CAOS

HiPEAC COMPILATION ARCHITECTURE

Third Workshop on

Computer Architecture and Operating System Co-design (CAOS)

January 25, 2012

Paris, France

In conjunction with:

the 7th International Conference on
High-Performance and Embedded Architectures and Compilers
(HiPEAC)

Message from the Organizers

Welcome to the Third Workshop on Computer Architecture and Operating System Co-design (CAOS), and thank you for helping us to make this event successful!

This meeting brings together researchers and engineers from academia and industry to share ideas and research directions in Computer Architecture and Operating System co-design and interaction. It is never easy to follow a successful edition, but this year we have three high-quality papers, spanning from single-chip multicore systems to large servers, and covering topics ranging from scalable shared-memory and storage architectures to integration of heterogeneous accelerators in the operating system. *Scalability* is surely this year's hot topic!

As last year, the third edition of CAOS presents another great keynote: "Blue Gene/Q: Architecture, Co-Design, and the Road to Exascale", from Dr. Robert Wisniewski (IBM Research). We hope to make great keynotes one of the CAOS's traditions.

This workshop is intended to be a forum for people working on both hardware and software to get together, exchange ideas, initiate collaborations, and design future systems. In fact, as multicore and/or multithreaded architectures monopolize the market from embedded systems to supercomputers, new problems have arisen in terms of scheduling, power, temperature, scalability, design complexity, efficiency, throughput, heterogeneity, and even device longevity. In order to minimize power consumption and cost, more and more cores per chip and hardware threads (contexts) per core share internal hardware resources, from the pipeline to the memory controller. Achieving high performance with these modern systems becomes increasingly difficult. Moreover, performance is no longer the only important metric: newer metrics such as security, power, throughput, and Quality of Service are becoming first-order design constraints.

It seems clear that neither hardware nor software alone can achieve the desired performance objectives and, at the same time, comply with the aforementioned constraints. The answer to these new challenges must come from hardware-software co-design. Computer Architectures (CA) and Operating Systems (OS) should interact through well-defined interfaces, exchange run-time information, monitor application progress and needs, and cooperatively manage resources.

We thank the Program Committee and the additional reviewers for their hard work in putting together an exciting proceeding for this edition.

Roberto Gioiosa (BSC)
Omer Khan (MIT)

Organizing Committee

Organization

Workshop Co-Chairs

Roberto Gioiosa	Barcelona Supercomputing Center	Spain	roberto.gioiosa@bsc.es
Omer Khan	MIT, CSAIL	USA	okhan@csail.mit.edu

Program committee

Buyuktosunoglu, Alper	(IBM T.J. Watson, USA)
Cesati, Marco	(University of Rome Tor Vergata, Italy)
Davis, Kei	(LANL, USA)
Etsion, Yoav	(Barcelona Supercomputing Center, Spain)
Falcon, Ayose	(Intel Barcelona Research Center, Spain)
Hempstead, Mark	(Drexel University, USA)
Holt, Jim	(Freescale, USA)
Koushanfar, Farinaz	(Rice University, USA)
Kursun, Eren	(IBM T.J. Watson, USA)
Lang, Michael	(LANL, USA)
Miller, Jason	(MIT, USA)
Nikolopoulos, Dimitrios	(University of Crete, Greece)
Schirner, Gunar	(Northeastern University, USA)
Tumeo, Antonino	(PNNL, USA)
Wisniewski, Robert	(IBM T.J. Watson, USA)

Web page

http://projects.csail.mit.edu/caos/

Advance Program

10.00-10.05 **Opening**

10.05-11:00

Keynote: **Blue Gene/Q: Architecture, Co-Design, and the Road to Exascale**
Dr. Robert Wisniewski (IBM Research)

In 2004 Blue Gene made a significant impact by introducing an ultra-scalable computer with a focus on low power. After that, Blue Gene/L maintained the number 1 spot on the top500 list for an unprecedented 7 lists. In 2007 Blue Gene/P was announced and a peak 1 PF machine installed at Juelich, and Blue Gene/P garnered the top position on the green 500 list. At Supercomputing 2011 we announced Blue Gene/Q, a 208 TF per rack machine, obtaining over 2 GF/watt of computing, which obtained the number 1 position on the green 500, and a 4 rack machine was ranked number 17 on the top 500 list. Blue Gene/Q also was number 1 on the graph 500 list. The announced LLNL Sequoia machine will be a 96 rack, 20 PF machine, and will be delivered in mid 2012.

Blue Gene/Q contains innovative technology including hardware transactional memory and speculative execution, as well as mechanisms such as scalable atomic operations and a wakeup unit to help us better exploit the 17 cores and 68 threads per node. In the talk I will describe the base architecture of Blue Gene/Q include the hardware, packaging, and software with a focus on the codesign process between the applications, system software, and hardware teams that lead to the above capability. I will also describe how Blue Gene/Q is a research vehicle for helping us explore the challenges that face us on the road to exascale.

11.00-11.30 **Break**

11:30-12:00 **NUMA Implications for Storage I/O Throughput in Modern Servers**
Shoaib Akram, Manolis Marazkis, and Angelos Bilas

12.00-12.30 **Judicious Thread Migration When Accessing Distributed Shared Caches**
Keun Sup Shim, Mieszko Lis, Omer Khan, and Srinivas Devadas

12:30-13:00 **Programming and Scheduling Model for Supporting Heterogeneous Accelerators in Linux**
Tobias Beisel, Tobias Wiersema, Christian Plessl, and André Brinkmann

13.00-14:30 **Lunch**

Keynote

Blue Gene/Q: Architecture, Co-Design, and the Road to Exascale

Dr. Robert Wisniewski (IBM Research)

Abstract:

In 2004 Blue Gene made a significant impact by introducing an ultra-scalable computer with a focus on low power. After that, Blue Gene/L maintained the number 1 spot on the top500 list for an unprecedented 7 lists. In 2007 Blue Gene/P was announced and a peak 1 PF machine installed at Juelich, and Blue Gene/P garnered the top position on the green 500 list. At Supercomputing 2011 we announced Blue Gene/Q, a 208 TF per rack machine, obtaining over 2 GF/watt of computing, which obtained the number 1 position on the green 500, and a 4 rack machine was ranked number 17 on the top 500 list. Blue Gene/Q also was number 1 on the graph 500 list. The announced LLNL Sequoia machine will be a 96 rack, 20 PF machine, and will be delivered in mid 2012.

Blue Gene/Q contains innovative technology including hardware transactional memory and speculative execution, as well as mechanisms such as scalable atomic operations and a wakeup unit to help us better exploit the 17 cores and 68 threads per node. In the talk I will describe the base architecture of Blue Gene/Q include the hardware, packaging, and software with a focus on the codesign process between the applications, system software, and hardware teams that lead to the above capability. I will also describe how Blue Gene/Q is a research vehicle for helping us explore the challenges that face us on the road to exascale.

Bio:

Dr. Robert Wisniewski is the chief software architect for Blue Gene Research and manager of the Blue Gene and Exascale Research Software Team at the IBM T.J. Watson Research Facility. He is an ACM Distinguished Scientist and IBM Master Inventor. He has published over 60 papers in the area of high performance computing, computer systems, and system performance, and has filed over 50 patents. Prior to working on Blue Gene, he worked on the K42 Scalable Operating System project targeted at scalable next generation servers and the HPCS project on Continuous Program Optimization that utilizes integrated performance data to automatically improve application and system performance. Before joining IBM Research, and after receiving a Ph.D. in Computer Science from the University of Rochester, Robert worked at Silicon Graphics on high-end parallel OS development, parallel real-time systems, and real-time performance monitoring. His research interests lie in experimental scalable systems with the goal of achieving high performance by cooperation between the system and the application. He is interested in how to structure and design systems to perform well on parallel machines, and how those systems can be designed to allow user customization.

NUMA Implications for Storage I/O Throughput in Modern Servers

Shoaib Akram, Manolis Marazkis, and Angelos Bilas[†]
Foundation for Research and Technology - Hellas (FORTH)
Institute of Computer Science (ICS)
100 N. Plastira Av., Vassilika Vouton, Heraklion, GR-70013, Greece
Email: {shbakram,maraz,bilas}@ics.forth.gr

I. Abstract

Current server architectures have started to move away from traditional memory buses that do not scale and towards point-to-point interconnects for communication among processors, memories, and I/O devices. As a result, memory modules are not equidistant from all cores leading to significant differences in memory access performance from different cores. Similar to memory modules, I/O devices are connected today to processor sockets in a NUMA manner. This results in NUMA effects for transfers between I/O devices and memory banks, as well as processor I/O (PIO) accesses to I/O devices. This trend towards NUMA architectures increases complexity for buffer placement, device data transfers, and code execution, creating a complex affinity space. In this paper, we discuss problems that arise when performing I/O and present a preliminary evaluation of the impact of different types of affinity. We use a server-type system with two Intel Xeon processors, four storage controllers, and 24 solid-state-disks (SSDs). Our experiments with various machine configurations show that compared to local transfers between devices and memory, remote transfers have the potential to reduce maximum achievable throughput from 8% up to 40%. Further, for I/O-intensive applications, remote transfers can potentially increase I/O-completion time up to 130%.

II. Introduction

A predominant number of servers deployed in data-centres today use multiple processors on a single motherboard. The processors, memory modules, and the I/O devices are connected together by a cache-coherent, point-to-point interconnect [26], [4]. Such architectures result in non-uniform communication overheads between different devices and memory modules. A known problem in this direction has been the non-uniform latency of memory accesses by a processor to a local or remote memory module. Each processor has faster access to memory modules connected locally to it and slower access to the rest of the (remote) memory modules. In addition, today, accesses from one processor to a remote memory module need to traverse other processors' sockets (also called NUMA domains), interfering with local traffic. Given the current trend towards increasing number of cores in each processor and also the number of sockets, we expect that this non-uniformity will become more diverse with multiple crossings from other processors' sockets for memory accesses. Solutions have been proposed to deal with this problem at the Operating System (OS) layer [13], [8] mainly using various memory management techniques as well as hardware caching approaches. However, these approaches alone are inadequate to deal with affinity issues that arise during transfers between I/O devices and memory. The affinity that a transfer of data exhibits, e.g. from a local memory module to a local I/O device can impact performance.

Figure 1 shows a typical modern server architecture based on a point-to-point interconnect. Note that the number of processors in NUMA architectures has been increasing [12] and the trend is projected to continue. In this paper, we quantify the impact of affinity in non-uniform architectures (NUMA) on storage I/O throughput. Our initial evaluation of a server-class machine with an architecture similar to the one shown in Figure 1 shows that the maximum achievable storage throughput degrades significantly if communication is done without considering proper affinity. In particular, we observe that the maximum achievable throughput can reduce significantly if processor (A) reads data from storage devices connected to chipset (b) compared to reading from devices connected to chipset (a).

The main objective of this paper is to give an initial evaluation of the impact of affinity on storage throughput. In particular, we present the impact of remote buffer placement (improper affinity) on application throughput, device throughput, time taken for completion of OS tasks (system time) and time taken for completion of I/O requests (iowait time). We quantify this impact by placing buffers and scheduling threads manually. We use a simple classification scheme to build four configurations with different approaches to buffer placement and scheduling threads. We evaluate the performance of various applications using these configurations.

Typically in real applications, buffers are allocated in memory modules closest to the processor. However, systems try to balance the use of memory across modules to allow for

[†]Also, with the Department of Computer Science, University of Crete, P.O. Box 2208, Heraklion, GR-71409, Greece.

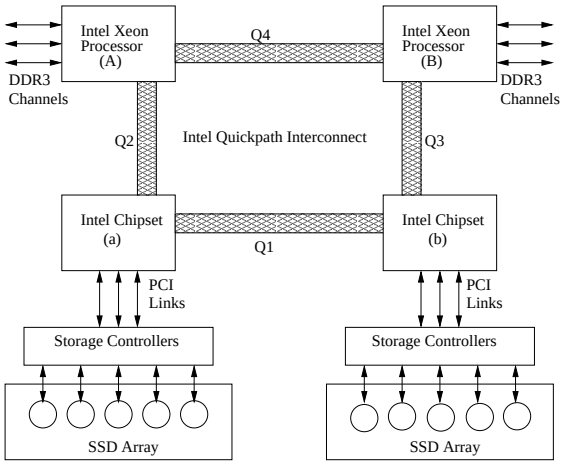


Fig. 1. The top-level architecture of a server machine with non-uniformity.

higher throughput. In addition, the system scheduler may move threads around resulting in the initiation of transfers between devices and memory modules with improper affinity: Data requested by a processor could be located on a device that is either closer to the processor or remote, as shown in Figure 1, whereas the buffers used for the transfer can have different affinity to the processor, resulting in significant variations in the observed performance.

Our work shows that compared to the configuration where transfers between devices and memory are local :

- I/O-intensive workloads suffer from 69% up to 130% increase in I/O-completion time due to remote transfers.
- Filesystem-intensive workloads suffer from 40% up to 57% increase in system time (time for performing OS-related activities) due to remote transfers.
- Throughput-oriented workloads such as state checkpointing or data-streaming suffer up to 20% drop in read/write throughput due to remote transfers.

The rest of the paper is organized as follows. Section III describes a taxonomy of NUMA affinity in modern servers involving memory and devices. In the same section, we describe four configurations with different policies for buffer placement and thread scheduling Section IV describes our methodology for evaluation and in Section V we discuss the results of our evaluation. We provide a discussion of the shortcomings of our work in Section V. We conclude this work in Section VIII.

III. I/O Affinity Taxonomy

In real applications, when a processor accesses a block device for a file, it first allocates a buffer in memory for reading a block from the block device. For instance, consider a worst-case scenario (Figure 1 where a process running on processor (A) allocates a buffer in memory module closer to processor (B) and requests a block of file to be read from the devices connected to the chipset (b). The three high-level operations are 1) issuing the I/O operation, 2) serving the I/O request, and 3) using the data that is returned. We ignore the first operation because unlike the other two operations, issuing an I/O request does not depend on the size of data. The second operation is the type of transfer (local or remote) and the third operation is the usage of data (local or remote). We further differentiate based on the type of transfer (read or write) and the type of usage (load or store).

Table I presents our taxonomy. The best case is when a transfer occurs with proper affinity between a memory module and an I/O controller that are located close to the same CPU socket. Conversely, the worst case is when the transfer buffer and the I/O controller are located in different sockets (also called NUMA domains). An even worse case is when not only the transfers are remote but the subsequent use of the data is by a processor that is located remotely to where the memory module is located. Some typical scenarios for real applications include:

- TLORP0I0 : I/O transfers are local, the transfer operation is read, and data is not used by the processor.
- TRORP0I0 : I/O transfers are remote, the transfer operation is read, and data not used by processor.
- TRORPRIR : I/O transfers are remote, transfer operation is read, and the data that is returned is accessed by remote processor.
- TLORPRIR : I/O transfers are local, transfer operation is read, and the data is used by a remote processor.
- TRORPRIR : I/O transfers are remote, transfer operation is read, and data usage is by remote processor (load).
- TLORPLIR : I/O transfers are local, transfer operation is read, and data is used by the same (local) processor, where data is returned.

The last three cases are depicted in Figure 2: circles denote CPUs or devices involved in the I/O operation. Arrows denote the transfer path taken by an I/O request. The first transfer is from chipset to memory DIMM. Next, we discuss buffer management and thread scheduling taking NUMA effects into account. Proper buffer management involves placing data in the same memory module that is connected to the socket as the storage controller responsible for the I/O operation. Thread scheduling involves running threads on the CPU that is connected to the memory module containing data needed by the CPU. In this paper, we do not propose new algorithms for scheduling and buffer placement. Instead, we place threads and buffers manually and build five configurations for evaluating the possible range in performance degradation. In order to understand the configurations, we first describe the copies that take place when data is transfered from from a device to application memory.

