Functional Partitioning to Optimize End-to-End Performance on Many-core Architectures

Min Li[1], Sudharshan S. Vazhkudai[2], Ali R. Butt[1], Fei Meng[3], Xiaosong Ma[2,3], Youngjae Kim[2], Christian Engelmann[2], and Galen Shipman[2]

[1]Virginia Tech, [2]Oak Ridge National Laboratory, [3]North Carolina State University

{limin, butta}@cs.vt.edu, {vazhkudaiss, kimy1, engelmannc, gshipman}@ornl.gov, {fmeng, ma}@cs.ncsu.edu

***Abstract*—**

Scaling computations on emerging massive-core supercomputers is a daunting task, which coupled with the significantly lagging system I/O capabilities exacerbates applications' end-to-end performance. The I/O bottleneck often negates potential performance benefits of assigning additional compute cores to an application. In this paper, we address this issue via a novel *functional partitioning* (FP) runtime environment that allocates cores to specific application tasks — checkpointing, de-duplication, and scientific data format transformation — so that the deluge of cores can be brought to bear on the entire gamut of application activities. The focus is on utilizing the extra cores to support HPC application I/O activities and also leverage solid-state disks in this context. For example, our evaluation shows that dedicating 1 core on an oct-core machine for checkpointing and its assist tasks using FP can improve overall execution time of a FLASH benchmark on 80 and 160 cores by 43.95% and 41.34%, respectively.

I. Introduction

As growth in processor frequency has stagnated, chip designers have turned to increasing the number of processing cores per socket to meet Moore's law scaling of processor capability. In the near future, each socket may contain 8, 16, or even 80 or more cores, e.g., Intel's 80-core chip prototype [1]. This push towards increasing peak CPU throughput in High Performance Computing (HPC) systems is not matched by a similar push towards improving the access bandwidth to other components: sustained I/O bandwidth significantly lags behind processor improvements [2]. With many-core processors driving up the per-socket memory and I/O bandwidth requirements, the "storage wall" problem that has long perplexed designers of parallel computing clusters is now moving to within each compute node.

Consider the current No. 1 machine on the Top500 list [3], the 224,256-core Jaguar petaflop supercomputer. A balanced petaflop system should sustain a bit of I/O operation per instruction, requiring the parallel file system (PFS) to provide 100 TB/s of I/O bandwidth. In reality, the currently used state-of-the-art PFS, Lustre [4], supports a peak I/O bandwidth of 254 GB/s [5] (based on an IOR benchmark), which is two orders of magnitude less than the ideal bandwidth. Furthermore, applications typically only realize a fraction of this peak performance due to software overhead or resource contention.

Simply assigning cores to an application does not scale: End-to-end application performance is not expected to grow linearly with the number of cores [6], mainly due to the bottleneck-prone HPC storage hierarchy and the contention for on-chip resources in the "sea of cores" of modern multicore systems. Adding to this is the overhead from using the needed sophisticated but complex programming techniques both in symmetric [7]–[11] and asymmetric [12], [13] multicores.

To underscore these challenges, we tested two common parallel programs, namely mpiBLAST [14], an I/O-intensive biological sequence alignment application, and FLASH [15], [16], a computation-intensive astrophysics simulation. We executed the applications on a cluster comprising four oct-core machines, using standard scheduling to map processes to cores on the four nodes. We used a 24 GB workload and Sod 3D for mpiBLAST and FLASH, respectively, and studied them with a fixed total problem size, i.e., under strong scaling.

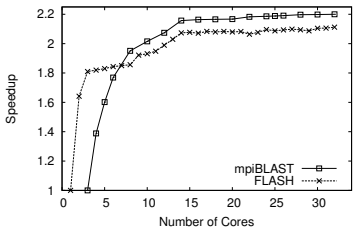

Fig. 1. Speedup obtained for mpiBLAST and FLASH.

Figure 1 shows the speedup achieved with increasing number of cores compared to the case of one core. It is observed that for the selected workloads, using increasing number of cores does not provide corresponding improvement in execution time. This behavior can be attributed to: (i) the storage wall as more cores contend for the data

SC10 November 2010, New Orleans, Louisiana, USA 978-1-4244-7558-2/10/$26.00

from PFS, and (ii) the memory wall as more cores in a node compete for the shared memory.

Functional partitioning of cores: The solution to the compute-I/O imbalance problem must scale with core count. In the post-petascale environment, everything off socket is too far away in terms of "clocks." Many system resources and tools will need to be present on a compute node and integrated and composed into system-level services at scale. Rather than continuing to assign more of the available cores to computation and intensifying the pressure on the memory and secondary storage systems, some of the cores may serve applications' overall performance better if they can intercept and absorb part of its data-intensive tasks. Such asymmetric division of labor among cores is not new. There exists I/O libraries that dedicate processors [17], [18] or use separate threads [18], [19] for handling parallel I/O operations. Similarly, BlueGene/L [20] uses distinct cores for compute and networking tasks. However, we propose a *functional partitioning (FP) runtime environment* as a generalized way to dedicate a subset of cores within a compute node to data processing services to help alleviate the I/O bottleneck. This way, our approach can improve the overall system resource utilization and speed up eventual application turnaround. As a proof-of-concept study, we explore several sample data services, such as peer checkpointing, data analytics, and file format conversion. The partitioning techniques we develop can also provide the basis for other activities, such as monitoring, result verification and validation, shadow computation, compression, and encryption.

A suitable platform for demonstrating the usefulness of FP is driven by the observation that future HPC systems are likely to be equipped with non-volatile devices such as solid-state disks (SSDs). SSDs offer excellent read/write throughput (e.g., up to GB/s with PCIe cards) when compared to secondary storage and larger capacity when compared to DRAM (hundreds of GBs). SSDs can be used either as slow memory or a faster cache and posses very desirable properties such as low power consumption and persistence. HPC systems are beginning to explore the use of SSDs in the storage hierarchy (e.g., SDSC's Gordon [21]) as a means to mitigate the pressure on storage systems. However, there is a lack of a coherent architecture in HPC to use SSDs in a unified fashion and in concert with secondary storage. Further, there is no clear set of guidelines as to where to place SSDs in the supercomputer (e.g., node-local or system nodes). Thus, in this paper, we also investigate the use of SSDs towards addressing the I/O bandwidth bottleneck and illustrate how FP can be used to dedicate cores that are geared towards performing different I/O services on the SSDs. However, the benefits of FP hold irrespective of whether a system employs SSDs or not.

A. Contributions

This paper makes the following contributions.

