Optimizing Latency and Throughput for Spawning Processes on Massively Multicore Processors

Abhishek Kulkarni Andrew Lumsdaine
Indiana University
Bloomington, IN 47405
{adkulkar, lums}@cs.indiana.edu

Michael Lang Latchesar Ionkov
Los Alamos National Laboratory
Los Alamos, NM 87544
{mlang, lionkov}@lanl.gov

ABSTRACT

The execution of a SPMD application involves running multiple instances of a process with possibly varying arguments. With the widespread adoption of massively multicore processors, there has been a focus towards harnessing the abundant compute resources effectively in a power-efficient manner. Although much work has been done towards optimizing distributed process launch using hierarchical techniques, there has been a void in studying the performance of spawning processes within a single node. Reducing the latency to spawn a new process locally results in faster global job launch. Further, emerging dynamic and resilient execution models are designed on the premise of maintaining process pools for fault isolation and launching several processes in a relatively shorter period of time. Optimizing the latency and throughput for spawning processes would help improve the overall performance of runtime systems, allow adaptive process-replication reliability and motivate the design and implementation of process management interfaces in future manycore operating systems.

In this paper, we study the several limiting factors for efficient spawning of processes on massively multicore architectures. We have developed a library to optimize launching multiple instances of the same executable. Our microbenchmarks show a 20-80% decrease in the process spawn time for multiple executables. We further discuss the effects of memory locality and propose NUMA-aware extensions to optimize launching processes with large memory-mapped segments including dynamic shared libraries. Finally, we describe vector operating system interfaces for spawning a batch of processes from a given executable on specific cores. Our results show a 50x speedup over the traditional method of launching new processes using `fork` and `exec` system calls.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*Threads*; D.4.8 [**Operating Systems**]: Performance—*Measurements*

General Terms

Design, Measurement, Performance

ROSS '12, June 29, 2012, Venice, Italy

Keywords

scalable process launch; many-core operating system interfaces; vector process spawning

1. INTRODUCTION

Emerging manycore general-purpose processors like the Intel Single Chip Cloud Computer (SCC), Intel Manycore Integrated Architecture (MIC) and the Tilera TILE-Gx family are packing an ever-increasing number of cores on a single silicon CPU chip. These processor chips typically pack between 64 to 128 low-power cores per chip and are expected to scale further in the future. As this results in higher performance at lower energy consumption, the trend of increasing core counts is making its way into leadership-class HPC computing systems. The Cielo system at LANL has 8944 compute nodes with a total of 143,104 cores. Exascale systems are estimated to have hundreds to thousands of cores in a single compute node within the decade[12]. Solutions to cross-cutting concerns such as resiliency, scalability and programmability are being investigated at multiple levels spanning new programming models and runtime systems. At this scale, it is requisite to turn our focus towards dynamic runtime systems, and to revisit some of the design choices of operating system services commonly used by these software runtime systems.

Efficient bootstrapping of applications on extreme-scale systems has been studied extensively [21, 1, 9]. Most of these investigations have focused on reducing the application startup time across the cluster. To launch application processes locally, these systems rely on the operating system to spawn a new process on its behalf. Until recently, launching large groups of processes locally has not posed a problem due to modest core counts. With hybrid HPC applications leveraging novel runtime systems to exploit intranode shared-memory parallelism, the latency and throughput of spawning batches of processes is becoming critical.

Typically, distributed runtime systems spawn $n \times p$ processes for a single parallel job where n is the number of allocated nodes with p processors each. Reducing the latency to spawn new processes locally results in faster global job launch. Even though it is normal for scientific applications to run for days and months, faster process spawning results in higher throughput. Emerging dynamic and resilient execution models are considering the feasibility of maintaining process pools for fault isolation. Task-parallel runtime systems launch several asymmetric tasks as processes in a relatively shorter period of time. In these runtime models higher throughput allows running many-task computing applications yielding a better overall performance.

The traditional approach to spawning new application processes in most Unix-based operating systems and its derivatives involves forking the parent process to create a new copy and then replacing

the copied child process image with a new process image loaded from a file. While this approach works fairly well, it results in poor spawn performance due to redundant operations executed serially and unpredictable behavior due to memory over-committing, especially when the system runs out of memory. The POSIX standard defines alternative process creation mechanisms like `posix_spawn` and `posix_spawnp` as a solution to the aforementioned problems. In Linux, however, these functions are implemented using a combination of `fork` and `exec` and face the above issues.