The I/O stack of a typical OS today is shown in Figure 3. For each I/O request made, there are two buffers involved in the transfer from the device to the application: One buffer in the application address space and one in the kernel. The placement of the application buffer is controlled in our experiments via `numactl` that is able to pin threads and buffers to specific sockets. Kernel-buffer placement cannot be controlled; I/O buffers in the kernel are part of the buffer cache and are shared by all contexts performing I/O in the kernel. Thus, a context

TABLE I
TRANSFER AFFINITY TAXONOMY.

Transfer (T)	Transfer Operation (O)	Core access (P)	Access type (I)
Local (L)	Read (R)	Local (L)	Load (R)
Remote (R)	Write (W)	Remote (R)	Store (W)
		None (0)	

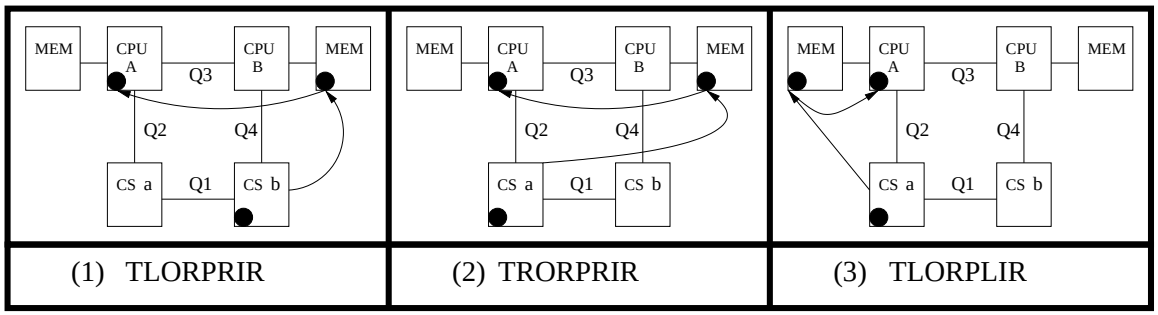


Fig. 2. Pictorial representation of three cases derived from the taxonomy described in Table I.

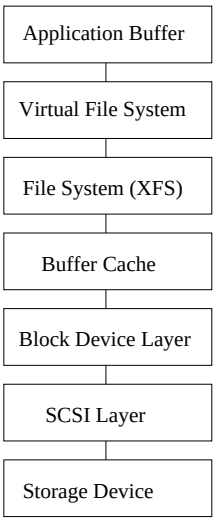


Fig. 3. I/O Stack in Modern Operating Systems.

might use a buffer that is located in any memory module. Creating different buffer pools for each socket could allow proper kernel buffer placement and use, however, requires extensive kernel modifications. In our experiments, buffer allocations are initiated by user contexts entering the kernel (we always start experiments with a clean buffer cache). This results in (properly) placing buffers initially in the socket where the user context is running. Although during each experiment buffers can be reused by other contexts performing I/O resulting in degraded affinity, this is not very pronounced due to the large memory size in our setup.

Based upon buffer placement and thread scheduling, we use five configurations shown in Table II. The axis for classification are: (1) local versus remote transfers between I/O device and memory and (2) local versus remote copy operation. This copy operation is between the application buffers and

TABLE II
CONFIGURATIONS.

Transfer (TR)	Copy Operation (CP)	Configuration
Local(L)	Local(L)	TRLCPL
Remote(R)	Remote(R)	TRRCPR
Remote(R)	Local(L)	TRRCPR
Local(L)	Remote(R)	TRRCPR

the buffers of the OS-managed cache. We manually control the source and destination of each copy operation by placing threads and their buffers appropriately via `numactl`.

IV. EVALUATION METHODOLOGY

In this section, we describe our experimental platform, applications for evaluation, and our methodology for evaluation.

A. Testbed for Evaluation

The top-level diagram of our evaluation platform is similar to the one shown in Figure 1. The server uses Intel Xeon Quadcore processors with four cores and eight hardware threads (two-way hyperthreaded). The server is equipped with two chipsets also from Intel (Tylersburg 5520). We populate the three memory slots with three DDR3 DIMMs. Each DIMM occupy a separate physical channel. We use four storage controllers (two per chipset). The storage controllers are form LSI (Megasas 9260). We use a total of 24 SSDs (Intel X-25 SLC). Each storage controller is connected to six SSDs. We create a software RAID device on top of six SSDs connected to each storage controller. Therefore, each processor has two software RAID devices that are local to it with better affinity and two that are remote with worst affinity. We use CentOS release 5.5 OS distribution with 2.6.18-194.32.1.el5 kernel (64-bit). For placing buffers and contexts, we use the `numactl` library for Linux (version 2.0.7).

B. Bandwidth Characterization of System Components

In this section, we describe the bandwidth of individual system components in order to understand the peak limitations in our system. The bandwidth of the QPI links (labeled Q1, Q2, Q3, Q4) is 24 GBytes/s. Each storage controller from LSI is able to achieve 1.6 GBytes/s. The SSDs can sustain a throughput of about 200 MBytes/s for sequential writes and 270 MBytes/s for sequential (or random) reads. To measure the memory bandwidth in our system, we use a in-house benchmark modeled after STREAM [14] called mstress. We run multiple instances of mstress and measure the memory throughput with local and remote affinity. Figure 4(a) shows our results. The peak bandwidth of storage controllers is much less than the memory subsystem and the QPI interconnect, neither of these is a potential bottleneck when performing I/O.

C. Methodology

To evaluate the impact of wrong buffer placement on application performance, we use the following benchmarks and applications:

1) zmIO: is an in-house benchmark that fully stresses the storage sub-system of our high-end server machines (4 storage controllers each capable of doing 1.6 GB/s). zmIO uses the asynchronous API of Linux for performing I/O operations [1]. zmIO issues multiple (user-defined parameter) I/O operations and keep track of the status of each of the operation in a queue called status queue. When the status queue is full, zmIO performs a blocking operation and waits for an I/O operation to complete. A new operation is issued after completing a pending operation. The completion of I/O operations by CPU and the completion of outstanding I/O operations by the storage devices happens in parallel. We run zmIO in direct mode. Note that in direct mode, zmIO performs I/O access to storage devices that does not go through the page cache in the kernel.

2) fsmark: is a filesystem stress benchmark that stresses various features of the filesystem. fs_mark runs a sequence of operations on filesystem layer. In particular, we use it to perform the operation sequence create,open,write,read, and close. We run fs_mark using 128 threads with each thread creating a single directory and 128 files within each directory. Each thread chooses a random directory and performs the specified sequence of operations on any of the files within the directory.

3) IOR: simulates checkpointing support in compute- intensive applications [18]. We use the MPI API for performing I/O operations. We run IOR on top of the XFS filesystem. We use 32 processes that checkpoint a 2 GB state to a shared file (aggregate file size is 64 GB). Each process works with a single file using sequential offsets within the single file.

4) Stream: is a synthetic application that simulates the end-to-end datapath of data streaming systems [11]. The application consists of a consumer thread that reads 64 KB records in a buffer. The consumer thread enqueues the pointer to buffers in a list of descriptors. The list has 128K entries. The producer thread reads the buffer from the list of descriptors, performs some conditioning on the buffer, updates the list of descriptors and stores the record to storage device.

5) Psearchy: is a file indexing benchmark in the MOSBENCH [10] suite. File indexing is mainly done as a backend job in data centres and web hosting facilities. We run Psearchy using multiple processes. Each processes picks a file from a shared queue of file names. Each process has a hash table for storing in-memory BDB indices. The hash tables are written to storage devices once they reach a particular size. We use 32 processes, 128 MB hash tables per process, and 2 KB reads and character oriented writes. We use 100 GB corpus, 10 MB file size, 100 files in each directory and 100 directories.

For evaluating NUMA effects, we run a workload consisting of four instances of the same application or benchmark. We assign one RAID 0 device consisting of six SSDs to each instance. Next, we define various metrics for our evaluation.

To project results to future systems with more components, it is important to use appropriate metrics for evaluation and observe how various components of the system are stressed instead of merely observing the application throughput. For this reason we use:

- Application Throughput (GB/s): The application throughput refers to the aggregate bytes accessed by the application divided by the execution time. Usually, read and write throughput is reported separately based upon the total bytes read or written during the execution time.
- Cycles per I/O (CPIO): In this work, we define and use CPIO as a new metric for characterizing behavior of applications that mainly process I/Os. We define CPIO as the total cycles spent by the application divided by the total sectors read and written by the device. We believe that CPIO is particularly important for data-centric applications that perform a one-pass over the dataset as it gives an estimate of the work performed per I/O sector. Ideally, as the number of cores increase, CPIO should remain the same. Thus, it is a measure of how well the applications scale on new generations of systems.
- Throughput per socket: For one application, we report the results in terms of throughput per socket. Because of non-uniformity in the server systems, it is important to maintain similar throughput across the entire system. We show that for one of the applications, the throughput is different for each socket depending upon the scheduling scheme.

Since CPIO is a new metric we use in this paper, we discuss it in detail below. We calculate cpio for each application by running each application in a *meaningful* configuration; applications when run, should generate I/O traffic. For instance, cases where the workload fits in the available memory and exhibit low I/O are probably not typical of future configurations since the demand for data grows faster than DRAM capacity. For this purpose, we select datasets that are big enough to not fit in memory and generate I/O throughout execution.

To calculate CPIO, we measure the average execution time breakdown as reported by the OS and consisting of user,

system, idle, and wait time. We also note the number of I/Os that occurred during the same interval. There are two issues related to the cpio calculation. First, what each of the components means and second which ones should be taken into account to come up with a meaningful metric. We next briefly explain what each component of the breakdown means.

user time refers to the time an application spends executing code in the user space. When the user application request services by the OS, the time spent is classified as *system time*. The time an application spends waiting for I/Os to complete is classified as *wait time*. *idle time* refers to the time that the application either has no more work to perform within the current quantum or because it is waiting for resources that are not available, for instance, locks. We use the modified terms called $CPIO_{iow}$and $CPIO_{sys}$respectively to describe the two components in terms of $CPIO$. In our evaluation, we use sector-size I/Os, with each sector being 512 bytes. Note that since CPU cycles proportionate to power [15], and given the increasing emphasis on energy efficiency in data centres, CPIO is an important metric.

V. Results

In this section, we describe the results of our evaluation.

A. zmIO

We run zmIO in *direct* mode, and therefore, I/O accesses do not go through the page cache in the kernel. Hence, there is no distinction between local and remote copies. For DMA transfers between devices and memory, the buffer provided by the application is used instead. Note that this buffer is aligned across the page boundary. In order to evaluate the impact of affinity on throughput of zmIO, we use the affinity taxonomy listed in Table I for describing our results. We mainly focus on three issues:

- The impact of affinity between source and destination of a transfer operation on storage throughput. Effectively, this shows how much better or worse I/O transfers can become by employing the wrong affinity.
- The impact of processor memory accesses on data transfers, in combination with affinity. Typically, programs that perform I/O also use CPU cycles to process data. We examine the impact of accessing memory from the processor to I/O transfer throughput.
- The impact of contention between processor and I/O memory accesses on maximum achievable memory throughput. Although this issues is similar to above, in this case we are interested in whether simultaneous accesses from processors and I/O devices to memory result in a degradation of the maximum throughput, rather than the impact on I/O throughput.

To evaluate the impact of affinity between source and destination on storage bandwidth, we run multiple instances of zmIO and measure the throughput. Figure 4(b) shows the throughput of zmIO with up to eight instances. The reduction in throughput with more than two instances and remote affinity is up to 40%.

At this point, it should be mentioned that we measured throughput of zmIO using different machine configurations. We observed that NUMA effects on throughput of zmIO depend on a number of factors including OS distribution, the version of Linux kernel, version of `numactl` library, and even the type of motherboard. We observed that while Figure 4(b) shows a 40% drop in throughput, one of the machine configuration with a newer OS distribution and kernel, we observed 8% drop in throughput due to remote transfers. We believe that the range of degradation that an application can potentially suffer due to remote transfers is important to quantify and improve.

Next, we optionally perform a summation operation over all the bytes returned by the I/O read to observe the impact of TLORPRIL and TRORPRIL. The variable that stores the sum of the bytes is pinned in memory. The size of each transfer is 1 MByte. Figure 4(b) shows results with zmIO touching the data. Note that the absolute throughput for local transfers and local use (TLORPLIR) is lower to that of TLORP0I0 because both the outstanding I/Os and the summation operation accesses memory simultaneously. The reduction in throughput for TLORPRIR when the data is used by a remote processor is 5% with four instances of the benchmark. Beyond four instances, TLORPRIL and TLORPLIL behave similarly. We do not show results for TRORPLIR as it is also bounded by the bandwidth of remote transfer operation and behaves similar to the second case (TRORPRIR).

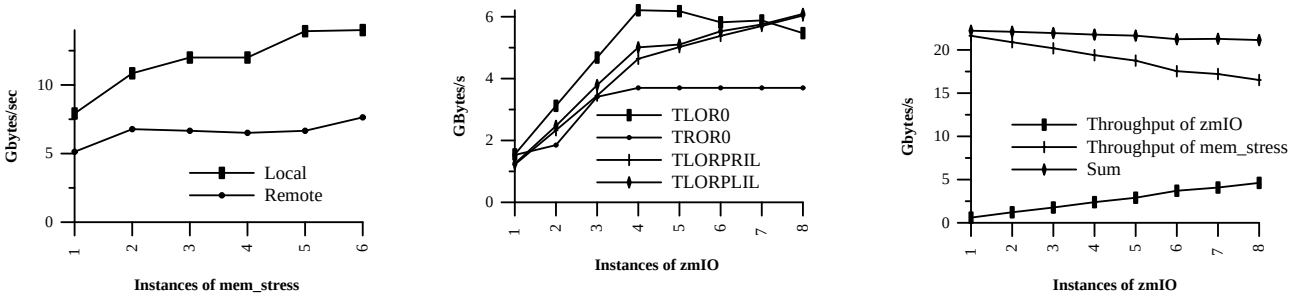
Next, we show how memory contention can hurt the performance of storage I/O throughput in case of TLORPLIR in Figure 4(c). We run instances of zmIO and mstress together. We run up to eight instances of zmIO. Neither mstress nor zmIO is bottlenecked by the CPU in this experiment. We run zmIO in TLORPLIR mode. The absolute throughput of zmIO drops by 23% for eight instances when there is contention for memory throughput i.e., mstress is running. The sum of memory bandwidth used by zmIO and mstress together is never greater than 22 GBytes/s which is the maximum memory bandwidth in the system.

B. fsmark

We discuss the results for fsmark in terms of cycles per I/O. Since fsmark mostly perform operations related to the filesystem, the system time is high. Also, due to contention from multiple threads for I/O devices, iowait time is high. Figure 5(a) shows the breakdown of CPIO in terms of $CPIO_{sys}$ and $CPIO_{iow}$. Remote transfers (TRRCPR) result in a 40% increase in $CPIO_{sys}$ compared to local transfers (TRLCPL). Also if transfers are local, remote memory copy operation (TRLCPR) result in a 15% increase in $CPIO_{sys}$ compared to TRLCPL. There is a 130% increase in $CPIO_{iow}$ due to remote transfers. The difference in $CPIO_{iow}$ due to remote copies is not noticeable.

C. Psearchy

The results for Psearchy are shown in Figure 5(b). Again, we discuss the results in terms of the cycles per I/O metric.