Functional Partitioning (FP) Design Paradigm: We propose a FP runtime environment as a novel generalized approach to partitioning many-core systems. Rather than focusing on raw performance scaling, FP enables the concerted use of the plethora of cores towards an application's own diverse set of activities. The runtime lays out an architecture that enables: (a) applications to specify the assist tasks in a job script, (b) an auxiliary application (aux-app) model for the assist tasks so they can co-execute with the applications, and (c) a foundation on which dynamic adaptation of the provisioning of cores to aux-apps can be performed in the future.

Building-blocks for FP: We show, via implementation and experimentation, that dedicating checkpoint and de-duplication cores is in fact a concrete first step towards functionally distinguishing cores.

Aggregate, Intermediate SSD Device: We built an architecture to harness SSD storage, in user space, from individual nodes to create a scalable, aggregate intermediate SSD device that can sustain high-speed writes. The device also facilitates diverse data operations potentially offered by our proposed service cores. Our approach is by far the first to propose a concerted use of distributed SSDs in a supercomputer environment.

Evaluation: We undertake a thorough evaluation of the FP approach using a large 160-core testbed, studying the resulting I/O throughput, impact of varying different parameters such as number of dedicated cores, and the overall impact on a real-world application's performance.

II. Related Work

Resource management in heterogeneous and specialized multi-processor systems has gained much research attention recently [22]–[26] with focus on scientific applications. There have also been studies on utilizing available cores for online execution monitoring or security checking [27], [28]. However, to the best of our knowledge, FP on general-purpose, homogeneous cores has not been studied for mainstream HPC applications. Although there exists I/O libraries that dedicate processors [17], [18] or use separate threads [18], [19] for handling parallel I/O operations, our FP approach for multicores directly targets the on-chip parallel computation efficiency problem, and presents a more general and versatile service model for balanced utilization of the increasing number of cores.

Several research efforts have also advocated a pipelined model — that assigns various computational tasks of an application to different cores — for homogeneous [29]–[32] and heterogeneous [33]–[36] systems for parallelizing applications. In contrast, FP is a novel runtime environment for core allocation and, in this

paper, we highlight its use in servicing I/O and the compute-intensive tasks related to it in order to achieve high overall system throughput.

This paper develops checkpointing and the compute-intensive operations surrounding it as sample services to be provided by dedicated core(s). Our implementation focuses on utilizing aggregated memory/SSD spaces, however, FP is general enough for incorporating other peer-checkpointing schemes, such as [37], [38].

Recently, supercomputers are being equipped with SSDs (e.g., Gordon [21]). Besides illustrating one potential mechanism of utilizing node attached SSDs, FP aggressively overlaps SSD accesses with computation, which may help in hiding from the user the SSD performance variance problem revealed by a recent study characterizing scientific I/O workloads on SSDs [39].

III. FP: Rationale and Environment

The advent of multicores implies that data production rates of computer systems are fast surpassing the consumption rates of the associated storage systems, thus creating a fundamental imbalance between the two. As applications try to scale to tens of cores within a single node, the storage-compute performance gap leads to I/O bottleneck. In essence, even though cores are available, they may not yield expected performance benefits.

There are numerous application activities, e.g., checkpointing, file reformatting, etc., in a typical application workflow that can benefit if just a few cores were exclusively allocated for the activities. Prior work in this area has often been relegated to application-specific solutions of running a few support threads on cores. Instead, we create a generic runtime environment for dedicating a portion of the cores allotted to an application towards application activities besides computation.

Partitioning Cores: To handle the aforementioned application activities, we specialize cores by assigning to them specific sets of functionality. The analogy being: just as a large supercomputer has compute, I/O and service nodes for different functions, we enable a "system on chip"-like design by partitioning the cores (within a node) based on their functionality, e.g., compute cores, checkpoint cores, shadow-computation cores, verification cores, etc. To achieve such a holistic solution, however, we need a sophisticated runtime environment for many-core systems. We argue that such an outlook brings a novel perspective to current multicore research, the vast majority of which is focused only on raw computational scaling of applications. Instead, we aim to *achieve a concerted use of the thousands of cores available to an application on a whole gamut of application activities.*

Functionally partitioning cores to conduct assist tasks in-situ, with the application, can be compared against *partitioning in time* approaches that schedule operations in a performance-optimal manner. FP offers several advantages in terms of programmability, transparency and simplicity. In contrast, partitioning in time does reduce resource contention, which might be an issue with functionally partitioned cores (e.g., contention for memory resources). However, the storage wall is still a looming issue with the time-partitioning model, which is a critical problem that we strive to alleviate with FP.

A. Runtime Environment for FP

FP is supported through a dynamic runtime component comprising a suite of application support services. We adopt a bottom-up approach and construct several sample data-oriented services, which illustrate the concrete steps in building and enabling application support services. They also serve as proof-of-concept case studies to evaluate the effectiveness of FP.

Realizing the Runtime: The support services are run as "auxiliary applications" (aux-apps) on the dedicated cores (co-located with the main applications). An aux-app monitors its associated main application and transparently performs the support tasks, e.g., an aux-app can be used to create an aggregate distributed storage device (using SSDs) for checkpointing. Moreover, the aux-apps do not have to run on each node; multiple aux-apps can be aggregated on specialized nodes from where they can monitor applications on other nodes.

The first step towards realizing the FP runtime is to provide an interface to the application writers for specifying what aux-apps should run along with the main application. The job submission script is the logical place where such information can be specified. Note that the scripts are used to specify the location of aux-apps as well as to invoke implemented aux-apps, and are not means for implementing the aux-apps.

The runtime utilizes a FUSE [40] enabled *driver component* that exports a mountpoint to allow interactions between the main application and the aux-apps. The driver supports an interface, the *aux-app API*, through which the aux-apps can be invoked on the data accessed by the application through the mountpoint. The aux-apps are thus implemented as pluggable modules behind the FUSE-based driver component. A number of standard aux-apps, e.g., checkpoint management, etc., are provided by the runtime, and customized aux-apps can be developed by the application programmer using the aux-app API. Consequently, the aux-app operation is transparent to the main application during execution, as it simply accesses data through the mountpoint. This ensures that the design and development of aux-apps is decoupled from that of the main application.

For each aux-app the driver creates a separate thread (or a set of threads), which is then scheduled on the dedicated core(s). The aux-app approach also supports

```
int aux_app_write (void * output_buffer, int size){
  int result=SUCCESS;
  //process output in chunks
  while((chunk=get_chunk(&out_buffer,size))!=null){
    // compute hash on output_buffer chunks
    char* hash=sha1(chunk);

    //write the new chunk
    if(!hashtable_get(hash))
      result=data_write(chunk);

    // update de-dup hash-table
    hashtable_update(&result,chunk,hash);
  }
  return result;
}
```

Fig. 2. Example *write* extension for a checkpointing aux-app.

advanced usage scenarios, e.g., aux-apps from different nodes can work together to provide I/O aggregation across nodes for reducing load on secondary storage.

