Hot-swapping in Self Healing Environment

Timi Tuohenmaa

***Abstract*—This paper introduces hot-swapping in different situations to implement self healing environment. It starts from the first techniques with hardware and inroduces user-level software hot-swapping. Last we show what is important when hot-swapping is implemented to operating system and present example implementantion of system-level hot-swapping for Linux operating system.**

***Index Terms*—Hardware, hot-swapping, self healing, software**

I. Introduction

KEEPING the system availability on maximum level important and in some cases crucial (like in nuclear plants). Still it is impossible to make unbreakable hardware and in practice completelly faultless software (some small programs can be proved to be faultless with mathematic models). To solve this problem there are some solutions and we introduce hot-swapping on hardware and software side.

On hardware side hot-swapping has been used in the late 1970's in high-end server systems [1]. It has been possible to replace a malfunctioning processor or memory unit without shutting down the system by using multiple processors and isolating the miscalculating processor or memory from the system and informing the end-user that the broken part needs to be replaced. Nowadays hardware hot-swapping is available in some ways in most server systems and in some ways (such as hard disks) in some workstations.

Software level hot-swapping is mentioned in some form in early 1980's [2] as dynamic modification of running program. Lots of research has been done, but still hot-swapping software components is not a basic tool in programming. Most commonly used operating systems and programming tools are lacking hot-swapping feature. Yet a lot of work has been done and most common programming languages like C, C++ and Java have technology reviews that show that we are close to the breakthrough.

Hot-swapping with software can be used to handle a wider scale of problems than hardware [3]. Still the primary advantage with hot-swapping is that it reduces or removes the downtime of the system. There are different ways to implement hot-swapping on user-level programs and some of them are introduced in this paper. As system-level hot-swapping is another area it is introduced separately and an example for implementing hot-swapping in Linux kernel is shown [4].

Manuscript received March 9, 2007.

Timi Tuohenmaa is with Department of Computer Science, Helsinki University, 00014 Helsingin Yliopisto FINLAND (corresponding author to provide e-mail: timi.tuohenmaa@helsinki.fi).

II. Hot-swapping with hardware

Replacement of the hardware component without halting the system is known as hot-swapping. Nowadays most components in computer systems have the ability to be replaced without halting the system. Workstations are not usually built in way they support hot-swapping (replacing keyboard or display etc. is not considered as hot-swapping), but server systems have ways to replace components online. It is up to the end-user how expensive computer he is ready to buy.

Replacing power supply is the easiest hot-swappable device since it is always in one state and with two power supplies components keep having electricity all the time even when one power supply is removed. Most commonly computers have hot-swappable hard drives [5]. These are called Redundant Arrays of Inexpensive Disks (RAID). See section II.A. Replacement of processor unit is also possible [1] as shown in section II.B. Many other components can be replaced too, but are not covered here [1].

A. RAID

Normally hard drives contains the critical information of the system. Therefore it is very important to keep critical information safe by either using backup or keeping data safe realtime by mirroring data to multiple hard drives [5]. Whereas mirroring is often the primary reason for RAID there is more relevant reason for self healing and hot-swapping. Since data is mirrored to multiple hard drives it is possible to replace one malfunctioning drive to another without interrupting the running system. Either hardware RAID controller or system-level software relieve the broken hard drive and when new functioning hard drive is replaced the controller will mirror the data to that drive.

B. Replacing processor unit

Professor D. Siewiorek [1] introduced computer by Stratus that had replaceable processor boards. Computer two processor boards each having two processors. Both processors in single board operates the same operations and compares results. If result differs it removes itself from operation and asks operating system to do a diagnostic. If operating system determines an error, it alarms the user to change the broken board. At the same time the other processor board works normally. This way system does not fail unless the both boards fails at the same time.

III. Software based hot-swapping

As for the hardware, hot-swapping can be made for software components too. In practise this is replacing parts of the software with newer code without shutting down the program. To strictly follow the requirements of hot-swapping, the program or its updated component must be exactly in the same state as it was before update. Hardware hot-swapping is used to replace broken (or soon-to-be broken) item to new, but hot-swapping software can be used in the wider sphere as will be represented in section III.A.

A. Applications

With hot-swapping broken components can be replaced without decrease of the system availability [3]. This makes it possible to fix bugs and security issues much faster than before since system does not need to be down at all and updates can be done even during the heavy traffic.

Some problems can be solved with many different algorithms. These work better (or worse) in different situations. The algorithms can be implemented as different components. These alternate components can be changed by special motitoring component that replaces algorithm when it notices poor performance [3].

