A Survey on Hardware and Software Support for Thread Level Parallelism

SOMNATH MAZUMDAR and ROBERTO GIORGI, Università di Siena, Siena, Italy

To support growing massive parallelism, functional components and also the capabilities of current processors are changing and continue to do so. Today's computers are built upon multiple processing cores and run applications consisting of a large number of threads, making runtime thread management a complex process. Further, each core can support multiple, concurrent thread execution. Hence, hardware and software support for threads is more and more needed to improve peak-performance capacity, overall system throughput, and has therefore been the subject of much research. This paper surveys, many of the proposed or currently available solutions for executing, distributing and managing threads both in hardware and software. The nature of current applications is diverse. To increase the system performance, all programming models may not be suitable to harness the built-in massive parallelism of multicore processors. Due to the heterogeneity in hardware, hybrid programming model (which combines the features of shared and distributed model) currently has become very promising. In this paper, first, we have given an overview of threads, threading mechanisms and its management issues during execution. Next, we discuss about different parallel programming models considering to their explicit thread support. We also review the programming models with respect to their support to shared-memory, distributed-memory and heterogeneity. Hardware support at execution time is very crucial to the performance of the system, thus different types of hardware support for threads also exist or have been proposed, primarily based on widely used programming models. We also further discuss on software support for threads, mainly increase the deterministic behavior during runtime. Finally, we conclude the paper by discussing some common issues related to the thread management.

Categories and Subject Descriptors: A.1 [General Literature]: Introductory and Survey; D.1.3 [Concurrent Programming]: Parallel programming; D.4.1 [Process Management]: Threads.

General Terms: Documentation, Management, Performance.

Additional Key Words and Phrases: Thread level parallelism, hardware support, software support.

ACM Reference Format:
Somnath Mazumdar and Roberto Giorgi. 2016. A Survey on Hardware and Software Support for Thread Level Parallelism ACM Comput. Surv. X, X, Article X (March 2016), 35 pages. DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Thread is a portion of a program aimed at providing an abstraction for defining parallelism. (Other definitions can be found in the literature [Pac11; CDK05]). Parallelism exploited at thread level is called thread level parallelism or TLP [BDMF10; HK09]. In application, TLP can be exploited using execution model such as fork-join, master/slave, dataflow, divide and conquer). Commonly, though, current architectures
Table I. Execution time (in secs) of N-body Problem

| Language | N=5,000,000 | N=50,000,000 |
|----------|-------------|---------------|
|          | Single core | Quad core     | Single core | Quad core |
| C        | 1.05        | 1.05          | 10.49       | 10.30     |
| C++      | 0.94        | 0.94          | 9.39        | 9.40      |
| Java     | 2.44        | 2.44          | 22.67       | 22.68     |
| PHP      | 67.83       | 66.46         | 657.77      | 698.86    |
| Perl     | 110.25      | 107.5         | 1088.37     | 1080.05   |

are multicore processors (like Chip multiprocessors (CMPs) and heterogeneous also [KTR+04] (such as Asymmetric CMPs [SPS+07]). They can run a variety of complex applications consisting massive number of concurrent threads. Today, even a single core can support simultaneous execution of multiple threads (known as simultaneous multithreading (SMT) [TEL95]. Few well-known examples of SMT supported architectures are Intel’s core architecture that supports two threads each core and Xeon Phi that can support four threads per core. Similarly, IBM's Power8 and Oracle's SPARC processor each can support up to eight threads per core.

Recent applications are long running, multithreaded and has different resource requirement. Only the micro-architectural advancement is not enough to execute these applications in the best possible way. To support heterogeneity, programming models like CUDA, OpenCL and also hybrid programming models (such as CUDA together with MPI and OpenMP) are commonly used nowadays. Along with proper programming model, both hardware and software support for threads are also necessary. But, there exists a trade-off between dedicated hardware and software thread support. Hardware based support for explicit multithreading can facilitate smooth execution of threads [URŠ03]. When threads are tiny or small, managing these fine-grain threads at hardware level offers advantages over managing them at the software level. Furthermore, compared to hardware based solutions, software based approaches can lead to a higher performance loss as the number of threads increases. The situation may worsen in large, complex and long running parallel applications. Generally, dedicated hardware support offers low overhead and faster execution time, but requires altering existing hardware or adding new hardware module. On the other hand, software components are easier to add and manage. Thus, added hardware support can increase performance, but at the expense of area and/or power consumption.

We can quantify application speedup by measuring the parallel segment to the serial segment of the application. When the speedup of an application exceeds the number of cores in a chip is called super linear speedup. One of the main criteria to achieve super linear speedup is to divide data sets so small, so that it can fit into the local cache of the cores. To calculate the relative speedup, two most widely used methods are Amdahl’s Law and Gustafson’s Law. Amdahl’s law is to compute theoretical upper bound of the speedup, but Gustafson’s law computes the speedup of the scaled parallel execution to the serial execution of the application. Programmer tries to extract as many independent threads as possible from application for better speedup, i.e., TLP. Thread scheduler assigns independent threads to processing cores to achieve better throughput. But, during the thread mapping onto the cores, physical location of input data is also very significant, because failure to take into account the physical location of shared data can create unnecessary communication traffic at the chip level [BWFM09]. Table I shows the CPU execution time for the classical N-body problem (a physical system of multiple, interacting objects) written in five different programming languages. With a large number of particles (e.g., N=50,000,000), moving the process from a single to a quad-core platform, performance improvement is very small [Ini15]. This simple example of performance comparison supports our statement that even in
a single threaded execution, increasing hardware capability is not enough, software support should also complement hardware to harness the inherent parallelism that these multicore processors has to offer.

2. THREAD BASICS

An application is a collection of threads and a basic thread consists of address space, access rights to cores, inter process communication (IPC), and I/O resources. Generally, threads creation can be done in a variety of ways, depending on the programming model in use. Figure 1(left) is a representation of thread creation in Pthreads, wherein a new thread is called in the calling process. Similarly, Figure 1(right) shows how threads are generated in OpenMP, when a portion of code is identified for parallel thread execution. After the newly created threads are started to execute, the state of the threads also started to evolve. Figure 2(left) represents the canonical state transition diagram that any threads can have during its execution life cycle. As a first step when thread has all its required input and processing core available, it transits from the ready state to running state. But at running state, if a thread is waiting for its input or resources (I/O stalls, memory stalls), then it changes it state from running to blocked state. Later, upon available of data helps it to resume its execution, which can further lead to its completion. In the following sub-section, we illustrate the typical thread characteristics that are exhibited very often by threads.
2.1. Thread Characteristics

A thread block wraps a sequence of instructions together with runtime related information. But, a thread can be classified according to the features (see Figure 3) that it inherits or exhibits either in compile time or in execution time, mostly influenced by the programming model and execution environment. They are such as execution flow, size of the work (granularity), level of parallelism, supported platforms, how safe they are during execution, and their special status (privilege level).

2.1.1. Execution Flow. A program is organized into several threads and its execution flow governed by either control dependency or data dependency [Mor10]. A control dependency means a thread instance is triggered when a condition is met (von Neumann based programming models), whilst a data dependency means a thread instance is triggered by the production of data coming from another thread (dataflow based programming model) (see Figure 2(right)).

2.1.2. Thread Granularity. Threads can have different granularity. Generally, granularity means the code length or the amount of work of a thread that has to be processed. Further, the granularity can be classified either by coarse-grain or fine-grain (also discussed in sub-subsection 2.2.1). In coarse-grain multithreading, switching to other threads only happens when there is a long I/O stall(s) (e.g., a cache miss) [IGHJS95]. In fine-grain multithreading, core switches to another thread on each instruction (multiple thread interleaving). Fine-grain multithreading can efficiently hide short and long stalls. In general, whether fine-grain threads or coarse-grain threads are performing better is typically dependent on the underlying hardware and the available TLP support.

2.1.3. Parallelism. Threads allow us to parallelize processes, or to execute a process on more than one processing unit, thereby reducing execution time. During TLP, if all the threads in an application were executed independently (ideal for the highest
level of parallelism), the total execution time would be equal to the execution time of the longest-executing thread. Applications can also exhibit other degree of parallelism such as instruction level parallelism (ILP) or data level parallelism (DLP). Figure 4(left) represents the parallelism that can be achieved at data, thread and loop level. Furthermore, TLP can also be divided into three classes according to their inter or intra-thread dependency exhibited during execution [Ott08]:

— Independent multithreading (IMT): As the name suggests, no thread dependency exists and all threads are executed independently and concurrently. Example: DoAll loops.
— Cyclic multithreading (CMT): In CMT, one or more inter-thread dependency exist in cyclic order. Example: DoAcross loops (i.e., loops with cross-iteration dependencies).
— Pipelined multithreading (PMT): As in CMT, in PMT application code is divided into multiple threads to fill the execution pipeline and is executed with recurrences. Inter-thread dependencies are managed by a directed acyclic graph (DAG). PMT can be considered as a superset of CMT with added features (e.g. toleration for long latencies). An example of PMT is DOPipe [Hai79].

2.1.4. Execution Platforms. Different forms of threads can run on different computing platforms like CPU, GPUs, and FPGAs. (Please refer to Section 4 for detailed discussion).

2.1.5. Thread Safety. In concurrent programming, thread safety is all about maintaining the integrity of shared data sets. Runtime thread management can exploit the concurrency capability of underlying cores, and can also address the issue of data-race prevention (or thread-safety) [But97]. In conventional programming languages (such as Java), where data as well as control flow are an integral part of process execution, relaxing consistency of shared data during concurrent access by multiple threads, makes thread unsafe. A classical solution to thread safety is to use locks for shared data and serialization for the entire function or methods.

2.1.6. Privilege Level. User applications typically start in user space and continue its processing by spawning child threads (fork-join model). In such case, we talk about
user level threads. User level threads have fewer thread management issues, such as creation, deletion, switching and are controlled by the runtime library [ABLL92]. When a thread's parent process is running in kernel or the parent is operating system (OS), then this thread is most privileged and is known as kernel level thread (e.g. Windows-NT Threads [ZY98]). Figure 4(right) show pictorial presentation of user and kernel level threads. Generally, a system call is required for context-switching among kernel level threads. The main difference between user level and kernel level threads are the costs of thread synchronization [VBCZ+03], and context switching [MSLM91]. Therefore, user level threads are often used for CPU-bound computations, while kernel level threads support I/O bound computations more efficiently.

2.2. Runtime Thread Management

Thread scheduler is a part of the OS, and is responsible for allocating threads to cores. Thread scheduling policy impacts the overall execution speed. Usually, a “quantum”-a number of clock ticks of a CPU is allocated to each thread. When a thread has exhausted its quantum, a context-switch is initiated, whereby the scheduler assigns a new quantum to the thread, also taking into account the thread's priority level. To increase the overall execution speed, determining the optimal number of concurrent threads that a core should support is also a challenge. In multithreaded architectures, a small number (typically ranging from 3 to 5) of concurrent threads can boost the performance [EJK+97].

Thread scheduling is a well-researched area. For homogeneous multithreaded processors, work stealing (WS) has become one of the most efficient algorithms till date [BL99]. To support heterogeneous multithreaded processors new scheduling policies have also been proposed. Such as Koufaty et al. proposed Bias Scheduling, which tries to increase the performance by assigning small or large cores to threads dynamically [KRH10]. Thread assignments are calculated by exploiting the affinity of a thread for a particular core type (small or big). The speedup associated with a thread on large core versus a small core is recorded (called biasness) and taken into account during thread mappings. Similarly, CAMP (Comprehensive Asymmetric Multi-Processor) scheduling algorithm, uses a utility factor to assign a thread to a faster core and increase the TLP [SPFB10]. The utility factor is a metric that approximates the application's speedup, when some threads are placed on fast cores and others on slow cores. Another interesting work, shared-thread multiprocessor (STMP) [BT08] has proposed an additional hardware unit (shared-thread control unit) to control the movement, activation, and deactivation of threads in SMT chips. It also supports thread's state sharing among cores inside the chip.

2.2.1. Multithreading

Multithreaded processors are very common nowadays. Basically, multithreading technique can be of three types, namely interleaved multithreading (IMT), block multithreading (BMT) and simultaneous multithreading (SMT).

— Interleaved multithreading (IMT) technique (also called fine-grain multithreading) is a simple thread execution methodology. Only after the successful execution of a previous instruction, current instruction from a new thread can get executed. In IMT, control and data dependency are not detected during the execution. This technique works best for multiple threads (ideally the number of concurrent threads should be equal to the number of pipeline stages) and shows poorer performance on single thread based applications. Generally, IMT provides the lowest overheads for context-switching.

— Blocked multithreading (BMT) technique (also called coarse-grain multithreading) is ideal for supporting fewer threads. Generally, in BMT, context-switch takes place or execution of a thread is blocked, when there is memory or I/O stalls. BMT can act
A Survey on Hardware and Software Support for Thread Level Parallelism

as a single core processor, if there are no stalls for a single threaded application. In BMT, context-switch overhead is higher than IMT.

