Generic Crash-Resilient Storage for Indigo and Beyond

Aviv Blattner, Ram Dagan, Terence Kelly

HP Laboratories
HPL-2013-75

Keyword(s):

fault tolerance; Indigo; storage; persistent heaps

Abstract:

Computer crashes threaten application data integrity. The threat is particularly acute for expensive industrial equipment such as high-volume printing presses, whose stringent uptime requirements demand both high performance during normal operation and rapid recovery following crashes. We have designed and implemented a novel general-purpose persistent memory buffer that protects application data from crashes and is easy to integrate into existing software. Our solution slid with remarkable ease beneath the mature, complex, highly tuned software that controls HP Indigo printing presses, reducing recovery times from days to minutes while adding negligible overhead to failure-free operation. The new system has been in successful production use at dozens of beta test sites for several months and will eventually ship with all new Indigo presses. Our novel crash resilience strategy is not specific to Indigo presses and is likely applicable in a wide range of HP products, so we have developed a portable implementation that is available upon request.

External Posting Date: November 6, 2013 [Fulltext] Approved for External Publication
Internal Posting Date: November 6, 2013 [Fulltext]

Generic Crash-Resilient Storage for Indigo and Beyond

Aviv Blattner* Ram Dagan* Terence Kelly†

*Indigo Division, Graphics Solutions Business †OST/Labs

{aviv.blattner, ram.dagan, terence.p.kelly}@hp.com

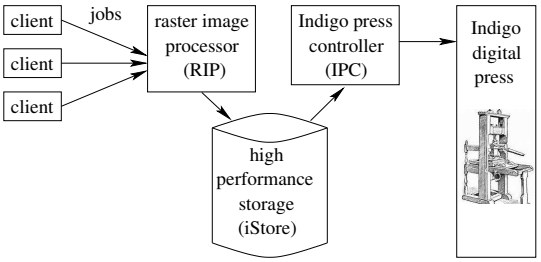
Abstract

Computer crashes threaten application data integrity. The threat is particularly acute for expensive industrial equipment such as high-volume printing presses, whose stringent uptime requirements demand both high performance during normal operation and rapid recovery following crashes. We have designed and implemented a novel general-purpose persistent memory buffer that protects application data from crashes and is easy to integrate into existing software. Our solution slid with remarkable ease beneath the mature, complex, highly tuned software that controls HP Indigo printing presses, reducing recovery times from days to minutes while adding negligible overhead to failure-free operation. The new system has been in successful production use at dozens of beta test sites for several months and will eventually ship with all new Indigo presses. Our novel crash resilience strategy is not specific to Indigo presses and is likely applicable in a wide range of HP products, so we have developed a portable implementation that is available upon request.

Problem statement

Power outages, OS kernel panics, and process crashes can corrupt or destroy application data. Applications can protect the integrity of their data by ensuring recovery to a consistent state following crashes, but existing mechanisms offer imperfect support: Conventional file systems do not directly provide transactional updates of application data; relational databases can be heavyweight and cumbersome; and transactional key-value stores present an awkward narrow interface. Furthermore none of these mechanisms are transparent, so they do not facilitate retrofitting crash tolerance onto applications that were not originally designed to withstand crashes. Fortifying mature, complex, high-performance software is particularly challenging: There are too many places where data must be committed to durable storage during normal operation and too many corner cases during recovery.

HP Indigo printing presses illustrate both the technical difficulty and the business value of making widely deployed, highly tuned production software crash tolerant. The business case is straightforward: An Indigo press can cost over $500K and is typically the centerpiece of the purchaser's business, which loses revenue whenever the press operates below full capacity. HP shares directly in such losses because most Indigo presses operate under a "click charge" model (the customer pays HP for every printed page). To ensure full utilization during normal operation, sophisticated software feeds print jobs to an Indigo press as rapidly as possible. Unfortunately this software was not designed for quick crash recovery, and power outages occur frequently in print shops.