Consider an aux-app for de-duplicating checkpoint data. The user specifies the aux-app in the job submission script, which is then executed as a thread by the FUSE-based driver component on application execution. When the application writes to our mountpoint, the de-duplication function (Figure 2) is invoked on the data. Note that although only the *write* function is shown, all I/O functions supported by the FUSE API are supported by the aux-app API.

Discussion: Our current implementation of the runtime supports a static partitioning of cores, which means that core allocation to aux-apps cannot change during an application run. Such dynamic re-allocation can be useful, as we illustrate later in our evaluation, if an aux-app can benefit from more cores or if an application can make sufficient progress without an additional core. The need for such flexibility is also warranted from the usability aspect of the runtime. For example, how will a user know what is an optimal partitioning? The ideal solution would be to start with a conservative partition and then to let the runtime provision the allocation based on an agreed upon progress metric. The advantage of the aux-app approach is that it provides the basis upon which such a dynamic provisioning of cores can be built. While the number of cores used by the compute component of an application is typically fixed, the functionally partitioned cores can be used to support multiple services as required to improve the application performance. For example, if two cores are available for running the aux-apps, dynamic provisioning may use one core for de-duplication and data compression and another for checkpointing, instead of using both for checkpointing. Or it may even run an additional format transformation service that shares the available cores with other services. The goal being to improve overall application performance. Such a dynamic approach will allow the aux-apps to adjust to the application demands while keeping the performance impact to a minimum. In this paper we present and evaluate static FP, and identify the need for future work on dynamic provisioning.

Another aspect that impacts the utility of FP is whether an application is designed to utilize all cores available to it, i.e., no dedicated cores are available for aux-apps. We argue that application writers should examine end-to-end performance, i.e., all tasks and not just compute, and decide what is best for their applications. Nonetheless, based on our experience with user allocation requests, such a scenario is highly unlikely as users, without exception, over-provision cores, which can then be used by aux-apps.

In summary, the runtime provides for flexibly synthesizing application support services and dynamically using the allocated cores to improve end-to-end application performance, and not just raw compute performance.

IV. Sample Core Services

In this section, we present several sample services that can be performed by dedicated cores. Note that a single service may be carried out by multiple cores if necessary, conversely, multiple services can be assigned to a single core. We illustrate dedicating cores to application tasks using a checkpointing service. Checkpointing and its associated tasks such as data draining, de-duplication, and format transformation provide an insightful case study for the application I/O activities that can be expedited using the functional partitioning runtime. We demonstrate the benefits of performing each of these operations in-situ by the aux-apps. This is in contrast to the extant approach of conducting such operations in an offline manner, which exacerbates the storage wall issue due to the constant writing and re-reading of TBs of data from secondary storage.

A. SSD-based Checkpointing

Why Checkpointing? Checkpointing is an important data operation routinely performed by parallel applications, both for fault tolerance and for user-initiated execution restart. Checkpointing is becoming increasingly expensive relative to computation, especially for large-scale jobs, and is a key consideration in designing supercomputers. For example, the Argonne Intrepid BG/P supercomputer was designed not to meet balanced machine criteria (a bit of I/O per second per instruction per second), but rather to be able to dump the contents of the entire system memory to secondary storage in 30 minutes [41]. This makes checkpointing an appealing candidate for being "outsourced" to spare cores that cannot further help towards improving the end-to-end application execution time.

Rather than simply handing the periodic checkpointing I/O to dedicated cores to store in their associated

memory (an approach exploited previously on dedicated processors within an SMP box [18]), in this paper we explore checkpointing using non-volatile memories. One can argue that checkpointing is I/O intensive and, therefore, does not benefit from a dedicated checkpoint core. We counter this hypothesis with the observation that checkpointing can benefit from a host of other operations, such as de-duplication and compression (both of which are compute-intensive), draining and format transformation, which when performed in-situ through FP can alleviate the storage wall problem mentioned earlier. Consequently, the checkpointing service core, in this work, can perform more sophisticated tasks beyond just performing background I/O from the main memory to PFS, which is what most applications are currently faced with and stymied by. Another question that arises in this context is whether the task that would have been performed on the dedicated core can be run on the same core as the application. This is not possible even when the application is performing I/O, unless the application is modified to run a helper thread that is pinned to another spare core. However, Leadership machines, such as Jaguar, are not designed to be time-sharing systems and an application's allocation of cores is its own for the entirety of the run. For these reasons, the checkpointing service can benefit from the FP runtime.

Why SSDs? Among the several types of non-volatile memories, flash memory-based SSDs are gaining popularity for persistent data storage. SSDs offer a number of benefits over the conventional mechanical disks, such as fast data access time, low power consumption, light weight, higher resilience to external shocks and high temperatures. SSDs become especially helpful as the system memory bandwidth, as a function of computation throughput (byte/FLOP), has been consistently dropping in the Top500 supercomputers over the past decade. In fact, the ratio for a 2018 exascale machine is projected to be 0.01, which has been explicitly identified by DOE as one of the chief exascale problems to be addressed [42]. Furthermore, the non-volatile nature of SSDs can help provide intermediate storage for checkpointing data and reduce disk I/O (and communication) load.

Sustainability of SSD-based checkpointing: To justify the use of SSDs for HPC checkpointing — a write-intensive and write-once workload — one must address two inter-dependent concerns of *cost* and *durability*. This is because there is a significant $/GB difference between current SSDs and hard disks, and the number of erase cycles supported on the SSDs is fixed (limited compared to the hard disks). To check whether it is feasible to use SSDs for HPC checkpointing, we examine production run checkpointing characteristics of seven leadership-class DOE applications on the ORNL Jaguar machine (Table I). At the time of the runs, Jaguar had four quad-core processors (16 cores/node). Based on this data, and the specification of the SSD we have used in our study, Intel-X25 SLC SSD (Table III), we estimate *how many compute nodes can share a single 32 GB SSD*, under the following constraints:

TABLE I
CHECKPOINT SIZES AND ESTIMATED NODES/SSD[1].

Appl.	Data size (MB/Core)	Data size (MB/Node)	# Nodes/SSD	
			C_1	C_2
GTC	180	2880	21	11
XGC1	120	920	31	16
GTS	220	3520	17	9
Chimera	10	160	380	200
S3D	14	224	271	142
GEM	20	320	190	100
M3D-k	14	224	271	142

- C_1 – lifetime: SSD's lifetime should be at least 5 years. This is based on the replacement cycle of hard disks in HPC setups and data centers (typically 3-5 years) [43]. Each SLC memory cell typically has a lifetime of 100 K–1 M erase operations [44]. Assuming each block can sustain the conservative 100 K erase cycles and that an update of 1 block (256 KB) results in one block erase under perfect wear-leveling scheme [45], the lifetime of an SSD (F) for a given workload is calculated based on the following Equation (1) [45]:

$$Lifetime(F) = \frac{Size\ of\ NAND\ flash \times Erase\ cycles(\#)}{Bytes\ written\ per\ day} \quad (1)$$

- C_2 – capacity: To allow lazy draining of data to secondary storage, each SSD should be large enough to hold one complete checkpoint from the nodes sharing the device.

