Operating System Structures for Multiprocessor Systems on Programmable Chip

Miaoqing Huang, David Andrews
University of Arkansas
Fayetteville, AR 72701, USA
{mqhuang,dandrews}@uark.edu

Jason Agron[†]
Intel Corporation
Santa Clara, CA 95051, USA
jason.m.agron@intel.com

Abstract—Chips are moving from single-core systems to much more complex, heterogeneous manycore systems. While heterogeneous architectures promise high performance, they are also challenging our ability to port our existing operating systems to abstract the heterogeneous components into a unified architecture. Baseline solutions to resolve heterogeneity issues within manycores use Remote Procedure Calls (RPC) for applications running on slave processors to access a traditional monolithic kernel running on a common master node. Microkernels are once again re-emerging to eliminate the central bottleneck of the monolithic kernel. In both cases the RPC methods used for communications increase the overhead of system services, counter to the desire of breaking threads up into finer grained services to match and scale with increasing numbers of processors. In this paper we show how new invocation mechanisms built as hardware primitives in combination with new a new hw/sw co-designed microkernel can resolve heterogeneity issues in a framework that supports the level of scalability required for next generation systems. We present experimental results as well as a new queuing model to show how both monolithic and microkernels that rely historical interrupt mechanisms cannot support scalability beyond small numbers of processors. We also show through these results the potential scalability of a hardware kernel based microkernel with new lightweight invocation mechanisms.

Keywords—microkernel, multiprocessor system, operating system, reconfigurable computing

I. INTRODUCTION

FPGAs continue to track Moore’s law [1] and now contain sufficient gates and diffused components to host complete multiprocessor systems on a programmable chip (MPSoPC). MPSoPCs promise productivity advantages over earlier smaller FPGA components with their ability to serve as an architecture framework upon which developers can work with modern programming languages, middleware and operating systems. Designers can work with these modern abstractions in place of hardware description languages and custom circuit synthesis. This can increase designer productivity levels to those more closely associated with modern software development methods while still delivering performance levels more closely associated with custom designs.

The productivity potential of MPSoPCs relies on our ability to successfully transition familiar higher level ab-

stractions and software protocol stacks from the general purpose computing domain. Unfortunately the general purpose computing domain itself is struggling to transition historical software protocol stacks for scalar processors to the parallel architectures that will make up the manycore era. From a historical perspective, operating system research for parallel architectures flourished during the prior parallel processing era but was damped with the dominance of commodity cluster architectures. Monolithic kernel structures from the earlier mainframe era were augmented with multithreaded shared memory and message passing middleware to enable domain scientists and not just computer scientists to program commodity clusters. The absence of any new foundational operating system structures have necessitated the continued adoption of familiar monolithic operating system structures within the general purpose and reconfigurable computing communities. While monolithic kernels have been successfully adopted for small numbers of homogeneous processors within SMP systems, the large scalability and heterogeneity needs of next generation manycores and MPSoPCs may end up retiring our monolithic kernels along with dynamic ILP scalar processors.

Heterogeneous processor requirements have risen based on the need to exploit parallelism at different levels of granularity [2]. Hill and Marty [3] discussed the design tradeoffs for symmetric and asymmetric (heterogeneous) architectures within the manycore era. They used Amdahl’s law to suggest that combining a subset of smaller homogeneous cores into fewer but more powerful heterogeneous cores will yield better performance compared to large numbers of smaller homogeneous cores. While promising from a performance perspective, heterogeneous mixes of processors introduce new challenges for operating systems when the processors have different Instruction Set Architectures (ISAs), Application Binary Interfaces (ABIs) and low level microarchitecture cache coherency support. Differences in atomic operations such as *load_linked*, *store_conditional* and *test_and_set* are particularly challenging as they form the basis upon which operating systems provide the fundamental synchronization primitives used in our modern programming models. These different atomic operations are not compatible between each other, and rely on shared bus snoopy cache protocols, which are known not to scale.