To launch a batch of processes across all cores in a massively multicore processor, new processes have to be *pinned* manually to the desired processor. To minimize the resource migration costs, the process is typically pinned to the core right after it is forked. We claim that the OS interface for spawning processes can be further optimized to account for spawning multiple instances at once, creating new processes directly from an executable image and actively launching processes on specific cores. The evolution of operating systems has been evolutionary largely dictated by the hardware they run on. With the emergence of massively parallel hardware, the scalar and synchronous nature of system calls is limiting with applications spending a significant time waiting for the operating system to serve its resource requests. Future manycore operating systems need to address this by either coalescing the system calls or exposing vector interfaces so that the OS can operate on a number of objects in parallel[23].

The remainder of the paper is organized as follows. Section 2 discusses the design and performance of the `pspawn` library to optimize the throughput of launching multiple instances of the same process. In Section 2.5, we measure the copy-on-write (CoW) kernel overhead and how it affects processes spawned using `libpspawn`. Section 3 propose NUMA-aware optimizations to process execution and assesses the benefits on a synthetic benchmark. The vector system-call interface is described in Section 4 discussing the ramifications and design choices to be considered when implementing such interfaces. Finally, the related and future work is discussed in sections 5 and 6 respectively.

2. THE PSPAWN LIBRARY

2.1 Overview

Intranode job launch is traditionally done by spawning new processes using a combination of the `fork` and `exec` system calls sequentially. A typical control graph of the traditional method of launching new processes is shown in Figure 1(a). In this method, a launcher (parent process) first forks itself into two independent threads of execution. The child process later replaces its existing process image with a new process image. This is done repeatedly to launch multiple instances of the same executable. The overhead involved in setting up a new address space after `fork` and then replacing it with a new one after `exec` adds up when launching a large number of processes. Besides, it also turns out to be expensive for launches across several distinct NUMA domains. Repeatedly calling the `exec` system call involves reading the same binary off the disk. Since every subsequent call after the first one is typically served by the operating system's page-cache, the penalty of reading the whole file from the storage again is not incurred in practice. However, since the page-cache in Linux kernel is interleaved, a large number of spawns eventually end up crossing their memory domain to fetch the data required for their execution.

Arguably, for HPC jobs the launch time could be insignificant compared to the running time which runs into days and even months. However, with the increasing benefits offered by adaptive dynamic runtimes, optimizing the throughput of process launch would facilitate job resiliency by on-demand spawning of shadow processes. Fast intranode process launch would eventually also result in fast inter-node job launches since launched processes almost always rendezvous after being launched. In cases where several instances of the same executable is to spawned, it is counterproductive to `fork` the launcher and load the same binary over and over again.

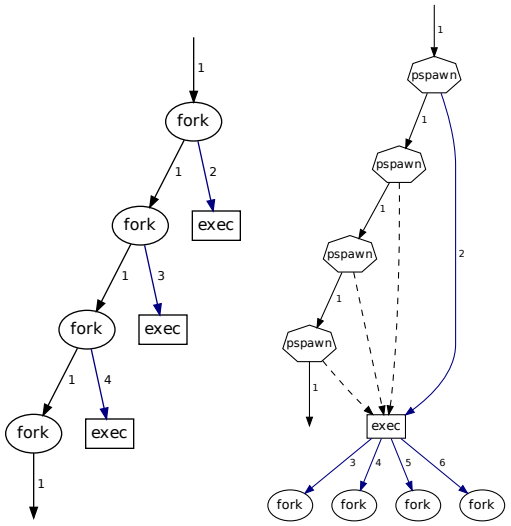


Figure 1: Control graphs for process spawn

An alternate way to efficiently launch multiple instances of the same executable is to load the process image once using `exec` and then `fork` it as many times as required. Forking is typically much cheaper compared to `exec`ing. Figure 1(b) shows the control graph for processes launched this way. As concluded in Section 2.4, this results in a significant speedup in launching multiple instances of the same executable under certain assumptions. There are two primary issues in doing this:

- Either the binary to be launched has to be modified to fork multiple instances of itself or a hook has to be inserted before the launched binary calls its `main` function.
- Multiple instances of the same executable can be launched with different arguments and different environment parameters. After forking multiple instances, the arguments and environment pointer variables (`argv` and `environp` respectively) need to be explicitly altered to point to the proper values.

To facilitate faster launching of executables and to overcome the two issues pointed above, we avoid putting the onus of modifying the launched binary on the user and have designed a library, `libpspawn`, that provides the necessary hooks to achieve this.