Table I shows the checkpoint characteristics for the applications considered, as well as the estimated number of nodes that can share a single SSD under C_1 and C_2. There are two key observations that can be made from the table. First, the lifetime is not the limiting constraint for sharing an SSD, rather the limit is determined by the SSD capacity. Second, even with both the constraints, a large number of nodes can share a single SSD. This is promising, as it indicates that an SSD-based checkpointing solution can be economically feasible.

Checkpointing Architecture: In our design, compute nodes contribute one or more cores and their associated SSDs as *checkpoint cores* to construct an aggregate, distributed checkpoint device. Each *checkpoint core* runs a benefactor process that contributes available, node-local SSD space (or a partition of it) to a *manager process* (running on one of the participating nodes) that aggregates such distributed SSD spaces and presents a

[1]We thank Scott Klasky for providing us with application data sizes.

collective intermediate storage device to checkpointing clients. Management tasks (such as benefactor status monitoring, space mapping, and data striping) can be done in a similar way as in existing storage aggregation systems [46]–[48]. For each checkpoint, the manager also maintains a *striping map* that contains information about where all the different parts (chunks) of the checkpoint are stored.

The aggregate SSD storage is made available to clients via a transparent file system mount point, */AggregateSSDstore*, using FUSE [40] as discussed in Section III. Here, we leverage our prior work on mounting an aggregate storage of node-local disks [48]. An application core that checkpoints data to the mount point will be redirected to the aggregate SSD storage, without requiring any other code modification.

While a single compute node's local memory is likely to be much smaller than its local SSD, checkpointing to aggregated SSD space from multiple nodes has several advantages. First, it provides fault tolerance in the event of compute node failure, which may render the persistent SSD-resident checkpoint data inaccessible. The globally accessible aggregate storage space also facilitates easy replication, e.g., using a simple copy in the aggregate space, of the individual node's checkpoint (or chunks of striped checkpoints) across multiple nodes, which would otherwise be complex, visible to the application, and cumbersome if nodes managed their associated SSDs individually. Second, when the SSDs are distributed across a set of system nodes, aggregation and access through a file system mount point offers an elegant abstraction to transparently access them from the numerous compute nodes, thus decoupling the placement of SSDs from the compute nodes and allowing for sharing of SSDs across multiple nodes. Finally, although we expect the growth in memory sizes to be matched with proportional growth in SSD space on all nodes, even if that is not the case and there is an imbalance, this abstraction allows for data to be striped over to other node-local SSDs. For example, the 512-core DASH system [49] at SDSC (the precursor to 8192-core Gordon system) is equipped with 4 TB of flash storage, compared to its 3 TB of DRAM. Currently, a high-end Fusion I/O PCIe MLC SSD card (io Drive Duo) at 640 GB is priced around $15K. Much like disk storage, SSD storage is increasing in capacity and decreasing in cost. Thus, growth in SSD space is currently outpacing memory increases.

B. De-duplication

Another sample data service function is de-duplication, which is used to identify and store unique data copies. For HPC applications, this service is useful when used in conjunction with checkpointing, to detect the similarity between two successive checkpoint images and only store their dissimilar parts. Such incremental checkpointing techniques have been explored earlier [48], [50]. The challenge in doing de-duplication on dedicated cores simultaneously with the main computation is to avoid significant memory contention.

To this end, we have built a service that computes hashes of the checkpoint data and identifies and removes duplicates. A dedicated de-duplication core (a checkpoint core can also double as a de-duplication core) is assigned the task of computing the chunk hashes that are then stored as metadata for that particular dataset at the manager (in checkpoint architecture above). When the checkpoint image for the next timestep, t, is to be written, the chunk hashes from $(t-1)$ are compared against the new, incoming image. A matching chunk hash indicates a duplicate and the chunk is not written, only the checkpoint's striping map is updated to point to the previously stored chunk. Consequently, depending on the degree of similarity between two successive checkpoints, the size of the checkpoint data written and the time to write it can be significantly reduced.

C. Format Transformation

Another potential data service for using dedicated cores is *file format transformation*. Large-scale parallel scientific simulations (and subsequent analysis/visualization tools processing their computation results) do not read/write data in plain binary formats. Instead, they often use high-level I/O libraries to create and access data in special scientific data formats. Intermediate checkpoint snapshot data is also saved in a specific data format so that it can be used by applications as a restart file in case of failure. Well-adopted formats, such as HDF5 [51] and netCDF [52], produce self-explanatory and self-contained files, with support for binary portability. However, accessing these files, especially through their parallel interfaces, has been substantially slower than reading/writing binary files [18], [53]. Checkpointing, while already cumbersome due to the storage wall, is often further stymied due to the need of being in proper scientific data formats. Recently, researchers have exploited dedicated data service nodes to form staging areas, where output data can be dumped in internal, faster formats, then asynchronously converted to HDF5 files on hard disks, producing a significant I/O performance improvement [54]. With dedicated cores, similar format transformation can be performed, especially with the SSD-based storage layer. As existing intermediate file formats (such as BP [53]) dump data in quite manageable units, the format transformation core can easily perform the conversion in a streaming manner. This reduces the memory requirement and performance perturbation to the computation and other concurrent data tasks running on the other cores. It helps to perform such format

conversions while the data is in transit (either in memory or SSD) and has not yet reached secondary storage. Further, performing conversion operations on each core within the compute node makes the data deluge more manageable. If the entire checkpoint or result snapshot is written as binary data to disk and format conversion is performed offline, the entire workflow suffers from constant re-reading of data. Offline format conversion also means that if a failure occurred right after a checkpoint, a valid restart file may not be ready yet, which wastes significant resources and delays job turnaround.

D. Adaptive Checkpoint Data Draining

Although an SSD can store data persistently, and its capacity will typically be manifold compared to node-local memory, large-scale, long-running jobs can generate overwhelming volumes of data that results in space on the SSD running out. This is especially true when not every node has an SSD attached to it, instead only a select set of system nodes has SSDs (due to budget concerns for example). Fortunately, checkpointing for fault tolerance does not require keeping all checkpoints: typically files are overwritten and saving up to two most recent checkpoints is enough. However, writing checkpoints to secondary storage supported via a parallel file system may still be needed: for some applications, checkpoint data doubles as result data for future analysis/visualization, or needs to be saved for elective restart. Even though draining is I/O bound, it cannot be done offline as the aggregate SSD space needs to be vacated for future checkpoint data. Growing memory size and the resulting increasing checkpoint size further stresses the need for in-situ data draining. With a checkpointing core, issues arising from growing memory sizes can be mitigated, and draining from SSDs to the secondary storage can be done in flexible and intelligent ways. As draining is I/O bound, it can be overlapped with other CPU bound checkpoint assist tasks.