Monitoring and logging is usually not needed and causes overhead for the operation that component really does. There could be two versions of components where first one does monitoring and logging in pursuance of it's primary work and second version does not [3]. This way there extra operations can be activated only when required whereas normally component achieves it's maximum effectiveness.

Usually components need to be able to manage all special cases to prevent unexpected failures. With hot-swapping and there can be one version of component that handles the usual cases and another component that handles all cases [3]. While all tasks are first given to simple and fast common-case version, it can fall back to the full version when it fails to handle special cases. This way the normal operations are substantially faster and rare special cases slower. Overally the performance improvers.

IV. User-level hot-swapping

In most user-level programs, like spreadsheet or WWW-browser, it is not important to be able to update software without restarting the program. It is easy to save current work and quit the program and update. But in the server environment it is often preferred to diminish downtime or even cut out the downtime completely. There can be hundreds of users using the server at the same time as the update needs to be done. Some might be in critical state paying their bills and others not, but server admistrator can not shut the system down when he is ready. In some cases these things can be done safely at night but in global world night time is not same for everyone.

Software hot-swapping is also referred as dynamic software updating [6], but both does mean the same, changing parts of the program (or the whole program) to newer version without interruption.

Software update can be made by using multiple computers and updating them one at the time [6], but it is expensive as it requires multiple server computers and other hardware to direct connections to right computer. For a more cost-effective way, software needs to be updateable without extra hardware.

There are different ways to gain the hot-swapping feature for user-level programs. Some of them are introduced in next sections.

A. Virtual machines

Virtual machines does not do any hot-swapping by default, but since they can be implemented to do things that underlying system can not, they can implement requirements for hot-swapping.

As example Java virtual machine does implement dynamic class loading, but not replacing already loaded class. S. Malabarba et al. [7] made implementation for Java virtual machine for dynamic classes. Their virtual machine allows hot-swapping class implementation with another. Current instances of hot-swapped class are replaced by copying object variable data from old to new object (implementation requires that all old variables exist in new implementation). This implementation also requires that object code is not used while hot-swapping is made. Still it is full hot-swapping able virtual machine.

B. Native programs

Native programs have the speed benefit over the virtual machines. Program is compiled for selected native instruction set unlike the virtual machines where program is first handled by virtual machine to native form.

Hicks et al. [6] implemented hot-swapping (which they call dynamic updating) that allows parts of the program to be patched at any time. Their approach was to make flexible, robust, easy to use and low overhead system.

To reach *flexibility* their solution allowed code updating to happen any time(also when code is in active use) and language to be C-like.

For *robustness* they used Typed Assembly Language (TAL) [8] which requires programs and patches to be verifiable native code.

old version f':

```
static int num = 0;
int f(int a, int b) {
  num++;
  return a + b;
}
```

new version f':

```
static int num = 0;
int f(int a, int b) {
  num++;
  return a * b;
}
```

state transformer S

```
void S ()
  f'::num = f::num;
```

Figure 1: Transformation [6]

Patch generation from source is automated to make system *easy to use*. Patch information contains the old and new code and also a transformer function that informs dynamic updater how to convert old code to new as can be seen in figure 1.

Low overhead comes with native code even with slight slowdown from dynamic linking.

Implementation shows that complete hot-swapping can be done with native code without special support from operating system.

C.Other solutions

To achieve some level of hot-swapping I suggest a few simple techniques. First user can use lightweight scripting language on top of the server system. Updating functions programmed by scripts is as simple as changing function to be used. This does not offer replacing objects that contain data, but is sufficient in some places.

On distributed systems like CORBA and DCOM client programs can be done so that they make a new connection if connection is lost. This allows services to be updated in traditional way by restarting them and possibly saving current state to disk. In many cases clients are able to wait couple of seconds for restarting service.

Simplest option is to place service to an other program and communication can then go throught sockets or shared memory etc. As updated component is then a separate program it can be restarted while main program can then reconnect and continue. Whereas this is not hot-swapping in strict manner, it still allows updating components while the main system keeps running.

V.Hot-swapping system software

Updating system components by hot-swapping is basically same thing as with user-level software. Options for implementantion are more limited since virtual machines etc. are not possible since kernel must be native for the underlying instruction set. Mistakes in hot-swapping implementation or hot-swapped code can and often will crash the whole system unlike with user-level software. One issue is also current common operating systems. None of them currently support hot-swapping components (though there are implementation for Linux shown in chapter VI). New operating system is rarely an option when a fairly small new feature is needed.