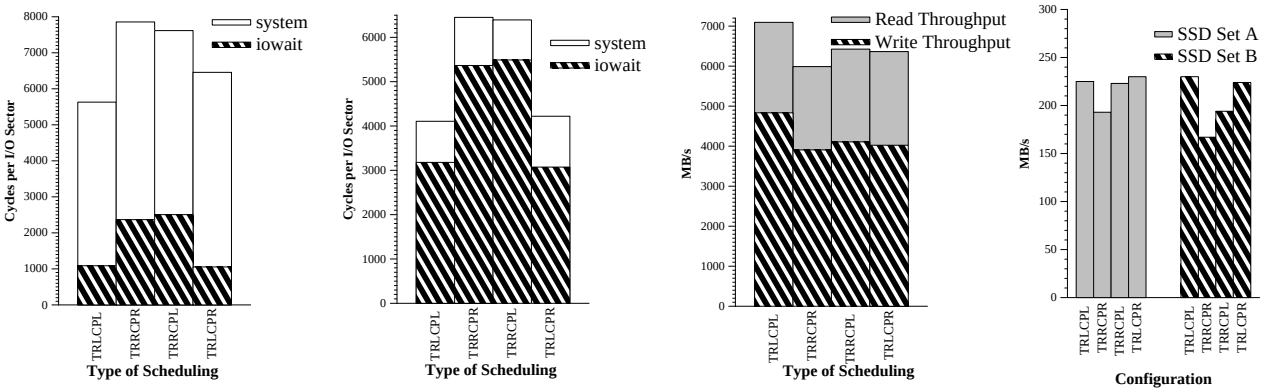


(a) Memory throughput

(b) NUMA affinity results for zmIO

(c) Contention results

Fig. 4. Results for mstress and zmIO.



(a) fsmark

(b) Psearchy

(c) IOR

(d) Stream

Fig. 5. NUMA affinity results for benchmarks and real applications.

First, we observe that remote transfers result in an increase in $CPIO_{sys}$ and $CPIO_{iow}$. However, remote copies does not show a noticeable difference. In particular, TRRCPR results in a 57% and 69% increase in $CPIO_{sys}$ and $CPIO_{iow}$ respectively relative to TRRCPL.

D. IOR

We report read and write throughput of IOR for different configurations in Figure 5(c). Note that for IOR, read operations can potentially complete in memory and thus the aggregate throughput in Figure 5(c) goes up to 7 GB/s. We observe that the read throughput decreases by 16% for the worst case (TRRCPR) compared to the best case (TRLCPL). Similarly, write throughput decreases by 19% due to remote transfers.

E. Stream

Figure 5(d) shows the results for the streaming workload. We show the throughput observed on each of the two sets of SSDs. Note that one set of 12 SSDs is connected to two storage controllers. Compared to TRLCPL, we observe a 14% and 27% drop in throughput respectively for the two set of SSDs in case of TRRCPR.

VI. Summary and Discussion

In this section, we first summarize the results of our evaluation. We then provide implications of our results for other important data-centric applications. We also discuss the shortcomings of our methodology for evaluation.

A. Summary of Results

We summarize our results as follows:

- Applications that are I/O-intensive suffer from 70% up to 130% increase in iowait time and from 40% up to 57% increase in system time due to remote transfers.
- For streaming workloads, remote transfers can potentially result in asymmetric throughput across the system i.e., some (NUMA) domains can provide more throughput compared to other domains.
- Checkpointing applications can potentially suffer a 20% degradation in write throughput due to remote transfers.
- Finally, raw device throughput, as measured by microbenchmarks such as zmIO, can drop from 8% up to 40% depending upon the machine configuration.

B. Discussion

Our main purpose is to discuss the I/O behavior of many emerging data-centric applications. In particular, we are interested in NUMA affinity effects on the performance of these applications. The applications we collected for evaluation comes from various domains. In particular, these applications are part of various benchmark suites including PARSEC [6], MOSBENCH [10], two OLTP workloads from the TPC foundation, and emerging data stores. A brief description of the

TABLE III
APPLICATIONS AND DATA SETS FOR EVALUATION.

Application	Description
zmIO	I/O subsystem stress test: direct mode (D) or through VFS.
fs_mark	File system stress test.
IOR	Application checkpointing.
Psearchy	File indexing: Directories (D) can be small (L) or large (L); files (F) can be small (L) or large (L).
Dedup	File compression: Files can be small (S) or Large (L).
Ferret	Content similarity search: Files could be Small (S) or Large(L).
Metis	Mapredce library: Word Count (C) or Linear Regression (LR).
Borealis	Data streaming: Record size could be 64 KB (Bor64), 128 bytes (Bor128), or 1 KB (Bor1024)
HBase	Non-relational database.
BDB	Key-value data store.
TPC-C	OLTP workload (Warehouse).
TPC-E	OLTP workload (Stock broker).
Tarrif	Profiling of Call Detail Records.

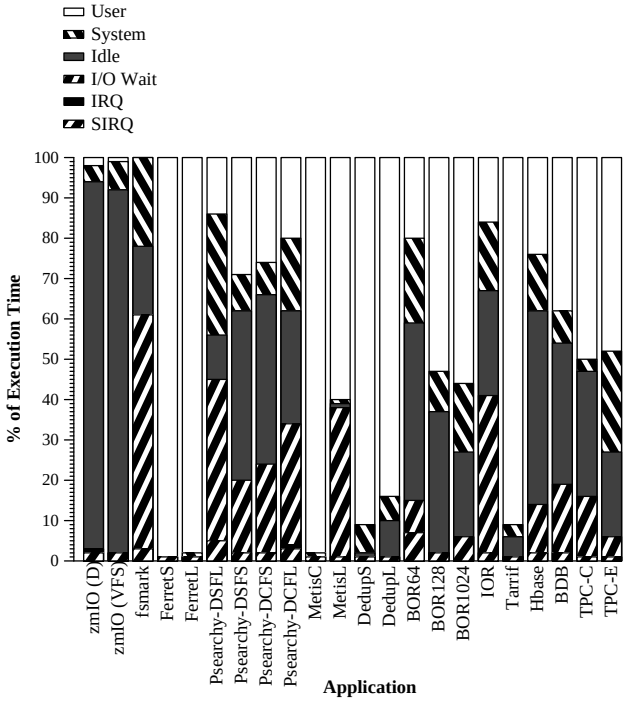


Fig. 6. Breakdown of time spent by various applications in terms of user, system, idle, iowait, serving hardware interrupts (IRQ) and serving software interrupts (SIRQ).

applications along with the type of data sets is given in Table III.

In terms of I/O behavior, most applications in Table III does not have high system or iowait times. Further, most applications does not stress the storage subsystem in a manner similar to applications we evaluate in Section V. For this reason, using different configurations do not show a noticeable difference in application throughput, CPIO, or physical device throughput. We suspect two potential reasons for this behavior as follows:

Figure 6 shows the breakdown of execution time of the applications in Table III in terms of user, system, idle, and iowait time. The breakdown is collected by running one instance of each application on top of a software RAID device consisting of 24 SSD devices. We note from the figure that most applications exhibit neither a significant component of system time nor iowait time. This lead us to the conclusion that in current NUMA systems, transfers from remotely located devices are detrimental to performance only if the application exhibit significant system or iowait time.

Finally, our experimental results, performed under controlled circumstances, strongly suggest that the kernel allocates buffers for paging purposes locally. Nevertheless, we can not manually control the placement of kernel buffers. Most applications in Table III have complex runtime layers and a large user-level application code base. Therefore, proper placement of kernel buffers can not be guaranteed.

VII. RELATED WORK

Much work has been done for NUMA-aware process scheduling and memory management in the context of shared memory multiple processors [24], [16], [23]. Here, we discuss recent work for modern server machines with multiple sockets on a single motherboard.

With the trend towards multiple cores on a single processor chip in commodity desktop and server machines, there is no longer a one-to-one mapping between I/O devices (network interface cards (NIC) or storage controllers) and processing resources (cores, virtual threads or even processors in motherboards with multiple sockets). For instance, a network interface card (NIC) can route the incoming traffic pertaining to a particular socket to a specific core and the rest of traffic to some other core. Recent 10 GBit/s Ethernet NICs from Intel (IX10GBE) provide multiple hardware queues and mechanisms to associate each queue in hardware to a particular software queue (which in turn is bind to a single core) [3], [2].

NUMA memory management is the problem of assigning memory in a NUMA processor such that threads use memory located next to the processor that they mostly run. These issues are discussed in the realm of traditional multiprocessor systems in [9], [17]. Recently, with multiple cores becoming commonplace, commodity OS developers have started to invest efforts to provide a NUMA API for programmers [5].

The authors in [20], [19] quantify NUMA effects in the memory subsystem of Xeon 5520 processor from Intel. The authors report that current memory controllers favor remote memory accesses to local memory accesses which implies that scheduling for data locality is not always a good idea. Also, they show that throughput of remote memory accesses are limited by QPI bandwidth. In this work, we show that along with remote memory accesses, accessing remote I/O devices

can also hurt performance of realistic workloads.

Recently, there is a surge in literature dealing with thread scheduling for modern servers. The authors in [25], [7] discuss scheduling policies that address shared resource contention. Their scheduling policies are built on a classification scheme for threads and addresses contention in the memory subsystem. In this paper, we use the transfer path from I/O devices to physical memory and the processor that subsequently uses the data to classify our scheduling policies.

Finally, energy efficiency in data centres is becoming more and more important. In [21], [22], the authors discuss the issues of co-scheduling processes considering both memory bandwidth and potential of frequency scaling.

VIII. Conclusions

In this paper, we described a problem in modern server machines that use point-to-point interconnects to connect CPUs, memory and devices. We discuss the performance degradation on applications if processes access data from memory modules or devices that are located remotely to the processor. As systems are built with more CPUs and sockets, with each CPU having many cores and various memory modules, the performance degradation due to the presence of NUMA affinity in the system will increase. We propose a taxonomy based upon the transfers from storage devices to memory modules and the use of data by the process running on the local or the remote socket. We describe four configurations with different buffer placement and process scheduling policies. We classify the configurations based upon how the transfers occur between the storage devices and the kernel memory, and from the kernel memory to the buffer reserved by the application. Our results show that NUMA effects are particularly degrading for I/O-intensive applications. As systems become more and more heterogeneous, a more general solution to the placement and scheduling problem will become essential for NUMA servers.

IX. Acknowledgments

We thankfully acknowledge the support of the European Commission under the 7th Framework Programs through the IOLANES (FP7-ICT-248615), HiPEAC2 (FP7-ICT-217068), and SCALUS (FP7-PEOPLE-ITN-2008-238808) projects. We are thankful to Yannis Klonatos for modifications to the original fsmark benchmark, Michail Flouris for modifications to Psearchy, and Zoe Sebepou and Markos Fountoulakis for providing the zmIO benchmark.

References

[1] Kernel Asynchronous I/O (AIO) Support for Linux . http://lse.sourceforge.net/io/aio.html .

[2] Receive flow steering . http://lwn.net/Articles/382428/ .

[3] Receive packet steering . http://lwn.net/Articles/362339/ .

[4] Advanced Micro Devices, Inc. AMD HyperTransport™ Technology. http://www.amd.com.

[5] A.Kleen. A numa api for linux. In *http://www.firstfloor.org/ andi/numa.html.*, 2004.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT'08, pages 72–81, New York, NY, USA, 2008. ACM.

[7] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali. A case for numa-aware contention management on multicore systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT'10, pages 557–558, New York, NY, USA, 2010. ACM.

[8] M. J. Bligh. Linux on numa systems. In *Proceedings of the Linux Symposium*, 2004.

[9] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but effective techniques for numa memory management. In *Proceedings of the twelfth ACM symposium on Operating systems principles*, SOSP'89, pages 19–31, New York, NY, USA, 1989. ACM.

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

[11] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. etintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *In CIDR*, 2003.

[12] M. Feldman. Dell unveils eight-socket hpc box. In *http://www.hpcwire.com/features/Dell-Unveils-Eight-Socket-HPC-Server-116201574.html*. HPCwire, 2011.

[13] http://lse.sourceforge.net/numa/. Linux Support for NUMA Hardware.

[14] John D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. http://www.cs.virginia.edu/stream/ .

[15] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid. Server engineering insights for large-scale online services. *IEEE Micro*, 30:8–19, July 2010.

[16] R. P. LaRowe, Jr., C. S. Ellis, and L. S. Kaplan. The robustness of numa memory management. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, SOSP'91, pages 137–151, New York, NY, USA, 1991. ACM.

[17] R. P. Larowe, Jr. and C. Schlatter Ellis. Experimental comparison of memory management policies for numa multiprocessors. *ACM Trans. Comput. Syst.*, 9:319–363, November 1991.

[18] llnl.gov. ASC Sequoia Benchmark Codes . https://asc.llnl.gov/sequoia/benchmarks/.

[19] Z. Majo and T. R. Gross. Memory management in numa multicore systems: trapped between cache contention and interconnect overhead. In *Proceedings of the international symposium on Memory management*, ISMM'11, pages 11–20, New York, NY, USA, 2011. ACM.

[20] Z. Majo and T. R. Gross. Memory system performance in a numa multicore multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR'11, pages 12:1–12:10, New York, NY, USA, 2011. ACM.

[21] A. Merkel and F. Bellosa. Memory-aware scheduling for energy efficiency on multicore processors. In *Proceedings of the 2008 conference on Power aware computing and systems*, HotPower'08, pages 1–1, Berkeley, CA, USA, 2008. USENIX Association.

[22] A. Merkel, J. Stoess, and F. Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European conference on Computer systems*, EuroSys'10, pages 153–166, New York, NY, USA, 2010. ACM.

[23] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the 5th international symposium on Memory management*, ISMM'06, pages 84–94, New York, NY, USA, 2006. ACM.

[24] M. Steckermeier and F. Bellosa. Using locality information in userlevel scheduling. Technical report, University Erlangen-Ng, IMMD IV, 1995.

[25] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS'10, pages 129–142, New York, NY, USA, 2010. ACM.

[26] D. Ziakas, A. Baum, R. A. Maddox, and R. J. Safranek. Intel® quickpath interconnect architectural features supporting scalable system architectures. *High-Performance Interconnects, Symposium on*, 0:1–6, 2010.

Judicious Thread Migration When Accessing Distributed Shared Caches

Keun Sup Shim, Mieszko Lis, Omer Khan and Srinivas Devadas
Massachusetts Institute of Technology

***Abstract*—Chip-multiprocessors (CMPs) have become the mainstream chip design in recent years; for scalability reasons, designs with high core counts tend towards tiled CMPs with physically distributed shared caches. This naturally leads to a Non-Uniform Cache Architecture (NUCA) design, where on-chip access latencies depend on the physical distances between requesting cores and home cores where the data is cached. Improving data locality is thus key to performance, and several studies have addressed this problem using data replication and data migration.**

In this paper, we consider another mechanism, hardware-level thread migration. This approach, we argue, can better exploit shared data locality for NUCA designs by effectively replacing multiple round-trip remote cache accesses with a smaller number of migrations. High migration costs, however, make it crucial to use thread migrations judiciously; we therefore propose a novel, on-line prediction scheme which decides whether to perform a remote access (as in traditional NUCA designs) or to perform a thread migration at the instruction level. For a set of parallel benchmarks, our thread migration predictor improves the performance by 18% on average and at best by 2.3X over the standard NUCA design that only uses remote accesses.