2.2 Design

To spawn multiple instances of a binary, a launcher application creates a new *pspawn context*. The context helps the library in locating the launched process (proxy) and its control channel. The proxy is essentially a frozen executable instance of the launched process listening for spawn requests. The proxy library is preloaded (using `LD_PRELOAD`) before launching the child process so that the `main` function of the launched process can be overridden. Before calling the `main` function, `libpspawn` first creates a control channel between the launcher and the proxy. Every subsequent

```
/* Create a new pspawn context */
int pspawn_context_create(pspawn_context_t *ctx,
                          char *path, int core);
/* Destroy a pspawn context */
void pspawn_context_destroy(pspawn_context_t *ctx);
/* Launch a single process */
pid_t pspawn(pspawn_context_t *ctx, char *argv[]);
/* Launch 'n' processes */
int pspawn_n(pid_t *pids, pspawn_context_t *ctx,
             int nspawns, char *argv[]);
```

Figure 2: The `libpspawn` API

`pspawn()` call results in a lightweight asynchronous IPC signal to the proxy asking it to fork itself. The `pspawn` call is, thus, a combination of `fork` and `exec` system calls with an additional directive to specify where to spawn the new process. Multiple `pspawn` calls to launch the same binary result in a fork avoiding the costs to `exec` the same binary again and again. pspawn is further optimized to do hierarchical intranode launches across NUMA domains to concurrently launch processes on all cores.

As shown in Figure 3, the launcher process and the proxy share a memory segment which is used to store the arguments and the environment to be passed to the launched processes. Once the launcher writes the argument to this shared buffer, it sends a real-time signal to the proxy commanding it to fork itself. The arguments and environment are set appropriately and all open file descriptors are closed before calling the `main` of the launched process. The process identifier (PID) of the launched process is sent back to the launcher using a real-time signal. `libpspawn` maintains semantics similar to `fork` and `exec` so that the modifications required to launchers are minimal.

In addition to the library interface, we have written a library which can be preloaded to transparently perform these optimizations to launchers following the traditional "repeated `fork` and `exec`" launching regime. A *pspawn context* is created when this library first encounters a `fork` and `exec` to a new executable. Every subsequent call to `fork` and `exec` referring to the same executable is intercepted and the launcher sends a signal to the launched process asking it to fork itself. The arguments and environment are set appropriately after the fork before calling the actual `main` function of the launched binary.

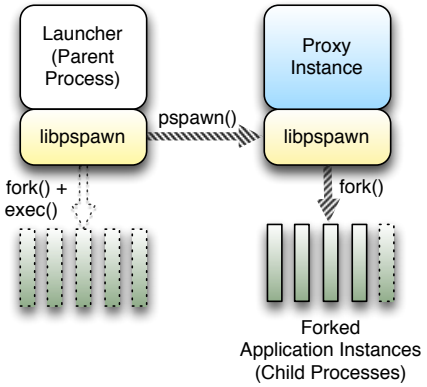


Figure 3: Spawning application instances with `libpspawn`

When two processes fork, they share the same address-space among other resources[1]. When processes launched using `libspawn` dirty their mapped pages, the kernel has to create a copy of the dirtied page. It uses copy-on-write to avoid the overhead involved in copying all virtual memory segments on `fork`. While the above approach is fairly beneficial for launching several short-lived processes, a measurable penalty is incurred for the copy-on-write overhead at higher core counts.

2.3 Experimental Setup

To evaluate the overhead of kernel operations at different core counts, we ran our tests on three machines with varying number of cores. The tests were run on

1. A quad-socket, quad-core (16 cores total, 4 NUMA domains) AMD Opteron node with 16GB of memory and running Linux kernel 2.6.32-220.13.1.el6.x86_64.

2. A two-socket, 12-core (24 cores total, 4 NUMA domains) AMD Istanbul node with 48GB of memory and running Linux kernel 3.4.0-rc7.

3. An eight-socket, 8-core (64 cores total, 8 NUMA domains) Intel Nehalem node with 128GB of memory and running Linux kernel 2.6.35.10-74.fc14.x86_64.

