Hardware-Modulated Parallelism in Chip Multiprocessors

Julia Chen, Philo Juang, Kevin Ko,
Gilberto Contreras, David Penry, Ram Rangan, Adam Stoler,
Li-Shiuan Peh and Margaret Martonosi
Departments of Computer Science and Electrical Engineering
Princeton University
Email:*peh@princeton.edu*

Abstract

Chip multi-processors (CMPs) already have widespread commercial availability, and technology roadmaps project enough on-chip transistors to replicate tens or hundreds of current processor cores. How will we express parallelism, partition applications, and schedule/place/migrate threads on these highly-parallel CMPs?

This paper presents and evaluates a new approach to highly-parallel CMPs, advocating a new hardware-software contract. The software layer is encouraged to expose large amounts of multi-granular, heterogeneous parallelism. The hardware, meanwhile, is designed to offer low-overhead, low-area support for orchestrating and modulating this parallelism on CMPs at run-time. Specifically, our proposed CMP architecture consists of architectural and ISA support targeting thread creation, scheduling and context-switching, designed to facilitate effective hardware run-time mapping of threads to cores at low overheads.

Dynamic modulation of parallelism provides the ability to respond to run-time variability that arises from dataset changes, memory system effects and power spikes and lulls, to name a few. It also naturally provides a long-term CMP platform with performance portability and tolerance to frequency and reliability variations across multiple CMP generations. Our simulations of a range of applications possessing do-all, streaming and recursive parallellism show speedups of 4-11.5X and energy-delay-product savings of 3.8X, on average, on a 16-core vs. a 1-core system. This is achieved with modest amounts of hardware support that allows for low overheads in thread creation, scheduling and context-switching. In particular, our simulations motivated the need for hardware support, showing that the large thread management overheads of current run-time software systems can lead to up to 6.5X slowdown. The difficulties faced in static scheduling were shown in our simulations with a static scheduling algorithm, fed with oracle profiled inputs suffering up to 107% slowdown compared to NDP's hardware scheduler, due to its inability to handle memory system variabilities. More broadly, we feel that the ideas presented here show promise for scaling to the systems expected in ten years, where the advantages of high transistor counts may be dampened by difficulties in circuit variations and reliability. These issues will make dynamic scheduling and adaptation mandatory; our proposals represent a first step towards that direction.

In Proceedings of Workshop on Design, Architecture and Simulation of Chip Multi-Processors Conference (dasCMP), held in conjunction with the 38th Annual International Symposium on Microarchitecture (MICRO), Barcelona, Spain, November 2005.

1. Introduction

Chip multiprocessors with modest numbers of cores are commonplace in product lines today, and technology trends will allow CMP designs with tens or hundreds of processor cores in the future. From a hardware perspective, multi-core designs are very appealing because replicated cores allow very high-performance and high-transistor-count chips to be built with manageable complexity and power-efficiency.

From a software perspective, however, problems of partitioning and mapping applications onto these multiple cores remain fraught with difficulty. On the one hand, traditional von Neuman parallelism (identified either by the programmer or the compiler) is often insufficient to effectively use the available cores on-chip. On the other hand, dataflow parallelism has frequently suffered from being too fine-grained, thus swamping hardware resources with many tasks that are difficult to schedule appropriately onto the cores and requiring programmers to fundamentally change the way they code.

Thus, the overall problem involves two parts: (1) partitioning a problem into parallel chunks of work, and (2) mapping and scheduling these chunks onto the underlying hardware. Currently, both phases are carried out in tandem—most applications are parallelized and mapped manually, hand-tuned specifically for a targeted underlying hardware platform. This requires programmers to not only uncover legal parallelism, and partition the application correctly, but also to load-balance application threads across the cores to ensure high performance. Moving to a CMP with a different configuration of cores often requires re-partitioning and mapping/scheduling of an application. Automatic compiler-driven partitioning and mapping of sequential programs is an appealing solution, but is difficult in its current form. In particular, run-time factors such as input data variability and long-latency memory stalls continue to make compile-time partitioning and mapping highly challenging.

In this paper, we argue instead for a new hardware-software contract or execution model based on *potential* parallelism. For (1), programmers or compilers aggressively partition applications into as many threads as possible, with the goal of exposing all potential parallelism rather than partitioning the program to run on a specific architecture. This philosophy draws somewhat from dataflow approaches. For (2), we propose and evaluate modest amounts of *run-time hardware support* for this style of parallelism. In particular, we investigate support to modulate the level of parallelism through dynamic scheduling and mapping of threads to cores and by leveraging program information exposed through the ISA. Such a hardware-software

contract hides the underlying hardware intricacies from the software, allowing programmers (and compilers) to partition a program only once for execution on multiple core configurations. Likewise, it empowers hardware to perform a range of dynamic adaptations that improve the power and performance efficiency of the underlying platform.

A key aspect of the system we propose and evaluate here hinges on the fact that efficiently supporting aggressive, heterogeneous (multi-granular) parallelism requires that thread creation, context-switching and scheduling be supported at very low overheads. Currently, support for parallelism is handled in *software*, either through user-level threads packages (Cilk [1], Pthreads [2]) or the operating system. User level thread management has low overheads but cannot leverage parallel hardware without going through OS thread management, since user-level threads belong to a single process. OS thread management, on the other hand, comes with significant overheads for thread creation, scheduling and context-switching. Just as an example, on a machine with an XScale processor running Linux, we measured POSIX-compliant thread creation times to be roughly 150K cycles and thread scheduling latencies of roughly 20K cycles. These overheads are aggravated by memory access delays, as the thread status data structures are, due to their infrequent access, not likely to be cached. Using hardware performance counters on the XScale measurement platform, we found that roughly 75% of the above cycles were spent accessing the memory for scheduling tables.

