting increasingly large and more complex. Maintenance, the longest and most difficult phase of the software process lifecycle [2] is therefore becoming increasingly hard. It is mentioned [3] that the development, management and configuration of the increasingly complex systems is starting to break current tools and paradigms.

IBM noticed that with the current rates of expansion there will be a shortage of IT workers and the work for IT-workers will be made hard “the complexity is growing beyond human ability to manage” [4].

System complexity is not the only thing growing, so is availability and performance demands. Standard manual offline configuration and adaptation is proving insufficient when it comes to handling these increased demands. IBM identified the need [5] to reduce the cost and complexity of owning an IT infrastructure and launched a long-term autonomic computing initiative [6]. In
October 2001 they released a manifesto [4] which they further elaborated in 2003 with their vision [7].

The term *autonomic* is inspired by biology and the human autonomous nervous system. Body temperature and heartbeat takes no conscious effort to control, the hope is to let computer systems manage themselves in much the same way. The aim is to allow administrators to specify high-level policies that define the goals of the autonomic system, and let the system perform self-management to accomplish these goals. The administrators work is thereby reduced from tweaking hundreds of parameters to defining high level policies for how the system should behave. IBM defined five levels ranging from basic to autonomic behaviour of a system. The basic level relies on skilled IT staff performing manual actions to configure and optimize the system while little effort is required to provide the high-level policies used on the autonomic level.

To achieve self-management IBM defined four general properties a system should have: self-configuring, self-healing, self-optimizing and self-protecting [7]. Since the launch of Autonomic Computing, the self-* list of properties has grown [8, 9]. It now also includes features such as self-anticipating, self-adapting, self-defining, self-destructing, self-diagnosing, self-governing, self-organizing, self-recovering, self-reflecting, and self-simulation [8, 10].

### 1.2.1 Arguments against autonomic computing

The main motivation for companies such as IBM to introduce the autonomic computing concept was to alleviate the growing complexity of today’s systems. While J. Miller and P. Thomson agrees on the problem of complexity they do not agree on the solution [11]. The authors make several arguments against autonomic computing, a important point is the catch 22 problem of fighting complexity with complexity.

> It is difficult to see how engineering more complex software or hardware can possibly help reduce the growing problem of software construction and maintenance since by assumption the proposed system is even more complex. [11]

The automation irony [12] is another arguable weakness of autonomic computing. Automation irony refers to the fact that introducing automation makes developers increasingly dependent on it. System administrators will be ignorant of the internal workings of the system since it usually adapts itself so when something goes wrong they are unable to fix it.

Another point mentioned in [12] is the small probability of fault free adaptation code. If developers have produced faulty systems in all times, what would make them produce fault-free adaptation code? Both these points are argued against in [12].

### 1.3 Terminology

*Autonomic computing*: An initiative to improve system performance and reduce the need for manual administration by letting systems manage themselves.

*Autonomic management engine* (AME): An AM implementation by IBM.
**Autonomic manager (AM):** A running implementation of the framework acts as an autonomic manager. The autonomic manager is an application managing some legacy component.

**Amadeus Booking engine:** Referring to the common code in the Amadeus Planitgo and Aergo products. The booking engine processes different kinds of bookings related to travels.

**Cacheable store procedures:** Store procedures that may be cached without breaking the application, store procedures like this returns static data that does not change in the database.

**Cached store procedures:** Store procedures being cached. Results from these store procedures can be put into the cache.

**Example implementation:** This is an implementation of the abstract framework providing autonomic capabilities to the Amadeus database handler cache service. The example implementation code is not necessarily reusable but large parts may be specific to for instance the cache or components supporting JMX.

**Framework:** In my case a set of abstract java interfaces specifying things like workflow and communication between components. The frameworks main components are sensors, gauges, controllers and effectors. Besides abstract definitions some code for communication i.e. sensor-gauge communication is included. The framework is supposed to be reusable and possible to fit to a range of components such as cache or queue services.

**Hooking code:** Code that is placed inside or outside a system to monitor or affect it.

**Item / Store procedure result:** Different calls to the same store procedure may result in different results from the database. A store procedure result stored in the cache is also referred to as an item.

**Java management extensions (JMX):** A protocol for monitoring and affecting a remote application.

**Legacy system:** Here refers to any already developed system not designed for autonomic behaviour.

**Sandbox:** Here refers to a cache simulation were tests can be made with no risk affecting the overall system performance.

### 1.4 Thesis purpose

This thesis has three main purposes.

1. To examine the feasibility of external retrofitting autonomic capabilities onto the Amadeus booking engine by an external application.
2. Evaluate if having an external autonomic manager is suitable in this context as opposed to internal adaptation.
3. Make a measure of the overall system improvement achieved fitting these autonomic capabilities. Examples of system improvements are reduced time needed for administrating the system and performance increase of the system or part of the system.

1.5 Problem formulation

Trying to achieve the first thesis purpose defined above there are questions that need answers.

- How can an external autonomic manager monitor and perform changes on a legacy enterprise application such as the Amadeus booking engine?
- How much of the work on one component in the system can be reused for other components? The probability for success in making the bookings engine autonomous is small if no work done on one component can be reused for another component.
- How big is the need of knowledge and analysis of a legacy component to improve its performance by adaptation?

Looking into the second purpose of the thesis I identified some important questions.

- How much can the need of manual administration be reduced?
- What amount of performance/dependability increase can be achieved?
- Can we provide additional functionality to the system?
- Is it possible to improve the system with an acceptable CPU and memory impact?

1.6 Method

I am to answer the problem formulation by developing a general framework and making an example implementation which can be fitted to a component/service in the booking engine. An iterative development approach will be taken. There are several parts of this iterative cycle: development of the framework, the service dependent implementation and changes to service code.

The initial workflow will be: exposing a few properties of the service - basic external orchestration of the service through simple adaptation – a first basic framework design – implementation of more sophisticated adaptation for the service.

The target of the example implementation is to provide self-configuring and self-optimizing capabilities to a database cache service. To achieve self-optimization I will try to improve performance of the cache in regards to cache hit ratio. This performance is measured by replaying database traffic from the production environment and study cache hit ratio through log analysis. There are already tools for log analysis and for replaying database traffic available at Amadeus. For self-configuration I will try to reduce need for manual administration of the cache with no performance decrease.

To make sure that my improvements to the cache in regards to hit ratio and decreased need of administration doesn’t impact memory and CPU usage of the cache I will use profiling tools.

Great care will be taken in balancing spent time on the service independent and dependent parts of the work. This is to get an autonomic manager that orchestrates the cache decently well and a framework that is decently reusable for other components.
1.7 Limitations

The booking engine is deployed in a cluster environment but developing my framework I have not addressed the problems a cluster environment introduces. Examples of this, deploying components and keeping track of components and network impact of the required communication.

2 Autonomic Computing

2.1 The control loop

An approach to achieve self-management is to introduce a closed control loop [14]. One such [13] is described in Figure 1. The management can be described as a monitor, analyze, plan, execute (MAPE) cycle. The element is a system component controlled by the autonomic manager (AM). The element may be a resource such as a server or application or a collection of resources such as a cluster. The AM is an implementation of a control loop and is in itself an autonomic element which may be controlled through sensors and effectors. The broad line is not a strict control flow but an asynchronous messaging bus, for instance, the effector may report success or failure back to the execute phase.

Autonomic managers may be layered in a hierarchical arrangement or collaborate in a peer-to-peer manner, communicating by their sensors and effectors.

Sensor

Information collection on the element targeted for management is done by sensors. Sensors may also be used to collect information on the environment of the element. As example, a cache’s memory usage may be monitored by a sensor on the cache and other sensors may be deployed on the server hosting the cache to monitor overall memory. Sensors on the AM are used manually or by another AM to monitor progress and to discover alerts.

Monitor
In the monitor phase the information from the sensors is gathered. The monitor phase only pass information to the analyze phase on certain conditions as to avoid passing large amounts of data for nothing. Examples of conditions are errors or performance degradations.

**Analyze**
The information collected must then be analyzed to see if any action should be taken.

**Plan**
What action should be taken or how can several actions be coordinated to remedy the situation?

**Execute**
When the correct action or sequence of actions is planned it is time to invoke the effectors.

**Effector**
Once the proper action is decided orchestration of the system is done through effectors on the target element. The effectors on the AM are used to inject high-level policies to determine the AM behavior.

Control loops like this can be found in many autonomic approaches [15-20].

### 2.2 DASDA infrastructure

A consortium including BBN, CMU, OBJS, Teknowledge, WPI and Columbia developed a reference architecture [2, 20] to externally retrofit autonomic capabilities to a system. The consortium consisted of members of DARPA’s DASDA community and was sponsored by DARPA which dropped the funding before finished. It is explicitly designed to be lightweight and modular to support a number of different implementations and technologies.

![Figure 2: Figure from [20].](image)
The sensors and effectors provide the same functionality as in IBM’s control loop. The gauges are responsible of the monitor and analyze part of the control loop and they collect information from the sensors. Once events or patterns of events from the sensors that deviate from supposed system behavior are detected the gauge emits a meta level event describing a situation to the gauge bus. Another gauge may collect this situation from the bus resulting in a layering of gauges. The situation can also be collected by a controller or several controllers constituting the control layer.

The control layer responsible for the plan and execute part of the control loop will react to the gauge emitted situation. The gauge may bundle the situation with the sensor data generating the event to provide the controller with additional information to be able to take the proper action. The control flow represented by the dashed lines illustrates how the controller can choose to deploy new or modify existing components.

Gauges and controllers need to have system specific information to know what to look for and to be able to plan proper actions respectively. This knowledge may be represented in several different ways, for instance a behavioral model as in [20] or an architectural model as in [21], more on this later. Much as the autonomic managers in the IBM control loop controllers and gauges may be layered or collaborate.

Before funding was dropped the gauge and sensor interaction protocols were standardized [22, 23] based on Siena (Scalable Internet Event Notification Architecture) [24]. Siena is a publish/subscribe type of event notification service. Publish/subscribe systems fall into two categories, topic and content based. In a content based system components register to events with certain content and this is the type used in Siena.

2.3 Examples

There are far too many efforts in the area of autonomic computing to list them here but I will give a few examples of where and how it is used.

K42 [25] is a research operating system providing dynamic adaptation mechanisms through hot swapping of the OS code. Using this functionality, resource-managers for physical resources such as memory, disc and network may be replaced on the fly with another implementation exposing the same interface.

An example of how the OS could be optimized is to use hot swapping to change policy for prefetching of memory blocks and memory pages from disk. Depending on application access patterns different policies are optimal [26, 27], using monitoring code to detect these patterns the best policy can be selected.