A.Pros and cons

Main reason for implementing hot-swapping to kernel is the same as with user-level software. Availability must be maximized, but still fixes should be applied as soon as they are available. Security issues need to be fixed before they are commonly known and other bug fixes before they cause crashes or corruption. These are so important that usually system reboot is concidered as smaller issue than keeping system available.

Adding new features or optimizing current components is also good feature though not as critical as adding fixes. Still servers on high load might need optimizations for example with network card drivers or file system drivers. If load is always high there is never time for update if hot-swapping is not possible.

On the other hand hot-swappable components are open for problems since keeping the component state while replacing it is much more complicated than just restarting it and initializing from start.

Implementation for existing system is also difficult and can cause surprising problems on anywhere in the kernel as it was not originally designed to have hot-swapping feature.

B. Way for implementation

Developers of K42 operating system [3] have described four requirements to make hot-swapping possible with operating systems. System needs to be able know component boundaries, the position of code and data. Component has to be in the safe state, where no other component is either using it or atleast not modifying it. Internal state of the component must be transferable to updated component. Also the external references to component must be possible to be updated through the whole system.

First requirement of *component boundaries* mainly requires good and disciplined design [3]. Component must be designed in way that its code clearly in one place and data are in one place without external state information. Using object-oriented programming language helps making component a logical packet, but is not required and often not even possible since often kernels in operating systems are written in C.

Quiescent state means state where component is not actively in operation [3]. State where no other component is currently operating with hot-swapped component. This state is required to make sure that component state is not changed during hot-swapping operation. Operations must be made as short as possible so that quiescent state would be available

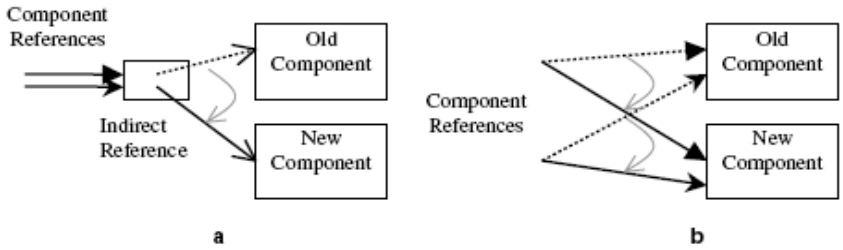

Figure 2: Options to manage external references [4]

more often. System monitors when such state is available [4].

Component state transfer requires that component designer makes an interface that is universal to hot-swapping components [3]. It is safe to assume that developer who creates the new component knows details about old version. This way new component gets old component's reference and reads all the required information directly from there.

Updating *external references* can be made in two different ways (see figure 2), both having good and bad sides. Indirect reference is easy to implement to new system [3]. It is also very simple to update since there is only one place where pointer to updated component needs to be set. Unfortunately this causes extra overhead since every call to component needs to go through indirection reference. Other option is to use reference counting where every reference is kept in memory [3]. This way allows hot-swap operation to update references to every required place. This does not cause complexity as indirection and other components are not required to handle indirection technique. Downside of reference counting is that it needs possibly large tables to keep references in memory and operation to update references is much more heavy than with indirection.

VI. Example: Kernel module hot-swapping in Linux

Linux Kernel is extendable using modules without need of kernel recompilation. As long as modules are not currently in use by other modules or any user level programs they can also be unloaded and then replaced by other updated module containing the same API. In many cases this is not good enough. If module is handling for example file system used on partition where is www-pages or database, module can not be removed without shutting down http-server before unloading old module. As indicated in section IV this is not acceptable in critical environments like online banks. If file system driver had some critical errors it still needs to be replaced immediately. With kernel able to do hot-swapping we can change the file system module on the fly without any downtime and end-users will never see problem.

A. Implementation

Implementation to Linux Kernel 2.6.11 has some difficulties since Linux kernel is not object-oriented nor component-based [4]. Kernel module system has also other difficulties than previously mentioned for problems.

Component boundaries needs to be solved other way than object-oriented style since modules are programmed in C [4]. Therefore state of the module needs to be explicitly defined in module or implicitly by module system. Former causes modifications to module but is also more flexible since programmer is able to control what needs to be kept safe for updated module.

Mutual consistency can be achieved by quiescence without need of synchronizations by monitoring module and replacing it immediatelly when modules is not in use [4]. Only change to old non-hot-swapping system is the monitoring system that detects the quiescence.

