

Model-T: Rethinking the OS for terabit speeds

Willem de Bruijn
Vrije Universiteit Amsterdam
wdb@few.vu.nl

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@cs.vu.nl

Abstract—This paper presents Model-T, an OS network stack designed to scale to terabit rates through pipelined execution of micro operations. Model-T parallelizes execution on multicore chips and enforces lockstep processing to maximize shared L2 data cache (d-cache) hitrate. Executing all operations without hitting main memory more than once (if at all) is the key design principle behind Model-T.

We show a prototype implementation that indeed handles terabit rate network traffic when accessing only the L2 cache and processing only headers. Additionally, we present a more practical implementation of Model-T that is under development for Linux 2.6. Finally, we introduce an affordable test infrastructure based on general purpose graphics processor computation (GPGPU) that can replay network streams at PCI Express saturation rates (up to 128 Gbps), to benchmark Model-T and similar research network stacks.

I. INTRODUCTION

While achievable networking speeds are approaching the terabit range, operating systems are sorely trailing, and are still clearing the bottlenecks to handle 10Gbps interfaces. We posit that the current direction of incremental OS changes is the real obstruction and present a radically different OS design. We will show that this design is not only likely capable of processing at these speeds in principle, we will demonstrate near terabit speed processing on a common laptop.

Terabit processing is not as far off as networking standards indicate: I/O buses capable of these speeds are on the market already, even on consumer-grade devices: the Nvidia GeForce 8800 Ultra has an official peak memory bandwidth of 830 Gbps. The Cell is an 8-way device with 200 Gbps sustained throughput *per node*, connected to a dual transport ring. The Cell does not scale to 1.6Tbps aggregate throughput only because the shared memory controller is also limited to 200 Gbps. This bottleneck is not coincidental, but representative of the field at large. In this paper we limit our attention to the predominant x86-64 multicore architecture with a shared L2 cache and its likely extensions.

A. Motivation

The main physical obstruction to terabit networking is the memory system. Firstly, peak bandwidth is restricted, with a current top of 200 Gbps peak rate. More importantly, sustained practical rates are substantially lower as traffic must flow (at least) twice across the bus and bus contention limits prefetching and burst mode effectiveness. Case in point is the fact that few systems are capable of processing data at 10 Gbps (sustained). Some network interface cards (NICs), like the DAG8.1X cards, have hardware support to push data to

memory at 10 Gbps, but in practice *handling* such data on the host is almost impossible, mainly because memory access is so slow.

The common technique of waiting for processors to become faster is therefore not a solution and neither is the introduction of specialized network processing logic. As network processing is a memory bound task it is imperative to improve memory bandwidth or use the memory hierarchy more efficiently. Memory bandwidth continues to grow slower than network speeds and latency is even worse (recent peak bandwidth increases are achieved through aggressive prefetching at the cost of latency). As alternative, we propose architectural OS changes to increase memory access *efficiency*.

Current OS design is founded on principles that are fundamentally at odds with high-speed I/O. OS design principles stem from an age when I/O was the slowest part of the system and data copying and per-block signaling did not affect performance. With today's constrained memory system, memory copying has become the principal bottleneck. Another mismatch lies in scheduling, which looks only at computational efficiency. To reach high system I/O rates, CPU utilization is simply the wrong scheduling metric as it fails to take into account task switching overhead and memory bus contention [1]. The arrival of multicore architectures only exacerbates contention. So far, scaling (near) linearly with number of cores has proven unsuccessful. Lock-contention on shared structures, cache-inefficiency and non-uniform load-balancing (e.g., using a hash over TCP connection state) are the main causes.

B. Contribution

We present Model-T: a network stack that has as sole aim to maximize I/O throughput and which is therefore targeted at networked machines, both within the network (routers) and at the edges (servers). We redesigned the stack from the ground up to match the computer architecture of today. Specifically, we exploit multilevel memory hierarchies and multicore processors. Also, we engineer for the hardware development trend from multicore to (a) manycore [2] (more than 8 cores), (b) performance asymmetric cores [3] (different implementations of the same instruction set architecture, or ISA) and (c) heterogeneous multicores (CPUs mixed with specialized cores for graphics, multimedia, networking and even FPGAs, such as the Cell "Broadband Engine" or the x86-64 based AMD Fusion). Our focus on I/O optimization enables radical solutions such as shutting off cores to prevent

working-set thrashing and using cores solely as prefetching machines to feed the cache. Before explaining how Model-T exploits caching and multicores, we first explain how these affect I/O.