Simultaneous Multithreading (SMT) aims to optimize pipeline utilization by interleaving multiple threads [TEL95]. In SMT, each core handles multiple threads' contexts and runs not-stalled threads from its thread pool. It allows multiple independent threads to issue multiple instructions at each cycle. When a cache miss occurs, multithreaded cores can hide long-latency memory accesses by replacing a waiting thread with another runnable thread [BR92] and can outperform fine-grain multithreading by a factor of two [TEL95]. SMT has been well-adopted in commercial processors e.g. Intel's Hyperthreading (HT) technology. Lang et al. have found that HT does not necessarily improve the performance of all running applications. Authors have shown that compute intensive and cache-friendly applications may suffer with the HT method, while I/O-bound parallel applications may gain performance [LAH +02].

2.2.2. Speculative Threading. Speculative threading policy can increase system performance considerably [SM98]. Speculative multithreading (SpMT) dynamically exploits parallelism in an application [HHS +00; KT99] and has mainly been used to increase the fine-grain parallelism in irregular applications. Two main steps for exploiting SpMT in side the application are, identification and partitioning of speculative thread candidates. The identification process can be done at compile time and also at runtime. At compile time, thread level speculation is done by checking the control dependency, whilst at runtime data dependency is verified. The process of selecting speculative threads is guided by performance factors (such as inter-thread data dependency, branch mis-prediction, load balancing). After successful selection, a sequential program is divided into multiple chunks called speculative threads. Compiler techniques can also be used to exploit parallel independent thread executions with high probability [CHH +03; BF02; ZC05]. Generally, software [TFNG08; LT +06] and hardware [BGM +00; HW098] support is needed for both processing speculative threads and also to recover from the dependency violations. Speculative data driven multithreading (DDMT) supports speculative thread execution in an out-of-order processor by forking new data-driven thread (DDT) in parallel to the main thread when main thread predicts an upcoming instance of a critical section [RS01].

2.2.3. Thread Migration. Generally, during execution, to distribute the workload among the available cores, threads are migrated to (and from) different cores (thread migration), by satisfying some conditions (like TLP improvement, system call support, or acceptable thermal temperature [ZHV13]) or costs [CSM +05]. During migration, thread scheduler can assign “lighter” threads to slower cores and thereby improve overall power/performance ratio [RWB09]. Frequent thread migration, however, can result in degraded performance. For example, during migration, if an associated memory location is not moved with the thread, access latency overheads can occur. Furthermore, unnecessary thread migration can also induce extra traffic in the interconnects.

2.2.4. Thread Synchronization. Critical section optimization is crucial for parallel thread performance. Lock based synchronization (mutexes, semaphores) is the classical way to synchronize concurrent threads, but many other ways that are more fit to thread execution have also been explored. Popular proposed solutions for critical sections can be grouped either hardware based [AFA11; SMQP09] or software based [LDX +12; SS06]. For example, Cai et al. proposed a so called meeting point thread characterization mechanism to detect threads that contains a critical section of a single multithreaded application [CGR +08]. Here, “thread-slack” is used to know the criticality level of the thread. The thread-slack is the time a thread can be delayed without affecting the
application’s performance. Similarly, speculative synchronization (SS) is proposed to resolve critical section and barrier related issues [MT02]. Transactional Memory (TM) [HM93] is another solution to the critical section problem, but TM as well SS fails when threads have data conflicts. In another interesting work, Godson-T [FZW+12] provides full/empty bit support for fine-grain control of shared data and also a lock-based synchronization manager for managing access conflicts.

2.3. Resource Sharing Among Threads

Different applications have different resource requirements. A typical hardware execution unit consists of various dependent components (like cores, local memory, DRAM controller, I/O controllers), but software plays a crucial part to tie them together for executing application. Figure 5 represents the block diagram of a superscalar and a chip multiprocessor with their typical components. With increasing core count contention reduction among shared resources is a challenge [CPV05; ZA05]. Cache performance can be improved by increasing the cache hits for memory intensive workloads by smart page replacement policy like least recently used (LRU) which is the most commonly used page replacement policy for caches [KCS04]. In practice, contention-aware scheduling [ZBF10; KBH+08] takes care of memory page and their migration.

In heterogeneous platform, where host CPU is connected to an accelerator, last level cache (LLC), bandwidth of interconnect as well as memory are becoming very critical components. Important shared resources that are crucial for better thread execution are LLC, DRAM memory controller and the interconnects. To increase overall chip throughput and fairness, thread-level schedulers such as symbiotic OS (SOS)-level job scheduler for SMT chips [ST00] try to mitigate shared-resource contention mainly in last level cache (LLC), and DRAM controller of multicore processors. Below we discuss the techniques that are proposed for LLC, DRAM controller and interconnects, to reduce the runtime access contention among threads.

2.3.1. Last Level Cache (LLC).

Cache contention is very critical to the performance [MTH+11] and it has been studied both in case of SMTs [PELL00; ST00] and also in CMTs [CGKS05; FSSN05]. Experiments showed that cache contention is influenced by the nature of threads that share LLCs [KCS04]. Few well-known algorithms for sharing LLCs among threads are work stealing [SBGH09] and parallel depth first (PDF) [BG04]. Generally, threads that affect each other performance are partitioned. Cache partitioning and sharing among concurrent threads, is also very critical to the effective performance of the multicore processors [LLD+08]. Cache partitioning policies are in place to optimize fairness [BMW+06; CPV05], performance [HKS+07; BMW+06], and
Quality of Service (QoS) [DS07; CS06]. Cache partitioning mechanisms include techniques both in software [ZDS09; CJ06] and hardware [QP06].

2.3.2. DRAM Controller. Generally, cache misses are managed by DRAM controller, but the performance of DRAM controller can severely be degraded due to concurrent access of threads. Treating all access requests from cores equally by DRAM controller is a popular policy and known as first-ready first-come-first-serve (FR-FCFS) [RDK00]. The uniformity policy of FR-FCFS could be the source of low throughput and may cause thread starvation in the system [MM07]. Hence, there exist few other throughput aware algorithms for DRAM controller to boost the system performance. To name a few are parallelism-aware batch scheduling algorithm (PAR-BS) [MM08], adaptive per-thread least-attained-service (ATLAS) [KHMHB10]. In an interesting work, machine learning technology has been employed to dynamically adapt the DRAM controller by interacting with the system and also optimizes its performance [IMMC08].

2.3.3. Interconnect. Generally, cores are connected inside a chip via topology. For smaller core count topology such as bus, ring [KK09] are very popular. Whereas for larger core count 2D mesh [CA95], its variants and hierarchical topology (such as mesh-ring [FYNM11], (global and local) bus based topology [UMB10]) are also used. Hence, interconnect or network-on-chip (NoC) [BDM02; KJS02; DT01] for larger core count is another critical component responsible for better performance on multicore systems. During thread execution, all the thread communication, data transfer is done via interconnects. NoC offers better scalability, productivity and more deterministic performance [DT01]. For better throughput, the available interconnect bandwidth should be enough to support generated traffic. Generally, external memory traffic, cache hierarchy and cache coherence policy are very much responsible for generating traffic patterns in shared-memory systems. Whereas, distributed systems are heavily relied on network interface cards, such as Peripheral Component Interconnect Express (PCIe) or other protocols such as AMBA/AXI [AMB03]. The thread mapping problem in NoC [HM03] is how to topologically place cores in order to optimize some certain constraints such as energy consumption, communication cost. NoC's architectural components such as channel width, buffer size, core mapping, and routing algorithms are very critical for better data traffic support [BM06; OHM05]. There have some works [DMMD09; GKM09] been done related to QoS of NoC. NoC based research works are mainly focused on reduction of either communication cost [SBG08; PBB03] or overall energy consumption [HM05; SK04] by placing threads on the cores.

3. THREADING IN PROGRAMMING MODELS
Parallelism is important to exploit efficiently the available computational resources. However, there is no consensus on a single style programming model. Several parallel programming models with different memory models are currently in use. We have grouped them into five broad categories and presented in Figure 6. They are shared-memory based, distributed memory based, partitioned global address space (PGAS) based, dataflow based, and heterogeneity based.

3.1. Shared Memory Programming Models
3.1.1. POSIX threads. One of the first-developed and popular approaches for parallel programming threads is POSIX (Portable Operating System Interface) threads or Pthreads [NBF96]. It is basically an API which has also been standardized. Pthreads was implemented via C language types and function calls [But97]. The advantages of using Pthreads for parallel programming are: light-weight thread creation [dSM99], low maintenance overheads and better communication speed [KC99]. In addition, Pthreads also provide TLP, but their supports rely on the shared-memory model and
Fig. 6. Categories of Parallel Programming Models

| Parallel Programming Model | Shared Memory | Distributed Memory | Database | Heterogeneity |
|----------------------------|---------------|--------------------|----------|---------------|
| Multithreaded (MT)         | Pthreads      | OpenMP             | DBCC     | CilkPlus      |
| Threaded (T)               |               |                    |          |               |
| Data Race (DR)             |               |                    |          |               |
| Language                   |               |                    |          |               |
| Run-time                   |               |                    |          |               |
| Database                   |               |                    |          |               |
| Execution Model            |               |                    |          |               |

they follow a fork-join programming execution model [MSM04]. Programmers manage workload distribution, task mapping, data races and other low-level compile time management issues. To manage critical section issues, Pthreads support mutual exclusion (mutex) and semaphores [Gra03].

GNU Portable Threads (GNU Pth) [Eng05] is somewhat less known portable user-space multithreading POSIX/ANSI-C based library. It supports co-operative, non-preemptive, priority-based scheduling for event-driven multithreaded applications. It works mainly as a co-routine based framework, but also supports backward compatibility to existing multithreaded applications written by Pthreads. Similar to Pth, nPth (New GNU Portable Threads Library) is another non-preemptive thread library [BK15], also developed as a replacement of Pth and also supporting the execution of Pth-incompatible libraries.

3.1.2. CilkPlus. CilkPlus is the commercial version of the C++ extension of Cilk [BJK+95]. Cilk was originally proposed as an extension of C. The aim of Cilk is to help programmers to build applications optimized for a maximum level of parallelism on shared-memory multiprocessors (SMPs). It supports both thread as well as data level parallelism. Its runtime provides support for scheduling, dynamic load balancing and communication. Cilk runtime uses a work stealing policy for scheduling work among the processors. It also defines a lock-free memory consistency model (called hyper object). During execution, it builds a DAG, which enables a thread based dataflow execution. It employs a fork-join execution model to provide deterministic multithreading.

3.1.3. Threading Building Blocks. Intel’s Threading Building Blocks (TBB) [Rei07] aim to enhance scalable parallel programming on multicore processors using C++ templates. It is a library to represent high level, thread based parallelism and also subsequently hides unnecessary platform details (such as thread pool management, distribution of load, cache affinity) from the programmer. TBB also follows a fork-join execution model. It also seamlessly supports other threading directives like OpenMP or MPI. Instead of dividing code among cores, it divides data sets among cores, thereby increasing the performance by exploiting larger core counts. TBB solves the user’s problem of finding the parallelism inside a code. It takes the code and its runtime automatically schedules them onto cores. But, in experiments, it has been seen that TBB runtime library had significant synchronization overheads, up to half of total execution time [CM08].

Intel’s Array Building Blocks (ArBB) provides a vector parallel programming solution for data intensive applications [SPS+07]. ArBB is based on standard C++ library and uses TBB’s runtime. ArBB lets a programmer to express computations as operations on arrays and vectors, which further can run on heterogeneous platforms using its library and runtime support. ArBB can also avoid race condition and deadlock issues.

ACM Computing Surveys, Vol. X, No. X, Article X, Publication date: March 2016.
3.1.4. **OpenMP.** Open Multi-Processing (OpenMP) [CJVDP08] is an API that eases the programming effort in a shared-memory environment. It supports fork-join execution model via its PARALLEL and END PARALLEL directives. OpenMP offers compiler directives, pragmas and a runtime to make thread management easy. OpenMP pragma directives can extract concurrent portion of code from a sequential code. Its work sharing runtime helps each thread to execute different tasks (e.g. for-loop parallelization). OpenMP provides support for intra-node communication via control structures, data environment (for efficient execution), (implicit and explicit) synchronization for data dependency and a runtime library with environmental variables.

3.2. **Distributed Memory Programming Model**

Message Passing (MP) is a way to distribute computation across separate machines (called “nodes”). Message passing based parallel programming model relies on the exchange of messages as a means of communication among nodes. Well-known implementations of message passing models are MPICH [GLDS96], Open MPI [GFB+04].

3.2.1. **MPICH.** MPICH (Message Passing Interface over CHameleon) is one of the early implementation of MPI aiming to combine mobility with high performance for parallel systems. MPICH has inherited its features from three main libraries such as Chameleon [GS93], P4 [BL94] and Zipcode [SSD+94]. MPICH process has separate address space and can communicate with another by sending and receiving messages. MPICH supports different hardware platforms as a model which provides the high portability. Similarly, MVAPICH [Pan14] is another implementation of MPICH (version 1) over InfiniBand and other Remote Direct Memory Access (RDMA) interconnects. It supports a wide range of platforms including Intel Xeon-Phi, GPUs.