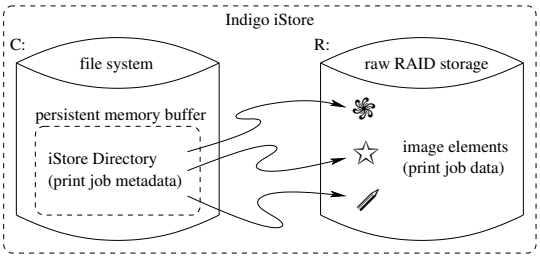
The difficulty of making Indigo presses crash tolerant stems from the streamlined efficiency of failure-free operation. Print jobs from clients (e.g., PDF files) pass through a computationally expensive raster image processor (RIP) that renders jobs into myriad image elements (e.g., bitmaps and font characters). Image elements and metadata describing their placement on the page are written to a proprietary storage system, iStore, from which the Indigo press controller retrieves elements as needed and sends them to the press. iStore caches and re-uses elements across jobs, reducing redundant RIP activity; over time, several terabytes of data may accumulate in iStore. To ensure that the press never stalls waiting for data, over the years iStore has incorporated sophisticated performance optimizations including a custom disk scheduler [3] and a custom memory allocator [1]. Crashes corrupt data in iStore, and reconstructing the lost data forces large numbers of re-submitted jobs through the very slow RIP all at once. Recovery can take *days* and sometimes requires technical support from HP Indigo personnel. A good crash-resilience solution would add substantial customer-visible value to HP Indigo presses by improving uptime; if generic and re-usable, it could similarly fortify other HP products.

Our solution

We considered and rejected several seemingly reasonable solutions. Manually inserting checkpointing and recovery code into iStore would be formidably costly, risky, and time-consuming, and furthermore would not be re-usable beyond iStore. An uninterruptible power supply offers no protection from crashes due to bugs in our own software or in the OS; moreover a customer-installed external UPS would be outside our control, whereas a built-in UPS would require design changes to new presses and would bring no benefit to the installed base. We sought a *software* solution that is both *generic* in applicability and *easy to retrofit* onto mature, complex, highly tuned code bases.

Our solution is a *persistent memory buffer*, a novel user-space software mechanism inspired by recent research on kernel support for failure-atomic updating of memory-mapped files [2]. Our interface enables applications to `Initialize` a region of memory in which arbitrary main-memory data structures may be manipulated with ordinary load and store instructions. Applications may `Commit` the current state of the buffer to the file system; following a crash, applications may `Recover` the buffer back into memory. Recovery restores the persistent memory buffer to the state corresponding to the most recent successful `Commit`, regardless of when the crash occurred. The intended mode of use is for applications to keep in a persistent memory buffer sufficient consistent state to enable recovery; we ensure that this state is not corrupted or destroyed by crashes.

We implemented the persistent memory buffer concept for Microsoft Windows because Indigo iStore runs on Windows Server. The most important library function underlying our implementation is `GetWriteWatch`, which allows a program to efficiently and conveniently track modifications of memory pages in a specified virtual address range. `GetWriteWatch` enables our `Commit` function to determine which pages in our persistent memory buffer have been altered since the previous `Commit`. The dirty pages constitute an incremental checkpoint, which is written to the file system. Finally, our `Commit` function re-sets the write watch so that the next call to `Commit` can find pages modified since the present call. Our `Recover` function reconstructs a consistent state of the persistent memory buffer from the incremental checkpoints.



We integrated our persistent memory buffer into the Indigo press iStore system, depicted in greater detail at right. iStore includes separate storage for print job data and metadata; the former is typically large (up to several TB) but the latter is relatively much more compact (less than one GB). Metadata are organized in a Directory that includes references to data in a raw RAID storage system on the `R:` drive. The iStore Directory previously resided in main memory only. We placed the Directory in a persistent memory buffer checkpointed to the `C:` drive file system. After a print job passes through the raster image processor, the resulting image elements are written to the RAID storage system, then the Directory is updated to reflect the newly entered job, and finally the persistent memory buffer containing the Directory is `Commit`ted. The net effect is that *job submissions are atomic*, even in the presence of crashes: Recovery restores the Directory to a state reflecting the most recent successful job submission, in which all of the "pointers" from the Directory into the RAID system are valid. Crashes effectively erase in-progress job submissions but do not corrupt or destroy completed submissions.