With these measurements and observations, it is clear that software thread management imposes significant costs, making it unsuitable for handling our proposed software model which encourages the aggressive exposing of parallelism of varying granularities. In this paper, we thus propose a CMP architecture that includes a suite of *hardware* microarchitectural mechanisms that significantly lower thread management overheads:

- Thread creation: Storing thread information entirely in hardware, in dedicated scratchpad memories at each core, thus enabling fast book-keeping;
- Thread scheduling: Performing scheduling in hardware, and placing these hardware structures at the network level to minimize the overheads of accessing scheduling information and placing threads remotely. ISA extensions allow program characteristics to be exposed to the hardware schedulers.
- Thread context-switching: Pre-fetching thread contexts, instructions and data initiated by the hardware scheduler;

The rest of this paper elaborates and evaluates the proposed architecture, the Network-Driven Processor (NDP). Section 2 presents the overall architecture, software model and hardware-software contract of NDP. Section 3 discusses each aspect of the architecture in detail, explaining our rationale for the various microarchitectural design decisions. Section 4 presents our simulation results on a variety of benchmarks exhibiting fairly diverse types of parallelism. Section 5 compares NDP with prior related work, and Section 6 wraps up the paper.

2. NDP architecture overview

The key driving goal behind NDP is the support for a new hardware-software contract where applications are aggressively partitioned in order to expose and exploit all potential parallelism. In particular, we view NDP as offering a hardware-software dividing line below which hardware dynamically modulates application parallelism. Above this line, we encourage a software model which exposes large amounts of multi-granular, heterogeneous parallelism. Such parallelism can be achieved via a range of techniques, including: traditional do-all parallelism, traditional and decoupled software pipelined parallelism [3], producer-consumer streaming parallelism as seen in multimedia codes [4, 5], transactional parallelism [6], and other techniques including speculative parallelism. While our current studies use hand-ported applications coded on sequential C, automated compilation and concurrent languages to ease exposing parallelism are among our longer-term future work plans.

The remainder of this section offers details on the hardware and software support we envision. Detailed explorations of the architectural issues faced in each component's design are further discussed in Section 3.

2.1 NDP software model

NDP provides API and corresponding ISA instructions which allow the software to create threads and expose thread characteristics to the hardware, while abstracting from the programmer/compiler the actual physical mapping and temporal scheduling of the threads onto cores.

Thread creation is supported directly by hardware, through the *thread_spawn* call in the API and its corresponding ISA instruction. To efficiently support aggressive do-all and streaming parallelism, NDP supports run-time thread cloning. A single *thread_spawn* can be dynamically parallelized into multiple threads at run-time. For do-all parallelism, this allows a programmer to spawn a single thread for all the iterations, while the hardware dynamically clones additional threads when it decides to further parallelize the program. This supports aggressive do-all parallelism without unnecessary overheads. When a do-all is running on a single core, it will be as efficient as single-thread performance, since the hardware will simply iterate repeatedly over the thread, rather than incur thread creation and context-switching delays as if each iteration were exposed as a distinct thread. This will also reduce the hardware per-thread overhead.

For streaming parallelism, the cloning support allows the programmer to create a single thread for each producer/consumer stage of the streaming application, rather than aggressively creating a thread for each streaming data item. In addition to saving on hardware resources and cutting down on threading overheads, this allows streaming threads to expose a long-lived producer-consumer data flow to the hardware, which can then be monitored and used to guide the run-time cloning of producer and consumer threads in order to balance the produce-consume rate and arrive at the ideal parallelization for streaming applications.

In addition to support for threading, a critical component of NDP's software model is the exposing of data flow between threads through queues. Queues expose inter-thread communication, allowing the hardware to appropriately provision network bandwidth. They also convey critical thread dependencies that enable the scheduler to track the critical path of a program and the data required to guide the hardware in prefetching prior to thread execution. This notion of logical queues is encapsulated in the software model as queue handles, or *queue IDs*. The hardware maintains the binding or connecting of queues to threads, as well as the current location of threads. For clonable threads, the hardware also transparently manage the splitting and merging of queues.

While our current design focuses on homogeneous tiles, future heterogeneous NDP systems might have broader sets of thread annotations in the API and ISA, such as a thread's need

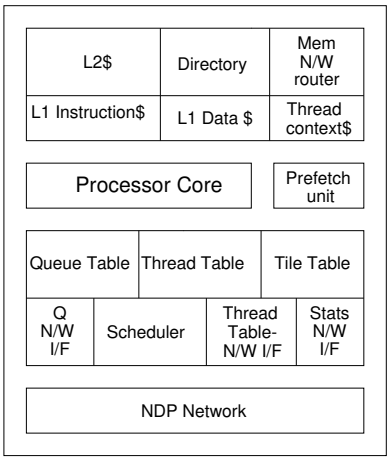


Figure 1. Sketch of NDP tile architecture. Hardware elements are not drawn to scale. We estimate that non-processor NDP hardware to represent roughly 10% or less of the total core area.

for specialized hardware accelerators, or real-time execution deadlines.

2.2 NDP hardware architecture

Field	Description
Thread Table	
ThreadID	Per-process thread identifier across the chip
Program counter (PC)	Program counter address for this thread
Stack pointer (SP)	Stack pointer for this thread
State	Thread status {invalid, blocked, active, ready}
Priority	Current thread scheduling priority
Flow Queues	Points to queues in the Queue Table
Cloneability	Whether or not the thread can clone itself
Queue Table	
Software QueueID	Per-process queue identifier
Hardware QueueID	Physical queue identifier
Producer ThreadID	ID of thread which produces into the queue
Producer TileID	Tile which currently houses the producer thread
Consumer ThreadID	ID of thread which consumes from the queue
Consumer TileID	Tile which currently houses the consumer thread
Tile Table	
TileID	Per-chip tile identifier
Load	Latest-known load situation

Table 1. Entries of the Thread, Queue and Tile Tables on each NDP tile.