3.2.2. **Open MPI.** Open MPI is an open source implementation of MPI. It supports a master/slave execution model and favors single program multiple data (SPMD) technique. MPI provides an API to code in distributed as well as shared-memory environments. Open MPI supports concurrent, multithreaded applications and provides support for internode communication. Each executing process has its own address space and supports thread safety. It also provides different levels of thread support. But, programmers are responsible for distributing tasks among processes during compile time.

3.3. **Partitioned Global Address Space**

Partitioned Global Address Space (PGAS) combines the features of shared-memory and message passing for supporting parallel programming models such as SPMD. It aims to provide support of shared-memory in distributed architecture (mainly NUMA platforms). The PGAS memory model is primarily based on distributed shared-memory (DSM) approach by solving its “locality issue” [CDMC+05]. PGAS implements global address space in distributed architecture where threads share the address space both as private space and shared space. PGAS threads can identify the difference between local and remote memory access. In PGAS, globally addressable memory spaces are divided into places. A place corresponds to a local node and has the minimal data access cost to its host. Places can be accessed from remote, with higher costs. PGAS supported languages either have a cheap data access cost or expensive data access cost. Data objects are distributed all over the address space and support both explicit and implicit data distribution. The data distribution model is responsible to distribute, access the data among the concurrent threads. PGAS supported compiler and runtime are responsible for supporting threads data access on a distributed system. APIs and libraries are also used to provide PGAS programming model support. Languages such as X10 [CGS+05], Fortress [ACH+05], Unified Parallel C (extension of the C language)
[EGS06], Chapel [CCZ07] are well-known PGAS programming languages. PGAS also has very efficient runtime systems such as Global-Address Space Networking (GASNet) [BJ02]. GASNet is used to exchange the active massages among the threads. It encapsulates the messages in such a way, so that it does not only carry the massages, but also the instructions to process it.

3.4. Dataflow Programming Model

Dataflow oriented programming models have been proposed to support dataflow based execution model [RRB69]. Dataflow execution model is different from traditional von Neumann model. Earlier von Neumann processors suffered from issues related to memory latency and synchronization [I88]. But, dataflow provides lower synchronization cost and also reduces processor idle time. Dataflow model can be classified as static [DM75] as well as dynamic [GKW85]. Dataflow architecture had initially been introduced for ILP [Den80] and also was initially implemented in a form of “restricted dataflow” in a superscalar processor [HP86]. Later, a similar concept had been investigated by [AN90] also to support thread. That latter effort brought to an explicit token store (ETS) architecture based machine known as Monsoon [PC90].

A process generated by dataflow language creates a DAG. In the produced DAG, each node represents a thread, and the arc is the data path to other threads. Computation is triggered only when required data arrive at the nodes via the arcs. All nodes that have data available can immediately start execution. Some of the well-known dataflow languages are Lucid [AW77], Lustre [HCRP91], hardware description languages (such as VHDL [Nav97]), actor model based (such as CAL [EJ03]). Readers can refer to the survey [YAMJGE14] of hybrid dataflow/von Neumann model for more details. There also exists dataflow based program execution model (PXM) such as Codelets [SZG13], Data triggered threads (DTT) [TT14; TT12; TT11], DF-Threads [GF14], Data driven multithreading (DDM) [KET06] and Scheduled Dataflow (SDF) [KGA01].

— Codelet is a fine-grain dataflow model for supporting multiple nodes. A codelet represents a non-preemptive, single unit of computation. For dataflow execution, multiple codelets are connected together to form a codelet graph (CDG).
— Data-triggered threads (DTT) architecture tries to exploit redundant computation to speedup execution. A DTT can be written as a function and can accept modification of addresses as an input. For runtime support, it adds four new instructions (ISAs) and an extra hardware support.
— DF-Threads is a variant of dynamic dataflow that can be easily interfaced to Pthreads based APIs.
— The DDM execution model has inherited a data availability based execution feature from the dynamic dataflow graph (more precisely from Decoupled Data Driven (D3) graphs [EG90]).
— The Scheduled Dataflow (SDF) architecture represents a decoupled memory/execution model that can be implemented using non-blocking threads that follow a dataflow activation rule.

In experiments, Nowatzki et al. showed that dataflow model can improve performance by reducing energy up to 40%, if a core provides support for both dataflow and out-of-order execution mode [NGS15].

3.5. Heterogeneous Programming Model

Now, heterogeneity is a common feature of recent execution platforms. Heterogeneity can be seen in inter-core domain (such as CPU and GPUs, CPU with Intel Xeon Phi) or intra-core domain (such as Asymmetric CMPs, ARM’s bigLITTLE). To support heterogeneity, CUDA and OpenCL are most widely used. But, there is also a trend to
Table II. Programming Model/Language Comparison

| Programming Model | Primary Programming Model | Supported Platforms |
|-------------------|---------------------------|---------------------|
| Name              | Shared                    | Distributed         | Dataflow | CPU | GPU | Accelerators (e.g. Intel Phi) |
| Pthreads          | ✓                          | ×                   | ×        | ✓   | ×   | ✓                      |
| CilkPlus          | ✓                          | ×                   | ×        | ✓   | ×   | ✓                      |
| TBB               | ✓                          | ×                   | ×        | ✓   | ×   | ✓                      |
| OpenMP            | ×                          | ×                   |          | ✓   | ×   | ✓                      |
| Open MPI          |                           |                     |          | ✓   | ×   | ✓                      |
| PGAS based Languages |                     | ✓                   |          | ✓   | ×   | ✓                      |
| Dataflow based Languages |                     | ×                   | ✓        | ✓   | ×   | ✓                      |
| CUDA              | ✓                          | ×                   |          | ✓   | ×   | ✓                      |
| OpenCL            | ✓                          | ×                   |          | ✓   | ×   | ✓                      |

combine multiple programming models (called as *hybrid programming*) due to their unique features to support heterogeneity.

3.5.1. **CUDA.** Compute Unified Device Architecture (CUDA) is a well-known parallel computing platform mainly for NVIDIA GPUs [OHL08]. GPU threads are very lightweight and they are intended for simple computational units and follow SPMD execution. Hence needed in higher number to perform a large task. The number of concurrent GPU threads is fully dependent on the underlying hardware. CUDA support shared-memory model and views the execution ecosystem as a collection of host unit and GPUs. Thread execution is managed implicitly by the runtime, but workload distribution is handled explicitly by the programmer. A CUDA kernel consists of a multi-dimensional array of threads. These array of threads are further divided into blocks and each block is viewed as a three dimensional array of threads. Fixed number of blocks is executed as *warps* and only one warp can be executed by Streaming Multi-processor (SM). All threads have thread id within a block and executed in a lock-step process. CUDA support only intra-block thread communication. Several generations of CUDA have been introduced in the last few years, with increasingly enhanced communication features among threads.

3.5.2. **OpenCL.** Open Computing Language (OpenCL) [SGS10] is the first industry standard language to address the need to write a single code for coordinating the execution among CPUs, GPUs and other accelerators like Intel Xeon Phi. OpenCL provides a high level abstraction of heterogeneous platforms and is also thread-safe. It supports four operational models. They are platform, execution and multilevel memory models, and programming models. The programmer is most often responsible for managing the execution order. An application written using OpenCL consists of a host program and a kernel which run on a device. Each processing element is called as *device*. OpenCL API specifies how to inquire, select and initialize the compute devices, whereas the related runtime manages the kernel across multiple compute devices.

3.5.3. **Hybrid Programming Model.** To support heterogeneity and the hierarchy of underlying different platforms, hybrid programming models combine the advantages of shared-memory (e.g. simplicity, efficiency, ease of programming) with those of distributed memory (like scalability) models. To name a few are: OmpSs [BMD+11], is based on OpenMP and StarSs [Lab10] programming model and supports SMPs, GPUs and also hybrid SMP/GPU-based execution. Other hybrid programming models include:
Table III. Characteristics of Programming Models/Languages

| Name         | Scalable | Ease of Coding | High Performance | API/Library | Compiler Support | Language |
|--------------|----------|----------------|------------------|-------------|------------------|----------|
| Pthreads     | ✓        | ✓              | ✓                | ✓           | ✓                | ✓        |
| CilkPlus     | ✓        | ✓              | ✓                | ✓           | ✓                | ✓        |
| TBB          | ✓        | ✓              | ✓                | ✓           | ✓                | ✓        |
| OpenMP       | ✓        | ✓              | ✓                | ✓           | ✓                | ✓        |
| Open MPI     | ✓        | ✓              | ✓                | ✓           | ✓                | ✓        |
| PGAS based Languages | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Dataflow based Languages | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| CUDA         | ✓        | ✓              | ✓                | ✓           | ✓                | ✓        |
| OpenCL       | ✓        | ✓              | ✓                | ✓           | ✓                | ✓        |

— CUDA with OpenMP and/or MPI: Another approach in a heterogeneous environment in which CUDA is combined with OpenMP and MPI. From a technical point of view, CUDA does not have the privilege of sharing the memories of both CPU and GPU. It needs some support (by using OpenMP or MPI) for receiving input data from CPU.
— MPI and OpenMP: In this model, MPI is mainly responsible for thread decomposition and OpenMP is for fine grain TLP.
— Pthreads with MPI: Pthreads are used to generate concurrent tasks that are to be executed on a single processor in the SMP environment while MPI takes care of communication between nodes.

Finally to conclude the section, we summarise the discussed programming models in Table II and Table III.

4. HARDWARE SUPPORT FOR THREADS

Multicore processors are developed to achieve scalability and a higher level of concurrency, while maintaining cache coherency. In multicore environment, assignments of threads to processing units are done either by multiplexing them in space or in time. In space multiplexing, threads assignment is made to optimize the use of spare resources. In time multiplexing, a time quantum is allocated among running threads. Controlling the number of threads per application can reduce power consumption and improve performance. Feedback Driven Threading (FDT) [SQP08] is a proposed framework for dynamically controlling the number of threads using running application behavior. Others [HCVC+14], proposed a methodology for dynamically reducing the number of concurrent threads to improve performance and energy efficiency. Generally, hardware or software multithreading hide latency. CDC 6600 developed by Control Data Corp in the 1960s and the Heterogeneous Element Processor (HEP) developed in the 1970s [Smi82] are known examples of hardware multithreading in earliest days.

— The CDC 6600 is an example of a barrel processor architecture that executes different threads on every clock cycle to support “fine-grain” multithreading. It was a distributed, RISC based general purpose computing system. Unlike SMT in modern superscalar architecture, interleaved multithreading generally does not allow execution of multiple instructions in one cycle.
— Similarly, the HEP system was based on multiple instruction, multiple data (MIMD) architecture and consisted of up to 16 processors. Each core could support a maximum of 128 threads and had eight pipeline stages. A large number of dynamically allocatable registers were used to support the massive thread execution.
— Unlike others, SCALE [KBH+04] is a prototype based on vector-thread (VT) architecture that combines the advantages of vector and multithreaded computing. VT
architecture was mainly developed for low-power, high-performance embedded systems.

— More recently, a family of massive thread throughput oriented processors have emerged to support larger core counts and a large number of threads. Commercial examples include GPGPUs, Intel’s Many Integrated Core (MIC) architecture, IBM’s PowerPC.

In the following three sub-sections, we group the hardware based thread supports proposed specifically for processor (sub-section 4.1), GPUs (sub-section 4.2) and reconfigurable architectures (sub-section 4.3) by academia as well as by industry.

4.1. Processor Based Thread Support

4.1.1. Multiscalar Processor. Multiscalar processor [SBV95] is a processing paradigm that initiates execution by constructing a control flow graph (CFG) of an application. During execution, the CFG is divided into large instruction sets basically tasks that are to be executed. After the tasks are identified, independent instructions are extracted for further parallel execution. To generate an efficient CFG, programs are coded with special information to guide the underlying hardware. The multiscalar hardware processes the CFG and assigns tasks to cores. The running threads of multiscalar processors are considered as different tasks of CFG, while being control and data dependent among one another.

4.1.2. Microthreading. Microthreading [BJM96] executes a small set of executable instructions in RISC microprocessor and supports speculative execution by executing instructions from non-resident threads by not flushing out the entire pipeline. A thread can be considered as a collection of multiple microthreads as each microthreads share the same pipeline and registers. Microthreading supports TLP by exploiting loops in the application (SPMD based concurrency) and schedules them on available cores. It also supports ILP by specifying a set of threads that generate MIMD concurrency in an execution block. Generally, the thread scheduler is responsible to distribute microthreads onto the cores and uses sleep-wake up mechanism for its synchronization and interleaving of microthreads. It supports explicit and parametric concurrency model using its extended ISAs which can be further applied to multiple ISAs that supports in-order execution.