If memory is the main bottleneck in network processing, then operating directly from the d-cache is the only short-term viable strategy for reaching terabit speeds. We must scale the combined system-wide working set to fit in the most critical cache level (predominantly L2 [4]) to bypass the memory bottleneck. Cache-aware scheduling is receiving interest with the arrival of multicore architectures [5]–[7]. So far, solutions are computation-centric and deal with issues such as fairness and random access. Network processing is a single task and therefore does not need to take into account fairness. Moreover, an I/O centric scheduler should take into account the largely sequential data access pattern of network processing. If scheduled optimally, co-scheduled network operations access the same packet so that no packet needs to be fetched into the cache more than once.

Transparent caching complicates cache-optimization. The simplified memory management improves base application performance without development effort, but frustrates efforts to reach truly high performance. To achieve terabit rates we want to make sure that everything fits nicely into the cache. Lack of explicit cache control renders working-set tuning non-obvious. To resolve the issue Intel recently introduced “Direct Cache Access” (DCA), a memory bus hint that enables peripheral devices to directly push data into the shared L2 CPU cache. DCA makes it theoretically possible to completely bypass the constrained main memory system, but requires software support.

Scaling the combined working-set to cache size is one leg of the task, the other is to co-schedule network operations in such a way that their working-sets overlap. Existing network processing is built from very coarse-grain tasks, commonly a bottom-half and an application process. Additionally, communication between the two is heavyweight as illustrated by POSIX-imposed copy semantics and per-packet synchronization. As a result, it is infeasible that such a stack can consistently overlap working-set sizes to optimize d-cache hitrate. Nor can it execute any parallelization strategy but packet- or connection-based distribution, both of which scale poorly for certain traffic types [8].

The stack can be spread across multicore architectures more efficiently if the tasks are broken up into smaller operations, because such items can be co-scheduled to increase cache-efficiency. Spreading I/O over multiprocessors in this manner is hardly new. The CDC6600 barrel processor in the mid 1960s solved the inverse of the problem we are facing today. Then, the CPU was slower than memory and I/O slower still. The designers’ solution was to have 10 peripheral processors that waited for I/O out-of-band, to allow a simpler CPU that ran at a higher speed. Heterogeneous multicores are still common in selective domains – notably in network processing – but now the difficulty lies in maximizing memory throughput. In previous work we implemented a Gigabit payload-inspecting

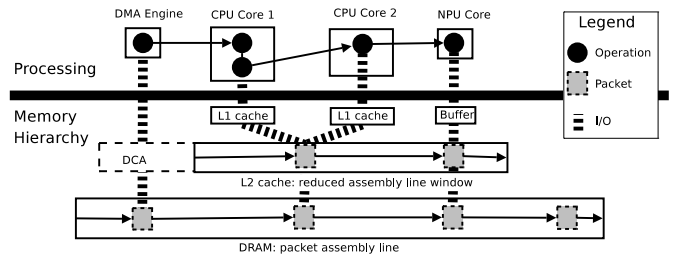


Fig. 1. The Model-T architecture

intrusion prevention system on top of the 9-core heterogeneous Intel IXP2400 network processor. Here, too, memory access proved to be the main bottleneck [9].

This paper builds on our previous work in high-speed I/O transport (Beltway Buffers [10]) and composite I/O stacks (FFPF and its actively maintained descendant Streamline¹ [11]) and draws from our experience designing high-performance networking stacks on Intel IXP1200 [12] and IXP2400 [9] network processors, but the complete architecture remains a work in progress. Specifically, throughput-oriented scheduling is lacking. As an indication of performance, we present the results of a more limited prototype stack alongside partial results obtained from the practical system.

The contribution of this paper is that it presents an architecture, *Model-T*, for OS network processing that

- 1) increases I/O throughput by overlapping working-sets and scaling the combined set to d-cache size;
- 2) increases core scalability by breaking up the stack into many small operations and scheduling these operations in parallel; and
- 3) incorporates asymmetric and heterogeneous cores by combining operation implementations for different ISAs into the same application

In the next Section (Section II) we introduce the two architectural features underlying our system. Then, in Section III, we show initial results of the new architecture using a prototype stack. In Section IV we show how the two features can be incorporated into a real OS by reusing Beltway and Streamline. In Section V we present a test-infrastructure for Model-T that can replay network traffic at speeds only limited by the system bus and memory system (as opposed to a physical link layer). We draw conclusions in Section VII.