Figure 1 sketches the proposed architecture for each tile in the NDP chip-multiprocessor[1]. In the middle of the figure, the network interface comprises much of the hardware support for run-time thread creation, context-switching and scheduling. In particular, NDP has dedicated hardware memories or scratchpads that store information of threads, queues and tiles. Table 1 depicts the specific fields in these tables. In particular, the *Thread Table* stores thread information that facilitates their execution, much like its counterpart *process* structure in the OS. Upon a non-blocking *thread_spawn* call, an entry is created in the Thread Table, and marked as ready for scheduling, setting its PC and SP appropriately. Every spawned thread will first have an entry in the *Thread Table* of the core at which it is spawned (i.e. where its parent thread is executed). The *Queue Table* keeps track of the exposed producer-consumer data flow queues between threads, interfacing with the network to effect communication when producer and consumer threads are mapped onto different cores. When *queue_create* is called, it creates an entry in the tile's Queue Table, setting up a uni-directional hardware queue. The producer and consumer threads of a queue are bound at spawn time for parent-child communications, or at the first *queue_send* and *queue_recv*. These bindings are maintained automatically by the hardware, which also tracks the locations of producer/consumer threads. When the producer and consumer of a queue is on the same tile, the hardware queue essentially functions like a register; when the receiver is remote, the queue interface logic maps the queue to the network, so its contents are automatically packetized, injected into the network and sent towards the tile on which the receiver thread is housed. The third hardware table, *Tile Table* keeps track of the load situation on all tiles across the chip, and is directly maintained by the network, to aid the scheduler in load balancing.

All the above structures are used by the *Scheduler* in determining when to map threads onto which remote cores, as well as when to run them locally. The Scheduler is thus embedded within the network interface where the hardware tables are housed to ensure fast, efficient updates. Clearly, there are a rich set of scheduling algorithms, with myriad design tradeoffs; we propose and evaluate simple policies in this paper that consume little hardware resources, but more elaborate schemes may be fruitful in managing complex interactions of performance, power, and reliability goals.

Both the Thread and Queue Tables are sized so they can be accessed within a cycle. Upon an overflow, entries are evicted by the hardware to off-chip DRAM. When an off-chip entry is accessed, say upon the arrival of a data item for a queue, it is swapped back in. The eviction policy is tied with the scheduling algorithm – lower priority thread entries are evicted over higher priority ones, along with their associated queue entries.

At the bottom of the figure, the NDP network (per-tile router) is architected to efficiently support the needs of hardware thread scheduling. Specifically, it provides fast access to thread scheduling information such as the Tile Table which is updated directly by the network through its statistics protocol. Producer-consumer rate information is also available, providing 'queue transparency' via virtual circuits. This means that schedulers can infer the rate of consumption at the remote end of a queue by looking at the output buffer of the queue's local producing end, without incurring any communication latency. The network also supports fast placement of threads through high-priority scout messages, which effect the moving of thread and queue table entries from the original tile to the destination tile and seamlessly manage communication through queues as threads move.

At the top of Figure 1 is a nearly-standard processor core and directory-based coherent cache memory system, with one primary exception – the NDP Scheduler is responsible for initiating execution on the processor core. We chose to support a shared memory space so as to ease application partitioning.

[1]While the tile counts are flexible, we are targeting 16 tiles in 50nm technology, and envision up to 100 tiles per chip. Furthermore, while we assume identical, homogeneous tiles in this paper for simplicity, the architecture remains unchanged for heterogeneous processor cores. All that would change is that the scheduler would need to account for varying resources per core when mapping threads onto cores.

3. Architectural Support for Parallelism and Scheduling

3.1 Support for Fast Thread Creation

In order to support exposing aggressive amounts of parallelism of heterogeneous granularity, thread creation delay has to be lowered, as this is on the critical path of each thread. High thread creation overheads have severe impact on the performance of fine-grained threads.

In NDP, through our hardware thread tables at each tile, spawning of new threads can be done efficiently. A thread spawn requires only the creation of a new entry in the tile's Thread Table and the allocation of space on the stack. The former can be performed within a cycle when an entry is available in the Thread Table, blocking only when the table is full and an entry needs to be evicted. Then the scheduler will decide which thread entry to swap out to memory.

While it is possible to provide hardware support for the management of stacks to minimize thread creation delay, we chose to keep stack allocation under the management of the OS, which can amortize the cost of a stack allocation across multiple threads, and lower it to approximately 30 cycles. This is done by initially allocating a large segment of memory and partitioning them into fixed-size thread stacks as necessary. Since stacks are fairly large in size (ranging from 1KB to 2MB in Linux), using hardware-managed scratchpad memories for their storage leads to rather high area overheads.

Also, on a *thread_spawn*, the parent thread creates a structure with the input arguments to the child thread, so the input arguments are contiguous in memory. A pointer to the input arguments is passed onto the newly spawned thread's stack. The scheduler can then access this pointer and begin prefetching from memory the arguments needed by the thread in advance, prior to launching its execution. This features benefits both fast thread creation and fast thread context switching.

We estimate that 50 cycles are needed for a spawn; threads in the thread table are implicitly ordered and can be quickly accessed, and thus the scheduling delay is on the order of 10 cycles. In an operating system, simply accessing the process table can cause the processor to incur cache misses and therefore induce a significant delay; executing the scheduling code to choose the new thread further increases the delay. Moreover, the NDP schedulers can be executing concurrently with an application on the same tile; much of the delay can thus be hidden, unlike OS or user-level scheduling code, which must uproot a running thread in order to execute.

3.2 Support for Fast Thread Scheduling

Thread scheduling involves three phases: first, accessing thread and tile information used by the scheduler, second, making the decision as to which thread to execute, and third, effecting scheduling decisions in moving threads onto remote cores. NDP provides hardware support to reduce the overheads of each of the above phases. More specifically, NDP proposes embedding such hardware support within the network, with the network designed from the ground up to lower the delays of each of the above phases. First, the network is architected for fast *access* of scheduling information such as tile load. Second, the scheduler ASIC which makes the *decisions* is implemented within the network interface, so it can very quickly access scheduling information maintained and directly updated by the network. In addition, once the scheduler decides to move a thread, the network is responsible for effecting the placement, with fast table updates and seamless queue management.