JAGR [28] is an example of incorporation of autonomic capabilities on the application level. JAGR is a modification to the JBOSS J2EE application server to make it self-healing. They use a number of failure detection components to detect anomalous behaviour. They use an automatic failure-path inference algorithm [29] to determine the true source of a problem.

Due to the increasingly complex DBMS systems these were early in trying to achieve self-management, and so with success [31]. The DB2 DBMS from IBM has autonomic features such as a query optimizer, index advisor and failure recovery.

Self-Managing And Resource Tuning [32] (SMART) is an IBM initiative to further extend the autonomic capabilities of DB2. The goal is to decrease the need of human interaction and optimize performance. Some of the new functionality is realized by advisors which model alternative designs and configurations to propose the best one. Self-validating mechanisms learning from past experiences to improve decisions are also part of the initiative.

### 2.3.1 Recovery oriented programming

The Recovery-Oriented Computing (ROC) project [33] is a joint Berkeley/Stanford research project often mentioned in the field of autonomic computing [34]. The ROC group is mainly focusing on increasing service availability, particularly in internet services. They motivate the work as an answer to the call of IBM [4] among others for more dependable and manageable systems.

Total unavailability of a system is roughly mean time to repair (MTTR) / mean time to failure (MTTF). Since more work has been done in the area of minimizing MTTF they are focusing on lowering the MTTR.

One presented way to better the MTTR is recursive restartability [35]. Reboot is a common way to resolve problems, deadlocks and memory leaks are examples. Unfortunately most systems do not support hard reboots and data loss and long startup time follows. By partitioning software it is possible to restart only parts of a system opposed to the whole system which lessens reboot time and limits data loss.

Another solution proposed by the ROC group is an undo-feature to alleviate mistakes by system operators. An operator making a mistake is able to undo his change, reset the system to the state before the change and then replay all user interactions with the system during the time the system was faulty.

### 2.3.2 External adaptation

To enforce self-management on legacy systems not designed for this an autonomic manager external to the application is often used.

IBM provides a solution with their autonomic computing toolkit [36] where resource models are fitted to an autonomic management engine (AME). The resource model is built using a tool where resources such as memory, CPU and file system content can be monitored and associated with constraints. Whenever a constraint is violated such as a certain threshold broken two poll cycles in a row a response action is taken. The actions are predefined and associated with a certain constraint violation.
The IBM toolkit is a ready made tool and easy to use but it is suited for management of an application and less so for a distributed system.

A more general approach focusing more on distributed applications is the Kinesthesetics eXtreme [20] or KX. This is an implementation of the DASDA reference architecture described in Figure 1. Since the system determines the preferred sensor technology they do not limit themselves to any specific but have so far employed among others AIDE [37] for hook insertion and OBJS’ ProbeMeister [38] for bytecode and binary modification. As different sensors technologies have different output format they have developed an event packager for transformation to their own format.

The main gauge component is called EventDistiller and can perform complex temporal event pattern analysis based on continuous data streams from several sensors. The EventDistiller is fed with rules in XML which describes what patterns to look for and the meta level event to emit when the rule applies. They have also started integrating support of the ACME architectural description language constraints, this is further explained in the next chapter.

EventDistiller is implemented as a number of state machines, whenever the fist condition of a rule is fulfilled a state machine will be launched waiting for subsequent states to be fulfilled. While this is a bit memory intensive they use timeouts and garbage collection to limit the impact.

Decision of when to take an action based on the gauge output and coordination and planning of performing the action is all done in the controller. To implement the controller they have used a workflow approach to coordinate and order the actions to be taken. Both the IBM AME and KX follows the control loop pattern explained above.

### 2.3.3 External adaptation with an architectural approach

A common way to support external adaptation is to use architectural models of the system. This is normally used for distributed systems with many components and readily describable architectures.

To describe the system architecture an architecture description language (ADL) is used. A common ADL core is to use a graph describing interacting components. Examples of components are servers, clients, databases, user interfaces and data stores. The arcs between these components are called connectors and may be realized in any number of ways. The elements in the graph can have properties, a connector may for instance have bandwidth and latency properties associated with it [15]. A few ADL:s using this core are Acme [39] xADL [40], and SADL [41].

The model may also incorporate constraints for these properties such as an upper bound for a latency property of a connector. A strength of the architecture description approach is the ability to automatically verify these constraints to find faults and repair strategies.

To be able to externally adapt a system using an architectural approach two basic prerequisites must be fulfilled; 1: The architecture of the system must be observable and describable 2: the autonomic manager must be able to affect the architecture of the system. Typical examples of
architectural adaptation are to change the topology of the system by adding/removing/updating components and connectors [42].

An architectural description may be further extended with a style as proposed in [15]. The architectural style defines a set of types for components, connectors and properties as well as rules for how they interact. To introduce a style is a tradeoff between the ability to express any system on one hand and additional constraints for use in for instance fault detection and code generation on the other.

The following approach is one way to realize architectural adaptation [15]. The target system is monitored through sensors and gauges abstract the values to architectural properties. Whenever a value changes a constraint evaluation takes place. If one or more constraints are broken repair mechanisms make changes to the architectural model which is then propagated to the system through predefined repair plans.

A similar system by R. Sterrit [10], has two main differences; firstly no architectural styles are used and secondly the approach is more general as there is no need to predefine repair strategies. They have developed a tool (ArchDiff) to calculate the difference between two architecture descriptions described in the xADL 2.0 ADL and symmetrically a tool (ArchMerge) for merging of a difference and a description.

A repair plan can be described as a difference and their ArchMerge tool is used to merge a copy of the existing architecture with the difference achieving a target architecture. The target architecture is then validated by a number of design critics checking that all architectural constraints are obeyed. If they are obeyed the difference can be used as a repair plan on the running system. Changing the running system is made by changing the architectural description of the running system and let it propagate to the running system. The work however does not so include a way to determine when and how to repair the system.

An architectural self-organizing approach is implemented in [43]. Each component is fitted with a component manager responsible for having the component fulfilling the constraints in its description. To have each component manager maintain a view of the entire architecture broadcast is used which limits the size of the system to a hundred components.

2.4 How to hook into a legacy system
Here I describe how an external autonomic manager can monitor (through sensors) and effect (through effectors) another application.

The ideal case would be to have one sensor/effector capable of monitoring and affecting any metric of any system. This is naturally not possible, for every system you need to come up with ways to gather and pass information. The approach in designing sensors is much like the one for designing effectors, you use whatever works for each specific target system.

Logs
Logging has long been a commodity to be able to follow the progress of a system and nothing stops an autonomic manager to use logs for information gathering. While a log may be easily
interpreted by a human being the same is not necessarily true for a computer. For starters, different systems may use different logging services and systems using the same logging service may still differ in format and language.

IBM identified the need to be able to convert the logs into a common format and analyze them and included two helpful products in their autonomic computing toolkit [36]. The generic log adapter converts logs to their common base event format and the log and trace analyzer helps in analyzing the logs.

**OS**
Most operating systems provide basic information such as available disk space and memory and CPU information for processes.

**Java and Middleware**
Java monitoring of an application through standard OS mechanisms was inaccurate in my environment as a JVM running several threads/applications only expose one process to the OS [44].

If a target system is deployed on top of middleware such as a J2EE application server, the application server as well as any application it is running on will share the same Java process in the operating system. This has led the application server developers to introduce out-of-the-box monitoring of their applications. The BEA WebLogic provides monitoring through the Java Management Extensions (JMX) protocol of attributes [45] such as memory and thread pool data and IBM WebSphere has similar capabilities [46].

The monitoring in many major application servers are ad-hoc and the implementations are unique to the application server, this monitoring is often insufficient and does not obey to any standard. A concept that can be used to improve the monitoring is to isolate java applications running in the same environment from each other [44].

**Helpful technologies**
With luck a system exposes properties through some protocol to allow for manual administration. Some technologies and protocols mentioned [18] that can be used for monitoring and in some cases effecting are JMX, WSDM, WSRF, SNMP and CIM. Older protocols such as telnet and SSH can also be helpful.

**Fake client**
Many systems provide some kind of client access or service and so the concept of fake client may be used. A fake client is used in the modified JBOSS application server JAGR [28]. The client is used as an end-to-end failure monitor and reports when a request seems to fail.

**Change source code**
Changing the source code of an application to enable monitoring and/or management can be done following any number of design patterns. A good overview of how source code can be changed is available in a thesis by P.W. Gill [47].
Source code does not have to be changed manually, there are systems such as WPI’s AIDE to automatically insert hooks into source code. AIDE has already been used with success in the context of autonomic computing [15, 20].

**Other alternatives**
There are several more alternatives such as byte/binary code manipulation, wrappers, instrumented connectors and eJava explained well in [47].

### 3 The Cache

#### 3.1 Caching in general

Caching is a way to trade memory for time. Data that normally takes a lot of time to collect from a data store such as a database can be stored in memory for faster access. This type of technique is used on nearly all systems in for instance caching of hard drive data in RAM memory and caching of instructions in the CPU cache. Unless explicitly stated I assume a uniform-cost for collecting data from the database and a uniform memory size of the results returned.

The cache can normally not contain all data so when it is full something needs to be removed for addition of new data. An algorithm performing this operation is called a replacement algorithm.

When the cache is full an optimal replacement algorithm will always remove the item from the cache that has the longest time till next reference. This requires that the future references are known which is uncommon. Normally a replacement algorithm tries to predict the future from the history. One way to make this prediction is based on temporal locality. Cache items having temporal locality means that the access time of an item can be used to predict when the next future reference will be made.

A very popular replacement algorithm using temporal locality is the *least recently used* algorithm (LRU). LRU will evict the item that has the longest time since last access. Due to performance reasons this algorithm is often approximated but in the Amadeus databasehandler cache this is not an issue so a true implementation is used.

LRU is an overall very strong algorithm on most types of traffic but has a few weaknesses. The standard LRU does not regard history, only the last access time is used which due to chance may be non representative. It is also weak when there is weak temporal locality. The LRU is also very weak in looping patterns where the opposite, MRU is actually optimal. When a loop of n+1 different call is made in with a cache sizes of n LRU will result in 0% hit ratio, it is said that the cache is trashing.

An overall weakness of replacement algorithms is the mandatory eviction. Replacement algorithms will always evict something from the cache when there is a miss.