II. ARCHITECTURE

To achieve our goals we modify network stack design in two fundamental ways: by (1) replacing coarse-grain tasks with an *assembly line* of smaller operations that can be trivially executed concurrently and by (2) introducing a throughput-oriented scheduler that indeed executes operations in parallel and in lockstep to maximize cache hitrate. Figure 1 shows the resultant architecture; we will now discuss each of the two techniques individually.

¹Sourcecode for Linux 2.6 is available from <http://netstreamline.org/>

A. Assembly line processing

First, we replace the common model of having few coarse-grained tasks with one built from many smaller processing steps. These *operations* are not independent. Each operation executes a single step in the network stack, such as IP defragmentation, TCP port routing or HTTP gzip decompression. For each packet, a stack of operations must be executed, for the most part in strict order, *as an assembly line*.

We named the architecture after the best-known example in assembly line manufacturing: the Ford Model-T automobile. In a physical assembly line, a complex manufacturing process is broken into very small operations. Dedicated workers each perform only one operation. The manufactured product moves between the workers, to minimize transport delay. As a result, workers only have a fixed, limited time to perform their operation. As the line moves forward at a fixed rate, maximal throughput is defined by the slowest worker.

Model-T applies the assembly line principle to network processing. Here, too, we set out to maximize global throughput. Designing the system around a fixed-rate conveyor belt makes bottlenecks stand out. Just as Ford could move employees among tasks to remove local bottlenecks, so can we increase or decrease the amount of resources spent on a given operation. Like the physical line, Model-T is not clocked, but moves as fast as its slowest link. We must move operations closer together if doing so improves locality of reference (cache hit-rate!) and space them further apart across cores to remove computational bottlenecks if they arise and to increase parallelization.

Pipelined processing causes predictable cache behavior. As we explained in the introduction, lack of cache control in CPUs was one difficulty in reaching high I/O rates. By giving exclusive access to the L2 cache to the highly parallelized pipeline, we minimize cache contention with other applications. By running the operations in lockstep, we also avoid contention between packets.

Besides improved d-cache utilization and trivial scalability, a design based on many loosely coupled operations has a third benefit: it enables transparent cooperation of resources with disparate Instruction Set Architectures (ISA). Model-T can therefore combine, say, an x86 control processor and an IXP coprocessor (μ Engine) into the same application. All these strong points are moot, if operations are not scheduled effectively. We now move to the discussion of the scheduler in Model-T.

B. Scheduling for throughput

In computer system I/O the throughput of an individual operation is greatly affected by its “co-runners”: the operations or applications executed concurrently on another core. The main sources of contention are cache conflicts and the memory bus bottleneck [1]. In Model-T, we aim to reduce contention through effective operation co-scheduling.

Operations are individually schedulable entities. An increase in schedulable entities increases the scheduling option space. We exploit this by scheduling as many operations

as possible across as many nodes as available at the same time. When operating in lockstep, this in turn increases d-cache hit-rate, as more operations are accessing the same data concurrently. A decrease in task granularity commonly leads to an increase in communication overhead, in the form of copying, task switching and cache conflicts. Clearly, we cannot endure extra overhead when scaling to terabit rates. We circumvent these problems through hierarchical scheduling. Instead of scheduling operations as first class hardware tasks, we schedule operation *container tasks*. These, in turn, schedule operations without providing isolation. This layering avoids protection overhead, such as saving registers or flushing the TLB, when switching between operations. This weakens isolation, but as all operations are part of the same stack hardware-enforced isolation is unnecessary. In Model-T, we implement hierarchical scheduling by having containers run Streamline, an I/O stack that connects operations through I/O streams, similar to Unix pipes. We introduce Streamline in more detail in Section IV-A. For now it suffices to say that operation and stream control, including heterogeneous issues, are taken care of. Model-T extends Streamline with runtime operation migration between cores based on throughput and with a scheduling component that replaces interrupt-based processing with the proposed *superscalar pipelined architecture*.