3.2.1 Fast access to tile table and queue rate

By embedding accessing and updating of the Tile Table within the network hardware, with routers directly maintaining the Tile Tables, the communication overhead involved can be reduced to the minimum, providing the scheduler with fast, up-to-date information without occupying processor resources.

Several alternative protocols with different trade-offs in recency vs. communication overhead were considered. At one extreme, the *snooping* protocol observes the movement of passer-by threads, recording thread destinations, thus updating the Tile Table without additional traffic. When there is substantial thread movement across the chip, snooping will be able to gather a fairly complete and accurate picture. Another alternative is the *token ring* protocol, where tiles are logically viewed as a ring. Periodically, each tile sends a token containing its load to its right immediate neighbor in the ring, so a tile always has a view of all tiles that is, at most, (N-1)-hops-old, where N is the number of tiles, trading off scalability for known delay and low communication overhead. A third option, the one we chose in NDP, is *neighbor broadcasting*. Whenever the tile workload changes passes a threshold level, it informs its four neighbors of its load situation, piggybacking on the same message the load information of its own neighbors as well, so that each message contains load information of five tiles. While this incurs a higher communication overhead, it delivers a fairly up-to-date view of the Tile Tables and scales well to a large number of cores.

By exposing queues in the ISA, NDP allows the scheduler to monitor the producing/consuming rate of the queues and balance the producer and consumer threads appropriately. The NDP network is architected for queue transparency, i.e. the local status of one end of a queue accurately reflects its status on the remote end. In other words, when an outgoing queue from tile A to B fills up, it can be inferred that tile B is not reading the queue fast enough, and thus there is an imbalance between the producer on tile A and its consumer on tile B. Without queue transparency, a local queue can fill up due to network contention from unrelated threads, so producer-consumer thread relationships can only be tracked through explicit messages to the remote tile, incurring costly round-trip communication delays.

Queue transparency is realized through virtual circuits—pre-reservation of bandwidth at every router between the sender and receiver. This pre-reservation is done with a scout packet which also installs thread and queue table entries at the remote node and initiates prefetches (see Section 3.2.3). At each router hop, the scout computes the next hop along the route and arbitrates for a virtual channel [7]. Once a virtual channel is obtained, the requested bandwidth is factored as a weight guiding the switch allocator to ensure that the virtual channel gets its allocated physical channel bandwidth. If there is insufficient bandwidth, the circuit is rejected through a return-to-sender nack. The reservation table stores this weight, as well as the input and output virtual channel mapping for the queue, so subsequent packets need not go through routing, virtual channel or switch allocation. Pre-reservation of bandwidth not only enables fast access to queue information, but also minimizes thread communication delay since routes are set up ahead of time. Reducing thread communication overhead is critical in our quest to support heterogeneous threads of variable granularities. With the ever-increasing wiring resources available on-chip, we feel that the potential bandwidth inefficiency due to pre-reservation can be mitigated.

3.2.2 Fast scheduler decision logic

We considered several alternatives in the implementation of the scheduler decision logic. First, it can remain in the OS, running as a thread in each core, accessing the hardware thread, queue, and tile tables. This requires context-switching to this kernel thread for thread scheduling, occupying processor resources during scheduling and incurring context-switching delays. Alternatively, the scheduler software thread can run on a separate hardware co-processor housed within the network interface. This allows for fast access to scheduling information without interfering with actual computation. We chose to go with a purely hardware ASIC implementation of the NDP scheduler in our aggressive push towards lower thread scheduling delays. Our simple scheduling policies result in manageable hardware area and energy overheads.

3.2.3 Fast remote thread placement

The execution of a thread on a remote tile involves first setting up the thread table entry on the remote tile, then maintaining and prefetching the thread's queues so communications occur seamlessly and efficiently. NDP lowers remote thread placement delay by architecting the network to support these operations explicitly.

When the thread is placed on a remote tile (or pulled from a remote tile), a network scout packet containing the thread and queue table entries is sent to the specific tile. Upon receipt of a scout packet, the network interface installs these entries into the local tables.

The network is also responsible for the seamless management of queues as threads move and clone. During the transit of threads, the queues at the source tile are frozen until the table entries have been set up and the queue contents forwarded. For clonable threads, the queue network interface logic is responsible for mapping software to hardware queues as well, maintaining the merging and splitting of queues. When a thread is cloned, new hardware queues are created and mapped to the original software queue IDs, and a confirmation message is sent back to the producer tile, informing it of these new hardware queue ids.

A possible concern is that cloning may destroy the ordering of data within queues, when doling out data in a round-robin fashion to different tiles. This is not an issue as there will be separate hardware queues from the producer to each of the cloned consumers, and separate hardware queues from each of these consumers to a later stage, and data items are tagged at the producer.

3.2.4 NDP Scheduler

In NDP, every tile has a lightweight, hardware scheduler which keeps track of threads spawned in the Thread Table and selects threads for execution on the local core in order of their priorities. By exposing thread characteristics through the hardware-software contract, NDP allows for myriad thread scheduling policies based on the rich information tracked efficiently by NDP's hardware mechanisms. In this paper, we evaluate a simple distributed, random work-stealing scheduler fashioned after that used in the Cilk's runtime software system [1]. It chooses another tile randomly from its Tile Table and sends a request to steal across the network to that tile. If that tile has a thread that is ready or blocked, it sends a confirmation, along with the information required to run that thread (Thread Table entry, associated Queue Table entries, state context). If it does not have a thread to spare, it sends a negative confirmation and the idle tile chooses another tile to steal from. The implementation of such a simple scheduler clearly has very low hardware overheads. Despite its simplicity, our simulation over a suite of applications with diverse parallelism profiles show that NDP is able to dynamically modulate the parallelism and realize significant speedups and power savings.

3.3 Support for Fast Thread Context Switching

With a large number of threads, the frequency of context switching becomes correspondingly higher. We therefore need hardware support for low-latency context switching between threads. A large portion of the context switching delay is involved in swapping in and out each thread's machine state, which can consist of hundreds of registers.