I. Background

In the recent years, transistor density has continued to grow [13] and Chip Multiprocessors (CMPs) with four or more cores on a single chip have become common in the commodity and server-class general-purpose processor markets [25]. To further improve performance and use the available transistors more efficiently, architects are resorting to medium and large-scale multicores both in academia (e.g., Raw [31], TRIPS [26]) and industry (e.g., Tilera [12], [4], Intel TeraFLOPS [29]), and industry pundits are predicting 1000 or more cores in a few years [5].

With this trend towards massive multicore chips, a tiled architecture where each core has a slice of the last-level on-chip cache has become a popular design, and these physically distributed per-core cache slices are unified into one large, logically shared cache, known as the Non-Uniform Cache Architecture (NUCA) [18]. In the pure form of NUCA, only one copy of a given cache line is kept on chip, maximizing the effective on-chip cache capacity and reducing off-chip access rates. In addition, because only one copy is ever present on-chip, no two caches can disagree about the value at a given address and cache coherence is trivially ensured. A private per-core cache organization, in comparison, would need to rely on a complex coherence mechanism (e.g., a directory-based coherence protocol); these mechanisms not only pay large area costs but also also incur performance costs because repeated cache invalidations are required for shared data with frequent writes. NUCA obviates the need for such coherence overhead.

The downside of NUCA designs, however, is high on-chip access latency, since every access to an address cached remotely must cross the physical distances between the requesting core and the *home* core where the data can be cached. Therefore, various NUCA and hybrid designs have been proposed to improve data locality, leveraging data migration and replication techniques previously explored in the NUMA context (e.g., [30]). These techniques move private data to its owner core and replicate read-only shared data among the sharers at OS level [11], [15], [1] or aided by hardware [33], [8], [28]. While these schemes improve performance on some kinds of data, they still do not take full advantage of spatio-temporal locality and rely on remote cache accesses with two-message round trips to access read/write shared data cached on a remote core.

To address this limitation and take advantage of available data locality in a memory organization where there is only one copy of data, we consider another mechanism, *fine-grained hardware-level thread migration* [19], [20]: when an access is made to data cached at a remote core, the executing thread is simply migrated to that core, and execution continues there. When several consecutive accesses are made to data assigned to a given core, migrating the thread context allows the thread to make a sequence of local accesses on the destination core rather than pay the performance penalty of the corresponding remote accesses, potentially better exploiting data locality. Due to the high cost of thread migration, however, it is crucial to judiciously decide whether to perform remote accesses (as in traditional NUCA designs) or thread migrations, a question which has not been thoroughly explored.

In this paper, we explore the tradeoff between the two different memory access mechanisms and answer the question of when to migrate threads instead of performing NUCA-style remote accesses. We propose a novel, on-line prediction scheme which detects the first instruction of each memory instruction sequence in which every instruction accesses the same home core and decides to migrate depending on the length of this sequence. This decision is done at instruction granularity. With a good migration predictor, thread migration can be considered as a new means for memory access in NUCA designs, that is complementary to remote access.

In the remainder of this paper,

- we first describe two memory access mechanisms – *remote cache access* and *thread migration* – and explain

the tradeoffs between the two;
- we present a novel, PC-based migration prediction scheme which decides at instruction granularity whether to perform a remote access or a thread migration;
- through simulations of a set of parallel benchmarks, we show that thread migrations with our migration predictor result in a performance improvement of 18% on average and at best by 2.3X compared to the baseline NUCA design which only uses remote accesses.

II. Memory Access Framework

NUCA architectures eschew capacity-eroding replication and obviate the need for a coherence mechanism entirely by combining the per-core caches into one large logically shared cache [18]. The address space is divided among the cores in such a way that each address is assigned to a unique *home core* where the data corresponding to the address can be cached; this necessitates a memory access mechanism when a thread wishes to access an address not assigned to the core it is running on. The NUCA architectures proposed so far use a *remote access* mechanism, where a request is sent to the home core and the data (for loads) or acknowledgement (for writes) is sent back to the requesting core.

In what follows, we first describe the remote access mechanism used by traditional NUCA designs. We also describe another mechanism, *hardware-level thread migration*, which has the potential to better exploit data locality by moving the thread context to the home core. Then, we explore the tradeoff between the two and present a memory access framework for NUCA architectures which combines the two mechanisms.

A. Remote Cache Access

Since on-chip access latencies are highly sensitive to the physical distances between requesting cores and home cores, effective *data placement* is critical for NUCA to deliver high performance. In standard NUCA architectures, the operating system controls memory-to-core mapping via the existing virtual memory mechanism: when a virtual address is first mapped to a physical page, the OS chooses where the relevant page should be cached by mapping the virtual page to a physical address range assigned to a specific core. Since the OS knows which thread causes a page fault, more sophisticated heuristics can be used: for example, in a first-touch-style scheme, the OS can map the page to the core where the thread is running, taking advantage of data access locality. For maximum data placement flexibility, each core might include a Core Assignment Table (CAT), which stores the home core for each page in the memory space. Akin to a TLB, the per-core CAT serves as a cache for a larger structure stored in main memory. In such a system, the page-to-core assignment might be made when the OS is handling the page fault caused by the first access to the page; the CAT cache at each core is then filled as needed[1].

[1]Core Assignment Table (CAT) is not an additional requirement for our framework. Our memory access framework can be integrated with any data placement scheme.

Under the remote-access framework, all non-local memory accesses cause a request to be transmitted over the interconnect network, the access to be performed in the remote core, and the data (for loads) or acknowledgement (for writes) to be sent back to the requesting core: when a core C executes a memory access for address A, it must

1) compute the *home* core H for A (e.g., by consulting the CAT or masking the appropriate bits);
2) if $H = C$ (a *core hit*),
   a) forward the request for A to the cache hierarchy (possibly resulting in a DRAM or next-level cache access);
3) if $H \neq C$ (a *core miss*),
   a) send a remote access request for address A to core H;
   b) when the request arrives at H, forward it to H's cache hierarchy (possibly resulting in a DRAM access);
   c) when the cache access completes, send a response back to C;
   d) once the response arrives at C, continue execution.

Accessing data cached on a remote core requires a potentially expensive two-message round-trip: unlike a private cache organization where a coherence protocol (e.g., directory-based protocol) would take advantage of spatial and temporal locality by making a copy of the block containing the data in the local cache, a traditional NUCA design must repeat the round-trip *for every remote access*. Optimally, to reduce remote cache access costs, data private to a thread should be assigned to the core the thread is executing on or to a nearby core; threads that share data should be allocated to nearby cores and the shared data assigned to geographically central cores that minimize the average remote access delays. In some cases, efficiency considerations might dictate that critical portions of shared read-only data be replicated in several per-core caches to reduce overall access costs. For shared read/write data cached on a remote core (which are not, in general, candidates for replication), a thread still needs to perform remote accesses.

B. Thread Migration

In addition to the remote access mechanism, fine-grained, hardware-level thread migration has been proposed to exploit data locality for NUCA architectures [19], [20]. A thread migration mechanism brings the *thread* to the locus of the data instead of the other way around: when a thread needs access to an address cached on another core, the hardware efficiently migrates the thread's execution context to the core where the memory is (or is allowed to be) cached and continues execution there.

If a thread is already executing at the destination core, it must be evicted and migrated to a core where it can continue running. To reduce the necessity for evictions and amortize the latency of migrations, cores duplicate the architectural context (register file, etc.) and allow a core to multiplex execution among two (or more) concurrent threads. To prevent deadlock,

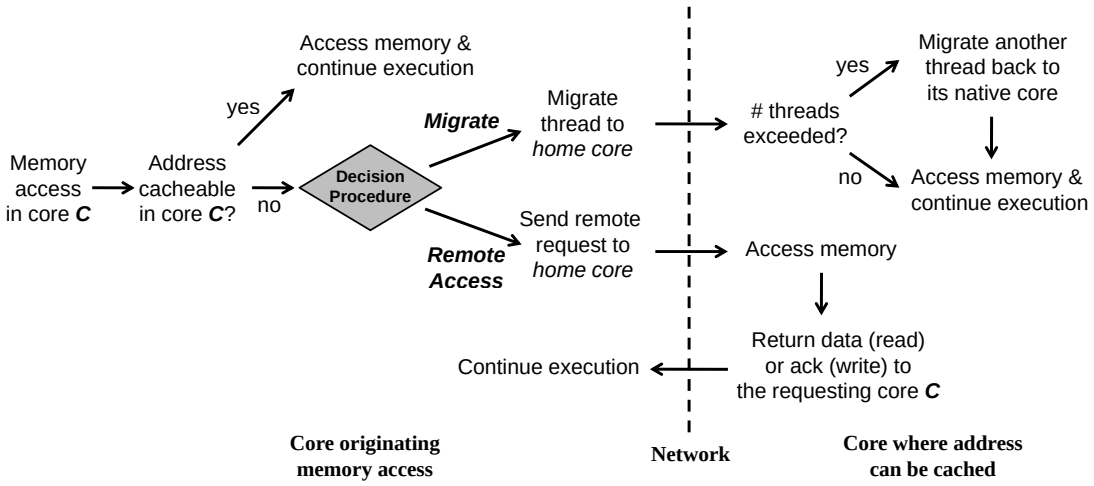


Fig. 1. In NUCA architectures, memory accesses to addresses not assigned to the local core result in remote data accesses. Here, we add another mechanism for accessing remote memory – migrating a thread to the home core where the data is cached. In this paper, we propose an on-line decision scheme which efficiently decides between a remote access and a thread migration for every memory access.

one context is marked as the *native context* and the other is the *guest context*: a core's native context may only hold the thread that started execution on that core (called the thread's *native core*), and evicted threads must migrate to their native cores to guarantee deadlock freedom [10].

Briefly, when a core C running thread T executes a memory access for address A, it must

1) compute the *home* core H for A (e.g., by consulting the CAT or masking the appropriate bits);
2) if $H = C$ (a *core hit*),
   a) forward the request for A to the cache hierarchy (possibly resulting in a DRAM access);
3) if $H \neq C$ (a *core miss*),
   a) interrupt the execution of the thread on C (as for a precise exception),
   b) migrate the microarchitectural state to H via the on-chip interconnect:
      i) if H is the native core for T, place it in the native context slot;
      ii) otherwise:
         A) if the guest slot on H contains another thread T', evict T' and migrate it to its native core N'
         B) move T into the guest slot for H;
   c) resume execution of T on H, requesting A from its cache hierarchy (and potentially accessing backing DRAM or the next-level cache).

Although the migration framework requires hardware changes to the baseline NUCA system (since the core itself must be designed to support efficient migration), it migrates threads directly over the interconnect network to achieve the shortest possible migration latencies, which is faster than other thread migration approaches (such as OS-level migration or Thread Motion [24], which uses special cache entries to store thread contexts and leverages the existing cache coherence protocol to migrate threads). In terms of a thread context size that needs to be migrated, the relevant architectural state in a 64-bit x86 processor amounts to about 3.1Kbits (16 64-bit general-purpose registers, 16 128-bit floating-point registers and special purpose registers, e.g., *rflags*, *rip* and *mxcsr*), which is the context size we are assuming in this paper. The thread context size may vary depending on the architecture; in the Tilera TILEPro64 [4], for example, it amounts to about 2.2Kbits (64 32-bit registers and a few special registers).

C. Hybrid Framework

We propose a hybrid memory access framework for NUCA architectures by combining the two mechanisms described: each core-miss memory access may either perform the access via a remote access as in Section II-A or migrate the current execution thread as in Section II-B. The hybrid architecture is illustrated in Figure 1. For each access to memory cached on a remote core, a decision algorithm determines whether the access should migrate to the target core or execute a remote access.

As discussed earlier, the approach of migrating the thread context can potentially better take advantage of spatiotemporal locality: where a remote access mechanism would have to make repeated round-trips to the same remote core to access its memory, thread migration makes a one-way trip to the core where the memory can be cached and continues execution there; unless every other word accessed resides at a different core, it will make far fewer network trips.

At the same time, we need to consider the cost of thread migration: given a large thread context size, the thread migration cost is much larger than the cost required by remote-access-only NUCA designs. Therefore, when a thread is migrated to another core, it needs to make several *local* memory accesses to make the migration "worth it." While some of this can be addressed via intelligent data layout [27] and memory access reordering at the compiler level, occasional "one-off" accesses

seem inevitable and migrating threads for these accesses will result in expensive back-and-forth context transfers. If such an access can be predicted, however, we can adopt a hybrid approach where "one-off" accesses are executed under the remote access protocol, and migrations handle sequences of accesses to the same core. The next section discusses how we address this decision problem.

III. Thread Migration Prediction

As described in Section II, it is crucial for the hybrid memory access architecture (remote access + thread migration) to make a careful decision whether to follow the remote access protocol or the thread migration protocol. Furthermore, because this decision must be taken on every access, it must be implementable as efficient hardware. Since thread migration has an advantage over the remote access protocol for multiple contiguous memory accesses to the same location but not for "one-off" accesses, our migration predictor focuses on detecting such memory sequences that are worth migrating.

A. Detection of Migratory Instructions

Our migration predictor is based on the observation that sequences of consecutive memory accesses to the same home core are highly correlated with the program (instruction) flow, and moreover, these patterns are fairly consistent and repetitive across the entire program execution. At a high level, the predictor operates as follows:

1) when a program first starts execution, it basically runs as on a standard NUCA organization which only uses remote accesses;
2) as it continues execution, it keeps monitoring the home core information for each memory access, and
3) remembers each first instruction of every sequence of multiple successive accesses to the same home core;
4) depending on the length of the sequence, marks the instruction either as a *migratory instruction* or a *remote-access instruction*;
5) the next time a thread executes the instruction, it migrates to the home core if it is a migratory instruction, and performs a remote access if it is a remote-access instruction.

The detection of *migratory instructions* which trigger thread migrations can be easily done by tracking how many consecutive accesses to the same remote core have been made, and if this count exceeds a threshold, marking the instruction to trigger migration. If it does not exceed the threshold, the instruction is marked as a remote-access instruction, which is the default state. This requires very little hardware resources: each thread tracks (1) *Home*, which maintains the home location (core ID) for the current requested memory address, (2) *Depth*, which indicates how many times so far a thread has contiguously accessed the current home location (i.e., the *Home* field), and (3) *Start PC*, which keeps record of the PC of the very first instruction among memory sequences that accessed the home location that is stored in the *Home* field. We separately define the depth threshold θ, which indicates the depth at which we determine the instruction as migratory. With a 64-bit PC, 64 cores (i.e., 6 bits to store the home core ID) and a depth threshold of 8 (3 bits for the depth field), it requires a total of 73 bits; even with a larger core count and a larger threshold, fewer than 100 bits are sufficient to maintain this data structure. When a thread migrates, this data structure needs to be transferred together with its 3.1Kbit context (cf. II-B), resulting in 3.2Kbits in total. In addition, we add one bit to each instruction in the instruction cache (see details in Section III-B) indicating whether the instruction has been marked as a migratory instruction or not, a negligible overhead.

The detection mechanism is as follows: when a thread T executes a memory instruction for address A whose PC = P, it must

1) compute the *home* core H for A (e.g., by consulting the CAT or masking the appropriate bits);
2) if $Home = H$ (i.e., memory access to the same home core as that of the previous memory access),
   a) if $Depth < \theta$,
      i) increment $Depth$ by one, then if $Depth = \theta$, $StartPC$ is marked as a migratory instruction.
3) if $Home \neq H$ (i.e., a new sequence starts with a new home core),
   a) if $Depth < \theta$,
      i) $StartPC$ is marked as a remote-access instruction[2];
   b) reset the entry (i.e., $Home = H$, $PC = P$, $Depth = 1$).