To maximize cache hitrate, container tasks are scheduled with long timeslices. This is a departure from interrupt-based network processing. In that model, (hardware) task switching may be frequent, which not only causes a high direct switching cost, but also an indirect cost in terms of cache-pollution. The logical extreme of long timeslices is dedicating cores solely to network processing [13]. On dual core machines, this fits with the split bottom half/userspace stack. It is less clear how such a configuration scales as we add cores.

Hierarchical scheduling limits task-switching overhead; to achieve high rates we must also limit the other two sources of communication overhead: copying and poor cache utilization. Both are trivially evaded by ensuring that every block of memory is loaded into the L2 cache only once. The pipelined architecture enables predictable cache fill-rate and static co-scheduling of operations that access the same block. Operations need not be *fully* synchronized (e.g., operation A may work on packet 1, while operation B works on packet 2), as modern L2 cache sizes may hold multiple packets at the same time. Because network processing is a largely sequential task, cache predictability is not affected by loosening constraints.

When operations run too far out of sync, the scheduler must intervene. It has two methods at its disposal: (1) prioritization of operation call sequences within each container to remove bottlenecks (and thus queuing) within the container and (2) it can initiate operation migration to another core. Because operations are interdependent, migration is limited to movement between the cores running the predecessor and successor operations or to an idle core.

The Model-T scheduler spreads operations across the the computation fabric according to these rules:

- 1 Container timeslices are maximized and only limited by

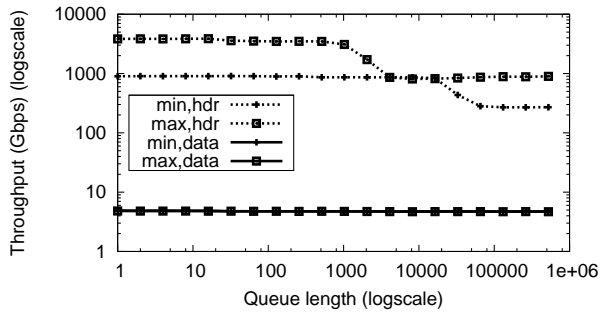


Fig. 2. Throughput for the prototype stack

latency constraints (of protocols, other active processes and with respect to overall system responsiveness). This rule minimizes task-switching overhead and cache-refresh cost.

- 2 All related network operations (e.g., the receive path) are scheduled concurrently during the same timeslice. This rule ensures that the executing operations have a high probability of overlapping working sets in the L2 d-cache.
- 3a If there are more operations than cores, some operations have to be scheduled sequentially. The pairs of operations with the shortest observed completion time will be joined recursively (using a container task).
- 3b Inversely, if there are more cores than operations, cores are dedicated to memory fetching to minimize memory delay within the pipeline, or left idle to preserve cache locality of reference.
- 4 If a queue builds up between two interconnected operations, their relative resource utilization is balanced in the manner we just described (call prioritization and operation migration).

III. PROTOTYPE

Figure 2 shows throughput obtained with a prototype implementation on an Intel Core 2 Duo T7300 2 GHz processor with 32KB split-level L1 and 4MB unified shared L2 caches. The Figure shows throughput for increasingly long packet queues for both header processing and full payload access, for minimally and maximally (1514B) sized Ethernet frames. Essentially, we pull a queue of packets in memory through the network stack. We observe that with header-only processing terabit network rates are indeed obtainable, as actual processing then occurs in only one or two cache-lines. Large packets show higher throughput as fewer packets are processed. L2 cache effects are clearly visible around 3000 large and 20000 small packet queue lengths, which coincides with the 4MB L2 cache. When accessing payload throughput is clearly memory-bound.

IV. IMPLEMENTATION

The prototype shows that near terabit speeds are within reach. The question then becomes how to reach these rates in a practical system. For Model-T we build on experience developing the FFPF high-speed packet filter [11], its descendant Streamline, and Beltway Buffers [10]. Streamline and Beltway

implement two features we require: composite tasks of small operations and shared packet queues. Model-T is the result of observing Streamline and Beltway in practice. We noticed the positive effect of increasing locality of reference by scaling working sets and timeslices, but also noted that automation of these steps requires scheduler support.

A. Filter optimization with Streamline

Streamline replaces monolithic applications with graphs of small operations connected through streams, similar to Unix pipes. It crosscuts the traditional OS layering to integrate network processing on peripheral devices, in kernelspace and within userspace processes. Streamline handles operation discovery, selection and allocation across a network of such environments: during discovery it searches through the network for operation implementations; during selection it chooses among competing implementations based on expected global throughput; during allocation, finally, it claims required resources.

