Journal of Physics: Conference Series **219** (2010) 062035 doi:10.1088/1742-6596/219/6/062035

Condor enhancements for a rapid-response adaptive computing environment for LHC

D Bradley, S Dasu, M Livny, A Mohapatra, T Tannenbaum and G Thain

University of Wisconsin, Madison, WI, USA

E-mail: dan@hep.wisc.edu, dasu@hep.wisc.edu, miron@cs.wisc.edu, ajit@hep.wisc.edu, tannenba@cs.wisc.edu, gthain@cs.wisc.edu

Abstract.

A number of recent enhancements to the Condor batch system have been stimulated by the challenges of LHC computing. The result is a more robust, scalable, and flexible computing platform. One product of this effort is the Condor Job Router, which serves as a high-throughput scheduler for feeding multiple (e.g. grid) queues from a single input job queue. We describe its principles and how it has been used at large scale in CMS production on the Open Science Grid. Improved scalability of Condor is another welcome advance. We describe the scaling characteristics of the Condor batch system under large workloads and when integrating large pools of resources; we then detail how LHC physicists have directly profited under the expanded scaling regime.

1. Condor Job Router

the Condor Job Router was created to serve the following specific use-case: a user with a large number of compute jobs wishes to run the jobs on a combination of grid and non-grid resources. By "non-grid" we mean vanilla universe condor jobs that use matchmaking and possibly Condor flocking to run on computers in one or more Condor pools. By "grid" we mean sites that will accept jobs from the user through any of the remote job submission protocols supported by Condor-G (e.g. Globus). By "Condor" we mean version 7.2.0 and beyond, with source and binaries (including Job Router) available for download from www.condorproject.org.

The user faces a trivial but annoying scheduling problem in this scenario. Jobs must be partitioned into those destined for the grid and those destined for the non-grid resources, because these are two distinct and immutable classes (*universes* in condor terminology). The user doesn't know in advance how quickly jobs in these two classes will complete, because resource availability, prioritization policy, and performance are all dynamic and difficult to predict.

The Job Router provides a mechanism for dynamically transforming jobs from one Condor universe into another. It addresses the load balancing problem using a simple strategy: the number of transformed jobs "in flight" may be limited. Therefore, the user may submit, say 10 thousand vanilla universe jobs and Job Router can just keep 500 at a time transformed into grid universe jobs. As these finish, it transforms more to replace them, until there are no more jobs left to transform, because they have all finished as either vanilla jobs or as transformed grid jobs.

In addition to or instead of having a fixed total limit, the number of idle (waiting to run) jobs can be limited. In this case, the Job Router transforms jobs until it reaches the idle job limit. Then it waits for some of the idle jobs to start running before it transforms more jobs. In this way, it can adapt to fluctuations in the availability of grid resources using direct evidence (idleness of a small number of jobs) rather than having to face the complex task of evaluating remote queue prioritization policies and resource usage in order to estimate availability of resources for making scheduling decisions.

Since the grid resources available to a user may actually be split between multiple distinct grid sites, this presents an additional load balancing problem. Condor-G site-level matchmaking has been successfully used to address this[1]. However, if one is already using Job Router to handle the scheduling task of vanilla to grid universe transformation, it is a simple extension to also handle the task of targeting several destinations. Instead of specifying a single transformation (vanilla-to-grid), one can specify multiple transformations (vanilla-to-site1, vanilla-to-site2, ...). Each of these transformations is called a *route*. The Job Router then chooses one of the available routes using fast round-robin scheduling, subject to the individual constraints of each route, such as maximum number of routed jobs, maximum idle routed jobs, and so on.

In the general case, arbitrary attributes of the job description can be changed by the transformation rules associated with a route in the Job Router's configurable policy. Special support is provided for inserting an X509 proxy into the job if this is desired. In addition, call-out hooks have been added (not by the authors but by Red Hat collaborators) so that external actions may be taken at key points in the routed job's life. Red Hat uses this for stuffing jobs into virtual machines and routing them to Amazon's EC2 service, a more complicated transformation than is possible by simply modifying the job description.

Routes may also have requirements for which types of jobs they are willing to accept, so not all paths are available to all jobs. The most basic application of this is to prevent jobs from being routed if they are incompatible in some way. For example, a job which relies on a shared filesystem that is not accessible from remote grid sites should not be routed to those sites. Since dependencies such as this are not always explicitly visible in the job description, the safest strategy is to require that jobs opt-in by declaring in the job description that they want to be routed.