#### 3.2 The Amadeus databasehandler cache

The Amadeus has a database handler handling all calls to the Amadeus database. The database stores a multitude of different data such as localization information and website configuration parameters for different airlines. This database has a lot of traffic, a single node in the Aerog
American cluster send on a sample of 10 hours about 630,000 calls. To ease the load of the database each node has a database handler cache, caching the results of chosen calls.

Amadeus has an approach where all data from the database is collected through stored procedures located on the database server. Each stored procedure takes zero or more parameters and then return a result. Calling different stored procedures or calling the same stored procedure with different parameters will generally result in different results.

The database handler cache will then store the result and associate it with the stored procedure call. The call is made up of the stored procedure name and the parameters used.

The cache has a number of configurable parameters such as: purge settings, definitions of what to cache, definitions of sub caches and the size of the cache. A subcache is a small cache that caches a set of calls belonging to a single stored procedure. The size of the cache is now measured in number of items which is synonymous with the number of results. I will explain later how the cache can be limited in other quantities.

3.2.1 The purge
The purge runs periodically and processes the whole cache to remove items not satisfying some predefined conditions, it has a multitude of configuration parameters. The original motivation for the purge was to remove stale items from the cache but in a correctly configured cache the LRU will remove stale results anyway. There is one purge for every cache so a sub-cache may run a purge with different settings than the main cache.

If the result constraints are set very strict the purge can be used as a periodical total clear of the cache to force result reload, this is currently done on a subcache in production. This exploitation of the purge functionality is a waste of CPU as a simple clear of the cache is sufficient. I have not addressed automatic configuration of the purge as I see no use for it and the purge is disabled in all my tests.

3.2.2 What to cache
The definition of what to cache is a long list of stored procedure calls that may be cached. Most commonly only a stored procedure name is given and the wildcard * given on all or some parameter positions, meaning any parameter is permitted on that position.

This list of what stored procedure calls to cache is lazily named by Amadeus and referred to me as the list of cached stored procedures. It is lazy because it is not really the list of cached stored procedures but in reality the list of cached stored procedure calls.

There are two reasons for a stored procedure or stored procedure call to not be represented in this list.

First, caching certain results will break the application. This basically means that the data in the result may be updated in the database so caching it could mean that the cache is returning outdated data to the user. Another possibility is that the call performs an update so preventing the database trip by caching would prevent the update. I have introduced the concept of cacheable
stored procedures, the definition of a cacheable stored procedure is any stored procedure that may be cached without breaking the application. A stored procedure sabotaging hit ratio may still be cacheable but is not suited to be listed as a cached stored procedure.

The other reason is that a stored procedure call may for instance include a timestamp as a parameter and hence only be used once. In the best case it would just be a waste of memory caching the result since the same call will never be done again. In the worst case the cache would fill up with worthless results and prevent caching of results that would generate cache hits. To have a cacheable stored procedure disincluded from the list of cached stored procedures may improve the performance of the cache. This is due to the mandatory eviction of the LRU replacement algorithm.

It is not sufficient for a stored procedure to sabotage cache hit ratio to regard it as a not cacheable stored procedure.

3.2.3 A list of cacheable stored procedures

As the list of cached stored procedures in the current configuration does not include all cacheable stored procedures I wished to extend this list. To do this I got a database query from the database team returning all stored procedures having to do with localization. The stored procedures associated with localization are guaranteed to be cacheable, some of them were already cached in other places though so had to be removed. I call this list the list of real cacheable stored procedures.

In some tests the list of all stored procedures is used as the list of cacheable stored procedures for testing purposes. There are several reasons for this, one is that the true list of real cacheable stored procedures is unknown so all might as well be used even though some of them are definitely not cacheable. Another reason is that testing is faster and more extensive as all calls recorded in the logs are used, no data is left unused. It is also a way to wary the traffic, the current traffic is not always the target of optimization but also alternate traffic such as possible future traffic. Using another list of cacheable stored procedures such as the full list of stored procedures is a way to simulate other traffic.

3.3 Amadeus needs

The cache is currently configured manually in two ways. When the application is starting a configuration is loaded from an xml file and so, making changes in this file will take affect on a reload.

The other way is to access the configuration of the node through the administration interface. Each individual node need to be accessed and the configuration changed, provided the parameter is settable through the administration interface. This configuration change will be lost if a restart of the node is made. To change configuration in this manner is tedious, all nodes need to be accessed and the changes made through a configuration file and/or the administration interface. Another issue is to determine when and how a configuration change is to be made.

At this moment Amadeus runs 14 active different server farms supporting two main products (Aergo, Planitgo). Different products have different database traffic but the same product
deployed on two different locations may also differ due to localization. For instance, the Europe Germany Planitgo farm, supporting mainly Lufthansa has different traffic from Aergo American farm due to localization as well as the fact that they run different applications.

To ensure an optimal configuration given these conditions a study need to be done for any new release of any of the products. This study would have to include all farms on which the product is deployed and the configurations on all nodes in the farms need to be tweaked to the traffic on the farm. This is of course too time consuming so a middle-road is taken where a study on one type of traffic will determine the configuration for several farms and products.

Even though there are internal tools to help optimizing the configuration is not done easily since there are a many parameters to take into consideration.

With this information I see a potential for autonomic adaptation. If configuration could be done autonomously from data collected at runtime I see room for several improvements.

- Since the configuration would be most up to date and tailored to the exact product/farm at all times there is a possible performance increase.
- The need for manual configuration would decrease as high level policies would determine parameters previously manually tweaked.
- Additional dependability of the system as the cache adapts to traffic changes.

4 The Framework

Since the goal is to retrofit autonomic capabilities I have chosen an approach where the autonomic manager (AME) is external to the target application. In my case this means the AME is not running in the same JVM as the booking engine. Motivation for this approach can be found in Motivation for external adaptation. It is worth noting that I have worked on a component level, focusing on one component only. Working on a distributed level with several applications/systems with higher level adaptation is more common for external approaches like this.

I have used the component model and terminology from the DASDA infrastructure but no code. While other systems use different models such as rules in xml or architectural models I have hard coded logic. This is more powerful as any type of rule can be created without any obligation to conform rules to some format, it was preferable for me due to the limited time I could spend on the framework. The mayor problem with this approach is the limited reusability and poor management capabilities of the adaptation behaviour.

I implemented a gauge component having hard coded logic for any type of situation in the cache. To enable the gauge for a certain type of situation another gauge or a controller needs to register itself in this gauge or perform a poll on the gauge and in doing so passing a number of parameters. These parameters include target situation and situation details. An example on this is a controller registering itself for a low memory event passing the parameter 100mb as a lower bound.
Here is a description of how deployment works. A controller is instantiated, the controller knowing what it needs to do will ask an instance handler component to instantiate relevant gauges and effectors. If another component asks for the very same type of effector/gauge the instance handler can reuse the already created components. The gauge in turn will in a similar way ask for the proper sensors or other gauges.

Gauges, sensors and controllers all support events and polling for interaction while the effector can be invoked by a call from the controller. This is much as how IBM pictured [48] the component interaction with one addition. They also define effector call-out-request as an alternate way to invoke an effector. In this case the effector knows it is supposed to do something but not what so it asks the controller what to do, I do not handle this case.

I have been working on a single machine so I have used inter-component communication by memory references. It is prepared for plugging of other communication protocols such as a publish/subscribe message passing solution used in [20] but there is still work to do on this. While there are several different XML events formats defined such as IBM’s common base event (CBE) [36] and smart event used in [20] I have used my own event format. It is a normal java object containing a hash map with data stored with predefined keys.

4.1 **Motivation for external adaptation**

The overall target of autonomic computing is to increase system performance and dependability while decreasing the need of system maintenance by self-management through runtime adaptation.

Adaptation is not a new concept and many new systems have adaptation code, here I present some of the advantages of having adaptation code outsourced to an autonomic manager instead of hardwired into code. I will motivate why 1: adaptation code should not be hardwired and 2: the benefits of an autonomic manager that is external to the application.

4.1.1 **Maintainability**

If adaptation and hooking code is inserted manually into the application policies and code may be hard to locate and analyze. Another issue is the mixing of functionality. The adaptation code and functional code is likely to go through different evolution phases which will be made hard due to the lack of separation. The component developer needs to understand the adaptation code and the adaptation code developer will need to have an understanding of the functional component code. It also makes responsibility of the component code diffuse.

4.1.2 **Code reusability**

Hardwiring adaptation code in applications is the most common approach in new systems. This tends to result in one-of solutions tailored to the specific application [2]. Ad-hoc solutions like this have several drawbacks and one is the poor reusability of such code.

In my current implementation the code reusability is weak since much of the logic is unique to the cache. In more advanced system in research adaptation policies can be derived from system models and sensors/effectors generated from code templates.
Large amounts of code in for instance KX [20] to analyze an architectural model described in an architectural language and transforming this to generic adaptation actions.

### 4.1.3 Changes in adaptation policy

Hardwiring adaptation code in application components such as the cache makes it difficult to change adaptation policies and code [16]. For instance, a change in adaptation behavior will need to be made on all nodes in the cluster. In the best case a configuration reload on all nodes is needed and in the worst case recompilation and redeployment of the application is necessary.

Using an autonomic manager you change the adaptation policy in the external AM and the effect will be cluster global. Even if you need to restart the autonomic manger the main system does not necessarily need to be taken down.

### 4.1.4 Work with what you have

Retrofitting adaptation directly into legacy systems may be impractical when you have a limited understanding of the design and implementation. You may not even have access to the source code of the target application and so you are forced to external adaptation [2].

Some systems exposing very few properties may be impossible to orchestrate externally while others are better suited. In short, fitting an autonomic manager is not always possible but when possible may save a lot in time spent understanding or reverse engineer the legacy system.

### 4.1.5 Deployment choices

Putting the adaptation code in the application means that in a cluster all nodes will run the same code which is not always the desired or optimal behavior. Using an external AM divided in components as in the DASDA infrastructure deployment is easily varied to the specific needs. Here are a few examples of different deployment options.

**Same configuration on all nodes in a cluster:**

One option to limit the impact of recording is to put a sensor on only one node and enable recording only on this node. This sensor can then send information to a gauge and controller on the same or another machine. The controller then affect the configuration on all nodes by effectors deployed on all nodes.

**Same configuration on all machines but more data:**

If more data is needed and data collection has a limited impact a sensor can be deployed on every machine reporting to the same gauge.

**Different configuration on each node:**

If the traffic for some reason differs between nodes in the cluster the above example can be extended to having a sensor, effector and gauge on each machine. The gauge will only collect data from one node and report to the controller who makes changes to the specific node configuration.
4.1.6 Reusability of information
Having an external system handle adaptation and storing system data and knowledge makes it possible to use the same information in making decisions on several different applications.