[†]Agron’s work was done when he was with University of Arkansas.

Figure 1. IBM Cell SPUFS programming environment

In light of these issues, both IBM and Intel have offered heterogeneous multiprocessor systems that use monolithic kernels. Both IBM’s Cell and Intel’s Exochi resolve heterogeneity issues by simply avoiding them through the use of Remote Procedure Call (RPC) methods. Typically the monolithic kernel is hosted on a master node, and applications running on slave nodes request services from the monolithic kernel using RPC calls. Figure 1 shows this approach for SPE and PPE processors within the Cell architecture [4]. While at first glance appealing this approach requires programmers to work with two separate models: one for the master node and one for the slave. This limits portability and is counter to the operating systems ability to seamlessly abstract platform specific implementations within a single unified virtual machine model. This approach also introduces unwanted contention and sequentialization for service requests between multiple slave processors and the single master. The unique needs of heterogeneous manycores are once again reinvigorating the monolithic versus microkernel debate [5]. Proponents of microkernels point out their ability to relieve the contention for services imposed by a monolithic kernel through the partitioning and distribution of services across parallel components. Microkernels also relieve heterogeneity issues through the use of message passing protocols between slave nodes and service nodes. Work such as the barrellfish multikernel [6], [7] is attempting to further refine the microkernel structure for heterogeneous manycores. A growing concern for both monolithic and microkernels is their reliance on traditional asynchronous interrupt invocation mechanisms for inter processor communications. Figure 2 shows the steps required in both cases to communicate between processors. The first column labeled CPU-to-CPU in Table I provides relative clock cycles for these operations. The heavyweight cost of interrupts is concerning as we consider partitioning applications into more finer grained threads to map over processors numbers that will be growing in accordance with Moore’s law.

II. EXPERIMENTAL RESULTS ON HTHREADS

The hthreads hardware microkernel was originally developed to create a unified programming model that seamlessly

Figure 2. IPC requests: asynchronous CPU-to-CPU communication

Table I
IPC AMONGST CPUs Vs. CPU-TO-CORE IPC

CPU-to-CPU (Figure 2)		CPU-to-Core (hthread)	
CPU ₁ Stores Parameter	10’s	Encode Parameters	10’s
Inter-CPU Interrupt	1000’s	Load Result	10’s
CPU ₂ Loads Parameters	10’s		
CPU ₂ Processes	Variable		
CPU ₂ Stores Result	10’s		
CPU ₁ Poll/Interrupt	10-1000’s		
CPU ₁ Loads Result	10’s		
Total	1000’s	Total	10’s

abstracted the CPU-FPGA boundary [8]. From the programmers perspective hthreads enables designers to create custom hardware components within the FPGA abstracted within the pthreads multithreaded programming model. Application designers can create custom threads that can synchronize, communicate, and be controlled within the scheduling envelope of a thread scheduler. A key design challenge for hthreads was to provide efficient mechanisms for both the software and hardware threads. To avoid the high overhead of interrupt invocations for software and hardware threads requesting services such as *mutex* operations, key services were transitioned into hardware components accessed using light weight simple *load* and *store* instructions. Figure 3 shows the hardware components of the hthreads system. The detailed design of hthreads can be found in [9]–[11] and are not elaborated further. Important for this discussion, the CPU-to-Core column in Table I shows the relative clock cycle counts for hthreads invocations.

To explore how operating system invocation mechanisms can effect heterogeneous manycores we developed two fully functional experimental systems: one using the hthreads microkernel and the second a monolithic-RPC approach. We modified our hthreads kernel to serve as a monolithic kernel on the PPC accessed by slave threads running on MicroBlazes through RPC calls. Both platforms were implemented on a Xilinx XC5VFX70T device using an ML507 board. Due to hardware resource constraints, we could only implement systems with up to 6 MicroBlaze cores. In both systems we created a synthetic program that created the same number of threads as MicroBlaze cores.