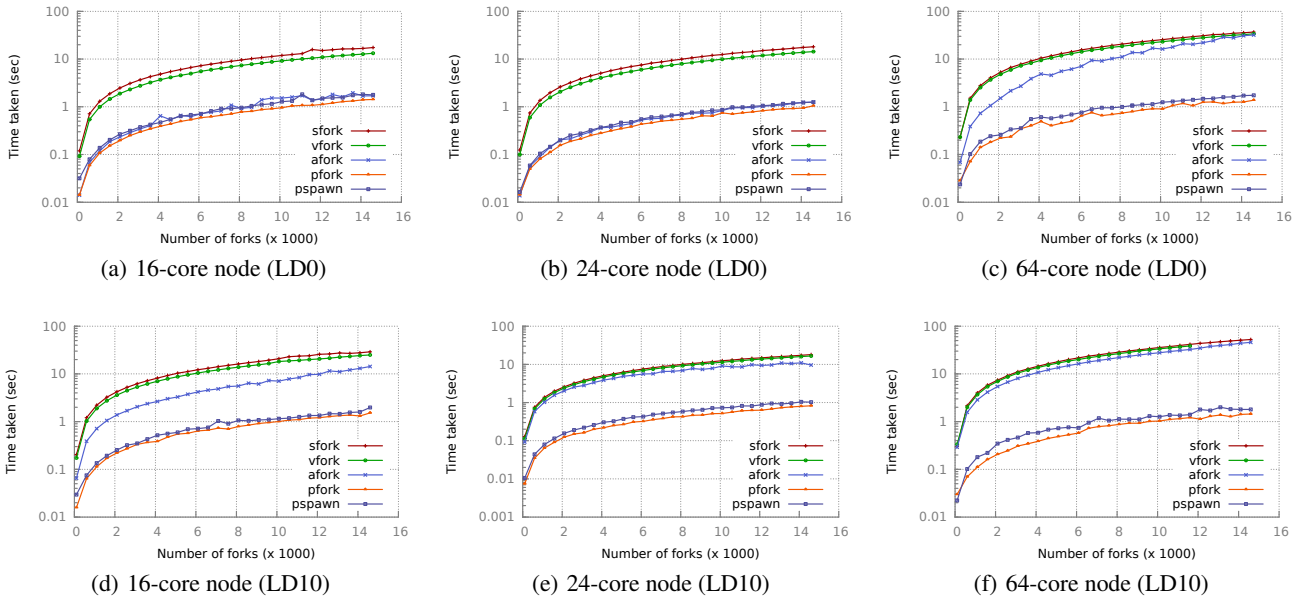
2.4 Performance Evaluation

In this section, we evaluate the performance of traditional process spawning and the approach described in the previous section, by running microbenchmarks that involve repeated `fork`ing and `exec`ing. Process launchers, resource managers and distributed runtimes like the MPI runtime use repeated `fork`ing and `exec`ing to launch children processes. In addition, they keep accounting information to monitor the progress of launched processes. Since the following microbenchmarks are similar in behavior to repeated launching of processes, they give a close estimate of the costs involved in launching of processes across different NUMA domains on a single node.

We classify the microbenchmarks based on several parameters like *control* semantics – synchronous (**sfork**) and asynchronous (**afork**) and *memory* semantics – no sharing of memory (**vfork**). All benchmarks launch a binary which immediately returns so as to minimize the execution time but account only for the overhead involved in loading and starting the binary. We added data to the *text* section of a binary to control its resultant size. In our tests, several runs with binaries of different sizes ranging from 8KB to 100MB were performed. Since Linux employs demand-paging, there is little to no observable difference in the spawn times for different executable sizes. We used a default size of 1MB for the following tests. Further, we ran two sets of the benchmarks, one with zero entries in `LD_LIBRARY_PATH`[2] and the other with 10 entries in `LD_LIBRARY_PATH` and with the standard entries in the environment.

Typically when launching new processes, `exec` is called immediately after `fork`. Hence, a new page-table mapping is created which invalidates the page-tables created on `fork`. Since this behavior is so common, Linux provides a system cal, `vfork`, in which the parent and child processes share the page-tables. After forking the child, the parent stops its execution until the child calls `exec` or `exit`. We use this call in our **vfork** tests to compare it against other mechanisms. Finally, in the **pfork** tests, we modified the launched

[1]Sharing of resources between forked processes can be controlled by specifying appropriate flags to the `clone` system call in Linux.
[2]The number of entries in `LD_LIBRARY_PATH` and the environment influences the time it takes for the `exec` system call to finish.



(a) 16-core node (LD0) (b) 24-core node (LD0) (c) 64-core node (LD0)

(d) 16-core node (LD10) (e) 24-core node (LD10) (f) 64-core node (LD10)

Figure 4: Comparing different process spawning mechanisms

process to fork itself multiple times whereas the **pspawn** tests use `libpspawn` to achieve the same.

The inconsistent behavior of **afork** can be attributed to the scheduling decisions made across the different Linux kernels the tests were run on. These tests were run with *random* scheduling without pinning the launched processes to specific cores. Our tests involving filling up the nodes by pinning them to cores linearly showed similar results with an additional overhead of pinning. **pfork** and **pspawn** perform significantly better on the 64-core machine as shown in Figure 4(c) but their throughput is similar to the throughput of **afork** for the 16-core and the 24-core machine as seen in Figures 4(a) and 4(b). As seen in Figures 4(d), 4(e) and 4(f), under more real circumstances with entries in `LD_LIBRARY_PATH` and the environment, **pfork** and **pspawn** attain higher throughput than **afork**. This is due to the fact that **afork** performs more `exec` system calls and hence more context switches than the other two.