Say that the Amadeus booking engine and another application is running on the same machine. Instead of having both applications collecting system information such as memory, CPU data or time since last reboot you can do it in one place.

4.1.7 Global adaptation
Localized error handling following from hardwiring code is weak when it is necessary to perform global adaptation or coordinate several adaptation operations [16]. Global adaptation refers to adaptation that involves several nodes and/or components.

For instance, a node with a sudden CPU peak may need to kill the defrag process running. A more complicated example requiring coordination may be to order another server running defrag to halt the defrag process and load balance more transactions to that server.

Another example of global adaptation is given above in “Deployment choices”. Data collected on one or several nodes may be used to affect all other nodes.

5 Solutions

5.1 How to affect and monitor the cache
The cache component is already exposing several properties and functions through JMX so exposing additional ones is a very simple matter. Some information can also be gathered from logs. There are 5 different logging levels, debug, error, info, fatal and warning. Normally error is used in production which does not provide any information I have needed in my approaches, on info level however all calls are logged.

Info level logs could be used as a source for some of the information collected but logging is not done on info level in production due to the big performance impact. Furthermore, as logs only can be used for monitoring and not monitoring/affecting like JMX I decided to use JMX exclusively. While using logs would save some memory the data still needs to be read into memory eventually.

Some of the additional functions and attributes exposed through JMX were a simple matter of defining existing code as remotely available while others involved changes to the cache code.

5.1.1 Compilation of exposed cache functions/attributes
The cache was exposing a number of attributes and functions through JMX but I have added some which I’m listing here.

To collect hit/miss data on stored procedures getting cached and number of calls to procedures not getting cached three maps were used. This was solved by addition of a few lines of code, the information is also available from info level logs.
A map describing the calls that should be cached is already available, methods for adding and removing calls from this map exists but were commented out for unknown reason.

The number of items in cache was a read only attribute, I made this writable with minor changes. If the maximum number of items is lowered below the current number of items in cache the oldest excess items will be removed on the next get or add to the cache.

Number of hits and misses in total was added as primitive attributes and methods to reset these values. The total amount of hits and misses of stored procedures getting cached are computable from the maps storing hit miss data per stored procedure.

5.1.2 Using time information

Different stored procedures may take different amount of time to execute. A query taking a large amount of time slows the booking engine and puts more stress on the database should therefore be cached while a fast query is less attractive for caching.

So far I have used information collection on a stored procedure level opposed to a call level, mainly for memory reasons. I continued doing this and so assumed that all calls to the same stored procedure takes a similar amount of time.

By getting system time with System.currentTimeMillis before and after the execution of the query an approximation of the time used can be made. Adding all these times together gave big differences (e.x. 72s vs 114s) playing the same traffic twice, indicating that the same query can take a different amount of time. There are several possible reasons for this. For one thing time slicing in the processor makes this kind of measurement unsure, if the load on the machine is stable this impact is probable to be small. Another point is the load on the database and network. Load peaks on the database and network could cause a query to take a long time. I also see this as a small problem as the production database is scaled to never use more than 50% CPU.

The biggest issue is the granularity of the measurement. When watching the times measured a lot of 0ms and 16ms queries are made but never anything in between. This depends on the Java 1.4 implementation on my platform. Windows XP and Windows 2000 are the platforms used at Amadeus. The windows native utility used by Java getting the current time on these platforms is GetTickCount which has a 10-16ms resolution. Other methods such as Thread.sleep() might or might not use a underlying low-resolution timer depending on platform. In this context it is impossible to use Thread.sleep so I did not conduct any experiments with that method.

100ms is mentioned as a limit [49] from which it is possible to get decently accurate results based on System.currentTimeMillis. This was also the limit from which I started seeing consistent results, that is, queries taking more than 100ms one time took a very similar amount of time next time. This however is of very limited use to me since a majority of the calls take 0-16 ms.

In Java 1.5 a method System.nanoTime is introduced using the most precise system timer available. This could be a great help in measuring time correctly but merging the Amadeus booking engine to Java 1.5 is not done overnight.
After these considerations I decided not to use any time information.

5.1.3 Memory size
The size a result uses in memory was computed from the byte stream returned from the database, how this is made is explained in more detail in Appendix I. The result object Amadeus uses to store the result was extended with a property describing the size of the byte stream returned from the database and the cache was extended with a property summing up the size of all the results currently in the cache. The cache is then exposing the total size of all results in the cache as well as a map containing the size of the last result of each cacheable stored procedure. Memory and code wise these changes were small and easily made.

The last result size is stored used as the size of the stored procedure, this can cause bad decisions of the autonomic manager since result size can vary between different calls to the same stored procedure. The most frequent call size is most probable to represent the size of the stored procedure.

Out of a sample of 120k Aergo calls ~2% had more than 10% difference between the most common frequent call of the stored procedure. The same value for a 50% difference was 1%. The difference between the most common call and the mean size had similar numbers since most stored procedures have identical sizes of all stored procedures. From this data we can see that most calls to the same stored procedure have similar result sizes. A size difference of 10% or even 50% is not a problem in the currently implemented rules. All calls to one particular stored procedure had 0 byte results while one had a 30kb result. However unlikely, if this call would be made right before the information was fetched by the autonomic manager it could cause a decision based on incorrect data.

5.1.4 Changes in source code and external adaptation
Most data collection is made by code inside the application and stored therein. This is not the desired way for monitoring as some of the benefits for externalizing adaptation are lost. Part of the benefits described in Maintainability, Code reusability and Work with what you have are lost for the monitoring and effecting code but not the intelligence which is still external in the gauge and controller.

Even though the monitoring code is inside the cache the intelligence of how and when to activate it and reset the data is external. This means the benefits described in “Deployment choices” are still viable even for the monitoring and affecting code.

5.2 Profiling of the cache
5.2.1 CPU profiling
It's important to note that the profiling is done on playing of database traffic and is so using the database handler directly. Most of the code of the booking engine is therefore not used.

While profiling the CPU usage of the application on 1 hour of Aergo traffic with caching of all cacheable stored procedures almost all CPU time is used accessing the database. In total 61890ms was used and 61515ms was used in database related functions. About 40% is used executing the
query and an equal amount getting the connection. Getting a connection in this context means reusing a connection from a pool. The CPU time is presumably used testing the connection before it is used. Of the 20% left 18% is used closing the connection and the rest is spread thinly among a number of functions.

The autonomic manager configured for maximum CPU impact used 31ms with a fast poll time and strict rule and querying the cache took in total 15ms. Of the 120036 calls made 58637 were cache misses, from this we can deduce that a single miss takes about 0.5ms of CPU time. This means that the existence of the framework is justified in CPU time if it can increase number of hits by approximately 60 or 0.05%.

Of all code I have added the computation of the true memory impact was the heaviest with less than 1% of the CPU time. This is not taken into consideration in the computation above since not all implementations use memory information. Although I did not see the need it is possible to further optimize this function. For instance, I create a string from a byte array to measure its length, creation of this object is the heaviest operation by far. This is not needed since the string length can be predicted directly from the byte array length.

5.2.2 Memory profiling
The maps added in the cache and used in the autonomic manager scales with the number of stored procedures. Since the number of stored procedures used is very limited (~400 on aergo and ~90 on Planitgo) these maps takes about 10kb of memory each. When the autonomic manager works it copies these maps from the cache and the memory usage is doubled at times. Since there are four different maps holding, hit, miss, call and size data the usage is maximum 80kb.

80kb should be put in relation to the memory usage of the cache which is about 10mb (lowest observed is 4mb and highest 27mb) at 5000 items. If the cache is limited in MB instead of items the difference an additional cache size of 80kb depends on the traffic and starting size of the cache. On Aergo traffic increasing cache size from 6mb to 6.08mb makes a difference of less than 0.01% in hit ratio.

5.3 Disable LRU in favour of static caching
Normally there exists more results than can be fitted in the cache and so the LRU policy will evict a result when a new is to be added. Another approach is to collect the most popular results and always keep these in the cache and entirely eliminate, called static caching. This approach was tested and compared to several replacement algorithms in caching of search results in the EXCITE web search engine [50]. On small cache sizes this approach achieved the highest hit ratio while on bigger the replacement algorithms were outperforming.

Using this approach a list over the x most popular calls would have to be compiled where x is the number of items that can be fitted in the cache. One weakness with this solution is that it by definition uses out of date information. Information collected during one period of time will be used for the coming traffic and so the set of popular queries may be different. In my tests I eliminated this factor by using a none-realistic scenario where the same traffic were replayed twice, once for crating the list of popular queries and once to test the result.
Another weakness with this approach is that the list of results to cache can be big and hence have a large memory impact. To record and store the amount of calls to each result is also memory intensive. This can be alleviated with by storing data on a bounded amount of calls, statistically a popular call should be made soon after recording is started and so be enabled for recording.

5.3.1 Result
Using this approach on one hour of Aergo American logs I achieved the following results. Hit ratio on number of items in cache with all stored procedures regarded as cacheable.

- 5000 items in cache LRU: 56%
- 5000 items in cache static caching: 45%
- 100 items in cache LRU: 40.61%
- 100 items in cache static caching: 38.99%
- 1 item in cache LRU: 10%
- 1 item in cache static caching: 1%

Using the list of only real cacheable stored procedures results in lower hit ratios overall, the reduction in hit ratio is bigger for static caching than the currently used LRU approach. In all cases using a replacement algorithm gave better performance than static caching, remember that the static caching has a handicap of using the same traffic in the learning and test phase.

It is possible to make hybrid approaches with part of the cache using a LRU policy and part having a static configuration but after these results it is unlikely that this would be an improvement.

5.4 Dynamically create sub caches
To create a subcache for a stored procedure can potentially increase or decrease overall hit ratio. A stored procedure with many different calls with a few popular calls is likely to increase overall hit ratio if sub cached. The popular calls will stay in the sub cache since they are not probable to be low on the LRU list while the unpopular calls are not contaminating the main cache since they are replacing each others in the sub cache. On the other hand it is a larger chance to have stale items in a sub cache since only one stored procedure populates it. If a sub cached stored procedure has little traffic during a period of time or if the sub cache size is too big the space occupied by the sub cache is not used effectively.

Some stored procedures may have one call used often and many calls that are seldom used, the calls and stored procedures involved may vary on different products and farms. For instance, a stored procedure with language as a parameter may have a very popular result for language US in the America while on the German farm GER is more popular and on the generic European farm no call is more popular than another.