4.1.3. Efficient Architecture for Running THreads. Efficient Architecture for Running THreads (EARTH) [The99] belongs to the group of hybrid dataflow/von Neumann execution model. The program execution model (PXM) in EARTH is customized to enhance support for dataflow threads. For efficient runtime execution, programs are divided into two levels of threads. The first level is called as a threaded procedure (similar to functions, but different with respect to frame allocation, thread invocation, scheduling and parameter passing). The second level is the runtime thread which is known as fiber. Fiber exhibits features such as sequential execution of its instruction sets and non-preemptive execution. It supports three states (such as enabled, active, and dormant) during its life cycle. When a thread is ready to execute, the system enables the fiber, and it executes inside the threaded procedure. Generally, procedures are invoked automatically by the application, but can be terminated explicitly. After invoking the procedure, the system creates a context for the procedure and executes other housekeeping jobs. Finally, when execution is complete, fiber is removed from the processor, but the associated threaded procedure may stay alive.

4.1.4. Hydra CMP. Hydra CMP [HHS*00] is a modified CMP that wraps four MIPS processors within a single die. Hydra CMP divide applications into arbitrary threads with or without dependencies. These threads run in parallel, but commit in sequential
order. Sequential application can be parallelized (to a limited extent) via thread level speculation and without any extra overheads. Hydra CMP also provides compiler and runtime support for thread level speculation. The runtime controls speculative threads via the co-processor interface and effectively handles all data dependencies related hazards (such as write after read (WAR), write after write (WAW) and read after write (RAW)).

4.1.5. Mini-threads. Mini-threads [REL03] was another attempt to increase the TLP and throughput of SMT processors by adding a new hardware module called thread-state. This hardware block is able to execute normal threads and mini-threads as well. From candidate application, multiple mini-threads are extracted to share the SMT hardware context’s register set, because mini-threads do not need the complete register set. Mini-threads have modified the way architectural register numbers are mapped to locations in renaming tables. When mini-threads within the same context need to access the same architectural register, they automatically address the same registers and access them.

4.1.6. WaveScalar. WaveScalar [SMSO03] is a tile-based processor architecture for executing conventional Pthreads, dataflow threads and also coarse/fine-grain threads with low overheads (like thread creation and destroy). To support multithreading, WaveScalar extended its instruction set. Its thread spawning mechanism tries to parallelize small loops of the application. Newly created threads do not have their own stack and cannot make functions call, but has its own thread identification. WaveScalar provides each thread with a consistent memory view and lets application manage the memory ordering directly. To support the concurrent thread execution, it supports multiple, independent sequence of ordered memory access. For inter-thread synchronization it provides lock.

4.1.7. TRIPS. TRIPS [SNL+03] is a chip prototype aimed at providing better performance at lower power by executing large instructions (up to 1,024 instructions) on the fly. TRIPS is a static placement, dynamic issue (SPDI) based architecture and employs dynamic issue methodology to tolerate latency. Unlike RISC or CISC, TRIPS block architecture is based on explicit data graph execution (EDGE) instruction sets, which do not perform operand encoding. TRIPS supports TLP and DLP on a single threaded application using its four, 16-wide, out-of-order cores. It can support a maximum of eight threads in parallel per core at runtime and provides an interface between the core and the compiler to enhance the performance. To achieve different levels of parallelism, it divides application at different levels of granularity. TRIPS compiler constructs a DAG and schedules each thread onto a block, where each block is a collection of 128 instructions. TRIPS partitions threads for enhanced, controlled execution. Similarly, block sequencing mechanism controls the execution of threads. Likewise to threads, cores are also partitioned in order to optimize communication and computation.

4.1.8. Cyclops-64. Cyclops-64 (C64) [ACC+03] is a single chip shared-memory multiprocessor which wraps 160 homogeneous cores together in a single chip. Each core in the C64 is a 64-bit single issue, in-order RISC processor and has two thread units. C64 supports non-preemptive thread execution and implements barrier in hardware. A core can switch the mode of a thread from running-mode to sleep-mode via hardware interrupts, although context-switching is not supported by C64. C64 provides three levels of memory hierarchy. They are SP, on-chip SRAM, and off-chip DRAM, which are fully exposed to the programmer. All memory modules are in the same address space, and directly accessible by the cores, but provide different latencies. Each cores do not have data cache, but the SRAM bank is used as a scratch pad memory. Code is stored in off-chip DRAM and the remaining portion of SRAM forms the global memory. The
Cyclops software stack has a compiler, kernel and runtime. The programmer controls data movement and can use pragmas to change data as well as code locations. OpenMP is the default programming model for Cyclops and TiNy Threads (TNT) [DCZHG05] is a native thread runtime library for the C64. A thread in TNN is a code section that can be executed by mapping to hardware threads, whereas the runtime is a virtual machine that provides API and features for runtime thread execution.

4.1.9. PowerPC. IBM's PowerPC [Fre05] is based on RISC architecture. Generally, PowerPC based cores are superscalar, symmetric and multiprocessor and are better suited for large, memory intensive applications. Recent processor model, like PowerPC8 can support three levels of on-cache hierarchy and off-chip L4 cache. All caches are dynamically shared among threads. L2 cache is a unified cache that maintains full hardware coherency and implements a snooping-based cache coherency protocol. Some of the PowerPC based processor models support coarse grain as well as fine grain multithreading. To provide massive LTP, each core can execute different numbers of threads simultaneously without involving applications or the OS. This type of processor executes a single thread for a long time until it terminates or stalls. After it stalls, it switches back to another thread, thereby using CPU cycles more efficiently. For efficient memory management, PowerPC architecture supports unified segment lookaside buffer (SLB) and a translation lookaside buffer (TLB) for both instruction as well as for data.

4.1.10. Cell Multiprocessor. Cell Multiprocessor [MS05] is a collection of small, but powerful Power architecture based processors (PPE) and multiple synergistic processor elements (SPEs). It supports two threads simultaneously and follows the SIMD based execution model. It was mainly developed to work as an accelerator. At runtime, instructions from two threads are issued in alternate cycles, hence improving pipeline utilization. To support Cell architecture, a source-to-source compiler and a runtime has also been developed and called as Cell Superscalar (CellSs) [BPBL06]. CellSs runtime is responsible for task scheduling and data handling for Cell processors. Given a sequential application in C, the compiler generates two files: one for the PPE compiler to generate the PPE objects (corresponding to the main program) and the other one for SPE objects to be compiled with the SPE compiler (further to be executed in a SPEs). Meanwhile, Sequoia [FHK+06] is developed as a programming language for Cell BE architecture and CPU clusters.

4.1.11. Network-Driven Processor. Network-Driven Processor (NDP) [CJK+05] provides a hardware extension to CMP to ease the runtime thread management. Author argued for a hardware-software co-design to exploit the inherent parallelism of compilers via hardware modulation. NDP lowers thread management overheads by providing support for thread creation, scheduling and context-switching. The thread scheduler is implemented in hardware, where thread cloning is performed. It can also dynamically map and schedules threads onto cores with its customized ISA support. NDP tile architecture consists of one block per core, called a thread table, to create a new thread. Thread scheduler tracks thread table, and selects threads for execution based on priorities.

4.1.12. Carbon. Carbon [KHN07] provides a hardware module for shared-memory CMPs to speedup dynamic task scheduling. It supports fork-join parallel execution model and provides two main components. They are a centralized global task unit and a distributed local task unit. The global task unit implements a thread stealing algorithm to increase system throughput. Similarly, the local task unit is added to each core as a task buffer to hide queueing latency. Customized ISAs are also proposed for managing thread distribution across cores.
4.1.13. Rigel. Rigel \cite{KJJ+09} is a programmable accelerator architecture for both DLP and TLP. Rigel supports thousands of small hierarchically-organized, in-order processing cores that in turn supports SIMD execution pattern. Equipped with a 32-bit RISC core with a fully-pipelined, single-precision floating-point unit, and an independent fetch unit, it supports a single-global-address-space memory model. Rigel supports arbitrary control flow among threads during execution. Rigel software stack consists of an API, an LPI (low-level programming interface) and its customized ISAs. The software API handles basic operations like managing resources in memory queues and updating those queues. Similarly, LPI can be used by programmers to distribute tasks.

4.1.14. Asynchronous Direct Messages. Asynchronous Direct Messages (ADM) \cite{SYK10} proposes an architectural extension of short message exchange support between cores. For better communication, ADM supports short and asynchronous message exchanges via registers without going through memory hierarchy. A hardware unit per core has been added to accelerate fine-grain thread scheduling. To implement the work stealing, a subset of worker threads communicates via (short) messages. To prevent deadlocks, a new virtual network has also been added to the router. ADM allows threads to maintain task queues and communication to be overlapped. A thread can be a worker or a manager, but also can change its role in future executions. Extra features have been added to the OS for supporting thread execution in ADM environment.

4.1.15. Cray XMT. Cray XMT \cite{Cra10} is built upon shared-memory, Cray multi-threaded architecture (Cray MTA) and Cray XT Infrastructure. Cray MTA architecture is based on a VLIW processor and IMT. The XMT programming model supports flat, globally accessible, shared-memory model and exploits loop level parallelism to achieve better performance. Each processor in the Cray XMT system can handle up to 128 independent, concurrent hardware threads. In XMT, hardware threads are called *streams*. Generally, threads are very lightweight software objects and are multiplexed onto hardware streams. A stream stores a thread’s state and executes it. Each core contains the associated contexts of a stream. XMT achieves massive parallelism via rapid context-switching among streams and provides extended memory semantics for better performance. On each clock cycle, the instruction issue unit selects a stream to execute from the pool of ready streams.

4.1.16. Many Integrated Core (MIC) Architecture. Intel’s Xeon Phi \cite{JR13} is an accelerator based on MIC architecture. MIC architecture has been developed for massive thread parallelism that follows a SIMD execution model. Each x86 cores follow in-order execution, but only has two levels of cache. It supports MESI (Modified/Exclusive/Shared/Invalid) protocol for cache and memory coherency. Each core can run its own OS and can provide acceleration via its x86 ISA based multithreaded cores. Phi can support both scalar and vector processing, and vector processing is crucial to the performance of MIC architecture. Furthermore, each core can support up to four threads. At least two threads are needed to attain maximum pipeline utilization. Experiments have shown that memory latency is the main performance bottleneck for the Xeon Phi \cite{SKC14}.

4.2. GPU Based Thread Scheduler

In general-purpose computing on graphics processing units (GPGPU), CPU and GPU are used to form a heterogeneous co-processing system in which CPU runs sequential code and GPU executes main compute intensive part. A GPU processing core is a SIMD based single pipeline, grouped together in clusters for better performance and share fetch, decode stages. To maintain execution order among threads, a barrier must
be used by the programmer. GPGPUs support either single or unified virtual-address space memory types, which are segmented further according to manufacturers. Generally, L1 and L2 caches are shared by the threads running on GPGPU cores. GPGPU supports CUDA programming model.

GigaThread scheduler [Gla09] is a commercial two-level, out-of-order distributed thread scheduler for GPUs. It provides fast application context switching and was first implemented on the NVIDIA G80. GPUs provide hardware support to manage huge quantities of simultaneous light weight threads (warps). Before initiating thread scheduling, GigaThread scheduler fetches input data from the system memory and copies it to appropriate frame buffers. Later, it schedules threads in two levels.

1. In the first level, it schedules thread blocks to SM thread schedulers and supports out-of-order thread block execution.

2. In the second level, via warp scheduler, warps are scheduled on GPU cores.

For better runtime thread management, GigaThread scheduler supports concurrent kernel execution, but to do so it must be aware of all kernels running in the system.

4.3. Reconfigurable Architecture Based Thread Support

In general purpose computing (CPUs, GPUs, accelerators), the hardware layout is fixed. Adding multithreading support to reconfigurable architecture (mainly based on FPGAs) is complex due to its application based flexibility. But, over the past few years, there has been growing interest in the reconfigurable hardware domain, especially adding thread support for multithreaded applications. When systems support reconfigurable architectures, they are called Field-Programmable Custom Computing Machines (FPCCM) [SVC+02]. To name a few are:

— OS4RS [MNM+04] is a project to design an OS for Reconfigurable Systems (OS4RS). It provides runtime as well as communication support to reconfigurable system-on-chip (SoC). It’s runtime supports the dynamic task relocation in the system. The system has a two-level thread scheduler, the low-level is implemented by the hardware and the high-level by the software.

— The HybridThreads Project (hthreads) is a real-time embedded OS that can run multithreaded Pthreads on both CPUs and FPGAs. It allows programmers to perform computations using threads as their building blocks. Hthreads [PAA+06] is an example of computational architecture that provides three co-operating layers of abstraction (one each for OS, reconfigurable devices and programmer) to connect general purpose computing systems to reconfigurable architectures. It supports thread migration, thread synchronization (using semaphores) and (hardware as well as software) thread scheduling. In fact, hardware threads can even be customized by user-defined components, controllers and interfaces.

— [UMKU06], proposes a real-time hardware scheduler that interfaces with a reconfigurable architecture to the host core. This model is capable of (no-preemption) scheduling only one thread at a time, and no additional ISAs are proposed for thread management on the reconfigurable platform.

— [MSIG07] proposed a multiprocessor system connecting a multithreaded Digital Signal Processor (DSP) and RISC processors with multiple polymorphic hardware accelerators (PHAs). An interleaved multithreading mechanism has been used to hide reconfiguration latency, and accelerators have been used to increase the execution speed of the core.