1.1. The Routing Table

The routing policy of the Job Router is specified as a list of ClassAds. Each ClassAd specifies one route, and the attributes of the ClassAd specify the requirements of that route and how jobs are to be transformed by it.

The routing table may either be hand-coded or it may be generated by a plug-in that polls some external information source such as the OSG Resource Selection Service. The details are best left for the manual, but a simple routing table depicted in Table 1 gives the flavor of how it works.

1.2. Performance and Scalability of Job Router

The memory requirements of Job Router scale linearly with the number of jobs in the job queue, because, like the schedd, the Job Router keeps a copy of the full job queue in memory. The Job Router requires in the range of 150MB to 1GB for 100,000 jobs, depending on the size of the job descriptions and the use of job clusters. Since the Job Router acts on jobs by placing the transformed copy of the job as a new (linked) job in the job queue, the size of the queue is increased by the number of concurrently routed jobs.

As shown in Figure 1.2, we found that there was little difference in throughput between queues of 10,000 jobs and 100,000 jobs, so the Job Router appears to scale well within that range. We expected it to scale reasonable well to large job queues, because it uses an efficient

```
[ name = "Site 1";
  GridResource = "gt2 site1.edu/jobmanager-condor";
  Requirements = other.WantJobRouter;
  MaxIdleJobs = 10;
]
[ name = "Site 2";
  GridResource = "gt2 site2.edu/jobmanager-pbs";
  Requirements = other.WantJobRouter;
  MaxIdleJobs = 10;
  MaxJobs = 100;
  set_GlobusRSL = "(maxwalltime=1440))(jobType=single)";
]
[ name = "Site 3";
  GridResource = "condor submit.site3.edu condor.site3.edu";
  Requirements = other.WantJobRouter;
  MaxIdleJobs = 10;
  set_remote_jobuniverse = 5;
]
```

Table 1. The routing table is a list of ClassAds in Condor's "new" ClassAd syntax. Attributes of the ClassAds specify scheduling policy and job transformation rules.

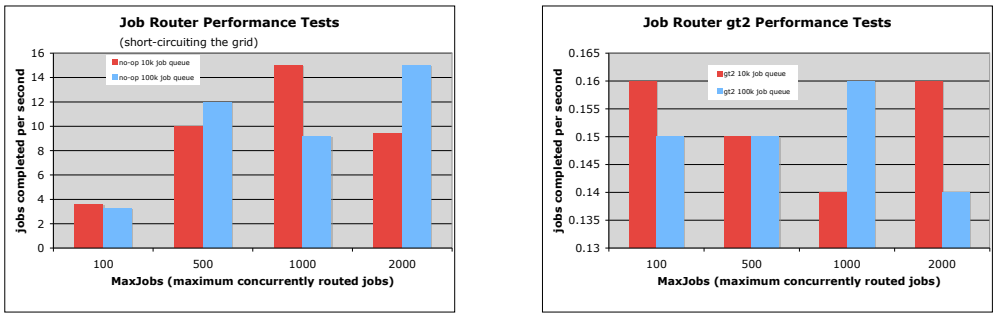

Figure 1. Tests of Job Router 7.3.2 on a 1.6GHz dual AMD Opteron with 7GB RAM. "no-op" indicates jobs that were transformed into a "done" state, bypassing any grid protocols, but still going through all the same motions in the Job Router. gt2 shows routing to a Globus GRAM 4.0.7 gatekeeper using the gt2 protocol.

method of maintaining a copy of the job queue. Rather than periodically querying the full state of the schedd's job queue, Job Router reads from the job queue transaction log and simply adjusts its copy of the queue as new transactions are committed to the log by the schedd.

In the "no-op" test, where jobs were routed without going to the grid, the throughput plateaued between 9 and 15 jobs/s. This is about half the maximum rate that one can submit jobs to the schedd. Since the Job Router has to finalize the jobs as well as submit them, it is reasonable that it achieved about half the maximum submission rate.

When routing jobs to Globus GRAM using the gt2 protocol, Job Router is doing all the same work that it did in the no-op case, so we attribute the considerably lower throughput in the gt2 case to something in Condor-G or Globus GRAM. We believe the Condor-G/Globus bottleneck is per-gatekeeper, so we expect total throughput to scale linearly with the number of target gatekeepers at least to the 10 jobs/s range achieved in the no-op case.