Figure 2 shows an example of the detection mechanism when $\theta = 2$. Suppose a thread executes a sequence of memory instructions, $I_1 \sim I_8$. Non-memory instuctions are ignored because they do not change the entry content nor affect the mechanism. The PC of each instruction from I_1 to I_8 is PC_1, PC_2, ... PC_8, respectively, and the home core for the memory address that each instruction accesses is specified next to each PC. When I_1 is first executed, the entry {Home, Depth, Start PC} will hold the value of {A, 1, PC_1}. Then, when I_2 is executed, since the home core of I_2 (B) is different from *Home* which maintains the home core of the previous instruction I_1 (A), the entry is reset with the information of I_2. Since the *Depth* to core A has not reached the depth threshold, PC_1 is marked as a remote-access instruction (default). The same thing happens for I_3, setting PC_2 as a remote-access instruction. Now when I_4 is executed, it accesses the same home core C and thus only the *Depth* field needs to be updated (incremented by one). After the Depth field is updated, it needs to be checked to see if it has reached the threshold θ. Since we assumed $\theta = 2$, the depth to the home core C now has reached the threshold and therefore, PC_3 in the *Start PC* field, which represents the first instruction (I_3) that

[2]Since all instructions are initially considered as remote-accesses, marking the instruction as a remote-access instruction will have no effect if it has not been classified as a migratory instruction. If the instruction was migratory, however, it reverts back to the remote-access mode.

accessed this home core C, is now classified as a migratory instruction. For I_5 and I_6 which keep accessing the same home core C, we need not update the entry because the first, migration-triggering instruction has already been detected for this sequence. Executing I_7 resets the entry and starts a new memory sequence for the home core A, and similarly, I_7 is detected as a migratory instruction when I_8 is executed. Once a specific instruction (or PC) is classified as a migratory instruction and is again encountered, a thread will directly migrate instead of sending a remote request and waiting for a reply.

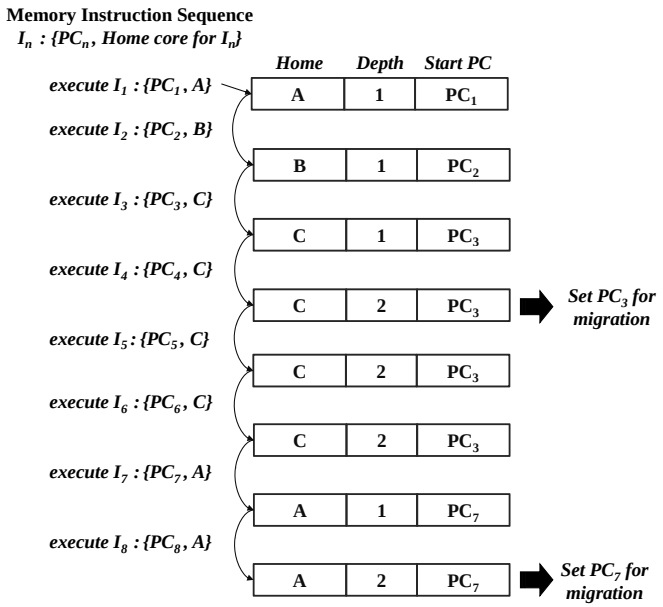


Fig. 2. An example how instructions (or PC's) which are followed by consecutive accesses to the same home location, i.e., *migratory instructions* are detected in the case of the depth threshold $\theta = 2$. Setting $\theta = 2$ means that a thread will perform remote accesses for "one-off" accesses and will migrate for multiple accesses (≥ 2) to the same home core.

Figure 3 shows how this migration predictor actually improves data locality for the example sequence we used in Figure 2. Suppose a thread originated at core A, and thus, it runs on core A. Under a standard, remote-access-only NUCA where the thread will never leave its native core A, the memory sequence will incur five round-trip remote accesses; among eight instructions from I_1 to I_8, only three of them (I_1, I_7 and I_8) are accessing core A which result in *core hits*. With our migration predictor, the first execution of the sequence will be the same as the baseline NUCA, but from the second execution, the thread will now migrate at I_3 and I_7. This generates two migrations, but since I_4, I_5 and I_6 now turn into *core hits* (i.e., local accesses) at core C, it only performs one remote access for I_2. Overall, five out of eight instructions turn into local accesses with effective thread migration.

B. Storing and Lookup of Migratory Instructions

Once a migratory instruction is detected, a mechanism to store the detection is necessary because a thread needs to

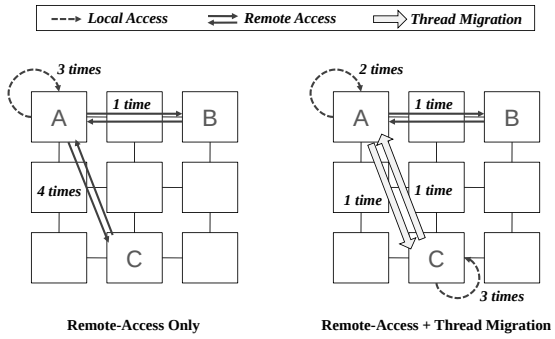


Fig. 3. The number of remote accesses and migrations in the baseline NUCA with and without thread migration.

migrate when it executes this instruction again during the program. We add one-bit called the "migratory bit" for each instruction in *the instruction cache* to store this information. Initially, these bits are all zeros; all memory instructions are handled by remote-accesses when the program first starts execution. When a particular instruction is detected as a migratory instruction, this migration bit is set to 1. The bit is set to 0 if the instruction is marked as a remote-access instruction, allowing migratory instructions to revert back to the remote-access mode. In this manner, the lookup of the migratory information for an instruction also becomes trivial because the migratory bit can be read together with the instruction during the instruction fetch phase with almost no overhead.

When the cache block containing a migratory instruction gets evicted from the instruction cache, we can choose either to store the information in memory, or to simply discard it. In the latter case, it is true that we may lose the migratory bit for the instruction and thus, a thread will choose to perform a remote access for the first execution when the instruction is reloaded in the cache from memory. We believe, however, that this effect is negligible because miss rates for instruction caches are extremely low and furthermore, frequently-used instructions are rarely evicted from the on-chip cache. We assume the migratory information is not lost in our experiments.

Another subtlety is that since the thread context transferred during migration does not contain instruction cache entries, the thread can potentially make different decisions depending on which core it is currently running on, i.e., which instruction cache it is accessing. We rarely observed prediction inaccuracies introduced by this, however. For multithreaded benchmarks, all worker threads execute almost identical instructions (although on different data), and when we actually checked the detected migratory instructions for all threads, they were almost identical; this effectively results in the same migration decisions for any instruction cache. Therefore, a thread can perform migration prediction based on the I-$ at the current core it is running on without the overhead of having to send the migratory information with its context. It is important to note that even if a misprediction occurs due to either cache eviction or thread migration (which is very rare), the memory

access will still be carried out correctly (albeit perhaps with suboptimal performance), and the functional correctness of the program is maintained.

IV. Evaluation

A. Simulation Framework

We use Pin [2] and Graphite [22] to model the proposed NUCA architecture that supports both remote-access and thread migration. Pin enables runtime binary instrumentation of parallel programs, including the SPLASH-2 [32] benchmarks we use here; Graphite implements a tile-based multicore, memory subsystem, and network, modeling performance and ensuring functional correctness. The default settings used for the various system configuration parameters are summarized in Table I.

Parameter	Settings
Cores	64 in-order, 5-stage pipeline, single-issue cores, 2-way fine-grain multithreading
L1/L2 cache per core	32/128 KB, 2/4-way set associative
Electrical network	2D Mesh, XY routing, 3 cycles per hop, 128b flits
	3.2 Kbits execution context size (cf. Section III-A)
	Context load/unload latency: $\left\lceil \frac{pkt\,size}{flit\,size} \right\rceil$ = 26 cycles
	Context pipeline insertion latency = 3 cycles
Data Placement	FIRST-TOUCH, 4 KB page size
Memory	30 GB/s bandwidth, 75 ns latency

TABLE I
SYSTEM CONFIGURATIONS USED

For data placement, we use the *first-touch after initialization* policy which allocates the page to the core that first accesses it after parallel processing has started. This allows private pages to be mapped locally to the core that uses them, and avoids all the pages being mapped to the same core where the main data structure is initialized before the actual parallel region starts.

B. Application benchmarks

Our experiments used a set of *Splash-2* [32] benchmarks: *fft*, *lu_contiguous*, *lu_non_contiguous*, *ocean_contiguous*, *ocean_non_contiguous*, *radix*, *raytrace* and *water-n*^{2}, and two in-house distributed hash table benchmarks: *dht_lp* for linear probing and *dht_sc* for separate chaining. We also used a modified set of *Splash-2* benchmarks [27]: *fft_rep*, *lu_rep*, *ocean_rep*, *radix_rep*, *raytrace_rep* and *water_rep*, where each benchmark was profiled and manually modified so that the frequently-accessed shared data are replicated permanently (for read-only data) or temporarily (for read-write data) among the relevant application threads. These benchmarks with careful replication[3] allow us to explore the benefits of thread migration on NUCA designs with more sophisticated data placement and replication algorithms like R-NUCA [15]. Rather than settling on and implementing one of the many automated schemes in the literature, we use modified Splash-2 benchmarks which implement all beneficial replications, and can serve as a reference placement/replication scheme.

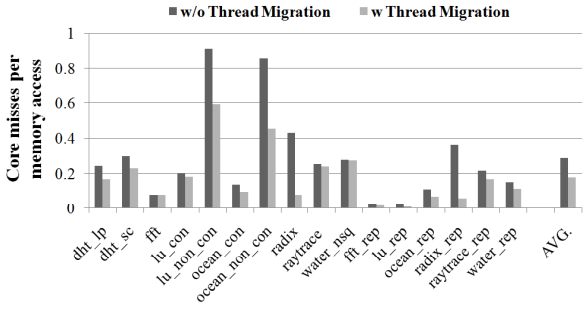


Fig. 4. The fraction of memory accesses requiring accesses to another core (i.e., core misses). The core miss rates decrease when thread migrations are effectively used.

Each application was run to completion using the recommended input set for the number of cores used. For each simulation run, we measured the average latency for memory operations as a metric of the average performance of the multicore system. We also tracked the number of memory accesses being served by either remote accesses or thread migrations.

C. Performance

We first compare the core miss rates for a NUCA system without and with thread migration: the results are shown in Figure 4. The depth threshold θ is set to 3 for our hybrid NUCA, which basically aims to perform remote accesses for memory sequences with one or two accesses and migrations for those with ≥ 3 accesses to the same core. We show how the results change with different values of θ in Section IV-D. While 29% of total memory accesses result in *core misses* for remote-access-only NUCA on average, NUCA with our migration predictor results in a core miss rate of 18%, which is a 38% improvement in data locality. This directly relates to better performance for NUCA with thread migration as shown in Figure 5. For our set of benchmarks, thread migration performance is no worse than the performance of the baseline NUCA and is better by up to 2.3X, resulting in 18% better performance on average (geometric mean) across all benchmarks.

Figure 6 shows the fraction of *core miss* accesses handled by remote accesses and thread migrations in our hybrid NUCA scheme. *Radix* is a good example where a large fraction of remote accesses are successfully replaced with a much smaller number of migrations: it originally showed 43% remote access rate under a remote-access-only NUCA (cf. Figure 4), but

[3]Our modifications were limited to rearranging and replicating the main data structures to take full advantage of data locality for shared data. Our modifications were strictly source-level, and did not alter the algorithm used.

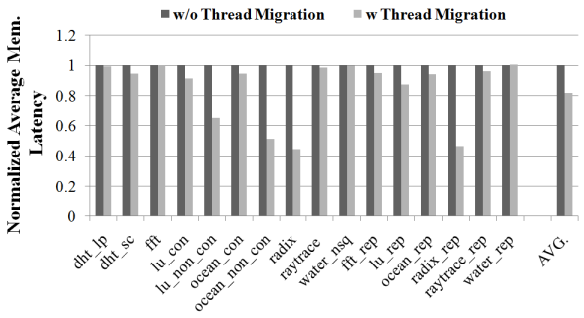


Fig. 5. Average memory latency of our hybrid NUCA (remote-access + thread migration) with $\theta = 3$ normalized to the baseline remote-access-only NUCA.

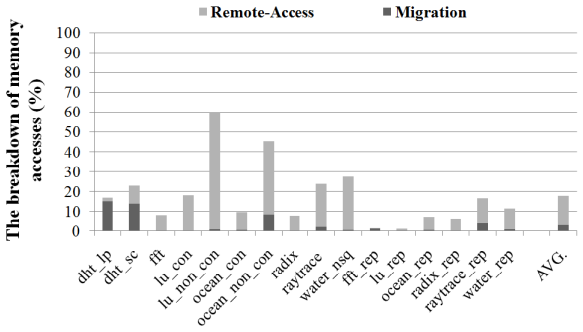


Fig. 6. The breakdown of core miss rates handled by remote accesses and migrations

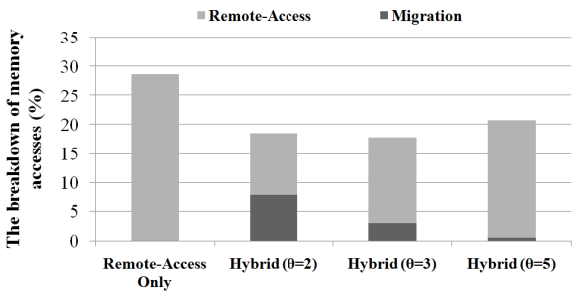


Fig. 7. The fraction of remote-accesses and migrations for the standard NUCA and hybrid NUCAs with the different depth thresholds (2, 3 and 5) averaged across all the benchmarks.

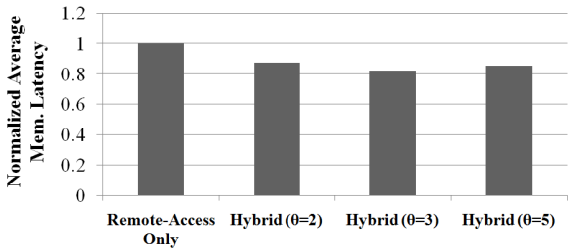


Fig. 8. Average memory latency of hybrid NUCAs with the different depth thresholds (2, 3 and 5) normalized to that of the standard NUCA averaged across all the benchmarks.

it decreases to 7.9% by introducing less than 0.01% of migrations, resulting in 5.4X less core misses in total. Across all benchmarks, the average migration rate is only 3% and these small number of thread migrations results in a 38% improvement in data locality (i.e., core miss rates) and an 18% improvement in overall performance.

D. Effects of the Depth Threshold

We change the value of the depth threshold $\theta = 2$, 3 and 5 and explore how the fraction of core-miss accesses being handled by remote-accesses and migrations changes. As shown in Figure 7, the ratio of remote-accesses to migrations increases with larger θ. The average performance improvement over the remote-access-only NUCA is 13%, 18% and 15% for the case of $\theta = 2$, 3 and 5, respectively (cf. Figure 8). The reason why $\theta = 2$ performs worse than $\theta = 3$ with almost the same core miss rate is because of its higher migration rate; due to the large thread context size, the cost of a single thread migration is much higher than that of a single remote access and needs, on average, a higher depth to achieve better performance.

V. Related Work

To provide faster access of large on-chip caches, the non-uniform memory architecture (NUMA) paradigm has been extended to single-die caches, resulting in a non-uniform cache access (NUCA) architecture [18], [9]. Data replication and migration, critical to the performance of NUCA designs, were originally evaluated in the context of multiprocessor NUMA architectures (e.g., [30]), but the differences in both interconnect delays and memory latencies make the general OS-level approaches studied inappropriate for today's fast on-chip interconnects.