The checkpointing core may decide to drain once every k checkpoints, in addition to maintaining the two most recent ones. The parameter k may even be configured and coordinated at runtime, through additional monitoring functions performed by the checkpointing core (such as watching the client checkpoint frequency). When the compute cores are back in the next computation phase, the checkpoint cores can collectively and lazily drain selected checkpoints to secondary storage. To enable this, the runtime supplies the aux-apps on different nodes with the location of the manager process. The system uses a soft-state protocol, where the aux-apps periodically announce their availability and sharing preferences, e.g., available SSD space, to the manager using *keep alive* updates. This not only allows the aux-apps to locate and communicate with each other, but also provides them with flexibility to change their preferences over time. The manager uses this information to instruct the aux-apps about the secondary storage location to where the checkpoints from the SSDs can be drained.

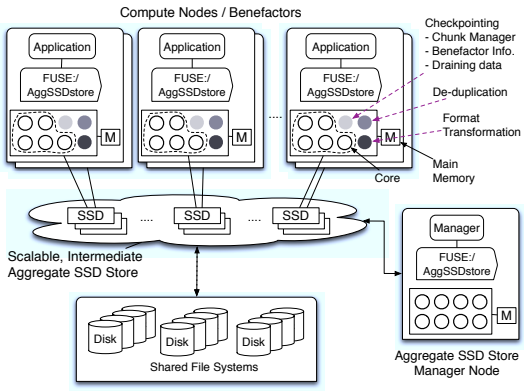

Fig. 3. High level FP system architecture.

V. Implementation

As a proof-of-concept, we have used the FP runtime to implement SSD-based checkpointing and various support services discussed in the previous section, using about 22.1 K lines of C code.

Figure 3 shows the components of our software that is run at the manager, benefactors, and clients. Note that every client node can also be a benefactor if it decides to provide its associated SSD to the aggregate SSD store. An example distribution of cores is also shown, where the white cores are used for computation and the shaded cores run aux-apps for services such as data draining and de-duplication. The manager (running on a separate node) works with the compute nodes to create a virtual aggregated SSD store, which serves as a transparent interface to the distributed SSDs. It also supports lazy draining to the very large but slower secondary disk-based storage. The aux-apps coordinate with each other across nodes using socket communication, and remain transparent to the main application.

The sequence of events when an application checkpoints is as follows. Upon receiving a request to write the checkpoint data, the FUSE module invokes the client component, which interacts with the manager to determine the benefactor that will handle the checkpoint for the client. If the client is on a node that has an associated SSD it is given preference and is utilized. The exception to this is if the local SSD is out of storage space, when a remote benefactor is chosen. Regardless of whether the client and benefactor are on the same node, the client directly contacts the benefactor to determine its current availability, divides the checkpoint data into fixed-size chunks, and transfers the chunks to the benefactor. The

TABLE II
TESTBED CONFIGURATION.

# of processing nodes	20
Capacity of storage server	2 TB
Network Interconnect	Infiniband QD 40 Gbit/s
HDD model	WD3200AAJS SATAII
Bandwidth	85 MB/s
Capacity	320 GB
Cores per node	8
Memory per node	8 GB
Max. cores available	160

TABLE III
SSD SPECIFICATIONS OF INTEL X25-E [56].

Model	Intel X25-E Extreme
Features	SATA-II SLC Flash Technology
Capacity	32GB
Bandwidth	Sequential read: 250 MB/s Sequential write: 175 MB/s
I/O Per Second	Random 4KB reads: >35K IOPS Random 4KB writes: >3.3K IOPS

benefactor stores the data on its associated SSD. Asynchronously, as discussed in Section IV the benefactor may drain data from the SSD to the secondary storage system. Once the checkpoint is complete, the benefactors inform the manager. The manager can then also invoke a merge component on the benefactors, which reads the checkpoint chunks from the secondary storage system and rearranges them into a merged checkpoint file, ready to be used by standard restart mechanisms if needed.

Finally, we have also built the checkpoint data manipulation services as discussed in Section IV, such as basic data draining, replication, and de-duplication.

VI. EVALUATION

In this section, we evaluate our implementation of FP and study its impact on application performance.

A. Methodology

Testbed Setup: Table II shows the configuration of our testbed, which uses 20 nodes from the systemG machine at Virginia Tech. All of the participating nodes are identical and run Linux Kernel 2.6.27.10. Each node is also equipped with an emulated SSD that has been validated against a real product (Table III) for sequential I/O throughputs within an error margin of 0.57%. The device uses DRAM for storage and emulates a real SSD by introducing artificial delays [55].

Our setup is not equipped with a PFS, so we used node-local disks for checkpoint data. While typical HPC setups do not employ node-local disks, we use them as a high-throughput alternative to an NFS server.

Workloads: We employ a real-world astrophysics simulation code, FLASH [15], [16], which generates checkpoint files in HDF5. We modified the Sod 3D version of FLASH for our evaluation: all the compute processes carry out parallel I/O, including checkpointing, using MPI-IO. The problem size remains fixed as the number of compute processes is increased. For more detailed testing, we also use a synthetic benchmark, which is a simple checkpoint application that generates same sized checkpoint data every barrier step. Specifically, we created an MPI program with 160 processes, each writing 0.25 GB of data per checkpoint, thus creating a total checkpoint of 40 GB per barrier step. Finally, we use static functional partitioning for the experiments.

B. Impact of FP

In our first set of experiments, we determine the impact of FP on overall application performance. We use the notation, $FP(X,Y)$, to denote a setup with a total of Y cores per node of which X have been functionally partitioned for support services. For this test, we use FLASH with a checkpoint size of 6.8 GB. No format transformation is performed on the checkpoint data. We consider four cases. (i) *Local disk non-FP(0,8)*: The baseline performance where all 8 cores per node are used for application computation. Checkpoint data from all the 8 cores in every node is written to the local disk on that compute node. (ii) *Local disk non-FP(0,7)*: Repeat (i), but with only 7 cores per node for application computation and the remaining core is left idle. All the 7 cores per node write to the local disk on that compute node. (iii) *Aggregate disk FP(1,8)*: FP where 1 core out of 8 on each node is used as a dedicated checkpoint core. An aux-app is run on these dedicated cores, which assists with checkpointing and its associated tasks. In this case, FP allows us to build sophisticated structures, such as an aggregate distributed store of node-local disks by pooling the aux-app services on each checkpoint core as explained in Sections IV and V. The 7 cores per node checkpoint to this aggregated storage, which stripes the data in parallel to distributed aux-app services. (iv) *Aggregate SSD FP(1,8)*: Similar to (iii), but with checkpointing to aggregate SSD storage.