From Figure 2.4, we see that **pspawn** and **pfork** perform significantly better than the other spawn methods across all of the above machines. In practice, considerably better results are seen since most HPC machines typically have a modest number of library path entries and environment variables. In summary, **pspawn** showed an overall increase of about 20% to 80% in throughput as compared to the other tests under all conditions.

2.5 Copy-on-Write Overhead

To account for the copy-on-write overhead because of shared address-space between the parent and child, we modified the tests above with a microbenchmark that writes to the first byte of every page in its address-space. These tests were run with an executable size of 100MB such that first byte of every page in the 100MB range was written to. Further, 200 instances of each benchmark were spawned. As shown in Figure 2.5, the copy-on-write overhead is typically negligible at modest core and NUMA domain counts. On the 64-core node, shown in Figure 5(c), the penalty incurred due to copying pages on write was significant and thus the **pfork** latency was much lower than **afork**.

3. NUMA-AWARE PROCESS SPAWNING

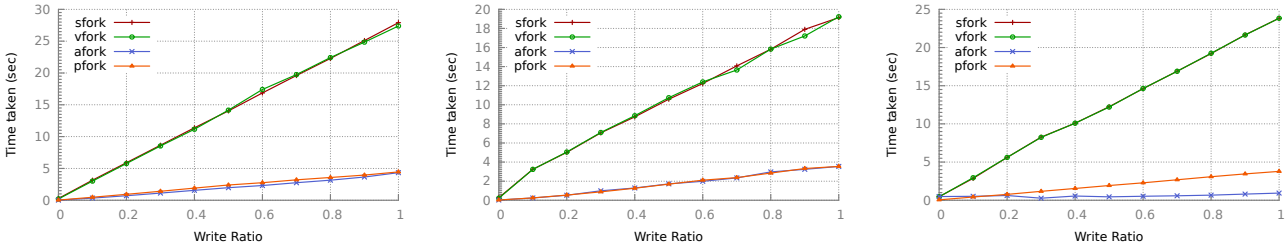
When the operating system forks a process into two distinct copies, the kernel data structures relevant to that task like `task_struct`, `mm_struct`, stack, VMA regions and page tables are allocated on the parent's NUMA node. The launched process can be pinned to a separate core using the scheduler's affinity functions so that further allocations made by the launched process are local to its NUMA domain[3]. By default, the memory policy in Linux dictates that pages be allocated local to the domain the process is running on. The kernel itself does a good job of load-balancing processes on `fork` and `exec`. The anonymous pages, mapped and unmapped page-cache pages are migrated or subsequently allocated locally so that the process is not penalized on accessing them. The remote kernel allocations can still, however, result in diminished performance at large scale. In this section, we describe some NUMA-aware optimizations for process spawning.

Table 1: NUMA-aware spawn of the Pynamic benchmark

Test	Avg. Import Time	Total Time
pynamic-first	65.6506	94.633
pynamic	45.9471	59.0187
pynamic-numa-first	65.3467	93.305
pynamic-numa	46.1293	56.64

To avoid crossing the NUMA-domains when a process is spawned, we dynamically replicated the shared libraries and executables local to each domain. This was done merely by copying and renaming the shared libraries at runtime. With this optimization, most of the memory accesses while launching the process would be local and hence, faster. For spawning multiple instances of the same application, after the first run most of the memory requests for mapped pages are served off of the page-cache. Since the page-cache in Linux is interleaved, a lot of requests end up crossing

[3]Where the memory gets allocated depends on the memory allocation policy that the child inherits from the parent process.



(a) 16-core node (b) 24-core node (c) 64-core node

Figure 5: Comparing different process spawning mechanisms with the copy-on-write overhead

the domain and result in poor performance. We dynamically partitioned the launched processes into separate process containers (using the `cgroups` API in Linux). The memory between these processes can be hard-walled such that the page-caches are not shared between processes across domains.

We used Pynamic, the Python Dynamic Benchmark [13], which is a synthetic benchmark to stress dynamic-linking and loading of applications. Python-based scientific applications tend to depend on a large number of sizeable shared-libraries. Launching of these applications is slow owing to the fact that every Python module they depend on has to be imported before the application can begin executing. We ran a Pynamic benchmark simulating 495 shared libraries with an aggregate total of 1.4GB. 64 instances of the benchmark were run across 4 NUMA-domains on the 16-node cluster. Table 1 shows the results for both the tests and their first runs. Although, the speed-up gained with the memory partitions in place was marginal largely due to the VM overhead caused by `cgroups`, we believe an isolated page-cache in Linux would show significant speed-ups at higher scales.