So the question is, can dynamically creating sub caches for stored procedures depending on traffic increase hit ratio. It should be noted that this is not practical to do manually since there are too many farms and products to cover.
To solve this question I first tried finding a rule to determine what stored procedure to add for sub caching leaving the sizing of the sub cache for later. First experiment was to try with and without the sub cache used in the production configuration on a language dependant stored procedure, I found no difference in hit ratio enabling/disabling this sub cache on any traffic. Next I tried finding stored procedures with lots of traffic that had an uneven call distribution. I could find no difference in hit ratio sub caching these and so I abandoned this approach.

This does not mean that dynamically creating sub caches is not a good idea, testing with other stored procedures on other traffic can potentially make a huge difference in hit ratio. The fact that creating sub caches dynamically is not currently supported in the cache and my inability to improve hit ratio on my traffic made me turn to other approaches with better effort/gain ratio.

5.5 Replacing cached procedures with others

A constraint programming inspired approach to decide what to cache is to remove one stored procedure for caching and add another. The cached procedure with the lowest amount of hits during the period of monitoring is removed and the stored procedure with highest amount of calls is added for caching. This way the list of stored procedures is always the same size preventing the LRU to trash due to too many stored procedures on a too small cache.

Once a stored procedure is added for caching it is compared to the stored procedure removed, the one with the most hits in the given period of time is chosen as the best one and will be cached from that point onwards. A stored procedure which is started for caching but performing worse than the currently worst stored procedure will be added to a blacklist and not tried again, a comeback can be introduced to reset this list after a time.

While maximizing the number of hits is the overall goal caching the stored procedures with the largest amount of hits is not necessarily the best way to achieve this goal. A stored procedure can potentially have a large number of hits but a huge number of misses which contaminate the cache and destroys performance for other stored procedures. To address this to some extent a hit ratio limit was added, no stored procedure with a hit ratio below 10% can be added for caching if trying to replace a stored procedure with higher hit ratio.

5.5.1 Results

I also used the complete list of stored procedures as a list of cacheable procedures. The results on two disjoint sets of Aergo logs were as follows.

<table>
<thead>
<tr>
<th>Hit ratio, set 1, approximately 30 minutes of traffic</th>
<th>Cacheables=real</th>
<th>Cacheables=all</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current configuration</td>
<td>42.27</td>
<td>42</td>
</tr>
<tr>
<td>AM</td>
<td>45.79</td>
<td>53</td>
</tr>
<tr>
<td>AM-start with endconfig</td>
<td>65</td>
<td></td>
</tr>
<tr>
<td>Everything from start</td>
<td>45.97</td>
<td>79</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hit ratio, set 2, approximately 60 minutes of traffic</th>
<th>Cacheables=real</th>
<th>Cacheables=all</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current configuration</td>
<td>24.90</td>
<td>24.9</td>
</tr>
<tr>
<td>AM</td>
<td>28.33</td>
<td>38.5</td>
</tr>
</tbody>
</table>
AM means that a autonomic manager using the specified rule was running. AM-start with endconfig refers to a test where the configuration achieved at the end of the AM run was used as a static configuration from the start. Everything from start is a static configuration where the list of cached stored procedures is containing all cacheable stored procedures from start.

From the tests we can see that the autonomic manager in all cases outperform the configuration currently used in production. If all changes made by the AM are made from the start performance is further increased.

Best performance was in all tests achieved having all stored procedures cached from the start. While the AM improves hit ratio it seems another easily achievable static configuration is much better hence limiting the use of the AM.

5.6 Recommending stored procedures for caching

Some cacheable stored procedures can be collected at runtime by running database queries, as mentioned in “A list of cacheable stored procedures”. All cacheable stored procedures can not be caught with this approach since currently there is no way to know the exact list without asking humans with insight in the workings of the booking engine.

This can be tedious work since it involves phone-calls to different developers who know about the usage of different stored procedures. To provide humans with a good starting point the stored procedures are rated on how good they are for caching. A human can then select the stored procedures providing the maximum benefit to spend their time on.

Previously stored procedures were only regarded cacheable when they were known to not break the application and known to behave decent in the cache. The idea is to add all cacheable stored procedures available through querying the database and adding those a human has deemed cacheable. The decision of which cacheable stored procedures to cache is left to another rule.

The approach taken is to introduce a new small cache were store procedures can be tested. I refer to this small cache as a sandbox as not cacheable procedures can be cached in it without compromising the system. Sandboxing a stored procedure is not proper caching but a simulation, all the results must still be collected from the database even when there is a hit in the sandbox.

I have used a cache size of 10 items for the sandbox and it only caches one stored procedure at the time. While the sandbox is an internal change to the cache the intelligence of it is kept external. The autonomic manager has to enable the sandbox by telling it when and what to cache. The hit/miss information of the stored procedure is stored in the same way as the information of cached stored procedures.

The autonomic manager will pick the stored procedures with the largest amount of calls first for sandboxing as these have potentially the most hits. In the end though all stored procedures will be tried in the sandbox. Stored procedures already defined as cacheable or not cacheable or already sandboxed will not be tried in the sandbox.
Once tried the stored procedures will be ranked according hit ratio, other information as number of hits in the given time span is also presented.

To decrease the number of stored procedures human decision is needed for a check is made while getting a result from the sandbox to see if a result currently stored in the sandbox differs from the result retrieved from database. If the same database call has two different results the data in the database has changed and the call is not cacheable. If one call of a stored procedure is not cacheable in this way the whole stored procedure is regarded not cacheable. Currently there is no known case where a stored procedure has some calls cacheable and some not. By known I refer to stored procedures documented in the production configurations of Aergo and Planitgo as well as a list of cacheable stored procedures retrieved from the database team.

Unfortunately I have had no way to test the effectiveness of this test since the data on the developer database I use is not changing. It is certain that it does not cover 100% nor 0% of the stored procedures which are not cacheable and the chances of a successful classification increase with time spent sandboxing and size of the sandbox.

To actually store the result in the sandbox is not necessary. A hash value of the result byte stream or simply the size of the byte stream can be used to classify two results as different. All calls to the same stored procedure and hence all identical calls have the same column names and types in the returning result set. As such the only difference between the results is the byte stream.

Since I introduced size information it is possible to limit the sandbox in memory size instead of number of items. Doing this removes the risk of sandboxing a stored procedure having huge results. To do this introduces another problem, a stored procedure having big results can not be sandboxed since none or only a few results can be fitted in the sandbox.

The sandbox does not need to have a low limit due to memory consumption since the results does not need to be stored, only an integer describing their size. It is however desirable to have a decently small sandbox as a sandbox the size of the cache would use too much memory from just storing the call strings (~1mb for 5k items). Furthermore it does not provide an accurate rating to compare stored procedures with small call dispersion and those with high in a huge cache dedicated to the certain stored procedure.

To limit the sandbox to 200kb which is a reasonable limit would exclude 12/350 stored procedures on Aergo and 3/90 stored procedures on Planitgo traffic if the stored procedures are to fit 10 results in the sandbox. Queries belonging to these procedures makes up less than 0.2% of the total traffic, this due to the fact that the often used stored procedures all have very small results. Not sandboxing the stored procedures with large results or reporting very bad results from sandboxing these is therefore a minor problem.

Another approach using size information is to use number of items and just report the latest size of the stored procedure.
5.6.1 Result
There is no guarantee that a stored procedure with many calls is good for caching, many calls mean that a stored procedure is potentially very good or very bad. The idea is to have all liable stored procedures tested in the sandbox eventually and so the order in which they are tested is not critical.

The important question is how accurate the hit and hit ratio ranking is. There are no guarantees of the behaviour correlation of a stored procedure in a cache with the size of 10 and in a bigger cache where there is a inter stored procedure cache space competition. It is important to have sufficient data, a too short sampling of traffic and the results may be inaccurate.

The test is done on ~10 hours of Aergo traffic with a sandbox time of ~50m. First a run is made where no stored procedures are cached and they are one by one tested in the sandbox. A second run is made re-playing the same traffic, all stored procedures are cached and the sandbox is not used. A comparison can now be made between the statistics reported by the sandbox and the actual statistics achieved in the real cache.

The 4 stored procedures having the highest hit ratio in the sandbox had a hit ratio from 85% to 94%. The difference between their hit ratio during the entire run in the real cache and the sandbox hit ratio was 6.5±2. Including the top 9 (26% to 94%) stored procedures gives a value of 19±16. These are satisfactory data for a human to make decisions based on. Remember, these values are only to avoid wasting human resources by classifying stored procedures that would not be cached in any case due to bad performance.

Two of the remaining stored procedures had no traffic on them during their time of sandboxing and was defined with 0% hit ratio. This was due to an overall low traffic to these stored procedures. This is also a satisfactory result, a human would like to classify the stored procedures with the maximum amount of gain in total cache hits, this is achieved classifying stored procedures with highest amount of traffic.

The last stored procedure had plenty of traffic both during sandboxing and in the real cache and was reported to have 0.2% hit ratio in the sandbox. In the actual cache however the hit ratio was 66% which is an unacceptable difference. This stored procedure can due to call dispersion and patterned traffic not be approximated in a small cache. It should be noted that it is not unfair to rate this stored procedure low as the call dispersion and big number of unique calls have a big impact on other stored procedures in the cache.

In total all but one important stored procedure had a satisfactory hit ratio approximation.

5.6.2 Usage with external manager
I do not try to determine how the deployment of sandbox solution is supposed to be done as it depends on the specific needs. For instance, if fast decisions are in order parallel sandboxing of several different stored procedures on different nodes should be the choice.

In this case the controller will get a report of candidates for sandboxing and distribute them to the different nodes and then collect the results. Having a local solution would mean that all candidates must be tested on each node.
5.7 Removing stored procedures from caching

Observing the previous results the static configuration including all cacheable stored procedures for caching is a better starting point than the current production configuration. It should be noted that this is not necessarily true for any traffic but all traffic documented in the logs I have access to. Based on this I made a rule which assumes all cacheable stored procedures are added for caching from the start and tried to improve from there.

The idea is to remove the bad stored procedures from the list of cached stored procedures, now, what is a bad stored procedure? A stored procedure with plenty of misses have a potential large negative impact on other stored procedures since each miss will cause results to be removed. This negative impact must be weighted against the gain in hits of caching this stored procedure and so hit ratio is a good approximation of how good a stored procedure is for caching.