Linux Kernel uses symbol exporting as standard mechanism for exporting services between different parts of kernel including modules [4]. Symbol needed by a certain module can be requested only when loading the module and it will be read from symbol table. If module exporting symbol is changed by hot-swap, it also changes address of the symbol and therefore it needs to be updated to every module that has requested the module before. This problem is solved by dynamic resolution and relocation which means that reloading new module also updates symbol links in symbol table and makes modules using the symbols to resolve required symbols again.

Relating to previous problem, modules often need to pass variable addresses to each other [4]. When hot-swapping occurs these addresses become invalid without any solution. There are three different solutions that work in different cases. First is modifying functions that pass the addresses to allow updating new addresses. Second is using external variable allocation where variable is allocated outside of the module and does not change when module is updated. Third one is static address section where variables do not change their addresses when hot-swapping occurs.

There is also a problem with module descriptor. It can be seen when module is initiated and it needs to be updated when module is hot-swapped. Module descriptor contains two problematic elements related to hot-swapping [4]. First is reference count which contains amount of modules that depend on the hot-swapped module. Second one is the use list. A use list contains information about modules depending on the hot-swapped module. Module descriptor must be updated correctly when module is hot-swapped and it's information must be available for modules depending it.

B. Evaluation

Hot-swapping has been implemented for Linux Kernel 2.6.11 successfully and vfat file system is working test module. Vfat requires also a fat module so for full test both needed to be modified for hot-swapping.

Test had Apache http-server running on vfat partition and multiple clients benchmarking the server. Using hot-swapping slows module loading less than 4% as seen in table 1 and hot-swapping currently running module is 39% slower than than loading module without hot-swapping environment (table 2).

Loading times of the vfat module

Original system (µs)	Hotswap system (µs)	Ratio (%)
2119	2192	103.45

Table 1: Loading times [4]

Loading and hotswapping times of the vfat module

	Hotswap system (µs)
Module loading	2192
Hotswapping	3042
Ratio	138.77%

Table 2: Loading and hot-swapping times [4]

VII. Conclusion

I have shown how widely hot-swapping can be used in both hardware and software areas. Hardware hot-swapping is very mature technology and have been used widely for over 25 years. Almost every component in high-end servers can be replaced without shutting down the system.

Software based hot-swapping has been on talked for almost as long as hot-swapping hardware has been available, but it is still not commonly used in production. In majority of systems hot-swapping seems not to be necessary or customers are not aware of the possibility and therefore are not demanding it.

Technology for wide use of hot-swapping in software side is available and working considerably well. But since it is not available in base systems it won't get publicity it deserves. Nect step should therefore be adding hot-swapping to standard Java Virtual Machine and Linux kernel. Both have implementations and are widely used in production and server environments. After offering tools there will be soon hot-swapping programs and drivers.

References

[1] D. P. Siewiorek, "Fault Tolerance in Commercial Computers," in *Computer*, vol. 23, no. 7, 1990, pp. 26-37.

[2] T. Bloom, *Dynamic Module Replacement in a Distributed Programming System*, MIT, 1983

[3] C. A. N. Soules, J. Appavoo, K. Hui, D. D. Silva, G. R. Ganger, O. Krieger, M. Stumm, R. W. Wisniewski, M. Auslander, M. Ostrowski, B. Rosenburg and J. Xenidis, "System support for online reconfiguration" in *Proc. USENIX Annual Technical Conference*, 2003

[4] Y-F. Lee and R-C Chang, "Hotswapping Linux kernel modules," in *Journal of Systems and Software,* vol. 79, issue 2, 2006, pp. 163-175

[5] D. A. Patterson, G. Gibson, R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)", *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, Chicaco, Illinois, USA ACM Press, 1988, pp. 109-116

[6] M. Hicks, J. T. Moore and S. Nettles, "Dynamic software updating," in *SIGPLAN: ACM Special Interest Group on Programming Languages*, Snowbird, Utah, USA, ACM Press, 2001, pp. 13-23

[7] S. Malabarba, R. Pandey, J. Gragg, E. Barr and J. F. Barnes, "Runtime support for type-safe dynamic Java classes" in *Proceedings of the Fourteenth European Conference on Object-Oriented Programming*, 2000

[8] G. Morrisett, D. Walker, K. Crary and N. Glew, "From System F to typed assembly language," in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* San Diego, California, USA, ACM Press, 1998, pp. 85-97