Our synthetic program is shown in Figure 4 with each

Figure 3. Hthreads hardware microkernel

```
void * worker_thread (void * arg) {
    for (x=0; x<iterations; x++) {
        time_delay(work_delay_count);
        hthread_mutex_lock(mutex);
        time_delay(crit_delay_count);
        hthread_mutex_unlock(mutex);
    }
}
```

Figure 4. The computation of the thread in the synthetic application

thread running the *for* loop for 1000 iterations. In each iteration, each slave thread waits for 250 clock cycles before attempting to request a *mutex* guarding a simulated critical region. Once inside the critical region, the thread waits for an additional 25 clock cycles before releasing the *mutex*. We are only interested in the overhead and contention of system call invocations and not performance degradations from the sequentialization imposed by the critical region. Each thread uses a different mutex to eliminate these effects. Figure 5(a) shows experimentally measured latencies due to the invocation mechanisms for the monolithic kernel-RPC and the hthreads microkernel. The average access time for hthreads is 408 clock cycles, compared to 8765 for the monolithic-RPC approach. These results are consistent with the clock cycle counts given in Table I. Figure 5(b) shows the effects of contention when six slave processors are attempting to access services. The monolithic-RPC approach suffers an increase in average access times from 8765 to 12249 clock cycles and deviation times from 443 to 1010 clock cycles. Hthreads invocations show only a few clock cycles of difference between the two systems. Of particular interest is the absence of any significant deviation when the number of slave processors increases. This can be attributed to the very low, single assembly instruction invocation overhead that dramatically limits the probability of contention.

Figure 6 shows how these latencies can affect application level speedup. For the monolithic-RPC approach, the performance stops scaling with more than 3 threads. Thus only 3 threads are *effectively* running no matter how many threads are live across the system. The high synchronization cost as well as the contention caused by accessing the monolithic kernel contributes to the poor scalability. On the other hand the hthread system demonstrated good scalability.

(a) Effects of synchronization latency reduction (b) Effects of contention on synchronization

Figure 5. Effects of hthreads on synchronization latency

Figure 6. Scalability Comparison between the HW microkernel approach and the monolithic kernel + RPC approach

Microprocessor vendors as well as FPGA manufacturers have already outlined road maps that show the availability of manycore and MPSoPC systems in the near future with hundreds of cores [12]. We are interested in evaluating how our low latency invocation mechanisms will scale for these systems. Clearly we do not yet have MPSoPCs of sufficient size to run actual systems and generate experimental results. In the following section we present a new theoretical queuing model developed to extend our analysis to systems with larger numbers of processors. As with all theoretical models, the accuracy of results are dependent on the accuracy of assumptions and parameters used within the model. We use the results from our experimental analysis to help set assumptions and parameters.

III. UNDERSTANDING OS OVERHEAD

Threads invoke operating system code encapsulated within linkable libraries through API calls. We generalize the processing within an API as:

$$T_{API} = T_{setup} + T_{rqst} + T_{serv} + T_{complete}. \quad (1)$$

T_{setup} and $T_{complete}$ represent the code executed to start, finish and return status. Both represent parallel code that can run locally on a calling, or slave processor. For the monolithic kernel running on a master node, T_{serv} represents service code that is executed sequentially in a time multiplexed fashion. T_{rqst} encapsulates code used to communicate between T_{setup} , $T_{complete}$ running on slave processors and T_{serv} running on the master processor.

When no request conflicts occur for the services running on the master node, $T_{rqst} = T_{invocation}$, the time taken by

Figure 7. The execution of an application on (a) a single-core system, and (b) a manycore system

the operating system to transfer the service request between processors. $T_{invocation}$ contributes to the sequential portion of operating system code, and is set by the arbitration and signaling mechanisms between the slave and master processors shown in Figure 2. Data and status information is passed through shared memory, and processing invocation is realized through external asynchronous interrupts.