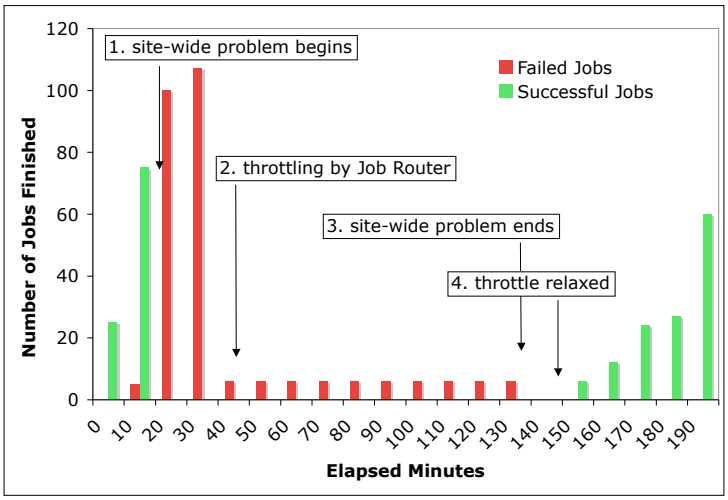

Figure 2. When the failure rate goes above the specified threshold (0.01 jobs/s in this example), routing of jobs to the failing site is limited.

1.3. Handling Errors

The simple Job Router scheduling algorithm outlined so far may be thought of as a "positive pressure" scheduler. It tries to keep all of the available sites busy by pushing jobs to them as fast as they run. A problem with this kind of scheduler is evident when one of the sites has a problem that causes jobs to rapidly fail for a period of time. This is commonly known as a "black hole". It can quickly consume (and fail) a large queue of jobs in a short span of time. Even if job failure is automatically handled by resubmission of failed jobs, rapid failure can create excessive stress on numerous parts of the system and can therefore make it difficult to keep the good sites busy when one bad site is generating so much work for the job-handling software.

To guard against this, the Job Router can set thresholds on the highest acceptable failure rate for each route. The test for failure is arbitrary and could even be a simple check that jobs do not complete unexpectedly quickly. When the failure rate exceeds the threshold, the rate at which new jobs are sent to the affected site is restricted in order to bring the failure rate down to the threshold. Evidence of successful job completion will cause the restriction to be gradually relaxed. Figure 2 shows a test run in which total failure of a site was induced.

In practice, we have found this "black hole" avoidance to be effective in a number of real-world circumstances in the Open Science Grid. Examples are shared filesystem failure, storage element failure, and unexpected state in the pre-installed application repository. However some types of problems are not problems of the whole site but rather of an individual worker node at the site, which may still be capable of producing a very high failure rate. In such cases, throttling the whole site has the undesirable effect of also reducing access to the large fraction of the site that is functioning normally. However, in the absence of some way of avoiding the black hole node, most jobs sent to the site may end up failing, because the bad node finishes jobs so quickly that it is always free to accept more.

We have found that a crude but effective strategy in this case is to wrap the job with a script that enforces a minimum run time in case of failure (e.g. 20 minutes). This "plug script" keeps the bad node occupied for long enough to keep it from rapidly consuming all idle jobs. The site-level throttle is still useful in case of a problem affecting the whole site, or in case of a worker

node that is broken in a way that causes the plug script itself to fail.

1.4. CMS Simulation using Job Router

The CMS experiment provides a Monte Carlo event simulation service. The computational workload for this "official production activity" is generated and tracked by a software package called prodAgent[2]. The simulation jobs are distributed across LCG and OSG sites.

For the portion of CMS simulation that runs on OSG, the Job Router is used to do the site-level scheduling. prodAgent is configured to generate vanilla universe condor jobs. The Job Router is given a routing table that includes an entry for each of the dozen or so desired compute elements.

At a given time, the system may have tens of thousands of jobs instantiated in the Condor queue with up to six thousand jobs routed to 10 grid sites. A number of different datasets may be in production concurrently. For ease of management, the production operator found it convenient to generate datasets at specific sites, rather than distributing jobs for all datasets uniformly across the sites. This was accomplished by advertising the dataset in the job ClassAd and setting the routing requirements appropriately.