Our solution is to prefetch the machine state and the input arguments of the desired next thread. Since the NDP hardware scheduler determines which thread to launch onto the processor core based on priorities, it can initiate a hardware prefetch of the machine state and input arguments of the highest-priority thread into the regular cache. This avoids adding the latency of warming up cold caches to the context-switching delays, as the prefetcher ensures that one or more threads have what they need to execute ready in the cache. With a hardware prefetcher, normal program execution will not be interrupted, though conflicts in the cache may hurt performance. Another alternative to keeping threads' hardware register contexts in the regular cache is to store them in local scratchpad memory to avert cache conflicts. An even more aggressive alternative is to have hardware register contexts within the local processor core so active threads need not suffer a pipeline flush when swapped out [8]. While we considered these alternatives, we chose to simply prefetch machine state into the regular cache.

Our design choice is heavily guided by the software model NDP is built to support—aggressive, heterogeneous threading, which renders techniques such as hardware register contexts unsuitable as they constrain the number of threads. Our approach allows for an unlimited number of thread contexts in the memory hierarchy; so long as the prefetcher can effectively predict and fully fetch the state of a ready thread before it is needed, a low-latency context switch can be performed on any number of threads. Moreover, machine state of blocked threads can be forwarded to or a prefetch initiated by the new core.

4. Simulation Results

4.1 Simulator setup

Cores	simple in-order
L1 Data/Instr Cache	32KB each: 4-way associative, 32-byte block size, write-back + write-allocate, 1 cycle hit
L2 Data Cache (shared, directory)	4MB: 16-way associative, 16 banks, 32-byte blocks, write-back+write-allocate, 15 cycle hit, 200 cycle miss
Memory Network	4x4 grid, dimension-order routing
Queue Network	implemented as hardware queues with ideal delay
Distributed Scheduler	random work-stealing

Table 2. Simulator Setup

We developed our simulator models using the Liberty Simulation Environment (LSE) [9]'s automatic simulator builder framework. Table 2 describes the various hardware blocks and their parameterizations. The processor core is a simple RISC core capable of executing one instruction every cycle and stalls on misses. We model all transactions in the cache hierarchy in full detail, including transactions for load reservation and store conditional instructions that are necessary to implement

locking. All contention including banking and cache port contention are also accurately modeled. The distributed L2 controllers model a MSI directory-based cache coherence protocol.

The memory network comprises a 4x4 grid that connects all the L2 cache controllers in the 16-way NDP configuration. This allows us to model memory traffic bottlenecks accurately in the network. The tile-to-tile network is modeled as requiring one cycle for injection into and ejection from the network, with three cycles per hop communication cost.

The distributed scheduler is modeled as interacting with its corresponding core through dedicated ports. These ports are used to exchange control and status information between the scheduler and the core (to activate context switches, to inform the scheduler of thread creations and destructions, etc.). When Thread and Queue Tables overflow, they are evicted to off-chip DRAM, with the same penalty as a L2 miss of 200 cycles.

For the power modeling, we use Wattch [10] capacitance-based power models. We assume 50nm technology factors taken from the ITRS 2003 report [11], with 1 GHz cores at 1.1V with perfect clock gating for unused functional units. For all our simulations, we fast-forward and warm caches until we hit the first *thread_spawn* instruction in the main thread, at which point we begin detailed simulation and statistics collection.

4.2 Benchmarks

Studies focusing on fine-grained thread-level parallelism face a chicken-and-egg problem in their evaluations. Since high per-thread overheads plague current systems, few fine-grained parallel applications exist to test future architectures that lower these overheads. For our evaluations, we have hand-ported applications from SPEC and MediaBench to expose the sort of fine-grained aggressively-parallel behavior we wish to encourage. We were able to port the suite of applications with relatively low effort, using the NDP API function calls, and were able to achieve high-degrees of threading (ranging from 60 to 2000+ threads per application, ranging in granularity from between 5,000 to 9,000+ dynamic instructions per thread). In future systems, we plan to expose this parallelism automatically, by exploring a range of automated techniques. In particular, approaches allowing modest speculation (such as transactional coherence [6]) are promising for encouraging aggressive parallelism well-suited to our architecture.

We parallelized *art* and *equake* from SPEC2000, *jpegdec* from MediaBench, and two other programs with highly dynamic behavior *quicksort* and *othello*. The types of parallelism found in these programs can be broken up into three categories: do-all, recursive, and streaming. The first type of parallelism, do-all, is constituted by a single parent thread spawning many independent computational threads, each corresponding to one or more iterations of a critical program loop. Each of these will return a result to the parent thread or a synchronization thread to signify its completion.

Programs *art* and *equake* are both partitioned with do-all style parallelism. The amount of parallelism and the granularity of the threads vary across these benchmarks. *Equake* has parallelism exposed in the computationally-intensive time integration loop. Each of the nodes of the simulation can be computed independently for each time step, but must be synchronized after each time step. In *art*, the parallelism is within an inner loop that is a significant portion of the total execution time. Here the parallelism is more fine grained with less work per thread and more synchronization points. In both programs, parallel sections occurs in phases of very similar threads.

The second type of parallelism we utilize in our benchmarks is recursive parallelism, exhibited by *othello* and *quicksort*. *othello* is an AI game player that calculates the best possible next move by looking ahead in the game tree of possible moves. At each level in the tree, new threads are spawned for each of the possible moves given the game board setup, so the number of threads spawned for each possible move is unpredictable until the board setup is known. *Quicksort* is a typical sorting algorithm with the array being divided into two smaller arrays based on a pivot value. A new thread is recursively spawned for each of the two sub arrays and the amount of work for each is based on the pivot's division of the main array.

The third type of parallelism we look at is the pipelined parallelism of streaming applications. Our parallel implementation of *jpegdec* involves a main thread that divides the image into smaller blocks that can be independently decoded. The first stage of the streaming pipeline takes the block and performs the inverse DCT operation on the data. The second stage takes this information and does the color mapping for the final image. The last stage takes the output of these previous stages and arranges the decoded blocks into the final image. The computational stages of the I-DCT and the color mapping can be duplicated as needed in order to match the rate at which the main thread produces the blocks for processing.