Conflicts occur when multiple slave nodes issue simultaneous requests, or requests when the master node is busy. When the monolithic OS kernel is busy, all other thread requests can be modeled as residing in a waiting queue. We let T_q reflect the time a thread spends waiting in the queue. We now update our request time as $T_{rqst} = T_q + T_{invocation}$.

To simplify further discussions, we combine T_{setup} , $T_{invocation}$, T_{serv} , and $T_{complete}$ into a single variable, T'_{serv} , i.e., $T'_{serv} = T_{setup} + T_{invocation} + T_{serv} + T_{complete}$. We call T'_{serv} the congregate service time. Therefore, (1) can be rewritten as follows,

$$T_{API} = T'_{serv} + T_q. \quad (2)$$

From our prior experimental results we observe T_q is influenced by three factors: (1) the number of threads (and processors) running in the system that can issue simultaneous request services, (2) the number of queues and servers available for handling service requests, and (3) the congregate service time, T'_{serv} . For a given number of threads, the more queues and servers a system has, and the shorter T'_{serv} , the less time a request should be required to wait in a queue.

IV. MODELING THE OS STRUCTURE

Our goal is to evaluate the probabilistic sequentialization effects of OS services without additional bias introduced by any particular lower level architecture platform specifics. Therefore, the following assumptions are made:

- We assume a bus or network-on-chip capable of providing sufficient bandwidth to handle all the communication due to synchronization between cores. This eliminates additional degradation due to probabilistic and application specific bus contention.
- Each slave processing core executes only one thread. This eliminates the additional effects of context switch-

Figure 8. Queuing network models for manycore synchronization

ing and scheduling overhead. Each thread assumes one of three states:

- Active: the thread/processing core is executing normal operations.
 - Queueing: the thread/processing core is waiting in the queue for service.
 - System service processing: the thread/processing core is interacting with the server for system services.
- The duration of system service and active periods of each thread/core are assumed to be random variables with exponential distribution.

A generic execution model for a multithreaded application including time spent idling during contention is illustrated in Figure 7. We model individual thread behaviors as shown in Figure 7(b). During normal operation periods each thread performs some useful computations. Threads initiate requests for operating system services with the host node (in RPC mode) or the corresponding microkernel server component. If the server is busy, the request is put into a queue and the thread idles. Once the request is granted on the server, T'_{serv} denotes the time attributed to the operating system to execute the service. We first analyze the behavior of a single queue, which models the contention of a monolithic OS + RPC as shown in Figure 8(a). We then extend the analysis to multiple queues with multiple servers for modeling the microkernel OS, as shown in Figure 8(b).

We use n to represent the number of cores in our manycore system. The distribution of random variables in random vector $\epsilon = \{\epsilon_1, \epsilon_2, \dots, \epsilon_n\}$, represents the active times (i.e., normal operation time) for each core. The distribution of the random variables in the random vector $\sigma = \{\sigma_1, \sigma_2, \dots, \sigma_n\}$ represents the service times for each core. We assume that the vector ϵ contains exponentially distributed random variables, whose parameters are entries of the vector of scalars $\lambda = \{\lambda_1, \lambda_2, \dots, \lambda_n\}$. Similarly, we assume all the random variables in vector $\sigma = \{\sigma_1, \sigma_2, \dots, \sigma_n\}$ are exponentially distributed random variables whose parameters are entries of the vector of scalars $\mu = \{\mu_1, \mu_2, \dots, \mu_n\}$. It is further assumed that $\lambda_i = \lambda$, and $\mu_i = \mu$ for all i to simplify the analysis.