Sometimes dataset priorities will change, or site productivity will change and it becomes desirable to alter the way the sites are being used. Since the routing table can be modified at any time, it is a simple matter to change the mapping of datasets to sites for the remaining jobs that are waiting to run.

Jobs that have already been routed to a site and which are waiting to run or waiting to finish running can be rerouted as well. When a job is routed, it appears twice in the Condor job queue, once with the original job description and once with the transformed job description. Removing the routed copy of the job resets the job back to the original state, after which it may be routed again using the latest routing table. In practice, this simple mechanism has proved useful.

2. glideinWMS

Another recent advance relevant to LHC is in the scalability of the Condor glidein component of glideinWMS (the glidein Workload Management System used by CMS). This topic is covered in further detail elsewhere[3]. In some ways, glideinWMS is a more elegant solution to the Job Router scheduling task. Rather than turning vanilla Condor jobs into grid jobs, it turns the grid into a Condor pool. Here we give a brief comparison of Job Router and glideinWMS to give an idea of their relative strengths.

One of the key features of the Job Router is that it delays scheduling decisions until it is forced to make them. When it sees that the sites are too busy to run more jobs, it delays deciding what to do with the remaining ones. glideinWMS takes lazy scheduling one step further. It doesn't send a job to a site until a specific worker node at the site has been allocated to run the job. glideinWMS is a specific case of the general approach known as job *pilots*[4].

Whereas Job Router pushes jobs to grid sites, glideinWMS instead pushes glidein pilot jobs to the sites. The glideins are then responsible for pulling jobs from the central queue directly to a worker node that the glidein has been given to run on. In this sense, Job Router schedules by positive pressure (pushing) and glideinWMS schedules by negative pressure (pulling).

The advantage of the pilot way is that the user's job is never at the mercy of a site's job queue. Job Router, on the other hand, pushes jobs from the central queue into the job queues of the grid sites. It makes no attempt to understand the policies and conditions of those queues except for keeping track of the state of the jobs it has already pushed there, so it has little idea which site will be able to run a given job the soonest. This means it could end up routing a job to site A, which will not start running the job for a few hours, when site B would have started running the job in a few minutes. glideinWMS, on the other hand, keeps some idle pilot jobs in

the queues of both sites, and the first one which begins running will pull the next waiting job from the central queue.

When the number of top priority jobs waiting to run is large compared to the size of the grid (i.e. the number of jobs the Job Router has concurrently queued at all sites), the choice of site for the next job to be scheduled doesn't really matter, because the scheduler can afford to push some jobs to each site and just try them all. This is the high-throughput workflow regime in which the Job Router approach is reasonable. When the number of waiting top priority jobs is small compared to the grid, the Job Router will sometimes make poor choices relative to a pilot system such as glideinWMS.

Even pilot systems can't always choose the site which would successfully *finish* the job soonest. For extremely urgent jobs in a small workflow, or at the tail of a high-throughput workflow when a few jobs are "holding things up", it may be advantageous to schedule concurrent runs of the same job and have them race to completion, such as in the JugMaster pilot system[5]. In principle, this could be done also in Job Router, analogous to multi-cast routing, but this functionality has not been implemented.

In the handling of errors, both the Job Router and glideinWMS have to be careful to avoid black holes. However, glideinWMS has the advantage of being able to test for any anticipated problems before pulling a job from the work queue. This may reduce churn in the job handling system and somewhat detangle analysis of site problems from job problems.

If there is one aspect where Job Router has an advantage over glideinWMS, it is simplicity. Setting up Job Router is a fairly trivial process–adding a few lines to a Condor configuration file and watching over one additional daemon. glideinWMS requires secure communication with the worker node, perhaps traversing firewalls and NATs, and it requires a system for pushing glidein pilots to the sites. It may also require interaction with a grid-aware execution service such as glExec on the worker nodes to correctly distinguish between the pilot job's identity and the job owner's identity. This added complexity means there are more things that need to be watched over and understood in case things go wrong.

Fortunately, recent developments in the glideinWMS system have significantly reduced complexity while increasing scalability and reliability[3]. Although we continue to prefer Job Router for large workflows with many equally prioritized jobs, or for situations in which the support footprint must be kept to a minimum, we think glideinWMS does offer advantages in other situations. We think it is especially attractive if it can be deployed and supported centrally as a service that benefits a whole community. Use of glideins in this way has been a successful model for CDF[6].