Figure 4 shows the result for 80 and 160 cores, under strong scaling. First, we observe that removing a core from the computation, *local disk non-FP(0,7)*, does not affect the overall performance significantly; in fact the 2.23% average difference between that and *local disk non-FP(0,8)* is within the error margin. Note that the small increase in execution time from 80 to 160 cores is due to contention in our testbed. Moreover, it can be observed that dedicating one core to handle the checkpoint, *aggregate disk FP(1,8)*, can improve the execution time by 15.42% and 27.05% for 80 and 160 cores, respectively. The benefit is also in part due to the ability to write to an aggregate store of node-local disks, pooled from the aux-apps on the dedicated checkpoint cores. Thus, FP is a viable approach and it also lets us build rich

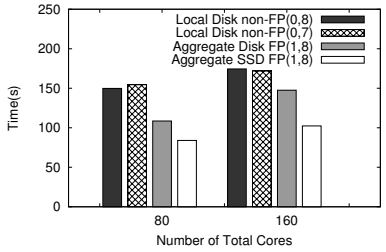

Fig. 4. Impact of FP on execution time.

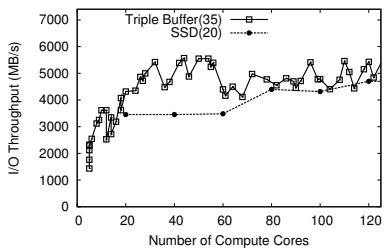

Fig. 5. Memory vs. SSD I/O rate.

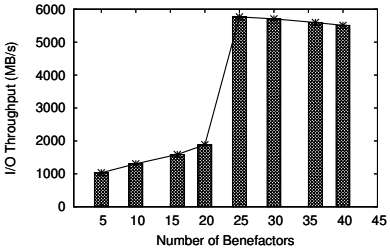

Fig. 6. Impact of varying benefactors.

composite services that are much beneficial. Moreover, the use of *aggregate SSD FP(1,8)* provides an additional gain of 43.95% and 41.34% compared to checkpointing on disks for 80 and 160 cores, respectively. This is promising as SSDs can enable efficient checkpointing in HPC setups that do not have node-local disks.

C. Checkpointing to Memory versus SSD Storage

In the next set of experiments, we compare SSD-based checkpointing with our previous work on in-memory checkpointing that aggregates memory buffers across nodes and employs triple-buffering to provide improved throughput. The workload comprises of a synthetic benchmark with up to 125 clients, each writing a checkpoint of 0.25 GB. For SSD-based checkpointing, each node has an SSD, thus the number of aux-apps (run as benefactors) for this case is 20, "SSD(20)". The number of benefactors for in-memory triple buffer is set to 35, "Triple Buffer(35)", which is large enough to avoid draining the data to secondary storage, and gives the best case performance of in-memory checkpointing.

Figure 5 shows the results. As the number of clients increases, the benefactor utilization increases and so does the sustained throughput. We also observe that I/O throughput for checkpointing to an aggregate SSD device is 14.7% lower compared to the in-memory technique, which is obvious. Nonetheless, our SSD-based checkpointing does not require the dedicated cores to give up their memory, which may reduce application performance, as can happen for in-memory checkpointing.

Next, we determine the effect of the number of benefactors on sustained checkpoint I/O throughput. We fix the client cores to 120 and vary the benefactors from 5 to 40. We perform in-memory checkpointing. Figure 6 shows that the I/O throughput does not increase beyond 25 benefactors and there is even a slight decrease, implying the futility of simply adding more benefactors as it takes cores away from computation.

1) Varying number of application processes: The next experiment limits the number of in-memory benefactors and compares its throughput against checkpointing to SSD. We repeat the experiment of SSD(20) and compare against in-memory triple buffering, but with only 20 benefactors, "Triple Buffer(20)". We use a maximum of 140 compute cores. Figure 7 shows that the initial performance of Triple Buffer(20) is similar to Triple Buffer(35). However, as the client cores are increased beyond 80, the benefactor buffers fill up faster than they can be drained, causing a significant throughput reduction. As the compute cores are further increased, the bandwidth eventually reduces to almost that of a direct checkpoint to disk, as the *draining agent* has now become I/O bound. Note that the steps pattern on the tail of the graph is an artifact of how clients and benefactors are distributed in our test. In contrast, SSD-based checkpointing achieves better throughput with increasing number of clients simply due to larger available space (32 GB SSD versus 1 GB memory aggregated).

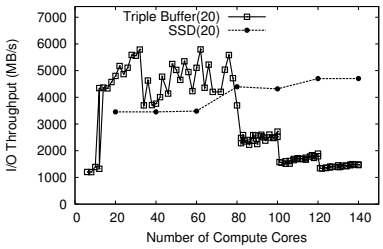

Fig. 7. Impact of insufficient benefactors.

2) Varying number of SSD benefactors: In this experiment, we study the impact of varying the number of SSD benefactors. Figure 8 shows the results. As expected, more SSD benefactors result in better I/O throughput. However, once a sufficient number of benefactors were available, i.e., $>= 5$, the overall throughput did not change much. A remarkable coincidence is that I/O throughput is limited by the available bandwidth of SSDs, and not the number of SSD benefactors (beyond a certain number, i.e. 5 in this case). Conversely, unless the proper number of SSD benefactors is available, the checkpoint nodes can be a bottleneck for the entire system performance in the worst case.

D. De-duplication of Checkpointing Data

We have seen the benefit of FP on reducing overall execution time. In the following, we observe how of-

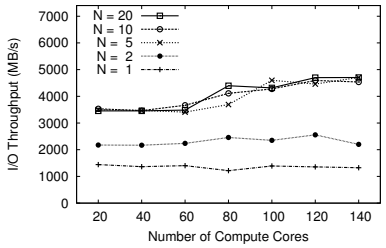

Fig. 8. I/O rates with 'N' benefactors.

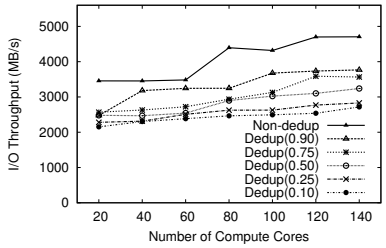

Fig. 9. Checkpoint I/O rates with FP(1,8).

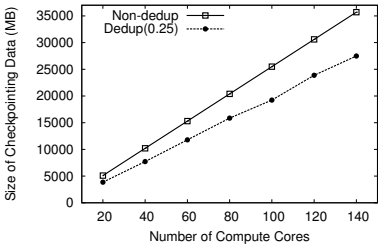

Fig. 10. De-dup data size with FP(1,8).