Streamline supports runtime compilation, shipping and code loading to integrate constrained resources, such as processors that lack a generic load-store mechanism or preemptive multi-tasking (such as the IXP μ Engines). We extended the three step process described above as follows: during discovery Streamline searches for a compiler as well as for an implementation. A compiler substitutes a request for an operation with one for an equivalent alternative. If a compiler is found, we annotate the request and issue a discovery for the alternative. This process is recursive and leads to an exhaustive search of all registered compilers. During the allocation phase, the compilation steps are actually carried out, to generate the operation implementation on demand.

The two presented features of Streamline – resource selection and peripheral hardware integration – implement points two and three of our contributions. We observed, however, that load-time resources selection cannot give firm throughput guarantees because it leaves runtime state out of the equation. We have implemented Streamline as a combination of a Linux kernel module, hardware drivers and partial LibC library replacement, with as result that Streamline performance depends on the Linux system scheduler. With Model-T we remove this dependency to increase predictability and replace load-time selection with runtime adaptation.

B. Cache-only transport with Beltway

A prerequisite for Streamline’s resource selection algorithm is that the algorithm need not worry about implementation details. Specifically, the algorithm expects that data can flow between any two operation implementations, irrespective of where the two reside (e.g., one in the host kernel and another on an IXP). To achieve this we implemented an independent, generic transport layer, Beltway Buffers, that enables system wide I/O and minimizes overhead.

I/O throughput is limited by three factors: copying, context-switching and cache misses. Legacy I/O interfaces, such as sockets, files and pipes, incur a large penalty by imposing

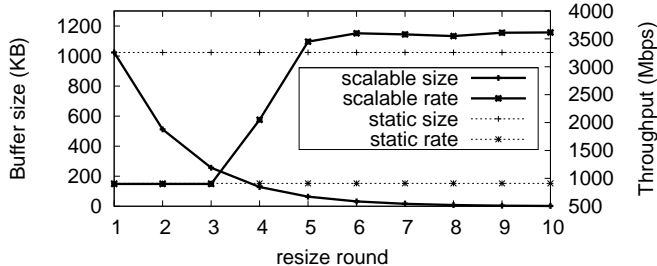


Fig. 3. Effects of working-set scaling on I/O throughput

copy semantics on their calls and tying their APIs to the application binary interface (ABI) of the kernel, which results in a user/kernel mode transition *for each packet*.

Beltway pulls the interfaces above the ABI to remove these bottlenecks. A large part of the work, which is outside of the scope of this paper, deals with guaranteeing adequate isolation without per-block access control. With this problem out of the way, Beltway can move network packets (or other blocks) through the I/O architecture through shared memory regions, without copy or per-packet switching overhead. Legacy interfaces are implemented as local function calls, as are the interfaces to Model-T operations.

Behind these interfaces, Beltway optimizes transfers as follows: (1) it moves all streaming I/O into *shared* memory regions as discussed above, (2) it integrates other important buffers (most notably the disk cache and peripheral device memory) and (3) it replaces copying with pointer forwarding when instructed to transfer data between buffers, to further reduce copy overhead.

Model-T extends Beltway by at runtime scaling buffers to fit the memory hierarchy of the currently available hardware. We observed when profiling Beltway that throughput can increase by orders of magnitude by tuning the buffers. By manually tuning we could outperform the standard Linux 2.6.22 network stack by about 100%. Figure 3 displays operation of our *resizable* sequential access (i.e., ring) buffer. This buffer has a static memory region, but adapts the amount in use at runtime to maximize locality of reference. It keeps a weighted moving average of the distance between the producer and consumer and whenever the producer wraps, it compares this value to a high and low watermark. If the distance is larger than the high watermark (85% of the ring) and it is not occupying the full memory area, it expands. Conversely, if it drops below the low watermark it temporarily reduces the size of the ring. In the example we begin with a maximally sized ring and show how it keeps reducing its size (by half). The figure also plots the effects this has on throughput: the function is largely bimodal, depending on whether the buffer fits in the (512KB) L2 cache together with the rest of the working set or not.