3. Condor Scalability Improvements

In recent releases, a concerted effort has been made to improve scalability and performance of the Condor batch system. This benefits users with large workflows. It also benefits glideinWMS, which is essentially a huge Condor pool, potentially dwarfing individual Condor pools in the grid, because it spans across multiple grid sites. Details of scalability enhancements that are most relevant to the glideinWMS case are described elsewhere[3].

The large workflows of LHC analysis demand scalability of the job queue to at least 10s of thousands of jobs. Experience in the field and tests of Condor with job queues of up to 100,000 jobs helped us identify a number of problematic algorithms that worked fine at small scale but at larger scale grew increasingly cpu-intensive in a non-linear fashion. Some of the improvements over time are demonstrated in Figure 3.

In many cases, the scalability and performance improvements have a much broader pay-off than is suggested by simple metrics. Our experience in busy environments, such as on the OSG compute elements of large sites, is that cases where the Condor schedd simply "melts down" and times out repeatedly are noticeably less common using the recent higher performing versions of

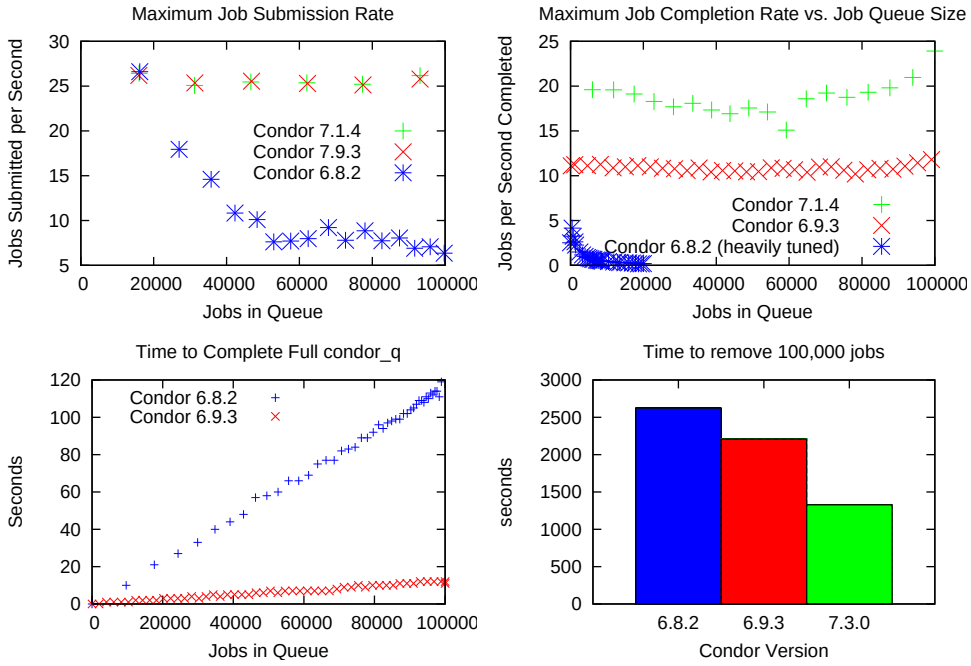

Figure 3. The Condor schedd demonstrates improved peak performance and scales well in tests of job queues ranging up to 100,000 jobs on a 1.6GHz AMD Opteron dual CPU machine with 7GB RAM.

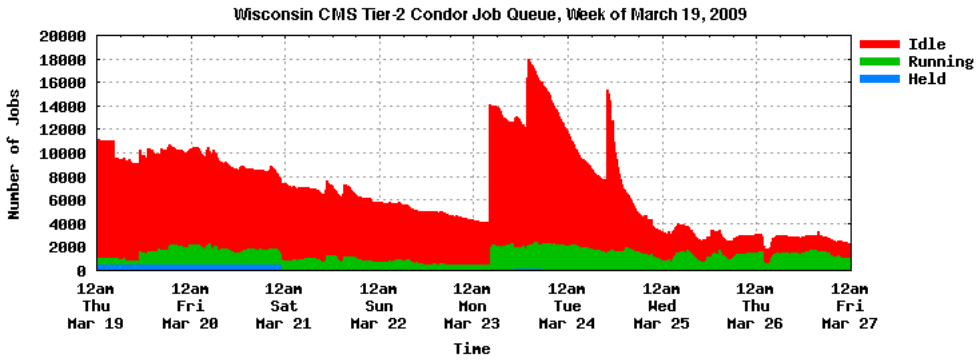

Figure 4. A typical week in the Wisconsin CMS Tier-2 computing center. Users quickly generate large workflows and manipulate collections of jobs exceeding the level of 5000 that was once a scalability limit.