4. PSPAWN SYSTEM CALL

4.1 Overview

The primary reason process spawning on a set of processors at larger scales can be slow is due to the limiting nature of the interface the operating system exposes to do these operations. The traditional system calls (`fork`, `clone` and `exec`) have been around since the uniprocessor days. We argue that future operating systems should expose richer interfaces to leverage the available hardware parallelism. With the traditional way launchers use the existing interface, it involves as many as 3-4 context switches to fork a process, pin it to a specific core and load the binary. When done repeatedly, this slows down the overall launch time. We combined the system call interfaces to directly launch a specified binary on a specified core. While limiting in general, this interface offers a much greater opportunity for the kernel to take advantage of the available parallelism. In Linux, the `fork` system call fundamentally works because of the memory-overcommit behavior of the kernel. A process with a heavy memory footprint can fork another process since only the page-tables are copied during fork. Memory over-commiting, at times, can lead to unpredictable behavior including excessive swapping or the kernel OOM (out-of-memory) killer killing random processes when the kernel is short of memory. With a combined `spawn` system call that we propose, new processes can be launched even when the memory-overcommit setting is turned off.

4.2 Design

We added synchronous and asynchronous `pspawn` vector system

```
/* Synchronous pspawn system call */
int pspawn(char *filename, char **argv, char **envp,
           unsigned int nspawns, unsigned int clen,
           cpu_set_t **mask, enum pspawn_flags flags,
           pid_t *pids);
/* Asynchronous ipspawn system call */
int ipspawn(char *filename, char **argv, char **envp,
            unsigned int nspawns, unsigned int clen,
            cpu_set_t **mask, enum pspawn_flags flags);
```

Figure 6: The vector `pspawn` system calls

calls that accept a list of arguments and environments for each instance of the process spawn. The interface for these calls is shown in Figure 6. In addition to the necessary information to launch a process, these calls accept a CPU mask that dictates what CPUs the process is allowed to run on. This facilitates launching a process directly on a given core.

Both the system calls launch a kernel thread that loads and starts the given process image with the security credentials of the current user. The process executing the call is set as a parent to the launched processes. These system calls avoid the redundant copying of resources (file descriptors, page tables) between the child and the parent. For the `pspawn` system call, the resulting list of pids is returned to the userspace through the `pids` structure. The asynchronous system call `ispawn` returns immediately without waiting for the kernel thread to launch all processes. The processes themselves are launched sequentially inside the kernel for the `pspawn` call, and in parallel for `ipspawn`. Currently, we do not have a way for the parent process to query the PIDs of the children processes launched through `ispawn`. A few ways in which this could be done is discussed in Section 6.

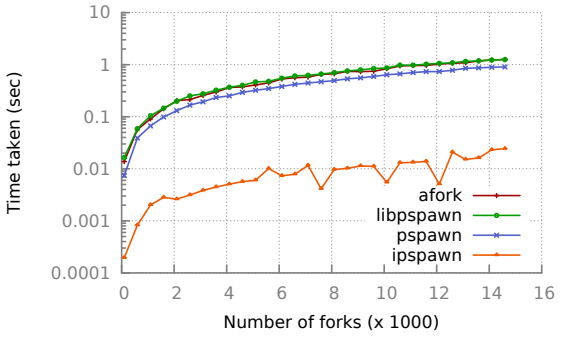


Figure 7: Performance of the vector system calls

The `exec` system call takes a list of strings as the argument and

environment parameters. We decided to vectorize the interface, yet keep it simple, by pushing some of the complexity to the user-space. The system calls shown in Figure 6 join these list of arguments and environments for each spawn into a long string. This enforces limitations on the length of these parameters. To overcome some of these limitations, we plan to evaluate the use of a filesystem interface to pass in some of these vector objects between the kernel and user-space. The tests to evaluate the performance of these calls were run on the 16-core machine described in Section 2.3. These tests were run with an empty environment and empty loader library path variable. As seen in Figure 7, **ipspawn** is almost 40-50x faster than its userspace counterpart, **afork**. The **pspawn** system call performs almost as good as the **afork** in this case. These results were run with random scheduling and without any explicit pinning. Both the newly introduced system calls, **pspawn** and **ipspawn** ran faster for the tests involving a linear launch across all available cores.

5. RELATED WORK

This work explores optimization of the several limiting factors for efficient spawning of processes on manycore architectures. There has been a significant focus on two related areas: i) improving operating system interfaces for increased scalability and ii) increasing throughput of global distributed process launch for faster job spawn.