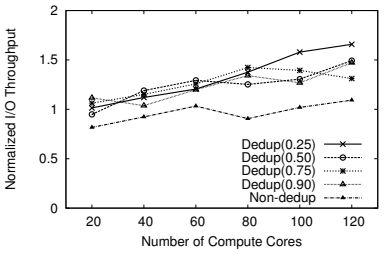

Fig. 11. Impact of increasing FP cores normalized to FP(1,8).

floading different application activities to dedicated cores can further improve overall application performance.

For this experiment, in particular, we consider the benefits of de-duplicating across consecutive checkpoints in reducing the amount of data that needs to be written. We use our synthetic benchmark program with one core on each node dedicated to de-duplication and checkpointing, and vary the number of compute cores from 20 to 140. We also observe the impact of varying the amount of de-duplication. Figure 9 shows the effective I/O rates for the checkpoint for different de-dup ratios (shown in parenthesis). Observe that although de-duplication reduces the amount of data that needs to be transferred, the I/O rates achieved for writing the checkpoint on average across the considered de-dup ratios are 30% lower than that without de-duplication. This is because of the compute-intensive nature of the de-duplication process and indicates that the service is under-provisioned (sharing one core with checkpointing). Figure 10 shows the amount of data transferred for a de-dup ratio of 0.25. As expected, de-duplication reduces the amount of data that needs to be written to the disk by 25% or as much as 8208 MB for 140 cores. Such decrease in the amount of checkpoint data to be written (which is all write I/Os) can help improve the lifetime of SSDs used for intermediate aggregated storage.

In the next experiment, we allocated two service cores, one bound for de-duplication and the other for checkpointing. The core services can be overlapped. Figure 11 shows the benefits of allocating more cores to support services by sacrificing computation resources. Overall, we see that with functional partitioning, FP(2,8) can provide higher throughput than FP(1,8). For instance, the I/O throughput is improved by about 60% with a de-duplication ratio of 0.25, when we use 120 cores for application computation. We speculate that these benefits are mainly obtained from pipelined effect of two cores executing in parallel. Finally, this experiment illustrates the need for dynamic and autonomic service core allocation and resource provisioning, as discussed earlier. We plan to pursue this as part of our future work.

VII. Conclusion

We have discussed FP of cores in large multicore systems to support different application activities, in contrast to the extant approach to allocating all cores to computation. We have applied FP to the critical problem of handling checkpoint I/O in supercomputers, where the large number of cores can result in a significant amount of application execution time spent in checkpointing. We have developed a flexible SSD-based checkpointing system that allows for transparent sharing of SSDs across different nodes, thus providing an economically viable solution. Our evaluation using a real implementation shows that our core allocation model is viable, and can provide significant benefits with minimal impact and even increase overall performance (dedicating 1 core on an oct-core machine for checkpointing can improve overall execution time of a FLASH benchmark on 80 and 160 cores by 43.95% and 41.34%, respectively). In summary, our work demonstrates FP's usability and in our future work, we will apply such partitioning to support other mission-critical application activities.

Acknowledgment

We are thankful to the anonymous reviewers and our shepherd, Dr. Toni Cortes, for their valuable feedback. We also thank Dr. John Cobb for several useful discussions. This work was sponsored in part by the LDRD program of ORNL, managed by UT-Battelle, LLC for the U.S. DOE (Contract No. DE-AC05-00OR22725), and by NSF grants CCF-0937827, CCF-0746832, CCF-0621470, and CCF-0937690, as well as Xiaosong Ma's joint appointment between ORNL and NCSU.

REFERENCES

[1] Intel. Advancing Multi-Core Technology into the Tera-scale Era, 2009. http://techresearch.intel.com/articles/Tera-Scale/1449.htm.

[2] Henry Monti, Ali R. Butt, and Sudharshan S. Vazhkudai. Timely offloading of result-data in hpc centers. In *Proc. ACM ICS*, 2008.

[3] Top500 supercomputer sites. http://www.top500.org/.

[4] Philip Schwan. Lustre: Building a File System for 1,000-node Clusters. In *Proc. Ottawa Linux Symposium*, 2003.

[5] Galen Shipman, Dave Dillow, Sarp Oral, and Feiyi Wang. The spider center wide file system: From concept to reality. In *Proc. Cray User Group*, 2009.

[6] Ami Marowka. Parallel computing on any desktop. *Commun. ACM*, 50(9):74–78, 2007.

[7] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[8] Nalini Vasudevan and Stephen A. Edwards. Celling shim: compiling deterministic concurrency to a heterogeneous multicore. In *Proc. ACM SAC*, 2009.

[9] Jack Dongarra. The impact of multicore on math software and exploiting single precision computing to obtain double precision results. In *Proc. ICPP*, 2006.

[10] Silas B. Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In *Proc.Usenix OSDI*, 2008.

[11] Herb Sutter and James Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.

[12] Duc Vianney, Gad Haber, Andre Heilper, and Marcel Zalmanovici. Performance analysis and visualization tools for cell/b.e. multicore environment. In *Proc. ACM IFMT*, 2008.

[13] Sain-Zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, and Wen-Mei W. Hwu. Cuda-lite: Reducing gpu programming complexity. In *Proc. LCPC*, 2008.

[14] Aaron E. Darling, Lucas Carey, and Wu-chun Feng. The design, implementation, and evaluation of mpiblast. In *Proc. ClusterWorld*, 2003.

[15] Anshu Dubey, Katie Antypas, Murali K. Ganapathy, Lynn B. Reid, Katherine Riley, Daniel J. Sheeler, Andrew Siegel, and Klaus Weide. Extensible component-based architecture for Flash, a massively parallel, multiphysics simulation code. *Parallel Computing*, 35(10-11):512–522, 2009.

[16] Robert Rosner, Alan Calder, Jonathan Dursi, Bruce Fryxell, Donald Q. Lamb, Jens C. Niemeyer, Kevin Olson, Paul Ricker, Frank X. Timmes, James W. Truran, Henry Tufo, Yuan-Nan Young, Michael Zingale, Ewing Lusk, and Rick Stevens. Flash code: Studying astrophysical thermonuclear flashes. *Computing in Science and Engineering (CSE)*, 2(2):33–41, 2000.

[17] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proc. SC*, 1995.

[18] X. Ma, J. Lee, and M. Winslett. High-level buffering for hiding periodic output cost in scientific simulations. *IEEE Transactions on Parallel and Distributed Systems*, 17(3):193–204, 2006.

[19] S. More, A. Choudhary, I. Foster, and M. Q. Xu. MTIO: a multi-threaded parallel I/O system. In *Proc. International Parallel Processing Symposium*, 1997.

[20] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas. Overview of the blue gene/l system architecture. *IBM Journal of Research and Development*, 49(2/3):195–212, 2005.