Table 3 characterizes our benchmarks in terms of the number of threads created as well as the granularity of each thread. All benchmarks aggressively expose parallelism and vary fairly widely in thread granularity.

4.3 NDP Performance

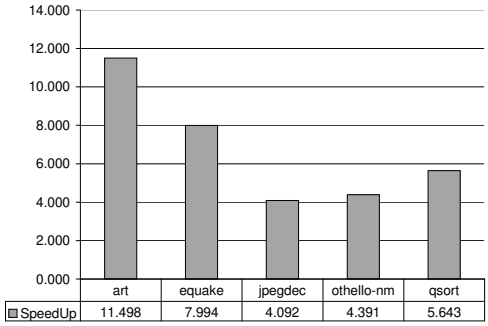


Figure 2. NDP 16 Core Speed-up vs Single Core.

To evaluate the *potential* performance from NDP's low-overhead hardware support mechanisms for dynamic parallelism, we simulated our parallelized applications under optimal scenarios in both 1-core and 16-core, 4x4 mesh configurations. Here, we define optimality by eliminating hardware constraints associated with thread and queue tables and by lifting congestion effects on the thread migration network. All other elements of the model remain unchanged, with the core and cache hierarchy following the description in Table 2.

Hardware constraints on thread and queue tables are twofold: size and execution time in processing the respective data. We handle the former by allowing for an infinite number of thread and queue table entries. This nullifies the need for paging on overflow and presents a best case scenario of a single-cycle hit per local thread or queue entry access. The simulated thread scheduler executes concurrently with the core as a state-machine and thus incurs a one-cycle delay on top of accesses to respond to thread-related events. For this ideal situation, we assume that register contexts are cached in the local thread

Benchmark	#Threads	Avg Inst/Thread	Inst/Thread Stddev	Avg Thread Life (cycles)	Life Stddev
art (phase)	60	6,745	26	320,216	1,544
equake (phase)	2,742	9,408	324	59,004	10,857
jpegdec	674	6,113	621	27,143	5,515
othello	722	5,640	5,219	26,679	12,597
qsort	469	8,468	7,236	24,701	16,024

Table 3. Granularity of parallelism in our benchmarks, in terms of thread count and number of instructions per thread.

entry and can be performed in three cycles.

In the 16-core case, we model randomized task stealing over a congestion-free free network by assuming a uniform 20-cycle penalty for each thread steal/migration. For the mesh topology, this pessimistically assumes the worst-case 6-hop traversal under dimension-ordered routing, allowing 3 cycles per hop and 1 cycle for extraction and insertion at the source and destination. The one-core model is directly connected to the cache hierarchy, but it possesses NDP structures so that it can process the exact same executable as the 16-core version. Since only one tile is available, task-stealing degenerates into a FIFO processing of spawned threads.

Figure 2 illustrates the performance of the 16-way configuration as compared to the 1-way instantiation. We observe promising speedups from 4X for *jpegdec* to 11.5X for *art*. In particular, the do-all style parallelizations for *art* and *equake* excel, showing the greatest benefit. While it is not surprising to expect healthy speed-ups from these sorts of applications under static orchestration, it must be emphasized that these speed-ups were achieved through dynamic scheduling of finely exposed parallelism.

qsort and *othello* both utilize recursive-style parallelism, which imposes a penalty due to dynamic workloads. Looking at data in Table 3, we see that both applications create children threads with small average lifetimes and high variability. This combined with the synchronization of threads in our parallelization decreased the potential for speed-up.

Although *jpegdec*'s parallel performance is the lowest among the applications, we found that our parallelization is fundamentally limited by a bottleneck in the main thread, which produces work for all subsequent stages. This stems from overlapping the preprocessing stage, which must perform a huffman decoding of the input bitstream, with the more independent block decoding. This 4X speedup actually beats a hand-tuned placement of *jpegdec*.

Subsequent subsections will extend on these potential results to show the effects of adding various real-world constraints.

4.4 Effect of hardware constraints

In order to determine the effect of finite hardware area, we ran the same set of tests on a more constrained version of the aforementioned best-case 16-core model. We imposed limitations on the thread table and evaluated performance over a varying number of local entries. In each case, overflow is handled by a least recently used eviction strategy that is enforced on every access to an entry. Hit costs remain unchanged from before; however, a miss to the local entry imposes a constant 210 cycle cost, which mimics a complete miss to off-chip memory in our model, along with a safety factor.

Across our application suite, we found that this constraint marginally affects performance and varies slightly as the number of entries changes. At worst, throughput drops to 95% of that seen previously. This occurred with *jpegdec* and is likely an offshoot of our parallelization, which introduces a large number of queue communications and hence opportunities for context switching.

The delays introduced for paging sometimes had a very minor *positive* effect. This can be seen in the 8-entry case for *qsort*, which performs better than the other versions. Here, the inclusion of delays alters scheduling behavior of the task stealing scheduler and manages to find a more optimal solution. In general, however, the significance of this constraint seems most dependent on the parallelization approach chosen.

4.5 Effect of thread scheduling and spawning delays

Alluding back to our motivation in the introduction, we measured thread spawn and scheduling delays on the order of tens to 100K cycles or more under a traditional operating system.

To model the effects of such costs on our simulation and application environment, we conducted a sensitivity analysis to observe the flexibility of our application suite's granularity with respect to thread spawn and scheduling latencies. Keeping our control points stable, we utilized the ideal resource model from before and inserted a fixed delay to all scheduler operations. Our scheduler still runs concurrently with the core, which is a benefit; however, its decisions are latent with respect to the specified cost. To mimic thread spawn delay, we appropriately restricted the scheduler's view of new threads.