Trying to remove stored procedures with low hit ratio gave bad results, this as a stored procedure may have a bad hit ratio in one instant and good in another. As an example, the first times the calls of a stored procedure is made the cache can not contain these results and so exclusively misses are generated. To address this, the number of misses is used as a measurement of the amount of traffic associated with the stored procedure. A decent amount of traffic has to be made to the stored procedure before a correct decision can be taken. The rule used was, a stored procedure with less than x% in hit ratio and more than y misses is to be removed. This excludes stored procedures with a low hit ratio and low traffic but due to their sparse traffic their impact is small.

5.7.1 Result

In most of these tests I regard all stored procedures as cacheable. One of the reasons for this is that testing is faster and the usage of more stored procedures provides for more disperse and interesting traffic. Using more stored procedures lowers hit ratio of individual stored procedures as they get less space per stored procedure.

The first test was on 1h of Aergo traffic tested with different cache sizes and two different cache setups. Using a small cache size the possibility of a trashing cache increases and so the rules increase performance.

<table>
<thead>
<tr>
<th>Max items in cache</th>
<th>AM(10%,100)</th>
<th>AM(20%,50)</th>
<th>Cache all</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>56.81</td>
<td>56.57</td>
<td>56.97</td>
</tr>
<tr>
<td>10</td>
<td>40.22</td>
<td>39.04</td>
<td>40.61</td>
</tr>
<tr>
<td>1</td>
<td>23.30</td>
<td>22.23</td>
<td>23.30</td>
</tr>
<tr>
<td>1</td>
<td>11.47</td>
<td>10.47</td>
<td>11.37</td>
</tr>
</tbody>
</table>

AM(x%,y) refers to rule implemented in the autonomic management engine removing the stored procedures with less the x% hit ratio and more than y misses while “cache all” is a static configuration of caching everything.

Apparently the size of the cache has a very small impact on the rule performance and none of the rules is justified by a performance improvement.

During a run on a set of logs containing 30mins of aergo traffic the hit ratio with all stored procedures cacheable was 71.1% while a 10%, 100 rule gave 71.4%. Redoing the same test with
a static configuration not using the stored procedures removed from caching in the previous test
gave a hit ratio of 71.5%. From this it is clear that the LRU is not optimal and can be
outperformed.

Testing on during longer time, here 10 hours, gave the possibility to demand more misses before
a decision is made and so accuracy of the decision is improved. Increasing the traffic limit is a
trade off between accurate decisions and reaction speed. If the traffic changes and the limit is
high it may take a while before it is removed from caching.

5000 is the maximum number of items in the cache in all following tests. Only the known
cacheable stored procedures are cached.

~10h of Aergo American, remove when less than 10% hit ratio.
AM off: 29.04
AM on, 500 misses: 29.06
AM on, 300 misses: 29.27

~10h of Planitgo Asia, remove when less than 10% hit ratio.
AM off: 86.80
AM on, 500 misses: no diff
AM on, 300 misses: no diff

~10h Planitgo Europe generic, remove when less than 10% hit ratio.
AM off: 78.32%
AM on, 300 misses: no diff

~10h Planitgo Europe Germany, remove when less than 10% hit ratio.
AM off: 88.57
AM on, 300 misses: no diff
AM on, 100 misses: no diff

In the following tests all stored procedures were considered cacheable.

~10h Planitgo Europe Germany, remove when less than 10% hit ratio.
AM off: 95.82%
AM on, 500 misses: no diff
AM on, 300 misses: 95.83%
AM on, 100 misses: 95.85%

~10h of Aergo American, remove when less than 10% hit ratio.
AM off: 54.60%
AM on, 500 misses: 54.65%
AM on, 300 misses: 54.31%
AM on, 100 misses: 54.34%

The Planitgo traffic produce higher hit ratio and fewer stored procedures are removed from
caching by the rule. Part of the explanation for this is that 4 times the number of stored
procedures is used in Aergo and the number of unique calls is much higher (20k and 200k respectively on 10 hours).

Both on Planitgo and Aergo the rule makes little difference in total hit ratio. On Aergo using 500 misses as a traffic limit consistently improves hit ratio. Taking the configuration used at the end of such a test and starting with it improves hit ratio further hinting that on a longer test the AM would perform better.

Using the rule 10% and 500 misses would on all traffic above make a small improvement or make no difference at all. The improvement is insignificant and does not motivate the use of the rule for performance reasons.

### 5.8 Setting the size of the cache

Currently the cache is limited in size by number of items, question is how many items should be used? To size the cache is a trade off between memory and cache hit ratio.

One suggestion on how to manually [ref Tristan study] do this is setting the cacheable stored procedures to achieve a 90% theoretical hit ratio. This means sizing the cache to accommodate 90% of the most frequent calls on a sample of traffic. As recording on call level is memory intensive I have chosen another approach.

![Figure 3](image)

<table>
<thead>
<tr>
<th>Max no# Items in cache</th>
<th>Total Hit ratio %</th>
<th>Cached procedures %</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>17.59599</td>
<td>54.23988</td>
</tr>
<tr>
<td>50</td>
<td>21.38877</td>
<td>66.00662</td>
</tr>
</tbody>
</table>
As seen in Figure 3 the hit ratio grows quickly to begin with and grows slower approaching the maximal hit ratio at 20,000 items. The sample traffic have less than 20k unique calls to the known cacheable stored procedures on the sample Aergo and Planitgo traffic used hence the maximal possible hit ratio is reached at this cache size.

One scenario is that the person trying to configure the autonomic manager knows what range or the desired granularity in determination of the size. A step size can then be provided and some constant describing the desired memory/hit ratio trade off. Me knowing that the current configuration is at 5000 items use an item step of 1000 items. I define the trade off constant to be 3%, when the hit ratio gain to step up is less than 3% no size increase is performed.

One way to implement this is to explore some points and find the lowest point that has less than 3% difference to the largest point. If no such point is discovered the exploration continues testing bigger and bigger cache sizes. To refresh the data on long-term changing traffic, aging is used.

Trying to implement this proved difficult because of the natural variations in the traffic. Here is a test where no configuration change is made and the hit ratio is studied over time.

We can see that the main cache hit ratio varies between 74.6% and 92.0% with a difference of 17.4. The standard deviation excluding the first point reserved for cache warm up is 4. The standard deviation for the other points in Table 1 is similar (3.8 to 6.2), same goes for maximal difference. These values should be put in relation to differences between the item steps, the difference between 1k and 2k items in cache is ~1.5% in hit ratio.
Trying to extend the period of sampling or taking an average of several points did not alleviate the situation. Comparing the first and last half of the traffic (excluding the start up time when the cache was not full) resulted in a difference of 5%. Trying to use this approach with an initial exploring to 5k items and maintaining an average of the hit ratio achieved at each point gave the following result.

![Graph showing hit ratio changes over time.](image)

**Figure 4**

Looking at Table 1 we can see that 1k or 2k items are the correct values depending on how far the exploration is made. This sizing clearly does not work as it hops wildly and tends to stay at 4k items. Total hit ratio varies more than the main cache hit ratio as seen in the graph and is therefore not a candidate for use.

Another scenario is that the item step is unknown. As the difference is relatively much higher between steps on small cache sizes it makes sense to exploit this by starting with small steps which gets increasingly bigger. A start of 10 items and increasing this 10 times on each having and having the constant slightly increased to accommodate for higher step granularity is still very unsafe. Taking a mean of 3 steps however and the standard deviation decrease to around 2.6. Looking at Table 1 on the steps 10, 100, 1000 and 10k we can now see that the correct decision of 1k would be safely taken. This approach however is crude since the granularity in the target range is poor and it takes about 9h of traffic before a decision can be made.

Testing on ~10h of Planitgo German traffic gave much smaller variations in hit ratio independent of cache size.

<table>
<thead>
<tr>
<th>Max no# Items in cache</th>
<th>Cached procedures %</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 2

The largest difference in hit ratio with a static configuration observed over time is 3% and the standard deviation is ~0.6 depending on size of the cache. Under these conditions decisions can be made with much better accuracy. Reusing the rule choosing the lowest size having less than 3% difference to the biggest known point provides stable results. 1k and 2k items in cache are correctly chosen depending on initial exploration.

Modifying the rule to avoid the need of exploration by taking the lowest of the first two points with less than 3% difference in hit ratio worked well as well. 1k items was chosen in the test run which is wrong however slightly, the difference between 1k and 2k items is 2.7. Using three times the data (3h) on each step alleviated this and reduced maximal observed difference to 1%.

Trying an approach without set step size using 10 as initial size and multiplying by 3 at each step correctly chooses 2430. The likelihood for a mistake is the biggest between 840 and 2430 items and it is less than 5% in this particular run (~91.2±0.6 vs ~94.8±0.6).

5.9 Removing stored procedures from caching with memory

Using memory information it is possible to optimize the cache both in memory usage and hit ratio. Stored procedures can be rated on (time saved)/(memory consumed). The amount of memory consumed is the number of results the stored procedure has in the cache times the size of these results.

Measuring the number of results a stored procedure has in the cache is difficult as the stored procedure name is not stored in the result. The result must be extended to contain the stored procedure name or the call string parsed when a result is removed from the cache. Parsing the call string provide unmanageable code since the call strings differs in format. I chose to use number of misses as an approximation of the number of results a stored procedure has in the cache.

Due to the timing problems described in “Using time information” all calls are defined to take the same time and so the saved time equals the number of hits. The formula in the end is scaling*hits/(misses*size).

Now that the rating is done the question is how many stored procedures to remove. The solution used is a threshold constant to define the stored procedures to be removed. While this may not seem ideal I have found no way to automatically determine this constant or eliminate it altogether.

As the autonomic manager will be taking decisions on events that have happened I have divided the test. For instance, if a stored procedure takes 100mb and is removed from caching the
maximal memory used by the cache is still 100mb. First a ~1h of Aergo traffic is used to let the autonomic manager make a configuration, all statistics is then reset and two different test sets of ~40min of traffic is used to evaluate the configuration. The two test sets is from another day, one from midday and one from night. It’s important to note that the constants used is not data mined to fit this traffic but is from the results on other traffic used in “Limiting the cache in memory size”.

Testing this on a cache running on 2k items and having all stored procedures as cacheable provides the following result.