Some of the earlier work focused on the performance of Berkeley UNIX and included various measurements of the `fork` and `exec` system calls [14], but this work did not take into account multicore processors which were not available at the time. The benchmark suite `lmbench` [3] benchmark suite was a set of operating system microbenchmarks, which included tests to measure the overhead of the `fork` and `exec` system calls. The benefits of *copy-on-write* (COW) with respect to UNIX `fork` was carefully measured and verified on two systems and reported in [19], although the systems under test are now dated. We reproduced these results on a modern architecture in the process of our investigation and described how this affects process launch. Other early work was focused on moving applications from static to shared libraries to reduce the memory footprint for frequently used libraries such as *libc* and large libraries such as the ones needed to support graphical environments such as X11 [8].

Recent work to investigate the scalability of existing operating systems has highlighted several bottlenecks that would prevent the scaling of these systems on manycore architecures. In [5], the authors compared the scalability of three commodity operating systems (Linux, Solaris and FreeBSD) using a benchmark suite they developed[4]. However, no solutions to address these limits were discussed. Further extensive analysis of the scalability of Linux kernel with respect to manycore processors has been done in [2]. The performance of spinlocks and other kernel data structures that are shared among processors were measured and optimized. While addressing the overall scalability concerns, they support the use of traditional operating system interfaces rather than proposing new ones. Asynchronous exception-less system calls have been demonstrated to perform much better on manycore processors[20]. Vector interfaces for operating systems have been suggested by [23] who advocate the use of these calls to leverage the increasing parallelism in hardware. Operating system abstractions to support accelerators and manycore processors as native compute devices[18] have been proposed. These interfaces could potentially demonstrate better application performance from asynchronous, vector system calls.

Exascale runtime systems need to be dynamic, resilient and scalable. The use of process replication for reliability has been demonstrated in [7]. Task abstractions for manycore architectures that include compressed task representation have been shown to improve efficiency and reduce the queuing overhead[17]. Browsers and runtime systems in mobile devices, both, depend on the capability to launch several short-lived processes in a short time. After this work, we found out that both Google Chrome[10] and the Dalvik Virtual Machine[6] that Android runs on use the technique described in Section 2. The *zygote* process is started at boot-time or when the browser is launched to facilitate the sharing of code between the parent and children processes.

6. FUTURE WORK

Some of the lessons learned in optimizing process spawn can be used to improve the local job launch times for parallel runtime systems. We would like to extend this work to optimize job-launch for MPI runtime such as Open MPI [22], MPICH2 [15] and MVAPICH [16] and resource managers like SLURM [11]. `libpspawn` could also be integrated into new task-parallel runtime systems for efficient spawning of tasks in a high-throughput environment. Dynamic resiliency process models that involve execution with shadow-processes [7] could benefit from the techniques described in this paper. By modifying the Linux kernel to allow isolated page-caches, cross-domain memory requests could be reduced. The speed-up that could potentially be gained with this optimization can be studied through a model that takes into account the memory accesses of an application and the latencies involved in fetching pages from local or remote memory domains.

We plan to extend the kernel interface to allow returning status information to the asynchronous calls. This could be done by associating a *launch-session* with each invocation of the call. Processes (including parent and children) can block and retrieve the PID list or error codes associated with each session. Some of this information can also be exported through the `proc` filesystem interface in Linux.

Finally, we intend to evaluate some of the ideas in this paper in the context of existing manycore operating systems like Tessellation and Barrelfish running on hardware like the Tilera processor architecture or the Intel MIC processors.

7. CONCLUSIONS

The path towards exascale computing has garnered a lot of interest towards dynamic runtime systems, and scalable operating system interfaces to support these runtime systems. Our investigation towards optimizing intranode process spawning was due to the emerging hardware trends that include massively multicore processor architectures. Operating systems designed to run on these architectures should explore ideas to make better use of the hardware, while exposing a richer interface to the runtime systems and applications. We showed that improving the throughput and latency for spawning processes would result in faster global application launches and more dynamic runtime systems. The factors affecting process management including shared resources between parent and children, the size of memory-mapped regions of the launched process and the launch patterns were explored with suggested optimizations that are applicable for future HPC manycore operating systems.

8. ACKNOWLEDGMENTS

The authors would like to thank Hakan Akkan for helpful discussions. This work was performed at the Ultrascale Systems Research Center (USRC), a collaboration between Los Alamos National Laboratory and the New Mexico Consortium (NMC). This publication has been assigned the LANL identifier LA-UR-12-21227.