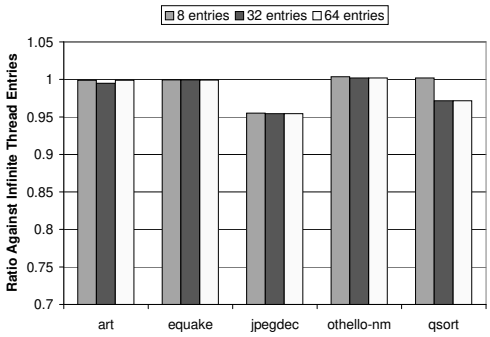
In every case, we found our program run times to degenerate severely between a 10,000 cycle and a 100,000 cycle delay, up to a 6.5X slowdown in *jpegdec* (Figure 3). Under 10,000 cycle delays, we see throughputs ranging from 74% to 96% of the ideal scenario, averaging 84% with standard deviation 10%. Our do-all style parallel programs likely do well because of their independent execution patterns, while benchmarks with highly variable dynamic instruction counts per thread (Table 3) show more susceptibility to such delays. Thus, by reducing thread spawn and thread scheduling delays to below 10,000 cycles, NDP's hardware support can adequately support the level of granularity exhibited by our applications.

4.6 Discussion of Parallelization Overheads

Benchmark	Unparallelized (cycles)	1 Core Parallelized
art	7,629,335	7,768,263
equake	28,611,779	62,569,725
othello	4,137,632	19,803,059
qsort	3,803,969	11,805,770

Table 4. Overhead of Parallelization.

To measure the overhead of parallelization, we simulated a single core configuration of NDP running the unparallelized application and compared it to a parallelized version using the NDP API function calls. Interestingly, the do-all application *art* had a trivial 1.8% run time overhead from parallelization, but a recursive-style program *qsort* suffered a much larger overhead. We traced the cause of this to cache effects due to the current lack of support in the simulator for prefetching—when running the simulator with an ideal cache, the parallelization



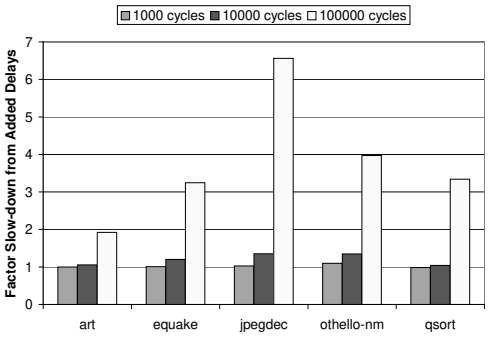


Figure 3. Sensitivity Analysis: (a) Thread Table Entry Constraints, and (b) Run Time vs. Spawn and Scheduling Delays

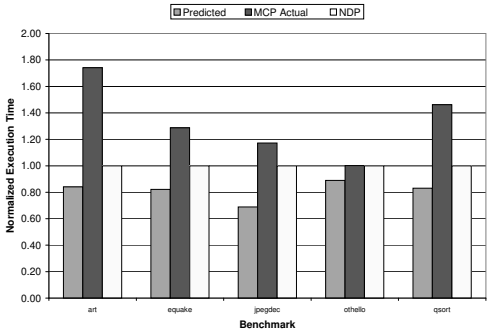


Figure 4. Predicted, actual, and NDP results with static scheduling using MCP.

overhead for all applications was low: 0.01% for *art*, 1.9% for *equake*, 5.0% for *othello*, and 1.5% for *qsort*. In part, this overhead is incurred, because we lose cache locality as NDP's non-blocking *thread_spawn*s spawn threads successively, allocating stack space and input argument space in the cache for each of them, before running them. This is done in order to uncover more of the task graph of the program for the scheduler. Unfortunately, by the time the first thread is run, its associated stack and arguments are long evicted from the cache. This is especially severe in a single-core run, where all threads share a single cache. In the future, we plan to design more sophisticated schedulers that can aggressively prefetch stack and local variables before executing a thread.

4.7 Effectiveness of hardware vs. software scheduling

As a yardstick for static software scheduling, we evaluated a static scheduling algorithm, MCP [12], that has been found to lead to superior schedules as compared to the other state-of-the-art scheduling algorithms [13]. MCP is based on critical-path scheduling, i.e. it schedules critical-path threads over non-critical-path ones. Since deriving an oracle schedule is an NP-hard problem [14], MCP is a heuristics-based algorithm. However, it is supplied with oracle thread load and critical path information that is obtained through profiling with the exact data inputs.

Figure 4 shows the performance of MCP's generated schedule, based on MCP's offline prediction, as well as the actual performance when MCP's schedule is fed back into the simulator and used to orchestrate thread placement in place of NDP's hardware scheduler. Across our applications, we see MCP's offline-predicted schedule resulting in 32-60% speedup as compared to our simple random work-stealing hardware scheduler on 16 cores. However, when MCP's schedule is fed back into the simulator, the performance gains are wiped out, with actually a 12-107% slowdown vs. our simple hardware scheduler. The reason lies in MCP's inability to factor in the new cache effects that are introduced by its schedule statically. These memory variabilities highlight the difficulties faced in static scheduling in handling runtime variabilities.

4.8 NDP's impact on energy-delay-product

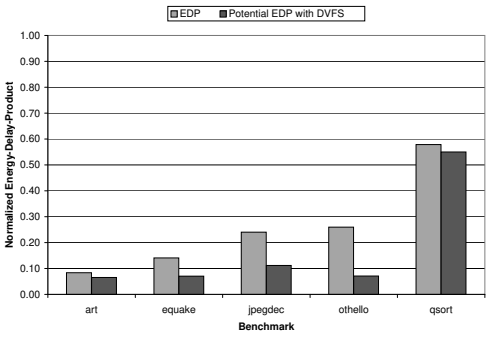


Figure 5. Energy-delay-product of our 16-core architecture normalized against the energy-delay-product of a single core, with and without oracle dynamic voltage and frequency scaling.

Figure 5 shows the energy-delay product (EDP) of a 16-core machine versus the EDP of a single core. Even with the additional cores added, as well as the additional performance impact due to contention in the memory system, contention for locks on shared values, as well as synchronization overhead, the 16-core machine still improves on energy-delay product by a significant amount. Note that each core in the 16-core machine is the same as the single core; thus, while the total energy of the 16-core machine is higher than the single core, because of the significant performance improvement one could use simpler, smaller cores to consume less energy and still achieve a lower energy-delay product than the single core machine.