— MTADRES (Multi-Threaded Architecture for Dynamically Reconfigurable Embedded Systems (MTADRES)) [WKMB07], aims to exploit TLPs in FPCCM systems. ADRES is based on a coarse-grain reconfigurable matrix layout. It is based on the
idea that threads have different resource requirements and the hardware can have different sized resource blocks. To support multithreading, the resources are partitioned as per the required size (adhering resource constraint for each partition).

Adding multithreading support to reconfigurable micro-architecture in highly customizable soft cores (in FPGAs) [Xil06] has also been experimented in some works. To name a few: ReconOS [LP09] proposed an execution environment (based on eCos and Linux) that extends Pthreads support in FPGAs, via a set of communication and synchronization primitives provided to both software and hardware threads. Although thread creation remains fully dependent on the programmer. Other works such as [FCVB06; DML05] proposed customized instruction sets for thread management in reconfigurable environment.

5. SOFTWARE SUPPORT

Similar to hardware support, software support is also needed to maximize the speed of execution while minimizing I/O stalls. In a massive parallel application, the deterministic behavior is very critical to produce correct output at various hardware platforms. Deterministic execution is very much necessary to scientific modeling, micro-architectural simulation, financial transactions and others. Deterministic behavior also helps programmer to track down the performance critical issues or debug the errors (if any), but it also add extra overheads. Deterministic execution support is not common among general purpose processors [WTM13]. Apart that, threads belonging to different programming models also show some different behaviors during execution. So, a thread either produces correct output due to its deterministic behavior or incorrect output due to its non-deterministic behavior. One of the main reasons for introducing non-deterministic behavior in application is the unprotected shared-memory access.

— At runtime, non-determinism is caused by threads that are generated without proper knowledge either of data dependency or update order of shared data. Non-determinism can be introduced into an application internally or externally. Generally, internal issues are coming from the application itself (like unpredictable thread interleaving) and this type of issues are relatively easier to fix. The non-deterministic execution of threads has four primary reasons: deadlock [Hav68], race condition [NM92], multiple thread update issues [FQ03] and access order of code or variables [LPSZ08]. In parallel programming, debugging is required to remove non-determinism. But, debugging a large multithreaded application is not trivial [MH89].

— Observing deterministic thread behavior in a shared-memory multithreaded application is not explicit and difficult [Lee06], because threads suffer from interleaving of memory accesses by shared data. Figure 7 shows the steps for executing deterministic threads in shared-memory. Deterministic multithreading can be classified into two classes [OAA09]. They are strong determinism and weak determinism.

1. In strong determinism, the same output will be produced all the time by a given program input (repeatability).
2. In weak determinism, the same output will be produced only, when the shared input data is protected by locks.

DMP [DLCO09] is one of the examples of hardware that supports deterministic program execution. The StreaMIT language [TKA02], developed for high-performance streaming applications, has explicit, built-in determinism. Like Intel’s TBB, Data Parallel Haskell [CLPJ+07] also offers deterministic sequential semantics. Furthermore, it has been argued that a deterministic thread behavior model should be added to
conventional object-oriented languages like Java [BAAS09]. Below, we discuss some proposal proposed from academia to add determinism at the software level for multi-threaded applications.

5.1. Software Support for Deterministic Threading

5.1.1. Kendo. Kendo [OAA09] finds non-deterministic bugs in shared-memory multithreaded applications, and provides weak deterministic multithreading via Pthreads API. Kendo proposes an algorithm based on each thread’s execution time to provide deterministic interleaving of lock acquisitions and non-protected reads. The lock based algorithm allows threads to acquire their locks based on an order as defined by deterministic logical time. The algorithm tracks the deterministic logical time of each thread. Logical time shows how thread execution progresses. Deterministic logical time is fully dependent on the number of running threads and their associated events. The deterministic logical time is computed separately and does not affect others. To maintain determinism, each thread holds a given lock at a given deterministic logical time. In other words, deterministic logical time allows threads to acquire locks one-by-one without causing deadlock. To be able to retrieve the deterministic logical clocks, a performance counter has also been added to the OS.

5.1.2. Grace. Grace [BYLN09] is a runtime system that removes concurrency errors in a fork-join execution model based multithreaded applications. It divides threads into number of processes via deterministic, sequential execution illusion (behaviorally equivalent to a sequential program). The runtime has been implemented using Pthreads API. Grace uses events and joins, to collectively divide threads into processes and, at synchronization points, combine them deterministically to perform sequential

Fig. 7. Deterministic Serialization of Shared-Memory [DLC009]
execution flow. Each newly-created independent thread becomes a sequential function, and locks are implemented as no-ops to eliminate deadlock. Furthermore, commit protocols, page protection and virtual memory mappings (using software transactional memory) are used to isolate processes and thereby, restrict atomicity and access order violations. Race conditions are handled by deterministically committing each thread’s state. Updates of one thread cannot affect another. To avoid race condition, it combines speculative threading (local to each thread’s memory space) with a sequential commit protocol that provides sequential thread commit.

5.1.3. Coredet. Coredet [BAD+10] is a runtime system that enforces strong deterministic execution in multithreaded applications (written by C and C++ language). Coredet works with most type-safe languages and also provide extra support for compiler. The runtime supports relaxed memory ordering. Threads are either scheduled in parallel (with no communication) or serially (with possible thread communication). As a first step, threads are scheduled in a round-robin process with a finite logical time slice. A barrier also exists between parallel and serial threads. In Coredet, parallel threads use private data and serial threads use shared data. To better manage thread execution, a hash table keeps track of ownership status for each memory location for serialization. Coredet implements deterministic runtime using Pthreads library, but suffers from slow execution speed.

5.1.4. Deterministic Process Groups. Deterministic Process Groups (DPGs) [BHCG10] is a virtual abstraction of OS to provide determinism between threads and processes inside, by serializing the communications between shared-memory space and external communication paths (like files, sockets, pipes). DPG can produce the same output given the same input and thread state information. A single DPG can run multiple multithreaded processes. DPGs can be thought of as boxes in which threads execute in deterministic fashion. A thread leaves the DPG when it completes its execution. DPG also provides an interface using which programmer can control external non-deterministic communication at desired granularity. The shim interface of DPGs is responsible for interposing deterministic and non-deterministic calls and also for monitoring all DPGs.

5.1.5. DTHREADS. DTHREADS [LCB11] is also a runtime that aims to include deterministic behavior in multithreaded applications written in C/C++. It provides deterministic behavior in both race conditions and deadlock conditions. DTHREADS divides applications into multiple processes with their relevant thread contexts and performs cross-thread memory isolation by replacing threads with processes. Processes have disjoint address space. The creation, cancellation and termination of threads are managed sequentially. DTHREADS ensures determinism by isolating and synchronizing each thread’s memory. A single global token is used to obtain deterministic updates and allows interleaving only at synchronization points. For synchronization, it inherits all Pthreads synchronization primitives.

Few alternative approaches to introduce determinism in applications are Parrot [CSL+13], Replay [LCFN12] and Record-Reply [CWG+11]. Below, we also like to mention two works for supporting TLP.

5.2. Decoupled Software Pipelining
Decoupled Software Pipelining (DSWP) [ORSA05] exploits TLP in application loops (mainly PMT) via non-speculative thread extraction. It can extract threads without knowing their architectural details, but to do so, it needs special hardware support. DSWP can offer improved latency tolerance, inter-core communication and per-core resource requirements, but based on some fixed assumptions. The DSWP algorithm
examines the code and builds two threads called *producer* and *consumer*. First, DSWP builds a dependency graph which contains data, control and memory dependency. Next, it builds a DAG by finding strongly-connected components (SCCs). It partially replicates a CFG for each thread for providing better TLP.

### 5.3. Open Community Runtime

Open community runtime (OCR) Project [wg15] aims to develop a low-level, task-based runtime to enhance power efficiency, programmability and application reliability while maintaining the performance. Currently, OCR supports both shared-memory and distributed memory model. In OCR, control and data dependency are equivalent. Generally, data are stored in data blocks and a task can only access its data if it owns it. A task is a non-blocking set of instructions and executed when its dependencies are met (with the help of events). An OCR program is defined as a DAG with event-driven tasks (EDTs) and computations are performed by executing EDTs. OCR lets the developer to exploit the parallelism by abstracting issues (like event-driven, asynchronous task based management issues). But, task based systems are not easy to maintain in large application as the task scheduler has significant overhead.

### 6. THREAD MANAGEMENT: IT’S ISSUES AND DISCUSSION

Generally, multithreaded applications do not scale easily due to issues such as complex thread management, synchronization, load imbalance, memory hierarchy. In this section, we are discussing the issues relevant to thread management.

#### 6.1. Primary Memory Issues

Primary memory is very crucial for the improved performance and throughput of the whole system. Generally, primary memory follows a hierarchical organization. It has registers, caches and memory banks. The more memory banks, the better the performance. Internally, memory banks are organized by rows and columns. A single cell constitutes the smallest unit of memory. A row-buffer is a tiny unit of memory comprising multiple rows and columns. Row-buffer locality (RBL) is the average hit-rate of the row-buffer across the banks. But, memory intensity is calculated from the number of DRAM memory access requests that follow a complete miss. Bank-level parallelism execute requests from threads in parallel and also reduces memory stalls. Basically, higher the degree of memory bank-level parallelism and row-buffer locality, the more DRAM bandwidth utilization. Employing smart techniques, the memory access behavior of threads can be optimized with bank-level parallelism [MM08], row-buffer locality [RDK00] and memory intensity [KHMHB10].

At runtime, memory is used by concurrent threads, so achieving high system throughput, and effective resource allocation entails optimizing memory access. The memory access behavior of running threads can be either latency-sensitive or bandwidth-sensitive. High latency memory access in shared-memory environments significantly reduces thread performance. On the other hand, bandwidth-sensitive threads are more vulnerable to starvation. To improve system throughput, latency-sensitive threads must be executed according to priorities, while bandwidth-sensitive threads must be executed to minimize slowdown. [YHK13] quantitatively analyzed that locality-aware scheduling in multicore processors can improve speedup and also reduces the energy cost.

#### 6.2. Cache Management

LLC is also one of the most important performance drivers of a CPU, and employing smart algorithms together with locality-aware data structures can also improve the hit-ratio in cache (more in sub-subsection 2.3.1). A popular solution for synchro-
nizing access to shared resources is to use locks, which, however, results reduction in parallelism, because serializing all threads waste time. Cache coherence protocol (like MESI) is used to reduce the false sharing of cache lines. Dynamically-allocated shared data can cause false sharing, but thread-local copies of data reduce its frequency. Recently, to reduce synchronized access overheads, compilers include functionality such as the maintenance of separate heaps for each thread in use. A special memory location is used to allocate or free the memory heaps for threads. Fedorova et al. have shown that a cache fair thread scheduling algorithm provides fairer thread scheduling, and also provide better performance stability on shared-cache multicore processors [Fed06]. In another work, a cache-aware thread co-scheduling algorithm was applied to reduce L2 cache contention by avoiding simultaneous scheduling in real-time systems [ACD06].

Generally, non-blocking locks and built-in synchronization APIs can increase overall system performance. In recent times, multiple page replacement policies have been proposed such as Early Eviction LRU (EELRU) [SKW99], Adaptive Replacement Cache (ARC) [MM03], Low Inter-reference Recency Set (LIRS) [JZ02], Dynamic Insertion Policy (DIP) [QJP+07]. Other works for employing better cache sharing/management policy for CMP chips are Cooperative Caching (CC) [CS06], Multiple Timesharing Partitions (MTP) [CS14], OWL [JKCN+13].

Generally, optimized application code can drastically reduce resource contention. By employing profiling tools at compile time, programmers can also optimize the cache access pattern, so that resource contention is minimized. There exist few well-known application profiling tools like Gprof [GKM82] for application profiling and performance optimization. The set of tools can be grouped either into application profiler or data/cache access profiler (such as Valgrind [NS07], Intel's Vtune [Rei05]). Application profilers gather runtime statistics about currently running applications, but suffer from high implementation overheads.

6.3. Synchronization

Synchronization is necessary, but also has its overheads. Generally, synchronization slows down execution at the critical section of the code by allowing only serial execution. Synchronization should be implemented in such a way, so that the cost does not surpass its execution time e.g. in Linux, the futex [Dre05] system call has less overhead than the Pthreads synchronization primitives.

At runtime, performance can be improved using per-thread storage and soft ordering of execution sequence. Local thread storage (TLS), like shared global data structure may reduce synchronization overheads. TLS does not remove synchronization, but only moves it from a critical to a non-critical section of the code. Another advantage of using TLS is to keep critical section code longer than shared data. If the cores do not share data cache, then data may stay in the cache unnecessarily. In this case the update is minimal and only affects the local memory by preventing the generation of unnecessary traffic. TLS can be implemented by allocating a stack variable in a given scope. In OpenMP, a thread-local variable can be assigned to the scope of a private clause on the parallel pragma by (#pragma omp [parallel] sections [clauses]).