<table>
<thead>
<tr>
<th></th>
<th>Max size (bytes)</th>
<th>Mean size (bytes)</th>
<th>Hit ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start run</td>
<td>21 780691</td>
<td>4 441135</td>
<td>53.92%</td>
</tr>
<tr>
<td>Start run c =1</td>
<td>21 613227</td>
<td>4 344979</td>
<td>55.00%</td>
</tr>
<tr>
<td>Start run c =0.5</td>
<td>21 606532</td>
<td>4 332539</td>
<td>54.30%</td>
</tr>
<tr>
<td>Test set 1, no am</td>
<td>21 667 597</td>
<td>6 066246</td>
<td>70.12%</td>
</tr>
<tr>
<td>Test set 1, c =1</td>
<td>4 448 655</td>
<td>3 561084</td>
<td>68.72%</td>
</tr>
<tr>
<td>Test set 1, c =0.5</td>
<td>3 485653</td>
<td>3 896513</td>
<td>68.37%</td>
</tr>
<tr>
<td>Test set 2, no am</td>
<td>5 393 652</td>
<td>3 785798</td>
<td>52.92%</td>
</tr>
<tr>
<td>Test set 2, c =1</td>
<td>4 173 690</td>
<td>3 729528</td>
<td>54.40%</td>
</tr>
<tr>
<td>Test set 2, c=0.5</td>
<td>5 243317</td>
<td>3 668633</td>
<td>53.68%</td>
</tr>
</tbody>
</table>

In the start run the size is as expected similar with and without the AM but hit ratio was increased. Using test set 1 there was a big difference in used memory, peak memory usage was decreased by 76% and average by 40%. The big difference is mainly due to a rarely used stored procedure “plhotelpropertyindex” consistently having results bigger than 15mb. Hit ratio however was also decreased.

“plhotelpropertyindex” was not used in test set 2 but memory usage was still decreased and hit ratio was increased. Overall these results are satisfactory for both constants. Using these constants on Planitgo traffic results in no configuration changes by the AM. Same goes for using the currently used configuration of caching only the known cacheable stored procedures.

Lowering the constant further works good to on long test runs due to the usage of number of misses in the formula. Decreasing the constant means a stored procedure has to have more misses before it is removed which is synonymous to more traffic. Using misses in this way is similar to the use of misses as a traffic requirement used in “Removing stored procedures from caching”. As mentioned demanding more traffic means more accurate results but slower reaction time. It is somewhat inconsistent with the original thought of the use of misses but works well as stored procedures with size in the mid-range will get a somewhat accurate hit/miss ratio before decision is made and stored procedures with huge results will be removed at the first miss.

Overall I find this rule usable and the constant should be put lower than my examples to ensure correct decisions. If implemented now it would make no difference but as traffic and database content is changed it works as a safeguard removing any stored procedure that is terrible for caching.
5.10 Limiting the cache in memory size

Having memory information available inspired me to make an internal change in the cache. When sizing the cache the administrator/AM has to choose number of items in the cache to weight memory usage against performance. This is a bit weird as he does not know the memory usage on different sizes of the cache, better than to choose a maximal memory size.

While the internal change in the cache to limit it in memory is a change that is outside the scope of this thesis it works as a change proposal to Amadeus as well as a “what if” scenario in my work. It also simplifies testing and evaluation as I do not need to take memory and hit ratio in account but can focus on hit ratio only.

5.10.1 Results

~10h Aerogo American traffic

<table>
<thead>
<tr>
<th>Size(mb)</th>
<th>Const.</th>
<th>On remove</th>
<th>AM</th>
<th>No AM</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>10</td>
<td>let be</td>
<td>50.6%</td>
<td>57.765%</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>let be</td>
<td>53.0%</td>
<td>57.765%</td>
</tr>
<tr>
<td>6</td>
<td>0.5</td>
<td>let be</td>
<td>58.0%</td>
<td>57.765%</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>let be</td>
<td>40.17%</td>
<td>55.13%</td>
</tr>
<tr>
<td>3</td>
<td>0.5</td>
<td>let be</td>
<td>55.56%</td>
<td>55.13%</td>
</tr>
</tbody>
</table>

~8h PlanitGo Asia

<table>
<thead>
<tr>
<th>Size(mb)</th>
<th>Const.</th>
<th>On remove</th>
<th>AM</th>
<th>No AM</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>10</td>
<td>let be</td>
<td>95,2%</td>
<td>95,17%</td>
</tr>
</tbody>
</table>

~8h PlanitGo Europe Germany

<table>
<thead>
<tr>
<th>Size(mb)</th>
<th>Const.</th>
<th>On remove</th>
<th>AM</th>
<th>No AM</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>15</td>
<td>let be</td>
<td>85.32%</td>
<td>89%</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>let be</td>
<td>93,2%, 90.5%, 87%</td>
<td>89%</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>no fetch</td>
<td>86%, 85%, 78%</td>
<td>89%</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>let be</td>
<td>86.7%, 90.6%</td>
<td>89%</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>let be</td>
<td>89.3%</td>
<td>82.01%</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>let be</td>
<td>90.7%</td>
<td>82.01%</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>clear</td>
<td>42.3%</td>
<td>82.01%</td>
</tr>
</tbody>
</table>

The “on remove” refers to the policy used when removing stored procedures from caching. When a stored procedure is removed it may still have results present in the cache. This is an issue in previous solutions as well but due to test setup and the memory limit the impact is bigger in these tests. The policy let be which I have used in the other solutions is to simply leave the results in the cache, most probably the stored procedure removed has few hits and the results will be evicted by the LRU soon. In the “no fetch” policy results belonging to not cached stored procedures are never fetched from the cache. This speeds up the LRU eviction from the cache. Clearing the cache after a batch of stored procedures is removed is referred to as “clear”. Of these the “let be” had the best performance and is the one implemented in the cache at present.

To go through the cache and remove all results belonging to certain stored procedures is only of academic interest since it scales poorly with bigger cache sizes and requires parsing of the call strings in the cache. This policy performed worse than let be on the best constant for the let be policy, not necessarily meaning that it always performs worse.
The constant 0.5 which gives a performance increase on the Aergo traffic results in no changes on the Planitgo traffic. On Planitgo constants 5 or 10 occasionally gives very good results, noteworthy is for instance how a cache of 3mb can outperform a cache of 6mb. The constants however must be tested and tailored manually to the specific farm and doing this is to approach the effort of manual configuration.

If the cache size and average result size drastically change the constant may perform bad removing too much or too little from caching. This can be alleviated making the constant a function of the cache size.

6 Analysis

6.1 External adaptation applied to the solutions

I described several general advantages in Motivation for external adaptation but how do they apply on my solutions? As mentioned in “Changes in source code and external adaptation” some of the advantages are lost as I change the source code to allow for adaptation of the cache while others are still viable.

As the traffic on different nodes in the cluster can be seen as the same the advantages described in “Deployment choices” becomes important. This is a luxury not always present, other data such as available memory of a node is unique and may have big variations between nodes in the same cluster.

I do not try to determine how the deployment of the solutions is supposed to be done as it depends on the specific needs. I have described several solutions on how to determine what to cache on a stored procedure level. All these solutions have in common that they can use all the deployment options in “Deployment choices”. This does work well for sizing the cache and recommending stored procedure for caching as well.

As an example, let’s look into how the recommendations may be done. If fast decisions are in order, parallel sandboxing of several different stored procedures on different nodes should be the choice. In this case the controller will get a report of candidates for sandboxing and distribute them to the different nodes and then collect the results. Having a local solution would mean that all candidates must be tested on each node.

Another alternative is to perform the sandboxing on one node only in a sequential manner. This sacrifices the time it takes to detect stored procedures that should be cached but saves memory and CPU as only one node need to be running the sandbox.

6.2 Utility of the solutions

6.2.1 Finding cacheable stored procedures

The sandbox is a useful tool to avoid making mistakes where stored procedures providing lots of hits are not correctly classified as cacheable. While providing additional functionality to the cache it also limits the amount of work a human need to make as there is no need classifying
stored procedures that perform badly in the cache or are detected as definitely not cacheable. The solution is not perfect as some stored procedures may be have large errors in simulation and actual performance, trading for memory this can be eliminated with a larger sandbox.

In addition to the sandbox there is the possibility to at runtime run queries to the database collecting stored procedures of certain types that are known cacheable. Automatically adding these stored procedures is a simple matter requiring no changes to the cache.

Overall, implementing these changes would decrease the need of manual administration, provide additional functionality and result in a performance increase. One example of this performance increase can be seen in the results of “Replacing cached procedures with others”. Using the query to extend the list of cached stored procedures systematically increase performance compared to the current configuration.

Extending the list of cacheable stored procedures may not always increase performance which I addressed with my solutions trying to remove stored procedures from caching.

6.2.2 Determining what to cache
As described before, the list of cacheable stored procedures can be collected and updated at runtime. This means that no human has evaluated the performance of the stored procedure.

Automating like this means that any stored procedure regardless of performance in the cache can be added for caching. Even if evaluation is done manually or in the sandbox before a stored procedure is added the evaluation can not be made with all future traffic in mind.

The addition to the list of cacheable stored procedures, traffic changes, deployment of a new product/version or start of a new farm are all possible causes for the start configuration to contain stored procedures that causes the performance of the cache to degenerate.

I have described several different solutions where stored procedures are removed from caching and one where a stored procedure was swapped out of the cached list in exchange for another. These all address this problem under different conditions such as with a memory limit and without.

A hypothetical scenario of trashing cache without limiting the cache in memory:
A new Planitgo version is deployed on a new farm in Russia and because of the patterns of the customers using this farm a previously unused stored procedure getBookingTimestamp is used. This stored procedure used on a cache of maximum of 100 items is called 100 times between every other query and each call is unique. Caching this would immediately cause the cache to reach a 0% hit ratio but running the above rule this stored procedure would be automatically removed due to low hit ratio.

While this is an extreme case stored procedures has had to be manually removed in the past to increase cache performance. The scenario with a cache limited in memory is similar, a stored procedure having results the exact size of the cache effectively empties the cache of other results and can cause serious performance degradation. To address changing traffic such as
getBookingTimestamp being called with the same parameter all the time a timed comeback can be used.

The solution “Replacing cached procedures with others” has worse performance under the traffic similar to the current but address another issue. If the list of cacheable stored procedures is huge we might not want to store this entire list as the list of cached stored procedures on every node. Limiting the size of this list we can keep the full list in the controller on one node only. This list is far from huge now (~200) so the usefulness of this approach is limited.

While performance can be increased on the current traffic if all stored procedures are regarded cacheable the improvement comes with different constants on different traffic. This is especially evident in the results of “Limiting the cache in memory size”. Maintaining different constants on different farms/products is too much manual work so my proposal in these situations is to set the constant very low, as a safeguard to remove any future stored procedures that are terrible for caching.

The great results achieved at times are as stated done with all stored procedures regarded cacheable which is not the actual case. I provided some motivated for this configuration in testing in “A list of cacheable stored procedures”.