The resizable ring-buffer uses watermarks to guide convergence to achieve optimal rate. As the example shows, this method does increase throughput, but it does not stabilize at the cache size. For a single consumer this is acceptable, but in the Model-T architecture, where many operations access data

roughly at the same time, a window over data of sufficient length must be kept. Because the increase in throughput is sharp and centers around L2 cache size, we want to stabilize as soon as this increase is observed. Complete convergence to a stable state is not sought, because packet arrival rate ebbs and flows during execution, as does memory bus contention with other processes, if any. For these two reasons, both compile time (as performed by StreamIt) and load time (Streamline) optimizations can only give rough estimates on cache fill-rate and will give less than optimal throughput.

C. Pipelined execution and I/O aware scheduling

The scheduler greatly affects Beltway throughput, because task switching caps caching and context-switch avoidance effectiveness. Computationally oriented schedulers do not take such issues into account. With Model-T we therefore replace the default system scheduler with the I/O aware replacement introduced in Section II-B.

To achieve predictable cache utilization it is further necessary to move to pipelined I/O processing. We replace Streamline’s interrupt driven processing with an independent buffer-fill phase. Only when the (cache-adjusted) buffer contains enough packets to fill the pipeline is the entire I/O task scheduled at once. The task remains scheduled until either the buffer has been processed completely or another task’s latency bounds force preemption. If dedicated resources, such as a NIC’s DMA engine, continue to fill the buffer in the background, the pipeline can remain scheduled indefinitely.

Maximal obtainable throughput is linearly dependent on the length of the I/O task’s timeslice as a ratio of the total CPU time aggregated over all cores. For the same total time, we can choose between scheduling shorter wall time across more cores (“horizontal”) or longer slices on fewer (“vertical”). With a system scheduler, neither is explicitly selected and thus operations may be scheduled asynchronously, which is the worst of all outcomes. Model-T requires horizontal scheduling to maximize operation parallelization and thus d-cache hitrate. The design leads to a longer buffering phase, which increases median latency but dampens the effects of bursty traffic.

V. TEST INFRASTRUCTURE

Lack of affordable high speed network replay hinders OS network processing research. Partly to be able to test Model-T, but also to demonstrate that truly high speeds are indeed feasible even with today’s hardware, we are developing a packet generator that replays network traffic at rates exceeding 10Gbps NICs at a fraction of the cost, by exploiting general purpose programming on low-cost graphics processors (GPGPU). In this section, we briefly sketch the generator, to convince the reader that the work described in this paper is both timely and urgent.

GPUs are well-suited to replay traffic because (1) they are consistently the peripheral devices with the highest I/O rates and (2) they have a large amount of fast on-board memory. As a result, they themselves are not the bottleneck during replay.

Moreover, compared to network processors or FPGAs they are far more widespread – and hence, affordable.

We can insert traffic in two modes: either have the CPU fetch, as is the common case for low-end NICs, or have the GPU stream to main memory. Streaming network traces from the GPU incurs no performance penalty on the CPU except for the already often mentioned memory bottleneck. For all practical purposes this makes the replaying GPU indistinguishable from high-end programmable network card like the IXP or DAG. Because the bandwidth of all elements on the card exceeds that of either the memory bus or the PCI Express channel, the GPU can saturate the system. As of writing, the highest rates PCI Express configuration (PCIe 2.0 x32) tops at 128 Gbps peak rate and the fastest DRAM (PC3-12800 DDR3) at 204 Gbps – both considerably faster than available NICs.

Besides replay mode, the GPU can also operate as a live stream multiplexer. Through the PCI Express switch the GPU can access NIC device memory and copy this directly into main memory, acting as an overly complex polling DMA engine. More interesting is the configuration where we replay the stream to main memory more than once, to increase observed throughput. Finally, by slightly modifying packets (e.g., changing IP source address), the CPU will receive high rates of unique packets. GPU logic is increasingly programmable and highly parallel, so we will leave open the possibility of more interesting transformations for domain-specific benchmarks.

Finally, GPU based replay must be combined with a DCA-capable motherboard and CPU to benchmark Model-T.

What we take away from this discussion is that sending data at hundreds of Gbps is imminent and indeed practical even today on commodity hardware. Unless we change the way in which the OS handles network processing, the full capacity of communication links will be left unused. We suggest that Model-T is a step towards such high-speed processing.