9. REFERENCES

[1] Ahn, D. H., Arnold, D. C., de Supinski, B. R., Lee, G. L., Miller, B. P., and Schulz, M. Overcoming Scalability Challenges for Tool Daemon Launching. In *ICPP* (2008), IEEE Computer Society, pp. 578–585.

[2] Boyd-Wickizer, S., Clements, A. T., Mao, Y., Pesterev, A., Kaashoek, M. F., Morris, R., and Zeldovich, N. An analysis of Linux scalability to many cores. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–8.

[3] Brown, A. B., and Seltzer, M. I. Operating system benchmarking in the wake of lmbench: A case study of the performance of NetBSD on the Intel x86 architecture. In *In ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (1997), pp. 214–224.

[4] Chen, Y., and Shi, Y. OSMark: A benchmark suite for understanding parallel scalability of operating systems on large scale multi-cores. *2009 2nd IEEE International Conference on Computer Science and Information Technology* (2009), 313–317.

[5] Cui, Y., and Chen, Y. Scalability comparison of commodity operating systems on multi-cores. *2010 IEEE International Symposium on Performance Analysis of Systems Software ISPASS* (2010), 117–118.

[6] Ehringer, D. The dalvik virtual machine architecture.

[7] Ferreira, K., Stearley, J., Laros, III, J. H., Oldfield, R., Pedretti, K., Brightwell, R., Riesen, R., Bridges, P. G., and Arnold, D. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2011), SC '11, ACM, pp. 44:1–44:12.

[8] Gingell, R., Lee, M., Dang, X. T., and Weeks, M. S. Shared Libraries in SunOS. In *Proceedings of the Summer 1987 USENIX Technical Conference* (1987), pp. 131–145.

[9] Goehner, J. D., Arnold, D. C., Ahn, D. H., Lee, G. L., De Supinski, B. R., LeGendre, M. P., Schulz, M., and Miller, B. P. *A Framework for Bootstrapping Extreme Scale Software Systems*. No. Whist. 2011.

[10] Google, T. C. T. LinuxZygote - Chromium - The use of zygotes on Linux.

[11] Jette, M. A., Yoo, A. B., and Grondona, M. SLURM: Simple Linux Utility for Resource Management. In *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003* (2002), Springer-Verlag, pp. 44–60.

[12] Kogge, P., Bergman, K., Borkar, S., Campbell, D., Carson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hill, K., and Others. Exascale computing study: Technology challenges in achieving exascale systems. Tech. rep., University of Notre Dame, CSE Dept., 2008.

[13] Lee, G. L., Ahn, D. H., de Supinski, B. R., Gyllenhaal, J., and Miller, P. Pynamic: the Python Dynamic Benchmark. In *Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization* (Washington, DC, USA, 2007), IISWC '07, IEEE Computer Society, pp. 101–106.

[14] Leffler, S. J., and Karels, M. J. Measuring and Improving the Performance of Berkeley UNIX*, 1991.

[15] MPICH2: High-performance and Widely Portable MPI. `http://www.mcs.anl.gov/mpi/mpich2`.

[16] Network-Based Computing Laboratory, Department of Computer Science and Engineering, Ohio State University. MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE.

[17] Orozco, D., Garcia, E., Pavel, R., Khan, R., and Gao, G. R. Polytasks: A Compressed Task Representation for HPC Runtimes. In *Proceedings of the 24th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2011), Fort Collins, CO, USA*. 2011.

[18] Rossbach, C. J., Currey, J., Silberstein, M., Ray, B., and Witchel, E. PTask: operating system abstractions to manage GPUs as compute devices. In *SOSP* (2011), T. Wobber and P. Druschel, Eds., ACM, pp. 233–248.

[19] Smith, J. M., and Jr., G. Q. M. Effects of copy-on-write memory management on the response time of UNIX fork operations'. *COMPUTING SYSTEMS 1*, 3 (1988), 255–278.

[20] Soares, L., and Stumm, M. FlexSC: flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–8.

[21] Sridhar, J. K., Koop, M. J., Perkins, J. L., and Panda, D. K. ScELA: Scalable and Extensible Launching Architecture for Clusters. In *HiPC* (2008), P. Sadayappan, M. Parashar, R. Badrinath, and V. K. Prasanna, Eds., vol. 5374 of *Lecture Notes in Computer Science*, Springer, pp. 323–335.

[22] The Open MPI Development Team. Open MPI: Open Source High Performance Computing.

[23] Vasudevan, V., Andersen, D. G., and Kaminsky, M. The case for VOS: the vector operating system. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems* (Berkeley, CA, USA, 2011), HotOS'13, USENIX Association, pp. 31–31.