6.3.1. Deadlock. Deadlock is one of the side effects of synchronization. It stalls the thread execution and reduces overall system throughput. Tools have also been developed to remove deadlocks, such as, RacerX [EA03], a static tool that uses flow-sensitive, inter-procedural analysis to detect race conditions and deadlocks in multithreaded applications. RacerX extracts the CFG from the application. Next it runs its deadlock and race checkers over the graph. Finally, it ranks the results. There are also tools developed to detect race conditions and deadlocks separately.
— Tools for detecting race conditions: FASTTRACK [FF09], Racetrack [YRC05], Eraser [SBN+97].
— Tools for detecting deadlocks: MagicFuzzer [CC12], [NPSG09; JPSN09] only for Java, Dreadlocks [KH08], Pulse [LELS05].

6.3.2. Memory Inconsistency Issues. Memory consistency model provide constraints to get consistent values across the system. Memory inconsistency issue arises when multiple concurrent threads have inconsistent view of their shared data [AG96], because the access order of variables differs from the order originally specified in the program. The issues are usually found in the compiler or in the core. In compiler, the instructions can be re-ordered for optimization, whereas in the core, thanks to out-of-order execution, data can be moved between registers, caches, and memories in any order.

6.3.3. Loop Management. Generally, loops are one of the most compute-intensive parts of multithreaded applications. Compiler plays an important role in extracting loops from code. Proper handling of critical regions within the loop can also improve the performance. Hence, efficient loop execution can increase overall execution performance. Two types of loop management techniques, namely, loop-split and loop-merge are popular. Loop splitting supports dependency and also helps to improve data locality. While, loop merging combines multiple nested loops to increase workload per iteration and lowers the overhead.

Usually, a compiler searches for loops to exploit parallelism (loop parallelism), but a compiler cannot analyze loops in depth nor determine thread safety. The compiler ensures that no data dependency exists in extracted parallel loops via dataflow analysis. Outer loop parallelization produces coarse-grain loops with multiple different iterations. Dividing these iterations among threads is critical to the load distribution among threads, because uneven division of threads can further lead to load imbalance. If the computations inside the loop are dependent, loop parallelization can only be achieved by meeting the following three hard constraints:

1. The number of iterations must be known at compile time. (In some cases, it is easy, e.g. “for-loop”; in others, more difficult, e.g. “while-loop”). Typically, the iteration number helps the compiler to distribute iterations evenly among threads at runtime.
2. Iterations must not share any data or control among them (like no aliasing of pointers or array references among threads).
3. Existence of jumps or branching violate the loop parallelization requirements.

6.4. Granularity
Granularity is an important factor in increasing CPU utilization in multithreaded applications (already discussed in sub-subsection 2.1.2). Ideal granularity is the amount of computation, each thread should execute with minimum workload imbalance and communication overheads (such as synchronization issues, message passing overheads or data sharing overheads across memory hierarchy). Too fine or too coarse granularity may hurt the system performance because less number of threads are unable to boost the CPU usage. Better data access patterns, I/O overlapping, and memory prefetching, can also increase the overall CPU utilization. In a recent work [AKG+15], authors have shown that optimal task granularity depends on the task scheduler. It also has been seen that increasing average task size may considerably increase the speedup at certain limit, but also may decrease the performance due to the absence of optimal granularity of the threads.
6.5. Thread Mapping

Thread mapping is another important process for achieving fairness, higher application throughput and better system utilization. A good thread scheduler should be able to reduce shared cache misses, cache contention and interconnect traffic, as well as be able to improve placements of runtime threads (as cores share TLBs). Sharing multiple threads among cores is essential for good load distribution. In data parallel applications, threads are mostly independent, so threads can be mapped to free cores, but scheduling threads of array-based applications is not trivial. Popular thread libraries use pinning threads to assign hardware threads and restrict their migration.

To conclude, going from Petascale to Exascale era, there has been a tremendous growth in micro-architectural domain. Number of cores as well as their capabilities are changing, which makes the runtime thread management more complex. Heterogeneity also has been induced into these multicore processors. But, we view the explicit thread support as very crucial for improving the performance of this cutting-edge hardware. Dedicated hardware support is very promising, but the generic applicability of the proposed approaches are also very essential. Similarly, the proposed software supports are also based on some critical assumptions, which also forces them not to become generic. Finally, to provide an efficient thread execution environment, we support for a software-hardware based co-design which is generic enough to be used by different programming models.

REFERENCES

Thomas E Anderson, Brian N Bershad, Edward D Lazowska, and Henry M Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems (TOCS)*, 10:53–79, 1992.

George Almasi, Clin Cascaual, Jose G Castanos, Monty Denneau, Derek Lieber, Jose E Moreira, and Henry S Warren Jr. Dissecting cyclops: A detailed analysis of a multithreaded architecture. *ACM SIGARCH Computer Architecture News*, 31(1):26–38, 2003.

James H Anderson, John M Calandrino, and UmaMaheswari C Devi. Real-time scheduling on multicore platforms. In *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, pages 179–190. IEEE, 2006.

Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L Steele Jr, Sam Tobin-Hochstadt, Carl Eastlund, et al. The fortress language specification. *Sun Microsystems*, 139:140, 2005.

Jose L Abellán, Juan Fernández, and Manuel E Acacio. Glocks: Efficient support for highly-contended locks in many-core cmps. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 893–905. IEEE, 2011.

Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *computer*, 29(12):66–76, 1996.

Dana Akhmetova, Gokcen Kestor, Roberto Gioiosa, Stefano Markidis, and Erwin Laure. On the application task granularity and the interplay with the scheduling overhead in many-core shared memory systems. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 428–437. IEEE, 2015.

AXI AMBA. Protocol specification. *ARM*, June, 2003.

Arvind and Rishiyur S Nikhil. Executing a program on the mit tagged-token dataflow architecture. *Computers, IEEE Transactions on*, 39:300–318, 1990.

Edward A. Ashcroft and William W. Wadge. Lucid, a nonprocedural language with iteration. *Communications of the ACM*, 20:519–526, 1977.

Robert Bocchino, Vikram Adve, Sarita Adve, and Marc Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, pages 4–4, 2009.

Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 53–64. ACM, 2010.

Luca Benini and Giovanni De Micheli. Networks on chips: a new soc paradigm. *Computer*, 35:70–78, 2002.

ACM Computing Surveys, Vol. X, No. X, Article X, Publication date: March 2016.
A Survey on Hardware and Software Support for Thread Level Parallelism

Geoffrey Blake, Ronald G Dreslinski, Trevor Mudge, and Krisztian Flautner. Evolution of thread-level parallelism in desktop applications. In ACM SIGARCH Computer Architecture News, volume 38, pages 302–313. ACM, 2010.

Anasua Bhowmik and Manoj Franklin. A general compiler framework for speculative multithreading. In Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures, pages 99–108. ACM, 2002.

Guy E Blelloch and Phillip B Gibbons. Effectively sharing a cache among threads. In Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures, pages 235–244. ACM, 2004.

Luiz André Barroso, Kouros Gharaeholoo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: a scalable architecture based on single-chip multiprocessing, volume 28. ACM, 2000.

Tom Bergan, Nicholas Hunt, Luis Coze, and Steven D Gribble. Deterministic process groups in dos. In OSDI, volume 10, pages 177–192, 2010.

Dan Bonachea and Jaein Jeong. Gasnet: A portable high-performance communication layer for global address-space languages. CS258 Parallel Computer Architecture Project, Spring, 2002.

Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system, volume 30. ACM, 1995.

A Bolychevsky, CR Jesshope, and VB Muchnick. Dynamic scheduling in risc architectures. In Computers and Digital Techniques, IEE Proceedings-, volume 143, pages 309–317. IET, 1996.

M Brinkmann and W. Koch. npth - the new pth library, 2015.

Ralph M Butler and Ewing L Luik. Monitors, messages, and clusters: the p4 parallel programming system. Parallel Computing, 20:547–564, 1994.

Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. Journal of the ACM (JACM), 46(5):720–748, 1999.

Tobias Bjerregaard and Shankar Madhavan. A survey of research and practices of network-on-chip. ACM Computing Surveys (CSUR), 38:1, 2006.

Javier Bueno, Luis Martinell, Alejandro Duran, Montse Farreras, Xavier Martorell, Rosa M Badia, Eduard Ayguade, and Jesus Labarta. Productive cluster programming with omps. In Euro-Par 2011 Parallel Processing, pages 555–566. Springer, 2011.

Bradford M Beckmann, Michael R Marty, David Wood, et al. Asr: Adaptive selective replication for cmp caches. In Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on, pages 443–454. IEEE, 2006.

Pieter Bellens, Josep M Perez, Rosa M Badia, and Jesus Labarta. Cellss: a programming model for the cell be architecture. In SC 2006 Conference, Proceedings of the ACM / IEEE, pages 5–5. IEEE, 2006.

Bob Boothe and Abhiram Ranade. Improved multithreaded techniques for hiding communication latency in multiprocessors. In ACM SIGARCH Computer Architecture News, volume 20, pages 214–223. ACM, 1992.

Jeffery A Brown and Dean M Tullsen. The shared-thread multiprocessor. In Proceedings of the 22nd annual international conference on Supercomputing, pages 73–82. ACM, 2008.

David R Butenhof. Programming with POSIX threads. Addison-Wesley Professional, 1997.

Nick Barrow-Williams, Christian Fensch, and Simon Moore. A communication characterisation of splash-2 and parsec. In Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on, pages 86–97. IEEE, 2009.

Emery D Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe multithreaded programming for c/c++. In ACM Sigplan Notices, volume 44, pages 81–96. ACM, 2009.

Chris M Cunningham and Dimitr R Avresky. Fault-tolerant adaptive routing for two-dimensional meshes. In High-Performance Computer Architecture, 1995. Proceedings., First IEEE Symposium on, pages 122–131. IEEE, 1995.

Yan Cai and WK Chan. Magicfuzzer: scalable deadlock detection for large-scale applications. In Proceedings of the 34th International Conference on Software Engineering, pages 606–616. IEEE Press, 2012.

Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. International Journal of High Performance Computing Applications, 21:291–312, 2007.

George F Coulouris, Jean Dollimore, and Tim Kindberg. Distributed systems: concepts and design. Addison-Wesley, 2005.

Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. An evaluation of global address space languages: co-
array fortran and unified parallel c. In Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 36–47. ACM, 2005.

Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on, pages 340–351. IEEE, 2005.

Qiong Cai, José González, Ryan Rakvic, Grigoris Magklis, Pedro Chaparro, and Antonio González. Meeting points: using thread criticality to adapt multicore hardware to parallel regions. In Proceedings of the 17th international conference on Parallel architectures and compilation techniques, pages 240–249. ACM, 2008.

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. ACM Sigplan Notices, 40:519–538, 2005.

Peng-Sheng Chen, Ming-Yu Hung, Yuan-Shin Hwang, Roy Dz-Ching Ju, and Jenq Kuen Lee. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. In ACM SIGPLAN Notices, volume 38, pages 25–36. ACM, 2003.

Sangyeun Cho and Lei Jin. Managing distributed, shared l2 caches through os-level page allocation. In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, pages 455–468. IEEE Computer Society, 2006.

Julia Chen, Philo Juang, Kevin Ko, Gilberto Contreras, David Penry, Ram Rangan, Adam Stoler, Li-Shiuan Peh, and Margaret Martonosi. Hardware-modulated parallelism in chip multiprocessors. ACM SIGARCH Computer Architecture News, 33(4):54–63, 2005.

Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. Using OpenMP: portable shared memory parallel programming, volume 10. MIT press, 2008.

Manuel MT Chakravarty, Roman Leschinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In Proceedings of the 2007 workshop on Declarative aspects of multicore programming, pages 10–18. ACM, 2007.

Gilberto Contreras and Margaret Martonosi. Characterizing and improving the performance of intel threading building blocks. In Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on, pages 57–66. IEEE, 2008.

Zeshan Chiashi, Michael D Powell, and TN Vijaykumar. Optimizing replication, communication, and capacity allocation in cmps. In Computer Architecture, 2005. ISCA05. Proceedings. 32nd International Symposium on, pages 357–368. IEEE, 2005.

XMT Cray. Programming environment user's guide. URL: http://docs.cray.com/books/S-2396-21/S-2396-21.pdf, 2010.

Jichuan Chang and Gurindar S Sohi. Cooperative caching for chip multiprocessors, volume 34. IEEE Computer Society, 2006.

Jichuan Chang and Gurindar S Sohi. Cooperative cache partitioning for chip multiprocessors. In 25th Anniversary International Conference on Supercomputing Anniversary Volume, pages 402–412. ACM, 2014.

Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A Gibson, and Randal E Bryant. Parrot: A practical runtime for deterministic, stable, and reliable threads. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pages 388–405. ACM, 2013.

Theofanis Constantinou, Yiannakis Sazeides, Pierre Michaud, Damien Fetis, and Andre Seznec. Performance implications of single thread migration on a chip multi-core. ACM SIGARCH Computer Architecture News, 33(4):80–91, 2005.

Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. Efficient deterministic multithreading through schedule relaxation. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pages 337–351. ACM, 2011.