NUCA architectures were applied to CMPs [3], [17] and more recent research has explored data distribution and migration among on-chip NUCA caches with traditional and hybrid cache coherence schemes to improve data locality. An OS-assisted software approach is proposed in [11] to control the data placement on distributed caches by mapping virtual addresses to different cores at page granularity. When adding affinity bits to TLB, pages can be remapped at runtime [15], [11]. The CoG [1] page coloring scheme moves pages to the "center of gravity" to improve data placement. The O^2 scheduler [6], an OS-level scheme for memory allocation and thread scheduling, improves memory performance in distributed-memory multicores by keeping threads and the data they use on the same core. Zhang proposed replicating recently used cache lines [33] which requires a directory to keep track of sharers. Reactive NUCA (R-NUCA) [15] obviates the need for a directory mechanism for the on-chip last-level cache by only replicating read-only data based on the premise that shared read-write data do not benefit from replication. Other schemes add hardware support for page migration support [8], [28]. Although manual optimizations of programs that take advantage of the programmer's application-level knowledge can replicate not only read-only data but also read-write shared data during periods when it is not being written [27], only read-only pages are candidates for replication for a NUCA substrate in general automated data

placement schemes. Instead of how to allocate data to cores, our work focuses on how to access the remote data that is not mapped to the local core, especially when replication is not an option. While prior NUCA designs rely on remote accesses with two-message round trips, we consider choosing between remote accesses and thread migrations based on our migration predictor to more fully exploit data locality.

Migrating computation to the locus of the data is not itself a novel idea. Hector Garcia-Molina in 1984 introduced the idea of moving processing to data in memory bound architectures [14]. In recent years migrating execution context has re-emerged in the context of single-chip multicores. Michaud shows the benefits of using execution migration to improve the overall on-chip cache capacity and utilizes this for migrating selective sequential programs to improve performance [21]. Computation spreading [7] splits thread code into segments and assigns cores responsible for different segments, and execution is migrated to improve code locality. A compile-time program transformation based migration scheme is proposed in [16] that attempts to improve remote data access. Migration is used to move part of the current thread to the processor where the data resides, thus making the thread portion local. In the design-for-power domain, rapid thread migration among cores in different voltage/frequency domains has been proposed to allow less demanding computation phases to execute on slower cores to improve overall power/performance ratios [24]. In the area of reliability, migrating threads among cores has allowed salvaging of cores which cannot execute some instructions because of manufacturing faults [23]. Thread migration has also been used to provide memory coherence among per-core caches [19], [20] using a deadlock-free fine-grained thread migration protocol [10]. We adopt the thread migration protocol of [10] for our hybrid memory access framework that supports both remote accesses and thread migrations. Although the hybrid architecture is introduced in [19], [20], they do not answer the question of how to effectively decide/predict which mechanism to follow for each memory access considering the tradeoffs between the two. This paper proposes a novel, PC-based migration predictor that makes these decisions at runtime, and improves overall performance.

VI. Conclusions and future work

In this manuscript, we have presented an on-line, PC-based thread migration predictor for memory access in distributed shared caches. Our results show that migrating threads for sequences of multiple accesses to the same core can improve data locality in NUCA architectures, and with our predictor, it can result in better overall performance compared to the traditional NUCA designs which only rely on remote-accesses.

Our future research directions include improving the migration predictor to better capture the dynamically changing behavior during program execution and to consider other factors than access sequence depths, such as distances or energy consumption. Furthermore, we will also explore how to reduce single-thread migration costs (i.e., the thread context size being transferred) by expanding the functionality of the migration predictor to predict and send only the useful part of the context in each migration.

References

[1] M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter. Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches. In *HPCA*, 2009.

[2] Moshe (Maury) Bach, Mark Charney, Robert Cohn, Elena Demikhovsky, Tevi Devor, Kim Hazelwood, Aamer Jaleel, Chi-Keung Luk, Gail Lyons, Harish Patil, and Ady Tal. Analyzing parallel programs with pin. *Computer*, 43:34–41, 2010.

[3] M. M. Beckmann and D. A. Wood. Managing wire delay in large chip-multiprocessor caches. In *MICRO*, 2004.

[4] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 - processor: A 64-Core SoC with mesh interconnect. In *Proceedings of the IEEE International Solid-State Circuits Conference*, pages 88–598, 2008.

[5] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of DAC*, pages 746–749, 2007.

[6] Silas Boyd-Wickizer, Robert Morris, and M. Frans Kaashoek. Reinventing scheduling for multicore systems. In *HotOS*, 2009.

[7] Koushik Chakraborty, Philip M. Wells, and Gurindar S. Sohi. Computation spreading: employing hardware migration to specialize CMP cores on-the-fly. In *ASPLOS*, 2006.

[8] M. Chaudhuri. PageNUCA: Selected policies for page-grain locality management in large shared chip-multiprocessor caches. In *HPCA*, 2009.

[9] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, 2003.

[10] Myong Hyon Cho, Keun Sup Shim, Mieszko Lis, Omer Khan, and Srinivas Devadas. Deadlock-free fine-grained thread migration. In *Proceedings of NOCS 2011*, pages 33–40, 2011.

[11] Sangyeun Cho and Lei Jin. Managing Distributed, Shared L2 Caches through OS-Level Page Allocation. In *MICRO*, 2006.

[12] David Wentzlaff et al. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, 27(5):15–31, Sept/Oct 2007.

[13] International Technology Roadmap for Semiconductors. Assembly and Packaging, 2007.

[14] H. Garcia-Molina, R.J. Lipton, and J. Valdes. A Massive Memory Machine. *IEEE Trans. Comput.*, C-33:391–399, 1984.

[15] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive NUCA: near-optimal block placement and replication in distributed caches. In *ISCA*, 2009.

[16] Wilson C. Hsieh, Paul Wang, and William E. Weihl. Computation migration: enhancing locality for distributed-memory parallel systems. In *PPOPP*, 1993.

[17] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. A NUCA substrate for flexible CMP cache sharing. In *ICS*, 2005.

[18] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches. In *ASPLOS*, 2002.

[19] Mieszko Lis, Keun Sup Shim, Myong Hyon Cho, Christopher W. Fletcher, Michel Kinsy, Ilia Lebedev, Omer Khan, and Srinivas Devadas. Brief announcement: Distributed shared memory based on computation migration. In *Proceedings of SPAA 2011*, pages 253–256, 2011.

[20] Mieszko Lis, Keun Sup Shim, Myong Hyon Cho, Omer Khan, and Srinivas Devadas. Directoryless shared memory coherence using execution migration. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing*, 2011.

[21] P. Michaud. Exploiting the cache capacity of a single-chip multi-core processor with execution migration. In *HPCA*, 2004.

[22] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In *Proceedings of HPCA 2010*, pages 1–12, 2010.

[23] Michael D. Powell, Arijit Biswas, Shantanu Gupta, and Shubhendu S. Mukherjee. Architectural core salvaging in a multi-core processor for hard-error tolerance. In *Proceedings of ISCA 2009*, pages 93–104, 2009.

[24] Krishna K. Rangan, Gu-Yeon Wei, and David Brooks. Thread motion: Fine-grained power management for multi-core systems. In *Proceedings of ISCA 2009*, pages 302–313, 2009.

[25] S. Rusu, S. Tam, H. Muljono, D. Ayers, J. Chang, R. Varada, M. Ratta, and S. Vora. A 45nm 8-core enterprise Xeon® processor. In *Proceedings of the IEEE Asian Solid-State Circuits Conference*, pages 9–12, 2009.

[26] K. Sankaralingam, R. Nagarajan, H. Liu, J. Huh, C. K. Kim, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP using polymorphism in the TRIPS architecture. In *International Symposium on Computer Architecture (ISCA)*, pages 422–433, June 2003.

[27] Keun Sup Shim, Mieszko Lis, Myong Hyo Cho, Omer Khan, and Srinivas Devadas. System-level Optimizations for Memory Access in the Execution Migration Machine (EM^2). In *CAOS*, 2011.

[28] Kshitij Sudan, Niladrish Chatterjee, David Nellans, Manu Awasthi, Rajeev Balasubramonian, and Al Davis. Micro-pages: increasing DRAM efficiency with locality-aware data placement. *SIGARCH Comput. Archit. News*, 38:219–230, 2010.

[29] S. R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *IEEE J. Solid-State Circuits*, 43:29–41, 2008.

[30] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. *SIGPLAN Not.*, 31:279–289, 1996.

[31] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to Software: Raw Machines. In *IEEE Computer*, pages 86–93, September 1997.

[32] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995.

[33] M. Zhang and K. Asanović. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *ISCA*, 2005.

Programming and Scheduling Model for Supporting Heterogeneous Accelerators in Linux

Tobias Beisel and Tobias Wiersema and Christian Plessl and André Brinkmann
Paderborn Center for Parallel Computing
University of Paderborn
Paderborn, Germany
Email: {tbeisel\|tobias82\|christian.plessl\|brinkman}@uni-paderborn.de

***Abstract*—Computer systems increasingly integrate heterogeneous computing elements like graphic processing units and specialized co-processors. The systematic programming and exploitation of such heterogeneous systems is still a subject of research. While many efforts address the programming of accelerators, scheduling heterogeneous systems, i. e., mapping parts of an application to accelerators at runtime, is still performed from within the applications. Moving the scheduling decisions into an external component would not only simplify application development, but also allow the operating system to make scheduling decisions using a global view.**

In this paper we present a generic scheduling model that can be used for systems using heterogeneous accelerators. To accomplish this generic scheduling, we introduce a scheduling component that provides queues for available accelerators, offers the possibility to take application specific meta information into account and allows for using different scheduling policies to map tasks to the queues of both accelerators and CPUs. Our additional programming model allows the user to integrate checkpoints into applications, which permits the preemption and especially also subsequent migration of applications between accelerators. We have implemented this model as an extension to the current Linux scheduler and show that cooperative multitasking with time-sharing enabled by our approach is beneficial for heterogeneous systems.

I. Introduction

Heterogeneous accelerator environments have become ubiquitous with the advent of multi-core CPUs and general-purpose graphics processing units (GPUs). This heterogeneity is also observable in compute centers, where cluster systems use combinations of multi-core processors, GPUs, and specialized co-processors, such as ClearSpeed CSX or FPGAs, to accelerate scientific applications [1].

The usage of such systems is still limited though, since most accelerators need to be programmed with unfamiliar programming languages and APIs. Developing efficient software for these architectures requires knowledge about the underlying hardware and software components. Hence many recent research efforts address the challenges to ease the development and use of hardware accelerated code. While this research area is of high relevance, we do not address the task of creating programs or configurations for hardware accelerators in this paper.

This work has been partially supported by the German Ministry for Education and Research (BMBF) under project grant 01|H11004 (ENHANCE).

Instead, we approach the challenge of performing scheduling decisions at runtime and treating hardware accelerators as peer computation units that are managed by the operating system (OS) kernel like CPU cores. The goal of scheduling tasks in the context of heterogeneous systems is to assign tasks to compute units in order to enable time-sharing of accelerators and to provide fairness among tasks that compete for the same resources. Scheduling tasks to heterogeneous accelerators raises a number of practical challenges, the most important being that some hardware accelerators such as GPUs do not support preemption and that the migration of tasks between different accelerators is complicated due to largely different execution models and machine state representations. Also, the scheduling process itself is more complex than process scheduling for homogeneous CPU cores, since each scheduling decision requires to incorporate specific hardware characteristics (e. g., the communication bandwidth or memory sizes) and needs to consider the current availability and state of the heterogeneous compute units. In addition, knowledge about the availability and suitability of a task for a particular hardware accelerator is required. This information is highly application specific and has to be provided by the application developer. Scheduling tasks to hardware accelerators has been neglected by OS developers so far and is managed as part of the application. This implies that hardware accelerators are used exclusively by one particular application without any time-sharing.

The contribution of this work is a general programming and scheduling model for heterogeneous systems and an example implementation in Linux. We provide an extension to the Linux Completely Fair Scheduler (CFS) that 1) provides awareness of installed accelerators, 2) enables scheduling of specific tasks to accelerators, and 3) allows time-sharing and task migration using a cooperative multitasking and checkpointing approach. Our approach is non-preemptive, but allows tasks to release a compute unit upon a request by the scheduler and thus increases the fairness among tasks. Dynamic task migration on heterogeneous systems is a major contribution of this approach.

The scheduler hardware selection decision is based on meta information provided by the applications. While we supply a basic scheduling policy based on static affinities to accelerators, our focus is to provide a framework for heterogeneous scheduling using a time-sharing approach. We evaluate our work with two example applications that prove the usability and benefits of the approach and supply data for an efficiency analysis.

This work is an extension and more comprehensive discussion of a previous work of ours, in which we already presented a prototype implementation of a linux kernel extension supporting heterogeneous systems [2]. In this paper we provide a more general view onto the problem and describe the kernel extension in more detail.

The remainder of this paper is structured as follows. We introduce a general scheduling model in Section II. Afterwards we describe our newly developed Linux kernel extension in Section III and present an according programming model in Section IV. Section V evaluates the contributions with two example applications. After a discussion of related work in Section VI, we finish the paper with discussing future work and drawing conclusions.

II. General Scheduling Model for Heterogeneous Systems

The CFS schedules processes in current Linux SMP systems. CFS is a preemptive scheduler that guarantees fairness with respect to the allocation of CPU time among all processes. The scheduler aims to maximize the overall utilization of CPU cores while also maximizing interactivity. An inherent precondition for the use of such a preemptive scheduler is that processes can be preempted and also migrated between computing resources.

In this work, we address scheduling in a heterogeneous computer system with non-uniform computing resources. We target computer systems, which include single- or multi-core CPUs operating in SMP mode and an arbitrary combination of additional hardware accelerators, such as GPUs, DSPs, or FPGAs. Scheduling such systems is more difficult due to several reasons:

1) Accelerators typically do not have autonomous access to the shared memory space of the CPU cores and explicit communication of input data and results is required. The most important impact on a scheduling decision is the introduction of data transfer times that rely on available bandwidths and the data to be copied. These overheads have to be known and used as input for a scheduling decision in heterogeneous systems. Further, the communication bandwidth, latency, and performance characteristics of accelerators are non-uniform. These characteristics also determine the granularity of the task that can be successfully scheduled without too much overhead (single operations, kernels, functions/library calls, threads). Scheduling decisions thus usually have to be more coarse-grained than on CPUs.
2) Most accelerator architectures do not support preemption but assume a run-to-completion execution model. While computations on CPU cores can be easily preempted and resumed by reading and restoring well defined internal registers, most hardware accelerators do not even expose the complete internal state nor are they designed to be interrupted.
3) Heterogeneous computing resources have completely different architectures and ISAs. Hence, a dedicated binary is required for each combination of task and accelerator, which prevents migrating tasks between arbitrary compute units. Even if a task with the same functionality is available for several architectures and if the internal state of the architecture is accessible, migrating a task between different architectures is far from trivial, because the representation and interpretation of state is completely different.

A. Design Decisions

In this section, we discuss and describe basic design decisions made for our scheduling framework.

1) Scheduler Component: Scheduling of homogeneous CPU cores is currently done in the kernel, as all needed input information for the scheduling decision is available to the system, so that the scheduling problem can be completely hidden from the application programmer. The heterogeneous scheduling problem is more complicated, as more decision parameters affect the decision, which are partly not available to the systems scheduler component.

Selecting an appropriate hardware architecture for a task to be scheduled dynamically at runtime is non trivial and has to be performed by a scheduler, which can be located at different locations in the system, either in the application, in user space or in the system's kernel.