The results achieved using the configuration proves that a constant can be used on this traffic to increase performance. We can also see how a low constant in worst case gives no changes to the configuration. Using only the known cacheable stored procedures no significant performance increase was achieved.

6.2.3 Setting the size of the cache
The varying hit ratio on a static configuration was the main issue with my approach on this topic.

Seeing how the probability for a mistake as well as the standard deviation can be calculated at runtime from data gathering. The granularity of the steps as well as time for data collection can be varied. Calculating the standard deviation can also be used to discover that proper sizing of the cache is impossible on the Aergo traffic. The main issue is that this requires many hours of data collection on the same size of the cache to calculate a proper mean and standard deviation.

With the good precision achieved on the Planitgo traffic data could be used to define a function to interpolate to points not discovered. Although I have not found such a function, as can be noted in Figure 3 a logarithmic function fails. Interpolating between two points may also be possible.

As a sum up, sizing the cache on Aergo is not possible to do in any satisfactory way but it is at least detectable by gathering of data at runtime. Sizing the cache on Planitgo we have several options depending on desired granularity and what is known to the person configuring the rule. I do not recommend trying this approach for sizing the cache as it would in practise only be usable on some farms.
6.3 Conclusion

One of the thesis purposes was to evaluate the external approach applied to the cache. Due to the clustered deployment and possibilities in deployment and global adaptation I find external adaptation to be very suitable. The grouping of queries in stored procedures and the possibility to merge or use node information independently adds to this.

In general I do not think it worth the effort to implement the described solutions for optimizing the cache by determining what to cache. The cache behaves well under current traffic and since there is no problem this is a low priority. I do recommend implementing the sandbox as it is easy to implement and adds extra value to the application. Automatic fetching of definitely cacheable stored procedures from the database should not be implemented if no solution trying to determine what to cache is implemented. I do not think that the database changes often enough to motivate introduction of both these rules.

In large the required manual administration of the system was not decreased with a large extent and performance is under variations of current traffic is good without the need of adaptation. With variations I refer to traffic from different farms/products and traffic varied by changing the list of cacheable stored procedures.

6.4 Future Work

6.4.1 Cache

As observed hit ratio and main cache hit ratio varies over time. If hit ratio was constant on unchanged configuration changes such as resizing the cache and removal of stored procedures could be evaluated. For instance, if total number of hits increase when a stored procedure is removed from caching a prediction can be made that this is also true in the future and the hassle a constant introduce can be avoided.

One way to solve this is to have the same traffic ran twice, number of hits/hit ratio is then guaranteed to be the same. In this evaluation phase the actual result is not interesting so database access and storage of the results can be skipped. Doing this requires recording of all calls made and their order which is about 60k/1h. The memory impact of such a recording is unacceptable. Another approach is to make the simulation in parallel to the real time traffic. This only requires storage of n call strings with length ~30 where n is the size of the cache. With a cache size of 5k items the call strings and an integer for storing size information the map used for simulating the cache takes about 700kb. This does not include all overhead such as the linked list a LRU needs. As the cache is doing fine on 5mb on some traffic/configuration I feel this is a bit much. If the average sizes of the results grow bigger the relative cost of keeping the call strings lowers and this would be well worth exploring.

There are approximations for caching using a markov model [51] but this reduces CPU usage more than memory and the maximal difference between the estimate and cache simulation is quite large at 12%.
Another interesting approach is described in [52]. It requires computation and storage of the popularity of a query. While this depending on implementation may reduce memory usage the precision is unknown.

Instead of one cache simulation with the size of the current cache two small caches can be made, both simulating the same traffic. While memory usage is reduced precision suffers, a change might be good on a large cache but bad on small. Earlier experiments with different cache sizes hint that the performance difference of removal of stored procedures is largely independent from the cache size. This is in my opinion decently easy to implement and is the most probable to work.

Prefetching is done in many caching environments such as caching of memory pages on disk. Since the overhead is large for making a disk access but small for reading subsequent blocks after a disc access is made this saves time. Prefetching can be used to make additional database calls when a connection is fetched and tested to reduce the overhead time of getting a connection. The AM would observe traffic patterns and inject a call list to the database handler, when a call is made that has a set of highly probable subsequent calls these would be made at the same time and the results cached.

The time between activation of the gauge has to be set explicitly right now. While I don’t see this as a big problem the impact can be reduced. For instance, the traffic can be described by a pattern and the gauge woken only when this pattern changes. How to make this pattern and the accuracy of this approach is to be explored.

Currently all decisions are made solely on stored procedure level and not on call level, mainly due to the memory problems recording on call level introduce. Approximate recordings can be made such as recording only the x first calls to be made or cycle the recording so that it in time covers all calls. Instead of disabling caching of all calls to a stored procedure certain calls can then be disabled. I see small or no performance gain doing this on the current traffic as most store have the calls evenly distributed.

A timer with better accuracy such as the one introduced in java 1.5 addresses one important problem of timing. Decisions based on correct time data could improve performance.

An alternate cache sizing method is to size it depending on available memory, there is no point in having unused memory. On this platform this is impossible due to the garbage collection in Java, used heap space increases till there is a garbage collection and heap space is freed.

7 References


8.1.1 Memory calculations

Getting the size of the cache in real memory was not a trivial matter. In languages c++ the developer may use a sizeof operator to get the size of a data type and is so able to allocate enough
storage space for it. In java allocation is handled by the JVM and so the need for a sizeof is smaller.

In java 5(1.5) the interface java.lang.instrument.Instrumentation with a method getObjectSize is introduced. Although this method returns an implementation and environment specific approximate object size it would have been a welcome addition for me. This implementation is JVM specific since it depends on, among other things, number of header words in an object, pointer size and word alignment.

Others identified this need pre java 5 as well [53, 54] and made sizeof operators. These are unusable in an application of the size of the booking engine for several reasons: forcing java garbage collection is a huge resource sink and the high grade of concurrency makes checking heap size before and after allocation of an object highly inaccurate.

Since there’s no provided way of knowing the size of the cache short of using a profiler or profiler like algorithm which is out of the question for performance reasons I made a approximation of my own.

Arriving data from the database is stored in a ResultSet which is wrapped into an Amadeus Result object. The ResultSet is a table with rows and columns, a set with no rows is considered empty but still consumes memory as the column names etc need to be stored. In the Result object several arrays are added and a map used to store result table column information. A result with little data from database may have any amounts of columns but I make the assumption that this is correlated.

The result is then wrapped in a number of objects before arriving at the top level, the cache, which is the target for size measuring.

Here are a few examples of sizes from calls to different stored procedures.

PSGETSITELANGUAGEPROVIDERTRAINTYPE
First level wrapper object 872 bytes
Result 456 bytes
Byte stream length 0

PLSITEWEBFARESRULESETTINGS
First level wrapper object 1848 bytes
Result 1520 bytes
Byte stream length 0

PSGETCOUNTRYNAMEFORHOTELSEARCH
First level wrapper object 31 568 bytes
Result 31 152 bytes
Byte stream length 2917 bytes

The raw byte stream contains the actual information that goes into the rows of the result set. It is clear from the results that an empty result can have different sizes, this depends on column data.
For instance, a result set with a hundred columns with 100 character column names will take more space than a single column with a one character column name if both contain no rows. Looking at the last example we can give some credit to the assumption that more data means bigger result. The size difference between the wrapper object and result object is almost constant, this holds true throughout the object hierarchy.

8.1.2 Measuring the size

To determine the constant overhead of all wrapping objects a test with results missing columns and rows was made. The test was done on several different settings of maxitems, this to ensure that the cache grows linearly with the number of items in it.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure5}
\caption{Overhead of all wrappers in the cache can from this graph be approximated to 940*items in cache.}
\end{figure}

Now I want to find the function:

\[ y = x \cdot k_1 + \text{items} \cdot k_2 \]  

\textbf{Equation 1}

Where \( y \) is the memory consumption of the cache and \( x \) is the summed byte stream sizes of all results. \( k_2 \) should be 940 from the graph above.

The first approach was to simply measure the length of the byte array coming from the database to have a value for \( x \) and data mine \( k_1 \). Only the first result row is processed and the size of that row is multiplied by the number of rows to get the total size.
A cache of 5k and 1k items were used to get data to determine the function in Figure 6. From Figure 6 the function fitting the data best is $k_1 = 7.4067$ and $k_2 = 1387.4$.

In Figure 7 this function is displayed as a light dashed line on a cache of 2k items for validation while the solid line is fitting the validation data only. The biggest difference between the validation data points and the data mined function in the graph is 1.3mb or 17% of the real cache size. On the data used for determining the function in Figure 6 the biggest difference is smaller with 1.2mb or 10% of the real cache size. While the differences are not huge an effort to improve them is made.
A more advanced approach was needed and so I also looked into the data types of the data received from the database. There are two differences from the previous approach, first all the rows are processed and second the column types are used to calculate a more precise approximation of used memory. An example of the difference in size when types are used, the previous approach would detect a stream of 1 byte for a character while on my platform a character stored in a string object actually takes 40 bytes [53].

Inspecting the JDBC type of every value as they are processed a total result size is approximated with formulas. For instance the string size is calculated as $38 + 2 + 2 \times \text{string length}$ [53] and an Integer is fixed at 16 bytes. It may seem strange that an Integer object is 16 bytes, the calculation is as follows [54], 8 byte for empty Object, 4 bytes for a primitive integer which makes 12 bytes, my standard 1.4 sun JVM then aligns in 8 byte words so it takes 16 bytes in total.

Using this approach the results were much better. Similar to the previous approach a cache of 5k and 1k items were used to determine the linear function which is validated on a cache of 2k items using different traffic.
5000 items

\[ y = 1.1512x + 6E+06 \]

1000 items

\[ y = 1.083x + 1E+06 \]
The dark line is a trend line with corresponding function displayed, it linearly fits the data in respective graph and the light dashed line is the linear function fitting the 1k and 5k cache data. Looking at the graphs of Figure 8 the function fits the original data well with a maximal difference of 5% or 120kb, the largest difference on validation data is 8% or 300kb. This function was calculated as in the previous approach and the result was: \( k_1 = 1.1258 \) and \( k_2 = 1337,1 \).

Noteworthy is that \( k_2 \) from Equation 1 calculated to 940 does not match 1337 in this formula. The difference is due to overhead in column name and type storage which is not directly related to result data.

The results from this approach are satisfactory but more CPU demanding than the previous result. About 1% CPU time is used when replaying database traffic (80% is used in db access). The results can be further improved by additional data for function determination. It is also possible to extend the memory calculations to the strings storing column names and types.