Since the active time of a thread is an exponentially distributed random variable with parameter λ , the average active time is $\frac{1}{\lambda}$. In other words, the probability for this thread to request a service (i.e., entering the queue) is λ . If we use N to denote the average number of active cores, the arrival rate of the queue is $N\lambda$. The number of threads that are in the queue (including waiting and performing the service) is $n - N$. By applying the Little's Formula [13], we can find the average delay due to a service request (including queueing time and real service operations) as

$$E[D] = T_q + T'_{serv} = \frac{n - N}{N\lambda} \quad (3)$$

Subtracting from $E[D]$ the real service cost (i.e., T'_{serv}), we have the average queueing time (i.e., the waiting time in the queue)

$$T_q = \frac{n - N(1 + \rho)}{N\lambda}, \quad (4)$$

where $\rho = \frac{\lambda}{\mu}$, i.e., the ratio between the average service period and the average active period.

Perhaps the most difficult challenge in developing a realistic queueing model is determining ρ . Clearly ρ will vary based on program behavior. ρ will decrease as the application increases the amount of processing between system calls. Conversely ρ increases as the application decreases the amount of processing performed between calls. We set ρ based on the execution behavior of our synthetic benchmark. Specifically we use the execution times of the *mutex_lock* and *mutex_unlock* APIs and the execution time between calls and within the critical region. Importantly setting ρ based on our synthetic benchmark allows us to verify within a reasonable error the correctness of the model with our prior existing experimental results.

To evaluate the effects of request conflicts and invocation mechanisms we can evaluate the number of requests in the queue (the number of idling threads). The average number of queued threads is

$$E[N_q] = T_q N \lambda = n - N(1 + \rho). \quad (5)$$

In accordance with [13] we can then calculate the average number of active cores as

$$N = \frac{1 - \pi_0}{\rho}, \quad (6)$$

in which

$$\pi_0 = \left(\sum_{k=0}^n \rho^k \frac{n!}{(n-k)!} \right)^{-1}. \quad (7)$$

We use the multiserver queueing model in Figure 8(b) to evaluate the effects of the microkernel OS. The n processors are divided into b sets, i.e., $S_1, S_2, \dots, S_b$, in which the cores belonging to S_i are performing service requests through server i . We assume a thread cannot issue multiple requests to multiple servers simultaneously, i.e., $n = \sum_{i=1}^b n(S_i)$,

Table II
THE AVERAGE NUMBER OF ACTIVE CORES IN THREE APPROACHES

Number of Total Cores	Number of Active Cores		
	HW Microkernel ($\rho = 0.005$)	SW Microkernel ($\rho = 0.05$)	RPC ($\rho = 0.05$)
4	4	4	4
8	8	8	7
16	16	15	14
32	32	30	20
64	64	57	20
128	127	80	20
192	191	80	20
256	254	80	20

Figure 9. The scalability comparison among three synchronization approaches

where $n(S_i)$ is the number of cores in set S_i . If we use $N(S_i)$ to denote the active cores in set S_i , then the total number of active cores in the system is

$$N = \sum_{i=1}^b N(S_i). \quad (8)$$

We apply (6) to calculate each individual $N(S_i)$ and then use (8) to find the total number of active cores. Based on our assumption of one thread per core, N gives us the number of parallel threads active at any instant within the system.

A. Scalability Comparisons

Table II and Figure 9 show the number of active cores (N) versus the number of total system cores (n) for our three different operating system structures. For these results we assigned ρ to 0.05 for the monolithic and software microkernel structures. We then conservatively adjust the efficiency of service by 10 $\times$, i.e., $\rho = 0.005$ for the hardware microkernel OS. In our real platform, this ratio typically runs much higher. For example Figure 5(a) shows an actual 21 $\times$ difference. All three results exhibit behaviors similar to those generated experimentally on our 6-processor system. The RPC approach provided the worst behavior, showing contention for the single master node and interrupt invocation

mechanisms prohibited scaling any further performance scaling at 20 cores. The software microkernel model exhibited better scaling, but also reached at plateau at 100 cores. As both models used the same values for invocation overhead and system service processing time, and the difference in scalability can more than likely be contributed to reducing the single point of contention through the distribution of services across multiple servers. In both cases, as more requests were placed in a queue, the queueing time naturally became longer. This is verified by (4), in which N remains unchanged when the system loses scalability and n becomes larger when more cores are added into the system. In contrast to these two approaches the hthreads hardware microkernel continued to show good scalability through 256 processors. As we set the operating system service time and number of servers of the microkernel and hthreads to be identical the difference in scalability can be attributed to the reduction in invocation times. These results support our concern that future heterogeneous manycore systems need new lightweight invocation mechanisms to allow application programs to achieve the desired performance increases through scalable parallelism.