To allow a holistic view on the applications and its execution environment, we perform scheduling in the system's kernel by extending the CF scheduler. That way the scheduling principles are still hidden from the application developer and the OS can perform global decisions based on the system utilization. Application specific scheduling inputs still have to be provided by the application developer to incorporate application's needs. Therefore we use a hybrid user/kernel level approach to perform heterogeneous scheduling. A specific interface has to be provided to allow communication between application and scheduler.

2) Adapting the Operating System: Kernel space scheduling is the current standard in operating systems. To provide support for heterogeneous architectures one could either extend an existing OS or completely rewrite and fully optimize it towards the heterogeneity. While heterogeneous systems will be more and more used in future and become standard in a foreseeable time, we believe that a complete rewrite of

the OS is not needed. An extension to the current system has several advantages: Providing a modular implemented extension to the CFS 1) keeps the management structures as well as the scheduler component exchangeable, 2) makes the changes easily applicable to other OS, and 3) reuses well established and well known functionalities of the current kernel that have been developed over years. That way our kernel extension will help to explore new directions for future OS, but does not yet try to set a new standard.

3) Delegate Threads: Tasks that execute on heterogeneous resources may have no access to main memory and use a completely different instruction set or execution model than an equivalent task on a CPU. In order to schedule and manage these tasks without requiring a major OS rewrite, we need to expose the tasks to the OS as known schedulable entities. We therefore represent each task executing on a hardware accelerator as a thread to the OS. This allows us to use and extend the existing data structures of the scheduler in the Linux kernel. We denote each thread representing a task on a hardware accelerators as a *delegate thread*. Apart from serving as a schedulable entity, the delegate thread also performs all operating system interaction and management operations on behalf of the task executing on the accelerator unit, such as transferring data to and from the compute unit and controlling its configuration and execution. The delegate threads must be spawned explicitly by the application and thus can also be used for co-scheduling on different architectures. Once created, all threads are treated and scheduled equally by the operating system.

4) Cooperative Multitasking: The CFS implements preemptive multitasking with time-sharing based on a fairness measure. Therefore, our scheduler has to include means to preempt a task and to migrate it to another computing unit. While non-voluntary preemption on FPGAs is possible, GPUs currently do not directly support it yet, even if it is planned for the future [3]. Therefore we use the delegate threads to forward requests from the kernel scheduler to the task on the accelerator.

Nevertheless, even enabling preemption on GPUs does not solve the migration problem. The major difficulty is to find a way of mapping the current state of a compute unit to an equivalent state on a different compute unit. To allow preemption and subsequent migration of applications on heterogeneous systems, their delegate threads need to be in a state, which can be interpreted by other accelerators or by the CPU. As it is not possible to interrupt an accelerator at an arbitrary point of time and to assume that it is in such a state, we propose to use a cooperative multitasking approach using checkpoints to resolve these limitations. After reaching a checkpoint, an application voluntarily hands back the control to the OS, which then may perform scheduling decisions to suspend and migrate a thread at these points. We believe that this currently is the only way to simulate preemptive multitasking on heterogeneous hardware.

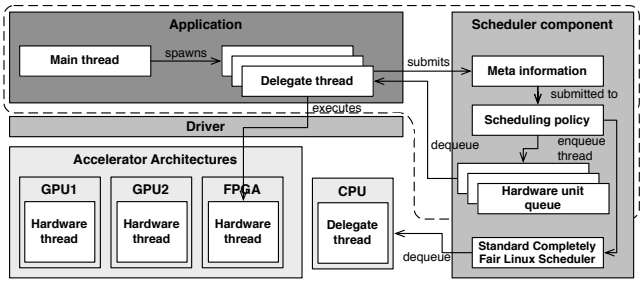


Figure 1. Novel scheduling model for heterogenous systems. New parts are surrounded by dashed lines.

In this paper, we focus on the underlying framework and assume that the application developer defines such states in his application and that he provides a delegate thread, which interacts with the scheduler and the accelerator. The structure of the delegate thread (cf. Section IV) as well as providing relocatable system states is generic enough that these can be automatically generated by a compiler in the future.

B. Scheduling Model

From the design decisions above we derive our scheduling model shown in Figure 1 that is not restricted to a certain class of operating systems or scheduling algorithms. Applications using the scheduler may spawn several threads that may possibly run on diverse architectures.

Thread information: As the scheduler needs information about the threads to be scheduled, we store this application provided information called *meta information* about each thread and submit it to the scheduler. The meta information can be individually set for an application. Currently we only use a type affinity towards a target architecture, which can be determined dynamically depending on the input data. Further application specific input data can possibly be determined using profiling prior to the first use of an application. While this is not in the focus of this paper, one could think of useful values like estimated runtimes, required memory sizes or data transfer sizes.

Scheduling: The scheduler component may be located in the kernel space as well as the user space. To assign tasks to certain hardware components, the scheduler has to provide a queue for each available hardware. The application provided meta information is used in a scheduling policy to map newly arriving tasks to one of the queues. Whenever a compute unit runs idle or the currently running task has used its complete time slice, the scheduler may dequeue a waiting task for that specific compute unit. In case this is a hardware task, the delegate thread receives the information that it may run its hardware counterpart. This includes using the proprietary drivers of the hardware, which are inevitable for the communication with some accelerators. As these currently may only be used from user space, this requires

a combined approach using the kernel space and the user space. For CPUs, the standard Linux scheduler is used.

Checkpointing: Checkpointing has to be performed when the application can safely interrupt its execution and store its state in main memory. The state has to be stored by the application itself in data structures of the corresponding delegate thread, which then can be migrated to a different architecture. The checkpoint data of the delegate thread thus has to be readable by all target architectures.

We define a checkpoint as a struct of data structures that unambiguously defines the state of the application. The scheduler does not have any requirements concerning the checkpoint data. Hence, the application has to make sure that all needed data is available in these data structures and thus stored in accessible memory at the end of each thread's time-slice. A checkpoint in most cases is a combination of 1) a set of data structures that define a minimum state that is reached several times during execution, and 2) a data structure that define the position in the code. The checkpoint data of an application is copied to the newly allocated accelerator and copied back to the host's main memory when the application's time slice is exhausted.

Checkpoints are to be defined by the application developer or to be inserted by a compiler. One has to identify a preferably small set of data structures that 1) unambiguously define the state of a thread, and 2) are readable and translatable to corresponding data structures of other compute units. The size of checkpoints may vary to a large extend depending on the application used. While MD5 cracking (cf. Section V) only needs to store the current loop index (i.e., a hash value) and the given search-string, image processing algorithms (e.g., medical image processing) require to store the complete intermediate results that might be of large extent. In general, a checkpoint could be simply defined by a list of already processed data sets. Therefore, the choice of the checkpoint is very important and influences the scheduling *granularity*. The checkpoint distance, i.e., the amount of work done between 2 checkpoints stored back, increases with the size of the checkpoint.

We here assume all checkpoints to be small enough to fit into the host's memory. The introduced checkpoint size is known at definition time and may be used to re-determine the scheduling granularity for a task. Please refer to Sections IV and V for examples and implementation details about the meta information and checkpoints, or directly to the example implentations (cf. Section VIII).

III. Linux Kernel Extensions

This section shortly introduces the changes made to the Linux kernel to enable the scheduling of heterogeneous hardware accelerators according to our scheduling model. Please refer directly to the source code for implementation details (cf. Section VIII).

A. Data Structures

Following the goal to extend the current Linux scheduler, we have to make the kernel aware of existing heterogeneous hardware accelerators. The CFS uses its queues and statistics to ensure a fair treatment of all tasks with respect to their priorities. Its queue is ordered by the amount of unfairness, i.e., the time the task would have to execute undisturbed to be treated fair. We extend the kernel with a specific task struct for hardware threads and a semaphore protected queue for each of the available accelerators.

The current implementation of the meta information includes the memory size to be copied and an array of type affinities. The higher a task's affinity to a compute unit is, the better is the estimated performance on this compute unit.

B. Scheduler API

With respect to the cooperative use of the scheduler, we provide an interface to the scheduler, which enables user space applications to request (allocate), re-request and free compute units. The allocation call requests and acquires that compute unit, which matches the calling task best by enqueueing the task to the associated waiting queue. The assignment is done using an affinity-based approach, where the given affinity, as well as the current length of the waiting queues and the load of the compute units are included.

Our CFS extension allows the migration of threads from one compute unit to another if the application provides implementations for both. Migration of a thread may be performed while it is blocked within a waiting queue or even if it is running on any of the available compute units. Since there are no means of directly migrating the tasks from one instruction set to another, migration is achieved by a combination of checkpointing and cooperative multitasking.

If the program reaches a checkpoint, it requests (re-requests) to further use the compute unit, but offers to voluntarily release it (also compare Figure 2). The scheduler decides if the task on the compute unit should be replaced by another, which depends on the type of compute unit and on the cost of switching the task. Re-requests inside the time window of an accelerator-specific granularity are always successful and will only be denied after the granularity has expired and if other tasks are waiting for the resource. The time a task may run on an accelerator follows the CFS approach. It is the sum of the fairness delta, i.e., the time to compute until the (negative) unfairness is equalized, and the granularity, i.e., the "positive unfairness" for this task.

To enable dynamic load balancing on CPU cores and GPUs, a load balancing component managing running and queued tasks was introduced. If the application has either finished its work or unsuccessfully re-requested its compute unit, it calls a *free* function. This releases the compute units semaphore and hands it to the next task or, in case no other tasks are waiting on this device, invokes the load balancer.

If a task is waiting for the semaphore of a compute unit and another suitable unit is running idle in the meantime then the load balancer wakes the task with a migration flag. The task then removes itself from the waiting queue and enqueues on the idle compute unit. Using this mechanism the scheduler achieves a late binding of tasks to units, which ensures a better utilization of the resources with only a negligible amount of computation overhead in the scheduler, as most tasks are blocked and thus migrated while waiting on the semaphore of a compute unit. The load balancer at first traverses the run-queues of all other compute units and tries to find the task with the maximum affinity to the idle compute unit. If the balancer is not able to find a suitable waiting task, it parses through all running tasks, which are currently being executed on other units.

C. Control API

Using most of todays hardware accelerators involves using their proprietary user space APIs to copy code or data to and from the device and to invoke programs on it. Since there are virtually no implementations to communicate efficiently with these devices from the kernel, our extension leaves all interaction with the accelerators to the user space.

We provide system calls to add a compute unit, to remove it afterwards, to iterate through all currently added units and to alter a device after it has been added.

IV. Programming Model

This section describes the design of applications using the provided system calls of our kernel extension. Additionally, we describe a design pattern for implementing an application worker thread (delegate thread), which is not mandatory for using the extended functionality of the CFS, but simplifies application development.

A. User Application Workflow

Figure 2 describes the typical lifecycle of a thread in our extended CFS. Besides the information about the system status the scheduler needs to have meta information about the thread to be scheduled. Additionally, the code to be executed and the needed input data has to be copied to the compute unit after it has been acquired by using the blocking allocation call. The worker then can execute its main function in a loop. If a re-request fails before the worker is done, it writes a checkpoint and waits for the allocation of a new compute unit for taking up its computation.

B. Worker Implementation

We provide code templates in C++ to simplify application development. We introduce a *worker* class that is the superclass of the delegate threads in an application. The worker class provides the virtual functions *getImplementationFor* and *workerMetaInfo*, which have to be implemented in the derived delegate threads.

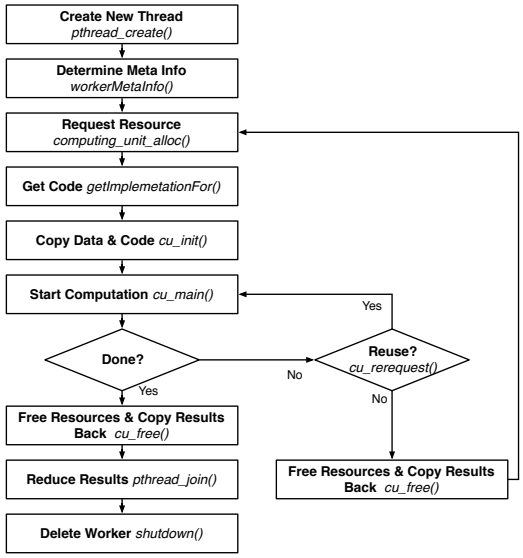


Figure 2. Typical workflow of a delegate thread.

```
void Worker_example::workerMetaInfo(struct meta_info *mi){
 mi->memory_to_copy=0;              // in MB
 mi->type_affinity[CU_TYPE_CUDA]=2;
 mi->type_affinity[CU_TYPE_CPU]=1;
}

void* Worker_example::getImplementationFor(int type,
     functions *af)
switch(type) {
  case CU_TYPE_CPU:
   af->init=&Worker_example::cpu_init;
   af->main=&Worker_example::cpu_main;
   af->free=&Worker_example::cpu_free;
   af->initialized=true;
   break;
  case CU_TYPE_CUDA:
   ... //similar
  default:
   af->initialized=false; }
```

Listing 1. Example implementation for mandatory worker functions.

The *workerMetaInfo* method implemented by a worker instance includes the mandatory values for the meta information, which ensures that only compatible compute units are assigned to the thread. The example *type_affinity* in Listing 1 defines the GPU to be twice as suitable for the worker as the CPU. Setting an affinity to zero tells the scheduler that no implementation for the specific compute unit exists. The application developer does not have to know the exact performance difference between implementations. The affinity only gives an approximate hint of how much the implementation for one compute unit outperforms the other.

The *getImplementationFor* function fills the functions array *af* with pointers to the implementation for the allocated compute unit type *type*. The worker implementation has to provide the three functions *cu_init*, *cu_main*, and *cu_free* for all supported compute units. While the CPU does not require anything to be copied, all other accelerators usually

```
typedef struct md5_resources {
 std::string hash_to_search;
 unsigned long long currentWordNumber;
 bool +;
} md5_resources_t;
```

Listing 2. Example checkpoint for MD5 cracking.

need the data to be copied explicitly. The *cu_init* function allocates memory on the compute unit and copies needed resources (including the checkpoint) to it. These resources can be used in the computation performed in the *cu_main* function, which does all the work between two checkpoints. The resources have to be copied back to main memory after finishing a computation or being denied a re-request and the memory on the compute unit can be freed, which has to be implemented in the *cu_free* function.

In addition to the code framework, the worker class provides system call wrappers to the Scheduler API.

The provided programming model is generic and can be easily used by a scheduler different than the CFS, e.g., by a new scheduler component to be used from user space. Applications that do not use the programming model are simply executed by the usual CFS scheduling mechanisms.

V. Evaluation

In this section we present applications using the extended scheduler, which dynamically switch between CPU cores and a GPU. We evaluate the overheads and show how adjusting the scheduler parameters affects the runtimes.

A. Example Applications

We used a brute-force MD5 hash cracker (MD5) and a prime number factorization (PF) as example applications to evaluate our scheduler extension. Both applications were implemented in C++ for the CPU and CUDA C to be run on NVIDIA GPUs. In both cases we examine the execution of several application instances concurrently.

1) MD5 Cracking: MD5 brute-force cracking enumerates all possible strings of a fixed length with letters of a given alphabet and computes their MD5 hash value until a match with a target hash value is found. Each string is identified by an ascending unique number, which can be used as a checkpoint specifying the progress of the search. Listing 2 shows how checkpoints usually are defined. The actual checkpoint within this struct is the *currentWordNumber*, which saves the current status of the computation. The other information is needed to restore the complete state of the application, i.e., by storing the needed inputs (*hash_to_search*) and the general status (*foundsolution*).