[21] Supercomputer uses flash to solve data-intensive problems 10 times faster. http://www.sdsc.edu/News%20Items/PR110409_gordon.html, 2009.

[22] Christos D. Antonopoulos, Filip Blagojevic, Andrey N. Chernikov, Nikos P. Chrisochoides, and Dimitrios S. Nikolopoulos. A multigrain delaunay mesh generation method for multicore smt-based architectures. *J. Parallel Distrib. Comput.*, 69(7):589–600, 2009.

[23] Ligang He, Stephen A. Jarvis, Daniel P. Spooner, and Graham R. Nudd. Dynamic scheduling of parallel real-time jobs by modelling spare capabilities in heterogeneous clusters. In *Proc. IEEE ICCC*, 2003.

[24] Jorge Manuel Gomes Barbosa and Belmiro Daniel Rodrigues Moreira. Dynamic job scheduling on heterogeneous clusters. In *Proc. IEEE ISPDC*, 2009.

[25] M. Mustafa Rafique, Ali R. Butt, and Dimitrios S. Nikolopoulos. Designing accelerator-based distributed systems for high performance. In *Proc. IEEE/ACM CCGrid*, 2010.

[26] M. Mustafa Rafique, Benjamin Rose, Ali R. Butt, and Dimitrios S. Nikolopoulos. Supporting mapreduce on large-scale asymmetric multi-core clusters. *SIGOPS Oper. Syst. Rev.*, 43(2):25–34, 2009.

[27] Shimin Chen, Babak Falsafi, Phillip B. Gibbons, Michael Kozuch, Todd C. Mowry, Radu Teodorescu, Anastassia Ailamaki, Limor Fix, Gregory R. Ganger, Bin Lin, and Steven W. Schlosser. Log-based architectures for general-purpose monitoring of deployed code. In *Proc. Architectural and System Support for Improving Software Dependability Workshop*, 2006.

[28] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. In *Proc. ACM ASPLOS*, 2008.

[29] Kue-Hwan Sihn, Baik Hyunki, Kim Jong-Tae, Bae Sehyun, and Song Hyo Jung. Novel approaches to parallel h.264 decoder on symmetric multicore systems. In *Proc. IEEE ICASSP*, 2009.

[30] S. Arash Ostadzadeh, Roel J. Meeuws, Kamana Sigdel, and Koen Bertels. A multipurpose clustering algorithm for task partitioning in multicore reconfigurable systems. In *Proc. IEEE CISIS*, 2009.

[31] Turgay Altilar and Yakup Paker. Minimum overhead data partitioning algorithms for parallel video processing. In *Proc. International Conference on Domain Decomposition Methods*, 2001.

[32] Robert Ennals, Sharp, and Mycroft. Task partitioning for multi-core network processors. In *Proc. Cluster Computing*, 2005.

[33] Marc de Kruijf and Karthikeyan Sankaralingam. MapReduce for the Cell B.E. Architecture. *IBM Journal of Research and Development*, 53(5):10:1–10:12, 2009.

[34] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *Proc. ACM PACT*, 2008.

[35] F.T. Hady, T. Bock, M. Cabot, J. Chu, J. Meinecke, K. Oliver, and W. Talarek. Platform level support for high throughput edge applications: the twin cities prototype. *IEEE Network*, 17(4):22–27, 2003.

[36] Sanjay Kumar, Gavrilovska, Karsten Schwan, and Srikanth Sundaragopalan. C-core: Using communication cores for high performance network services. In *Proc. IEEE NCA*, 2005.

[37] J. S. Plank, K. Li, and M. A. Puening. Diskless Checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, 1998.

[38] G. Bronevetsky and A. Moody. Scalable I/O Systems via Node-Local Storage: Approaching 1 TB/sec File I/O. LLNL Technical Report LLNL-TR-415791, Lawrence Livermore National Laboratory, 2009.

[39] S. Park and K. Shen. A Performance Evaluation of Scientific I/O Workloads on Flash-Based SSDs. In *Proc. Workshop IASDS*, 2009.

[40] FUSE. File System in Userspace, 2007. http://fuse.sourceforge.net/.

[41] Kamil Iskra, John W. Romein, Kazutomo Yoshii, and Peter H. Beckman. Zoid: I/o-forwarding infrastructure for petascale architectures. In *Proc. ACM SIGPLAN PPoPP*, 2008.

[42] International exascale software project roadmap. In *Proc. Cross-cutting Technologies for Computing at the Exascale Workshop*, 2010.

[43] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an mttf of 1,000,000 hours mean to you? In *Proc. USENIX FAST*, 2007.

[44] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proc. SIGMETRICS/Performance*, 2009.

[45] Numonyx. Wear leveling in nand flash memories. http://www.numonyx.com/Documents/Application%20Notes/AN1822.pdf.

[46] Ali R. Butt, Troy A. Johnson, Yili Zheng, and Y. Charlie Hu. Kosha: A peer-to-peer enhancement for the network file system. *Journal of Grid Computing: Special issue on Global and Peer-to-Peer Computing*, 4(3):323–341, 2006.

[47] S. Vazhkudai, X. Ma, V. Freeh, J. Strickland, N. Tammineedi, and S. Scott. Freeloader: Scavenging desktop storage resources for bulk, transient data. In *Proc. SC*, 2005.

[48] Samer Al-Kiswany, Matei Ripeanu, Sudharshan S. Vazhkudai, and Abdullah Gharaibeh. stdchk: A checkpoint storage system for desktop grid computing. In *Proc. ICDCS*, 2008.

[49] Dash User Guide: Technical Summary, June 2010. http://www.sdsc.edu/us/resources/dash/index.html.

[50] Eliezer Levy, Avi Silberschatz, and Avi Silberschatz. Incremental recovery in main memory database systems. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):529–540, 1992.

[51] http://hdf.ncsa.uiuc.edu/HDF5/doc/. *HDF5 - A New Generation of HDF*.

[52] http://www.unidata.ucar.edu/packages/netcdf/docs.html. *NetCDF Documentation*.

[53] Jay Lofstead, Fang Zheng, Scott Klasky, and Karsten Schwan. Adaptable, metadata rich io methods for portable high performance io. In *Proc. IPDPS*, 2009.

[54] Hasan Abbasi, Jay Lofstead, Fang Zheng, Scott Klasky, Karsten Schwan, and Matthew Wolf. Extending i/o through high performance data services. In *Proc. Cluster Computing*, 2009.

[55] Pavan Konanki and Ali R. Butt. An exploration of hybrid hard disk designs using an extensible simulator, 2008. Masters Thesis, Virginia Tech.

[56] Intel. Intel x25-e extreme sata solid-state drive. http://www.intel.com/design/flash/nand/extreme/index.htm.