Juan Del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R Gao. Tiny threads: A thread virtual machine for the cyclops64 cellular architecture. In Parallel and Distributed Processing Symposium, 2005. Proceedings. 18th IEEE International, pages 8–pp. IEEE, 2005.

Jack Bonnell Dennis. Data flow supercomputers. Computer, (11):48–56, 1980.

Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. Dmp: deterministic shared memory multiprocessing. In ACM SIGARCH Computer Architecture News, volume 37, pages 85–96. ACM, 2009.

Jack B Dennis and David P Misunas. A preliminary architecture for a basic data-flow processor. In ACM SIGARCH Computer Architecture News, volume 3, pages 126–132. ACM, 1975.

ACM Computing Surveys, Vol. X, No. X, Article X, Publication date: March 2016.
A Survey on Hardware and Software Support for Thread Level Parallelism

Robert Dimond, Oskar Mencer, and Wayne Luk. Custard—a customizable threaded fpga soft processor and tools. In Field Programmable Logic and Applications, 2005. International Conference on, pages 1–6. IEEE, 2005.

Reetuparna Das, Onur Mutlu, Thomas Moscibroda, and Chita R Das. Application-aware prioritization mechanisms for on-chip networks. In Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, pages 280–291. ACM, 2009.

Ulrich Drepper. Futexes are tricky. Red Hat Inc, Japan, 2005.

Haakon Dybdahl and Per Stenström. An adaptive shared/private ncu cache partitioning scheme for chip multiprocessors. In High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on, pages 2–12. IEEE, 2007.

Bronis R de Supinski and John May. Benchmarking pthreads performance. In PDPTA, pages 1985–1991, 1999.

William J Dally and Brian Towles. Route packets, not wires: on-chip interconnection networks. In Design Automation Conference, 2001. Proceedings., pages 684–689. IEEE, 2001.

Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In ACM SIGOPS Operating Systems Review, volume 37, pages 237–252. ACM, 2003.

Paraskevas Evripidou and Jean-Luc Gaudiot. A decoupled graph/computation data-driven architecture with variable-resolution actors. Technical report, University of Southern California, Los Angeles, CA (United States). Dept. of Electrical Engineering, 1990.

Tarek El-Ghazawi and Lauren Smith. Upc: unified parallel c. In Proceedings of the 2006 ACM/IEEE conference on Supercomputing, page 27. ACM, 2006.

Johan Eker and Jorn Janneck. Cal language report. Technical report, Tech. Rep. ERL Technical Memo UCB/ERL, 2003.

Richard J Eickemeyer, Ross E Johnson, Steven R Kunkel, Beng-Hong Lim, Mark S Squillante, and C Eric Wu. Evaluation of multithreaded processors and thread-switch policies. In High Performance Computing, pages 75–90. Springer, 1997.

Ralf S Engelschall. Gnu pth—the gnu portable threads. URL http://www.gnu.org/software/pth, 2005.

Blair Fort, Davor Capalija, Zronko G Vranesic, and Stephen D Brown. A multithreaded soft processor for socp area reduction. In Field-Programmable Custom Computing Machines, 2006. FCCM’06. 14th Annual IEEE Symposium on, pages 131–142. IEEE, 2006.

Alexandra Fedorova. Operating system scheduling for chip multithreaded processors. PhD thesis, Harvard University Cambridge, Massachusetts, 2006.

Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In ACM SIGPLAN Notices, volume 38, pages 338–349. ACM, 2003.

B Frey. Powerpc architecture book v2. 02. International Business Machines Corporation, 2005.

Alexandra Fedorova, Margo I Seltzer, Christopher A Small, and Daniel Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. Technical report, Harvard University Computer Science Group Technical Report TR-09-05, 2005.

Chris Fallin, Xiangyao Yu, Greg Nazario, and Onur Mutlu. A high-performance hierarchical ring on-chip interconnect with low-cost routers. Technical report, Carnegie Mellon University, 2011.

Dongrui Fan, Hao Zhang, Da Wang, Xiaochun Ye, Fenglong Song, Guoje Li, and Ninghui Sun. Godson-t: An efficient many-core processor exploring thread-level parallelism. Micro, IEEE, 32(2):38–47, 2012.

Roberto Giorgi and Paolo Faraboschi. An introduction to df-threads and their execution model. In Computer Architecture and High Performance Computing Workshop (SBAC-PADW), 2014 International Symposium on, pages 60–65. IEEE, 2014.

Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In Recent Advances in Parallel Virtual Machine and Message Passing Interface, pages 97–104. Springer, 2004.

Susan L Graham, Peter B Kessler, and Marshall K Mckusick. Gprof: A call graph execution profiler. In ACM Sigplan Notices, volume 17, pages 120–126. ACM, 1982.

ACM Computing Surveys, Vol. X, No. X, Article X, Publication date: March 2016.
Boris Grot, Stephen W Keckler, and Onur Mutlu. Preemptive virtual clock: a flexible, efficient, and cost-effective qos scheme for networks-on-chip. In Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, pages 268–279. ACM, 2009.

John R. Gurd, Chris C. Kirkham, and Ian Watson. The manchester prototype dataflow computer. Communications of the ACM, 28:34–52, 1985.

Peter N Glaskowsky. Nvidias fermi: the first complete gpu computing architecture. Nvidia, September, page 23, 2009.

William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. Parallel computing, 22:789–828, 1996.

Ananth Grama. Introduction to parallel computing. Pearson Education, 2003.

William D Gropp and Barry Smith. Chameleon parallel programming tools users manual. Technical report, Technical Report ANL-93/23, Argonne National Laboratory, 1993.

David Alejandro Padua Haiek. Multiprocessors: Discussion of some theoretical and practical problems. Department of Computer Science, University of Illinois at Urbana-Champaign, 1979.

James W. Havender. Avoiding deadlock in multitasking systems. IBM systems journal, 7(2):74–84, 1968.

Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. Proceedings of the IEEE, 79:1305–1320, 1991.

Undersubscribed threading on clustered cache architectures. In High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on, pages 678–689. IEEE, 2014.

Lance Hammond, Benedict Hubbert, Michael Siu, Manohar K Prabhu, Michael Chen, K Olukolun, et al. The stanford hydra cmp. Micro, IEEE, 20(2):71–84, 2000.

Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In ACM SIGARCH Computer Architecture News, volume 37, pages 152–163. ACM, 2009.

Jaehyuk Huh, Changkyu Kim, Hazim Shafi, Lixin Zhang, Doug Burger, and Stephen W Keckler. A nuca substrate for flexible cmp cache sharing. Parallel and Distributed Systems, IEEE Transactions on, 18:1028–1040, 2007.

Maurice Herlihy and J Eliot B Moss. Transactional memory: Architectural support for lock-free data structures, volume 21. ACM, 1993.

Jingcao Hu and Radu Marculescu. Energy-aware mapping for tile-based noc architectures under performance constraints. In Proceedings of the 2003 Asia and South Pacific Design Automation Conference, pages 233–239. ACM, 2003.

Jingcao Hu and Radu Marculescu. Communication and task scheduling of application-specific networks-on-chip. In Computers and Digital Techniques, IEE Proceedings-, volume 152, pages 643–651. IET, 2005.

Wen-reel Hwu and Yale N Patt. Hpsm, a high performance restricted data flow architecture having minimal functionality. In ACM SIGARCH Computer Architecture News, volume 14, pages 297–306. IEEE Computer Society Press, 1986.

Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor, volume 32. ACM, 1998.

Robert A Iannucci et al. Two fundamental issues in multiprocessing. Springer, 1988.

Robert A Iannucci, Guang R Gao, Robert H Halstead Jr, and Burton Smith. Multithreaded computer architecture: A summary of the state of the art, volume 281. Springer Science & Business Media, 1995.

Engin Ipek, Onur Mutlu, José F Martinez, and Rich Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In Computer Architecture, 2008. ISCA’08. 35th International Symposium on, pages 39–50. IEEE, 2008.

Open Source Initiative. Benchmark, 2015.

Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. Owl: cooperative thread array aware scheduling techniques for improving gpgpu performance. ACM SIGARCH Computer Architecture News, 41:395–406, 2013.

Pallavi Joshi, Chang-See Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. ACM Sigplan Notices, 44(6):110–120, 2009.

James Jeffers and James Reinders. Intel Xeon Phi coprocessor high-performance programming. Newnes, 2013.

Song Jiang and Xiaodong Zhang. Lirs: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. ACM SIGMETRICS Performance Evaluation Review, 30:31–42, 2002.

ACM Computing Surveys, Vol. X, No. X, Article X, Publication date: March 2016.
A Survey on Hardware and Software Support for Thread Level Parallelism

Ronny Krashinsky, Christopher Batten, Mark Hampton, Steve Gerding, Brian Pharris, Jared Casper, and Krite Asanovic. The vector-thread architecture. In ACM SIGARCH Computer Architecture News, volume 32, page 52. IEEE Computer Society, 2004.

Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using os observations to improve performance in multicore systems. pages 54–66, 2008.

Bhupesh Kothari and Mark Claypool. Pthreads performance. Technical report, Citeseer, 1999.

Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, pages 111–122. IEEE Computer Society, 2004.

Costas Kyriacou, Paraskevas Eviropidou, and Pedro Trancoso. Data-driven multithreading using conventional microprocessors. Parallel and Distributed Systems, IEEE Transactions on, 17(10):1176–1188, 2006.

Krishna M Kavi, Roberto Giorgi, and Joseph Arul. Scheduled dataflow: Execution paradigm, architecture, and performance evaluation. Computers, IEEE Transactions on, 50:834–846, 2001.

Eric Koskinen and Maurice Herlihy. Dreadlocks: efficient deadlock detection. In Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures, pages 297–303. ACM, 2008.

Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on, pages 1–12. IEEE, 2010.

Sanjeev Kumar, Christopher J Hughes, and Anthony Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In ACM SIGARCH Computer Architecture News, volume 35, pages 162–173. ACM, 2007.

John H Kelm, Daniel R Johnson, Matthew R Johnson, Neal C Crago, William Tuohy, Aqeel Mahesri, Steven S Lumetta, Matthew I Frank, and Sanjay J Patel. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. In ACM SIGARCH Computer Architecture News, volume 37, pages 140–151. ACM, 2009.

Shashi Kumar, Axel Jantsch, Juha-Pekka Soininen, Martti Forsell, Mikael Millberg, Johny Öberg, Kari Tienyryjä, and Ahmed Hemani. A network on chip architecture and design methodology. In VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on, pages 105–112. IEEE, 2002.

John Kim and Hanjoon Kim. Router microarchitecture and scalability of ring topology in on-chip networks. In Proceedings of the 2nd international workshop on network on chip architectures, pages 5–10. ACM, 2009.

David Koufaty, Dheeraj Reddy, and Scott Hahn. Bias scheduling in heterogeneous multi-core architectures. In Proceedings of the 5th European conference on Computer systems, pages 125–138. ACM, 2010.

Venkata Krishnan and Josep Torrellas. A chip-multiprocessor architecture with speculative multithreading. Computers, IEEE Transactions on, 48(9):866–880, 1999.

Rakesh Kumar, Dean M Tullsen, Parthasarathy Ranganathan, Norman P Jouppi, and Keith J Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In ACM SIGARCH Computer Architecture News, volume 32, page 64. IEEE Computer Society, 2004.

Jesus Labarta. Starss: A programming model for the multicore era. In PRACE WorkshopNew Languages & Future Technology Prototypes at the Leibniz Supercomputing Centre in Garching (Germany), 2010.

Tongping Liu, Charlie Curtisinger, and Emery D Berger. Dthreads: efficient deterministic multithreading. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pages 327–336. ACM, 2011.

Dongyang Lee, Peter M Chen, Jason Flinn, and Satish Narayanasamy. Chimera: hybrid program analysis for determinism. In ACM SIGPLAN Notices, volume 47, pages 463–474. ACM, 2012.

Jean-Pierre Lozi, Florian David, Gael Thomas, Julia L Lawall, Gilles Muller, et al. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In USENIX Annual Technical Conference, pages 65–76, 2012.

Edward A Lee. The problem with threads. Computer, 39:33–42, 2006.

Tong Li, Carla Schlatter Ellis, Alvin R Lebeck, and Daniel J Sorin. Pulse: A dynamic deadlock detection mechanism using speculative execution. In USENIX Annual Technical Conference, General Track, volume 44, 2005.

Jiang Lin, Qingda Lu, Xiaoming Ding, Zhao Zhang, Xiaodong Zhang, and P Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In High Per-
Enno Lübbers and Marco Platzner. Reconos: Multithreaded programming for reconfigurable computers. ACM Transactions on Embedded Computing Systems (TECS), 9(1):8, 2009.

Shan Lu, Soyeon Park, Eunseo Seo, and Yuanjuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In ACM Sigplan Notices, volume 43, pages 329–339. ACM, 2008.

Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. Posh: a tls compiler that exploits program structure. In Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 158–167. ACM, 2006.

Charles E McDowell and David P Helmbold. Debugging concurrent programs. ACM Computing Surveys (CSUR), 21(4):593–622, 1989.

Nimrod Megiddo and Dharmendra S Modha. Arc: A self-tuning, low overhead replacement cache. In FAST, volume 3, pages 115–130, 2003.

Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, pages 146–160. IEEE Computer Society, 2007.

Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In ACM SIGARCH Computer Architecture News, volume 36, pages 63–74. IEEE Computer Society, 2008.

Théodore Marescaux, Vincent Nollet, J-Y Mingolet, Andrei Bartic, Will Moffat, Prabhat Avasare, Paul Ceone, Diederik Verkest, Serge Vernalde, and Rudy Lauwereins. Run-time support for heterogeneous multitasking on reconfigurable socs. Integration, the VLSI journal, 38(1):107–130, 2004.

J Paul Morrison. Flow-Based Programming: A new approach to application development. CreateSpace, 2010.

TR Maeurer and D Shippy. Introduction to the cell multiprocessor. IBM journal of Research and Development, 49(4):589–604, 2005.

Suman Mamidi, Michael J Schulte, Daniel Iancu, and John Glossner. Architecture support for reconfigurable multithreaded processors in programmable communication systems. In Application-specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on, pages 320–327. IEEE, 2007.

Brian D Marsh, Michael L Scott, Thomas J LeBlanc, and Evangelos P Markatos. First-class user-level threads. In ACM SIGOPS Operating Systems Review, volume 25, pages 110–121. ACM, 1991.

Timothy G Mattson, Beverly A Sanders, and Berna L Massingill. Patterns for parallel programming. Pearson Education, 2004.

José P Martinez and Josep Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In ACM SIGOPS Operating Systems Review, volume 36, pages 18–29. ACM, 2002.

Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, pages 248–259. ACM, 2011.

Zainalabedin Navabi. VHDL: Analysis and modeling of digital systems. McGraw-Hill, Inc., 1997.

Bradford Nichols, Dick Buttlar, and Jacqueline Farrell. Pthreads programming: A POSIX standard for better multiprocessing. " O’Reilly Media, Inc.", 1996.

Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. Exploring the potential of heterogeneous von neumann/dataflow execution models. In Proceedings of the 42nd Annual International Symposium on Computer Architecture, pages 298–310. ACM, 2015.

Robert HB Netzer and Barton P Miller. What are race conditions?: Some issues and formalizations. ACM Letters on Programming Languages and Systems (LOPLAS), 1(1):74–88, 1992.

Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In Proceedings of the 31st International Conference on Software Engineering, pages 386–396. IEEE Computer Society, 2009.

Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In ACM Sigplan notices, volume 42, pages 89–100. ACM, 2007.

Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. ACM Sigplan Notices, 44(3):97–108, 2009.

John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. Proceedings of the IEEE, 96(5):879–899, 2008.
A Survey on Hardware and Software Support for Thread Level Parallelism

Umit Y Ogras, Jingcao Hu, and Radu Marculescu. Key research problems in noc design: a holistic perspective. In Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, pages 69–74. ACM, 2005.

Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. In Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture, pages 105–118. IEEE Computer Society, 2005.

G. Ottoni. Global instruction scheduling for multi-threaded architectures. PhD thesis, PhD thesis, 2008.

Wesley Peck, Erik Anderson, Jason Agron, Jim Stevens, Fabrice Baijot, and David Andrews. Hthreads: A computational model for reconfigurable devices. In Field Programmable Logic and Applications, 2006. FPL’06. International Conference on, pages 1–4. IEEE, 2006.

Peter Pacheco. An introduction to parallel programming. Elsevier, 2011.

Dhabaleswar K Panda. Mvapich: Mpi over infiniband, 10gige/iwarp and roce, 2014.

Peter Poplavko, Twan Basten, Marco Bekooij, Jef van Meerbergen, and Bart Mesman. Task-level timing models for guaranteed performance in multiprocessor networks-on-chip. In Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems, pages 63–72. ACM, 2003.

Gregory M Papadopoulos and David E. Culler. Monsoon: an explicit token-store architecture. In ACM SIGARCH Computer Architecture News, volume 18, pages 82–91. ACM, 1990.

Sujay Parekh, Susan Eggers, Henry Levy, and Jack Lo. Thread-sensitive scheduling for smt processors, 2000.

Moinuddin K Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steelly, and Joel Emer. Adaptive insertion policies for high performance caching. In ACM SIGARCH Computer Architecture News, volume 35, pages 381–391. ACM, 2007.

Moinuddin K Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, pages 425–432. IEEE Computer Society, 2006.

Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling, volume 28. ACM, 2000.

James Reinders. VTune performance analyzer essentials. Intel Press, 2005.

James Reinders. Intel threading building blocks: outfitting C++ for multi-core processor parallelism. " O'Reilly Media, Inc." , 2007.

Joshua Redstone, Susan Eggers, and Henry Levy. Mini-threads: Increasing tlp on small-scale smt processors. In High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on, pages 19–30. IEEE, 2003.

JE Rodrigues and Jorge E Rodriguez Bezos. A graph model for parallel computations. Technical report, Massachusetts Institute of Technology, 1969.

Amir Roth and Gurindar S. Sohi. Speculative data-driven multithreading. In High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on, pages 37–48. IEEE, 2001.

Krishna K Rangan, Gu-Yeon Wei, and David Brooks. Thread motion: fine-grained power management for multi-core systems. In ACM SIGARCH Computer Architecture News, volume 37, pages 302–313. ACM, 2009.

Sander Stuijk, Twan Basten, Marc Geilen, Amir Hossein Ghamarian, and Bart Theelen. Resource-efficient routing and scheduling of time-constrained streaming communication on networks-on-chip. Journal of Systems Architecture, 54:411–426, 2008.

Daniel Spoonhower, Guy E Blelloch, Phillip B Gibbons, and Robert Harper. Beyond nested parallelism: tight bounds on work-stealing overheads for parallel futures. In Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures, pages 91–100. ACM, 2009.

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems (TOCS), 19(4):391–411, 1997.

Gurindar S. Sohi, Scott E Breach, and TN Vijaykumar. Multiscalar processors. In ACM SIGARCH Computer Architecture News, volume 23, pages 414–425. ACM, 1995.

John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. Computing in science & engineering, 12(1-3):66–73, 2010.

Dongkun Shin and Jihong Kim. Power-aware communication optimization for networks-on-chips with voltage scalable links. In Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, pages 170–175. ACM, 2004.

ACM Computing Surveys, Vol. X, No. X, Article X, Publication date: March 2016.
Erik Saule, Kamer Kaya, and Ümit V Çatalyürek. Performance evaluation of sparse matrix multiplication kernels on intel xeon phi. In *Parallel Processing and Applied Mathematics*, pages 559–570. Springer, 2014.

Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. Eelru: simple and effective adaptive page replacement. In *ACM SIGMETRICS Performance Evaluation Review*, volume 27, pages 122–133. ACM, 1999.

J Gregory Steffan and Todd C Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *High-Performance Computer Architecture, 1998. Proceedings., 1998 Fourth International Symposium on*, pages 2–13. IEEE, 1998.

Burton J Smith. Architecture and applications of the hep multiprocessor computer system. In *25th Annual Technical Symposium*, pages 241–248. International Society for Optics and Photonics, 1982.

M Aater Suleman, Onur Mutlu, Moinuddin K Qureshi, and Yale N Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 253–264. ACM, 2009.

Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. Wavescalar. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 291. IEEE Computer Society, 2003.

Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaebyuk Huh, Doug Burger, Stephen W Keckler, and Charles R Moore. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 422–433. IEEE, 2003.

Juan Carlos Saex, Manuel Prieto, Alexandra Fedorova, and Sergey Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *Proceedings of the 5th European conference on Computer systems*, pages 139–152. ACM, 2010.

M Aater Suleman, Yale N Patt, Eric Sprangle, Anwar Rohillah, Anwar Ghuloum, and Doug Carmean. Asymmetric chip multiprocessors: Balancing hardware efficiency and programmer efficiency. *Univ. Texas, Austin, TR-HPS-2007-001*, 2007.

M Aater Suleman, Moinuddin K Qureshi, and Yale N Patt. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmps. *ACM Sigplan Notices*, 43(3):277–286, 2008.

Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM (JACM)*, 53(3):379–405, 2006.

Anthony Skjellum, Steven G Smith, Nathan E Doss, Alvin P Leung, and Manfred Morari. The design and evolution of zipcode. *Parallel Computing*, 20:565–596, 1994.

Allan Snavely and Dean M Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. *ACM SIGPLAN Notices*, 35(11):234–244, 2000.

Mihai Sima, Stamatis Vassiliadis, Sorin Cotofana, Jos TJ van Eijndhoven, and Kees Vissers. Field-programmable custom computing machines-a taxonomy. In *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, pages 79–88. Springer, 2002.

Daniel Sanchez, Richard M Yoo, and Christos Kozyrakis. Flexible architectural support for fine-grain scheduling. In *ACM Sigplan Notices*, volume 45, pages 311–322. ACM, 2010.

Joshua Suettlerlein, Stéphane Zuckerman, and Guang R Gao. An implementation of the codelet model. In *Euro-Par 2013 Parallel Processing*, pages 633–644. Springer, 2013.

Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ISCA ’95, pages 392–403, New York, NY, USA, 1995. ACM.

Chen Tian, Min Feng, Vijay Nagarajan, and Rajiv Gupta. Copy or discard execution model for speculative parallelization on multicore. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 330–341. IEEE Computer Society, 2008.

Kevin Bryan Theobald. *EARTH: AN EFFICIENT ARCHITECTURE: FOR RUNNING THREADS*. PhD thesis, McGill University, Montréal Québec, Canada, 1999.

William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction*, pages 179–196. Springer, 2002.

Hung-Wei Tseng and Dean M Tullsen. Data-triggered threads: Eliminating redundant computation. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 181–192. IEEE, 2011.

Hung-Wei Tseng and Dean Michael Tullsen. Software data-triggered threads. *ACM SIGPLAN Notices*, 47(10):703–716, 2012.
A Survey on Hardware and Software Support for Thread Level Parallelism

Hung-Wei Tseng and Dean M Tullsen. Cdtt: Compiler-generated data-triggered threads. In High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on, pages 650–661. IEEE, 2014.

Aniruddha N Udipi, Naveen Muralimanohar, and Rajeev Balasubramonian. Towards scalable, energy-efficient, bus-based on-chip networks. In High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on, pages 1–12. IEEE, 2010.

Sascha Uhrig, Stefan Maier, Georgi Kuzmanov, and Theo Ungere. Coupling of a reconfigurable architecture and a multithreaded processor core with integrated real-time scheduling. In Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, pages 4–pp. IEEE, 2006.

Theo Ungere, Borut Robič, and Jurij Šilc. A survey of processors with explicit multithreading. ACM Computing Surveys (CSUR), 35(1):29–63, 2003.

Rob Von Behren, Jeremy Condit, Feng Zhou, George C Necula, and Eric Brewer. Capriccio: scalable threads for internet services. In ACM SIGOPS Operating Systems Review, volume 37, pages 268–281. ACM, 2003.

OCR working group. The open community runtime interface, version 1.0.1, september, 2015, 2015.

Kehuai Wu, Andreas Kanstein, Jan Madsen, and Mladen Berekovic. Mt-address: Multithreading on coarse-grained reconfigurable architecture. In Reconfigurable Computing: Architectures, Tools and Applications, pages 26–38. Springer, 2007.

Vincent M Weaver, Dan Terpstra, and Steven Moore. Non-determinism and overcount on modern hardware performance counter implementations. In Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on, pages 215–224. IEEE, 2013.

IARR Xilinx. Microblaze processor reference guide. reference manual, 23, 2006.

Fahimeh Yazdanpanah, Carlos Alvarez-Martinez, Daniel Jimenez-Gonzalez, and Yoav Etsion. Hybrid dataflow/von-neumann architectures. Parallel and Distributed Systems, IEEE Transactions on, 25(6):1489–1509, 2014.

Richard M Yoo, Christopher J Hughes, Changkyu Kim, Yen-Kuang Chen, and Christos Kozyrakis. Locality-aware task management for unstructured parallelism: A quantitative limit study. In Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures, pages 315–325. ACM, 2013.

Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In ACM SIGOPS Operating Systems Review, volume 39, pages 221–234. ACM, 2005.

Michael Zhang and Krste Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In ACM SIGARCH Computer Architecture News, volume 33, pages 336–345. IEEE Computer Society, 2005.

Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In ACM SIGARCH Computer Architecture News, volume 38, pages 129–142. ACM, 2010.

Antonia Zhai, Christopher B Colohan, J Gregory Steffan, and Todd C Mowry. Compiler optimization of scalar value communication between speculative threads. ACM SIGPLAN Notices, 37:171–183, 2002.

Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In Proceedings of the 4th ACM European conference on Computer systems, pages 89–102. ACM, 2009.

Dali Zhao, Houman Homayoun, and Alexander V Veidenbaum. Temperature aware thread migration in 3d architecture with stacked dram. In Quality Electronic Design (ISQED), 2013 14th International Symposium on, pages 80–87. IEEE, 2013.

Fabian Zabatta and Kevin Ying. A thread performance comparison: Windows nt and solaris on a symmetric multiprocessor. In Proceedings of the 2nd USENIX Windows NT Symposium, 1998.