We chose the interval in terms of searched strings between two checkpoints different for the CPU (e. g., 500 strings) and the GPU (e. g., 1 billion) in order to consider the costs for re-transferring the kernel to the compute unit and doing a re-request at a checkpoint that are much higher on the GPU.

The meta information for this application is very simple, as practically no memory has to be copied. The affinity can be set depending on the size of the string length and the used alphabet, which defines the problem size. The performance differences between CPU and GPU are known to be very high in the test case, hence the meta data is set up to express a clear affinity to the GPU.

The vertical axis in Fig. 3 represents the search space that has been searched, while the horizontal axis denotes the runtime in seconds. We run 15 GPU affine threads, with a limited queue length of 5 for the GPU. As we spawn 15 threads at the same time, 9 threads initially run on CPU cores, 6 (1 running, 5 queued) on the GPU. Each thread is displayed by a different color. The GPU threads can be identified by the fast progress in the search space. The ascend of the curves representing CPU threads can hardly be seen, only a minor progress can be noticed in region e). The ability of the tasks to be stopped and replaced for later continuation on the GPU can, e.g., be seen in region a), where 6 GPU threads share the computation time based on their fairness, the granularity, and the time needed to reach the next checkpoint. In this example the load balancer refills the GPUs queue as soon as a thread finishes its computation (regions b) and d)). Regions c) and e) show how CPU enqueued threads are migrated to the GPU and afterwards compute much faster. Fig. 3 shows that cooperative multitasking is possible using the introduced checkpoints. Although the CPU does not lead to great speedups one can see that the heterogeneous system is fully utilized and computing resources are shared among the available tasks.

2) Prime Factorization: Prime factorization algorithms decompose numbers into their prime factors. Our sample application searches through all possible divisors of the number up to its square root. Whenever a divisor is encountered the number is divided by it as often as possible. Hence the application yields the divisor and its multiplicity. It then continues the search, now using the current remainder and its square root instead of the original number. Checkpoints here are defined by the pair of current remainder and potential

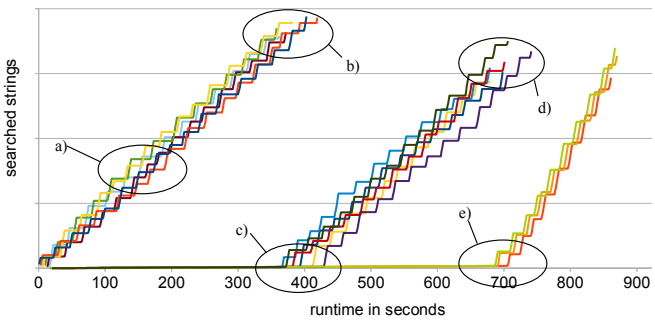


Figure 3. Running 15 independent MD5 instances.

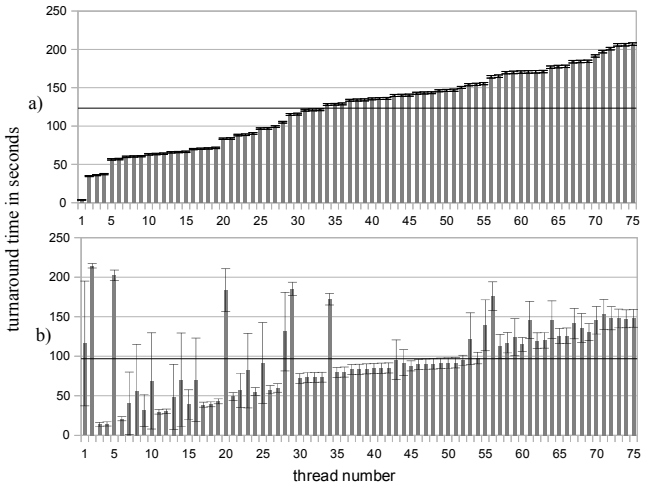


Figure 4. Turnaround times for 75 concurrently started threads without time-sharing (a)) and with time-sharing and 4s granularity (b)) using 25 MD5 and 50 PF threads.

divisor. Between two checkpoints the algorithm advances the divisor to the value of the second checkpoint, while keeping the remainder correct during the process. In our version the checkpoints are 1000 divisors apart.

B. Scheduler Evaluation

The following setup was used for the evaluation:

- SMP system with 2-way 4-Core Intel Xeon CPU E5620 @ 2.4GHz, hyperthreading enabled, 12 GB DDR3
- NVIDIA Geforce GTX 480, 480 thread-processors, 1536 MB GDDR5 memory
- Ubuntu Linux 10.04.1 LTS, 2.6.32-24 kernel
- CUDA 3.1, GCC 4.4.3

1) Performance Evaluation: The turnaround time is the interval between the submission of a process and its completion [4]. The effect of time-sharing on turnaround times can be seen in Fig. 4. Subfigure a) shows the turnaround times of 25 MD5 and 50 PF instances with one thread spawned by each instance in batch mode. As tasks are started at the same time and scheduled one after another, long running tasks (e.g., tasks 1, 2, 5, 20) block all other tasks, such that the average turnaround time is increased. Using time-sharing, short tasks are not blocked, so that the average turnaround time is lower. Subfigure b) shows that the tasks are not finished in the same order as they are started. Longer jobs do not block the shorter ones, as their time slice is of the equal length as that of a short job. This increases interactivity, as response times are decreased. After 150 seconds only long running threads remain in the system.

In addition to the reduced average turnaround times, the overall performance of several running applications may be increased, if using more than one compute unit. This is shown in Fig. 5, which depicts the total runtime of a varying number of PF applications on the GPU alone and on both

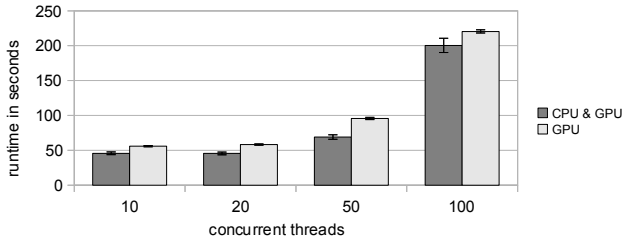


Figure 5. Average runtimes of different counts of PF instances on a GPU and on a combination of GPU and CPU cores.

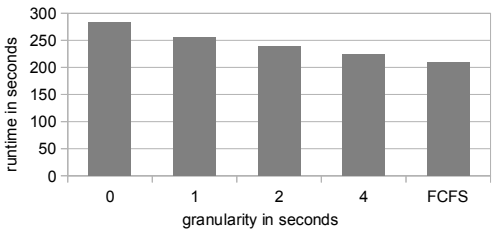


Figure 6. Mean of total runtime for 30 runs with 25 MD5 threads (string length 6) and 50 PF threads on the GPU.

the GPU and the available CPU cores. As can be seen, the average runtime of all instances can be reduced by using the scheduler extension. All threads compete for the use of the GPU, but profit from the fallback computation on the CPU.

2) Overheads: Fig. 6 shows the influence of the granularity of time slices on the runtime of the example applications. All tasks in this test were run on the GPU. Decreasing the granularity raises the total runtime, as task switching overheads are introduced. Introducing time-sharing is therefore a trade off between overheads and interactivity, as a higher granularity decreases the response times of the threads and FCFS scheduling obviously has the smallest overhead.

This is also emphasized in Fig. 7, which shows the average turnaround time depending on the used granularity. Using a granularity of 0 leads to fast-paced task switching and a very high interactivity and thus introduces huge overheads. On the other hand, submitting all tasks at the same time to the GPU queue and using FCFS for the tasks on the GPU results in higher average turnaround times due to the fact that long running tasks are blocking short tasks.

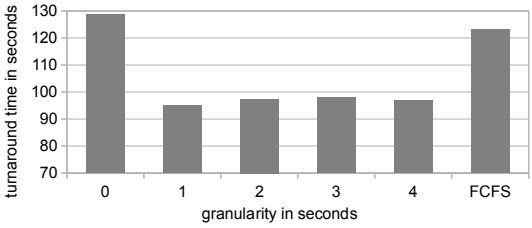


Figure 7. Avg. turnaround times per thread with 25 MD5 threads (string length 6) and 50 PF threads on the GPU.

VI. Related Work

Heterogeneous systems are widely examined in research. Many groups work on the programmability and utilization of these systems (e.g., [5], [6], [7]). OpenCL [8] is a recently well discussed approach that aims on using heterogeneous architectures. Our approach is more general. We do not rely on a complete library, but we provide a simple programming model, allow easy integration of further architectures into our scheduling framework, and allow multi-user scenarios and dynamic runtime scheduling. OpenCL does not support task scheduling. This work addresses OS integration of accelerators by using delegate threads and the problem of reading back their state to the system.

Delegate threads have also, e.g., been used by Bergmann et al. [9], which discuss an approach using "ghost processes" to make reconfigurable System-on-chip (rSoC) hardware processes appear like software processes in a Linux environment. Software ghost processes are associated to hardware processes, used to control these and allow communication with the associated hardware threads. Ghost processes load modules to a FPGA and set up the communication channels. The authors used processes combined with inter process communication (IPC) instead of threads to encapsulate hardware processes. This makes the handling of hardware tasks less lightweight and more complicated for OS integration. Scheduling hardware accelerators has not been discussed.

Lübbers et al. [10] extended the Linux and eCos operating systems by a uniform interface for software threads on the CPU and hardware threads on FPGA accelerators. They extended the multi-threaded programming model to heterogeneous computing resources. Every hardware thread is associated with exactly one software thread, which allows communication between FPGA threads and OS data structures. Cooperative multitasking has been discussed to be possible by storing state information on FPGAs.

Other groups also present work on enabling the OS to read the hardware state. Early work has shown that migrating the state of an application between heterogeneous CPU cores is possible. [11] presents a technique that allows objects and threads to be migrated between machines using heterogeneous nodes at native code level. They introduce so called "bus stops" as machine-independent formats to represent program points. We extend this idea to use time-sharing for current hardware architectures like FPGAs and GPUs. In contrast to GPUs, preemption has been shown to be possible on FPGAs (e.g., [12]). as well as non-preemptive multitasking (e.g., [13]). Nevertheless, none of these approaches has extended the OS scheduler to become responsible for hardware scheduling.

So et al. [14] also modify and extend a standard Linux kernel with a hardware interface. They use a message passing network to provide conventional IPC mechanisms to FPGAs. Communication between hardware and software processes was implemented by FIFOs and mapped to file system-based OS objects. FPGA processes are bound via the Linux */proc* directory and behave similar to software processes. FPGA resources are provided as virtual file system.

Integrating time-sharing using the Linux thread model on heterogeneous systems is a novel approach that increases the interactivity of the system and optimizes the components utilization and the performance of universal applications. None of the previous approaches presents such a global view on the system allowing a holistic scheduling decision by using meta information of the applications as well as incorporating the systems status. This is possible by providing an extension of the operating systems scheduler that can easily be extended to also support other hardware resources.

VII. Conclusions And Future Work

We presented an general model to perform scheduling of tasks on heterogeneous components. We introduced the use of cooperative multitasking to heterogeneous systems and provided an implementation that allows the preemption and a subsequent migration of threads between GPUs and CPU cores. The migration is done automatically based on an affinity metric associated with the compute units and the current system status. This not only reduces the average load on the CPU while at least preserving the application's performance, but also allows intelligent task scheduling to maximize application performance and system utilization.

Considering the fact that preemption is not possible on GPUs and reading back the state of accelerators is generally challenging, we introduced a programming model that uses checkpoints to define an unambiguous state of a running application, allowing its suspension and later continuation based on a time-sharing paradigm. This approach is in line with the goal of the current Linux scheduler to provide a fair treatment of available tasks and to increase interactivity.

Our programming model does not yet completely decouple the application development from the use of heterogeneous systems, but relieves the programmer from managing the scheduling of independent threads within the application. Nevertheless, we assume that this decoupling can be achieved by compiler extensions. The Linux kernel handles hardware threads as if they were software threads, which is a continuation of the traditional CPU scheduling approach. We have shown that the automatic migration of threads is possible and that task switching overheads are acceptable.

In future work, we will compare this work to a similar user space scheduling library approach, simplify the usage of our programming model by automatic detection and extraction of checkpoints in applications, and provide example applications incorporating FPGAs.

VIII. Source Code

To promote the uptake of our work by other researchers and users, we made our extended kernel available to the general public as open-source at http://github.com/pc2/hetsched.

REFERENCES

[1] K. Volodymyr, R. Wilhelmson, R. Brunner, T. Martïnez, and W. Hwu, "High-Performance Computing with Accelerators (Special Issue)," *IEEE Computing in Science & Engineering*, vol. 12, no. 4, Jul./Aug. 2010.

[2] T. Beisel, T. Wiersema, C. Plessl, and A. Brinkmann, "Cooperative multitasking for heterogeneous accelerators in the linux completely fair scheduler," in *Proc. IEEE Int. Conf. on Application-Specific Systems, Architectures, and Processors (ASAP)*. IEEE Computer Society, September 2011.

[3] T. Tamasi, "Evolution of Computer Graphics," in *Proc. NVISION 08*, 2008.

[4] W. Stallings, *Operating Systems: Internals and Design Principles*, 6th ed. Pearson Prentice Hall, 2009.

[5] N. Moore, A. Conti, M. Leeser, and L. King, "Vforce: an extensible framework for reconfigurable supercomputing," *IEEE Computer*, vol. 40, no. 3, pp. 39–49, 2007.

[6] F. Blagojevic, C. Iancu, K. Yelick, M. Curtis-Maury, D. Nikolopoulos, and B. Rose, "Scheduling dynamic parallelism on accelerators," in *Proc. of the 6th ACM conference on Computing frontiers*. ACM, 2009, pp. 161–170.

[7] R. Chamberlain, M. Franklin, E. Tyson, J. Buckley, J. Buhler, G. Galloway, S. Gayen, M. Hall, E. Shands, and N. Singla, "Auto-Pipe: Streaming Applications on Architecturally Diverse Systems," *IEEE Computer*, vol. 43, no. 3, pp. 42–49, 2010.

[8] A. Munshi, *The OpenCL Specification Version 1.2*, November 2011.

[9] N. Bergmann, J. Williams, J. Han, and Y. Chen, "A process model for hardware modules in reconfigurable system-on-chip," in *Proc. Dynamically Reconfigurable Systems Workshop (DRS) at Int. Conf. on Architecture of Computing Systems (ARCS)*, 2006, pp. 205–214.

[10] E. Lübbers and M. Platzner, "ReconOS: Multithreaded programming for reconfigurable computers," *ACM Trans. on Embedded Computing Systems (TECS)*, vol. 9, no. 1, pp. 1–33, 2009.

[11] B. Steensgaard and E. Jul, "Object and native code thread mobility among heterogeneous computers (includes sources)," in *Proc. of the 15th ACM Symp. on Operating systems principles*. ACM, 1995, pp. 68–77.

[12] L. Levinson, R. Männer, M. Sessler, and H. Simmler, "Preemptive multitasking on FPGAs," in *Proc. Symp. Field Programmable Custom Computing Machines (FCCM)*. IEEE CS, 2000.

[13] H. Walder and M. Platzner, "Non-preemptive multitasking on FPGAs: Task placement and footprint transform," in *Proc. Int. Conf. on Engineering of Reconfigurable Systems and Architectures (ERSA)*. CSREA, 2002, pp. 24–30.

[14] H. So and R. Brodersen, "Improving usability of FPGA-based reconfigurable computers through operating system support," in *Int. Conf. on Field Programmable Logic and Applications (FPL)*, 2006, pp. 1–6.