In our experience, the persistent memory buffer is a powerful and general *foundation abstraction* that composes conveniently and harmoniously with higher-level abstractions layered above it, e.g., the iStore Directory with its sophisticated custom memory allocator. Overall we found it remarkably easy to slide our persistent memory buffer beneath the large, complex, highly tuned iStore code base, which was never designed to support failure-atomic job submission; the required modifications were neither numerous nor difficult. We are confident that a persistent memory buffer could integrate into a wide variety of other software systems with comparable ease.

Evidence the solution works

We experimentally verified that our fortified iStore operates as intended. Following the methodology of Park et al. [2] we subjected a computer running iStore to repeated whole-system electric power interruptions. During a marathon 46-hour test session we crashed the server 425 times; recovery succeeded correctly every time. In stark contrast to the multi-day recovery times of the original iStore, our crash-tolerant iStore recovered in minutes. We also mea-

sured the performance overhead of `Commit`ting the persistent memory buffer, which is the last step of "ingesting" a print job from a client. Because our design separates iStore metadata from data and places only the former in the persistent memory buffer, incremental checkpoints are small and thus `Commit` is fast. `Commit` typically writes a few MB and takes at most a few seconds, which is negligible compared to the several *minutes* typically required to process a job in the raster image processor.

Competitive approaches

The prior system most similar in spirit to our persistent memory buffer is failure-atomic `msync` [2], which offers comparable ergonomic attractions via a modified Linux kernel. By contrast, our persistent memory buffer is implemented in portable user-level code and runs on unmodified OSes, which facilitates adoption. Object-oriented databases and object-relational frameworks offer transactional updates [6]. Our approach imposes fewer framework constraints and thus offers more flexibility to retrofit crash resistance onto complex legacy C/C++ software. For emerging non-volatile memories (NVRAM), researchers have enhanced transactional memory (TM) to atomically update NVRAM [4]. Our persistent memory buffers provide failure atomicity, allow unrestricted use of existing concurrency-isolation mechanisms (including but not limited to TM), and do not require NVRAM. The user-space persistent heap in the "Ken" fault-tolerant distributed computing platform efficiently takes incremental checkpoints and presents a `malloc/free` interface [5]. Persistent memory buffers neither mandate nor preclude any particular memory allocator, which is useful for existing software that has its own, e.g., iStore. Compared with most prior systems, persistent memory buffers are simpler, more convenient, and more widely applicable.

Current status

We implemented our persistent memory buffer and integrated it into Indigo iStore in the first half of 2013. Beta tests on dozens of production presses began in the second half of the year. Inspired by the success of Windows-based persistent memory buffers for Indigo, we developed an analogous POSIX-based portable library implementation. A very small `Commit` to very fast solid-state storage takes under three milliseconds on the new implementation, which is being used in several HP Labs projects and is available HP-wide upon request.

Next steps

Eventually, all new Indigo presses will be shipping with our new crash-tolerant iStore, which will also be available as an optional software upgrade to the entire installed base of Indigo presses. We are eager to help HP colleagues to use persistent memory buffers in other HP products and we will gladly share our Windows and POSIX code within HP; we plan to release the POSIX code as open source software. Our fortified iStore design is easy to imitate and offers high performance, RAID redundancy, and crash resilience—an attractive combination of benefits likely to be useful in many contexts. Looking forward to emerging non-volatile memories such as HP's memristor, HP Labs research is developing persistent memory buffers as a software abstraction well suited to NVRAM programming.

References

[1] A. Blattner, "Data Consistency over Power Loss for iStore," HP Restricted internal memo, Oct. 2013.

[2] S. Park et al., "Failure-Atomic `msync()`," *EuroSys*, April 2013. `http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Park.pdf`

[3] C. Staelin et al., "CSched : Real-time disk scheduling with concurrent I/O request," HP Labs Tech Report HPL-2011-11, Jan. 2011. `http://www.hpl.hp.com/techreports/2011/HPL-2011-11.pdf`

[4] H. Volos et al., "Mnemosyne: Lightweight Persistent Memory," ASPLOS 2011.

[5] S. Yoo et al. "Composable Reliability for Asynchronous Systems," *USENIX ATC*, June 2012. `https://www.usenix.org/system/files/conference/atc12/atc12-final206-7-20-12.pdf`

[6] `http://en.wikipedia.org/wiki/Object-relational_database`