VI. RELATED WORK

Model-T implements stream-based I/O [14]. Most directly related to Model-T are Click [15], which statically schedules a layer-2 processing network for high throughput, Dryad [16] which schedules streams across distributed resources, StreamIt [17] which compiles stream-based tasks to run on multiprocessors and optimizes cache utilization and SEDA [18], which schedules application-level operations in a highly concurrent fashion to optimize global end-to-end performance. Model-T sets itself apart from all but Click by operating at the lowest level and the highest rate of events. Unlike Click, it does scale up to application-layer processing and like Dryad it adapts operation scheduling at runtime.

Independent from stream processing, Model-T has related work in cache-aware scheduling [4]–[7]. These approaches are not tailored to I/O. Only Model-T handles working-set scaling and large scale operation co-scheduling.

VII. CONCLUSION

We have presented Model-T, a highly parallel architecture for network processing that is inherently more scalable

to multicore chips than existing approaches. Through wide scale parallel execution of operations in lockstep, Model-T maximizes L2 cache hit-rate. We have presented a prototype implementation that indeed achieves rates of above one terabit for header processing and showed that most logic necessary to implement Model-T in a real OS is already available (in the form of Beltway and Streamline). Finally, we complemented our design with a test-framework for very high throughput I/O benchmarking through general purpose GPU programming.

REFERENCES

- [1] T. Moscibroda and O. Mutlu, “Memory performance attacks: Denial of memory service in multi-core systems,” in *Proceedings of the 16th USENIX Security*, 2007.
- [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,” Tech. Rep. UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, December 2006.
- [3] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn, “Efficient operating system scheduling for performance-asymmetric multi-core architectures,” in *Proceedings of Supercomputing’07*, 2007.
- [4] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum, “Throughput-oriented scheduling on chip multithreading systems,” Tech. Rep. TR-17-04, Harvard University, August 2004.
- [5] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum, “Performance of multithreaded chip multiprocessors and implications for operating system design,” in *ATEC’05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*, (Berkeley, CA, USA), pp. 26–26, USENIX Association, 2005.
- [6] J. Nakajima and V. Pallipadi, “Enhancements for hyper-threading technology in the operating system: seeking the optimal scheduling,” in *WISS’02: Proceedings of the 2nd conference on Industrial Experiences with Systems Software*, (Berkeley, CA, USA), pp. 3–3, USENIX Association, 2002.
- [7] G. E. Suh, S. Devadas, and L. Rudolph, “A new memory monitoring scheme for memory-aware scheduling and partitioning,” in *HPCA*, pp. 117–, 2002.
- [8] P. Willmann, S. Rixner, and A. L. Cox, “An evaluation of network stack parallelization strategies in modern operating systems,” in *USENIX Annual Technical Conference, General Track*, pp. 91–96, 2006.
- [9] W. de Bruijn, A. Slowinska, K. van Rheeuwijk, T. Hruby, L. Xu, and H. Bos, “Safecard: a gigabit ips on the network card,” in *Proceedings of 9th International Symposium on Recent Advances in Intrusion Detection (RAID’06)*, (Hamburg, Germany), September 2006.
- [10] W. de Bruijn and H. Bos, “Beltway buffers: Avoiding the os traffic jam,” in *Proceedings of INFOCOM 2008*, 2008.
- [11] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis, “Ffpf: Fairly fast packet filters,” in *Proceedings of OSDI’04*, 2004.
- [12] T. Nguyen, M. Cristea, W. de Bruijn, and H. Bos, “Scalable network monitors for high-speed links: a bottom-up approach,” in *Proceedings of IPOM’04*, 06 2004.
- [13] T. Brecht, G. J. Janakiraman, B. Lynn, V. Saletore, and Y. Turner, “Evaluating network processing efficiency with processor partitioning and asynchronous i/o,” *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 4, pp. 265–278, 2006.
- [14] D. M. Ritchie, “A stream input-output system,” *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, pp. 1897–1910, 1984.
- [15] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, “The click modular router,” *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 263–297, 2000.
- [16] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: Distributed data-parallel programs from sequential building blocks,” in *Proceedings of Eurosys’07*, 2007.
- [17] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe, “Cache aware optimization of stream programs,” *SIGPLAN Not.*, vol. 40, no. 7, pp. 115–126, 2005.
- [18] M. Welsh, D. E. Culler, and E. A. Brewer, “Seda: An architecture for well-conditioned, scalable internet services,” in *Symposium on Operating Systems Principles*, pp. 230–243, 2001.