V. CONCLUSION

Platform FPGAs already support the creation of complete multiprocessor systems on programmable chip (MPSoPC) systems. MPSoPCs promise productivity advantages with their ability to allow designers to work with modern programming abstractions. Unfortunately our modern operating systems were never created to support the scalability and heterogeneity capabilities of MPSoPCs. We have shown through results generated from experimental systems as well as analytical modeling that the current work-arounds to make traditional OS frameworks operate in heterogeneous environments compromise both uniformity and efficiency. This results in complex programming models that make it increasingly harder for programmers to write portable code and eliminate the ability for an application to seamlessly scale with large numbers of processors. We believe that the importance of OS services warrants their partial implementation in hardware, especially as doing so eliminates many serious heterogeneous incompatibilities at the ISA- and microarchitectural-levels. At a minimum operating systems are in need of more streamlined invocation mechanisms that are efficient and uniformly accessible from *any* computational component. As the number and type of on-chip cores increases, so too will the contention for OS services, thus making the operating system itself a key component that must be optimized and distributed for achieving scalable performance.

REFERENCES

- [1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, vol. 38, no. 8, pp. 114–117, Apr. 1965.
- [2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006. [Online]. Available: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html>
- [3] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *IEEE Computer*, vol. 41, no. 7, pp. 33–38, Jul. 2008.
- [4] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell multiprocessor," *IBM Journal of Research and Development*, vol. 49, no. 4/5, pp. 589–604, Jul./Sep. 2005.
- [5] A. Baumann, S. Peter, A. Schüpback, A. Singhanian, T. Roscoe, and P. B. R. Isaacs, "Your Computer Is Already A Distributed System. Why isn't your OS?" in *12th Workshop on Hot Topics in Operating Systems*. USENIX, May 2009.
- [6] A. Schupbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, and R. Isaacs, "Embracing Diversity in the Barrelfish Many-Core Operating System," in *Proceedings of the Workshop on Managed Many-Core Systems (MMCS)*, 2008.
- [7] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpback, and A. Singhanian, "The Multikernel: A New OS Architecture for Scalable Multicore Systems," in *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. New York, NY, USA: ACM, 2009, pp. 29–44.
- [8] D. Andrews, D. Niehaus, and P. J. Ashenden, "Programming Models for Hybrid CPU/FPGA Chips," *IEEE Computer*, vol. 37(1), pp. 118–120, 2004.
- [9] J. Agron, W. Peck, E. Anderson, D. Andrews, E. Komp, R. Sass, F. Bajot, and J. Stevens, "Run-Time Services for Hybrid CPU/FPGA Systems On Chip," in *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS)*, December 2006.
- [10] J. Agron and D. Andrews, "Building Heterogeneous Reconfigurable Systems With a Hardware Microkernel," in *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*. New York, NY, USA: ACM, 2009, pp. 393–402.
- [11] D. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbee, J. Ortiz, E. Komp, and P. Ashenden, "Programming Models for Hybrid FPGA-CPU Computational Components: A Missing Link," *IEEE Micro*, vol. 24(4), pp. 42–53, 2004.
- [12] J. Howard et al., "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," in *Proc. 2010 IEEE International Solid-State Circuits Conference (ISSCC)*, Feb. 2010, pp. 108–109.
- [13] J. D. C. Little, "A proof for the queuing formula: $L = \lambda W$," *Operations Research*, vol. 9, no. 3, pp. 383–387, May 1961.