Condor. The cost of some operations, such as syncs to disk of the job queue log, can be quite variable, depending on other activity on the computer. An optimization that amounts to a 25% performance boost in an idealized environment can make the difference between reasonable operation and complete failure in a more chaotic environment.

4. Impact for LHC Physicists

We work closely with physicists using the Wisconsin Tier-2 CMS Computing Center. For them, the benefits of our recent Condor developments are quite simple: they can quickly generate

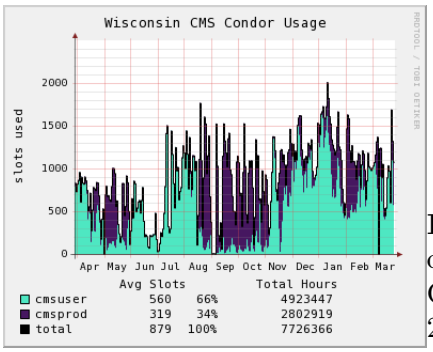

Figure 5. Usage of University of Wisconsin Madison campus Condor pools by CMS, March 2008 to 2009.

larger workflows and they get their results back faster than ever before. A snapshot of the job queue in a typical week is shown in Figure 4. Cases where users have to be asked to limit how many jobs they put into the queue no longer occur. Cases where the number of concurrently running jobs have to be limited to protect Condor are also a thing of the past.

Figure 5 shows the significant Condor usage by CMS in the University of Wisconsin Madison campus over the past year (over 7.5 million CPU-hours). In addition to the "cmsprod" activity, which is centralized Monte Carlo simulation, a significant amount of user-generated activity has been taking place. In order to balance the need for fast turnaround in user analysis with the need for longer-running simulation jobs, we created a "fast queue" for analysis jobs. When jobs run in the "fast queue," jobs in the normal queue are suspended by Condor. When the fast queue job finishes, the suspended job resumes where it left off. In this way, users can get results quickly while avoiding preemptive killing and restarting of longer running simulation tasks.

5. Conclusion

Recent releases of Condor include a number of enhancements triggered by the needs of LHC physicists. A much more scalable job queue is probably the most important outcome. In addition, flexible and high-throughput grid scheduling is provided by Job Router and by numerous improvements to Condor glidein. Many scheduling challenges remain as we approach the dawn of LHC data, but we are pleased to look back and see how far the software has come.

Acknowledgments

This work was supported by the U.S. National Science Foundation grants PHY-0427113 (RACE) and PHY-0533280 (DISUN).

References

[1] Garzoglio G, Levshina T, Mhashilkar P, and Timm S, *Grid Computing Int. Symp. on Grid Computing (ISGC 2007)* ed Simon C. Lin and Eric Yen (Springer US), p89-98.
[2] Evans D, Fanfanib A, Kavkac C, van Lingend F, Eulissee G, Bacchib W, Codispotib G, Masona D, De Filippisf N, and Hern JM, *Nuclear Physics B - Proceedings Supplements* **177-8** p285-6.
[3] Bradley D, Sfiligoi I, Padhi S, Frey J, and Tannenbaum 2009 Interoperability and Scalability within glideinWMS *Preprint* JPCS.
[4] Sfiligoi I, Bradley D, Holzman B, Mhashilkar P, Padhi S, Würthwein F 2009 The pilot way to Grid resources using glideinWMS *Preprint* CSIE.
[5] Dasu S, Puttabuddhi V, Rader S, Bradley D, Livny M, and Smith W Use of Condor and GLOW for CMS Simulation Production *Proceedings of the CHEP'04 Conference, Interlaken, Switzerland*, Published in InDiCo.
[6] Compostella G, Delli Paoli F, Jeans D, Lucchesi D, Sarkar S, and Sfiligoi, I *Nuclear Science Symposium Conference Record, 2006. IEEE* **2** p873-8.