For the apps in Figure 5, all except quicksort consume approximately the same energy as the single core machine, within 10%. Quicksort consumes 3.3X the energy of the single core machine, due to massively increased overhead in obtaining a lock on mallocs. However, because of the performance improvement, quicksort still has a lower energy-delay product. With 16-cores, on average NDP improves EDP by 3.8X.

By moving scheduling into the hardware, the NDP hard-

ware scheduler is uniquely placed to also schedule dynamic voltage and frequency scaling of the cores. Here, we explore the further energy-delay-product savings achievable with an oracle DVFS scheduler that knows precisely the idle times between scheduled threads and during failed thread steals. For some programs, such as *othello*, many threads complete and sit idle, waiting for other cores to finish. Coupled with a significant number of failed steals—the performance impact of which is masked due to the sheer number of cores attempting to steal— this translates into significant opportunities to use DVFS. Figure 5 shows the potential EDP savings of an oracle DVFS scheduler – up to 6.25X savings in EDP.

5. Related Work

Dynamically vs. statically-mapped parallelism. Research projects in chip multiprocessors have focused on statically-mapped parallelism, either explicitly uncovered by programmers, or automatically extracted by compilers [5, 15, 16, 17, 18]. RAW [5] not only statically partitions code onto the 16 tiles, but also statically orders communications between these tiles so as to lower communication delay, enabling fine-grained parallelism. Morphable architectures such as TRIPS [15] and Smart Memories [16] can be configured dynamically for different execution granularities, but the configuration occurs occasionally to suit different application classes; applications are then statically mapped onto the specific configuration. Synchroscalar [18] goes further with statically-mapped parallelism, with the compiler/user statically controlling power as well, determining the suitable paritioning of code and corresponding frequency/voltage to run each cluster of cores at for optimal power-performance.

Run-time mapping of parallelism has been proposed in various forms. Cilk [1] advocated *software* management of thread scheduling and mapping with a user-level library. To map that onto a CMP, Cilk requires underlying OS support, thus incurring high OS thread management delays.

The hardware mapping of parallelism in NDP is reminiscent of dataflow architectures, which have been proposed and shown to be feasible for future CMPs in Wavescalar with the addition of a Wavecache [17]. Threaded data flow architectures have similar hardware mechanisms for ensuring low-cost thread creation and context-switching delays [19, 20, 21, 22, 23]. However, prior architectures assume naive greedy local scheduling policies which round-robin instructions or threads onto cores. As pointed out by Culler in [24], these scheduling policies, which do not have access to program information such as thread relationships and resource usage patterns, lead to poor mapping of instructions or threads onto cores. This motivated dataflow architectures such as TAM [25] and Monsoon [19] where the compiler statically maps threads onto processing elements while the dataflow hardware schedules the threads locally in each processing element's queue. While TAM proposes hardware exposing this scheduling mechanism to compilers or programmers, NDP takes it one step further – have the hardware itself perform this more sophisticated scheduling and mapping of threads onto processing elements. This not only ensures very low scheduling overheads as the scheduler can quickly access thread and tile information, but also gives the scheduler access to run-time information.

Current uniprocessor architectures provide dynamic, hardware-modulated parallelism at the instruction level. Out-of-order superscalar architectures dynamically issue instructions to multiple functional units, while simultaneous multithreaded architectures [26] dynamically modulate the degree of parallelism of each thread onto the multiple processing resources. Architectures such as ILDP [27] and clustered microarchitectures [28, 29] dynamically manage parallelism at the instruction level as well. Multiscalar [30] explores dynamic paralellism at the coarser function level, focusing on speculative parallelism.

Previous architectures have targeted streaming parallelism – both Imagine [4] and RAW [5] used streams which capture data flow relationships between threads to better orchestrate memory and communication. However, both statically place and parallelize programs, and hence will not be able to dynamically modulate the degree of parallelism and flow balance streaming programs.

Hardware support embedded at the network level. The efficiency of embedding hardware support at the network level has been leveraged to speed up execution in parallel processors in the past. Message-driven processors (J-Machine [31] and M-Machine [32]) embed hardware support for message sending and handling at the network level, proposing register-mapped interfaces, message-triggered interrupts, and dedicated message handling threads to lower the overhead of sends and receives in multicomputers. Cache-coherent shared-memory processors have coherence controllers at the network interfaces [33, 34] to minimize the coherence protocol overhead, and architect the network to ensure ordering and priority for different classes of coherence messages. Recent fine-grained CMPs such as Raw and TRIPs re-architect the network to lower communication overhead, with Raw having statically-scheduled operand networks [35], and TRIPs proposing dynamic pre-reservation of communication bandwidth [15].

6. Conclusions

Chip multiprocessors are already a commercial reality, and are likely to continue scaling to higher degrees of parallelism in the near future. Given the long-standing difficulties in parallelizing applications and workloads, our work here seeks to address these difficulties by rethinking the hardware-software dividing line. In particular, the hardware methods we propose are effective at reducing the overheads of thread creation, scheduling, and context switching. As a result, hardware can now play an active and effective role in managing parallelism in the face of runtime variations due to memory-system effects and dataset dependencies.

This paper provides initial proof-of-concept for several of the key building blocks we propose. In particular, we learn that:

- Our proposed approach offers good speedups (4X-11X) for programs with quite diverse styles of parallelism, including recursive, streaming and do-alls.
- Dedicated support for thread scheduling and management is crucial to these performance numbers. Embedding thread creation, context-switching and scheduling in the hardware allows these operations to have low delays (tens or hundreds of cycles), which is critical for supporting aggressive parallelism. Our sensitivity studies showed that higher delays caused significant slowdowns: up to 85% slowdown with 10000, and up to 6.5X slowdown with 100000 cycle thread delays. While we explore a hardware-oriented approach in this paper, there are still possibilities for further variants. In particular, we are also planning future work addressing some of the remaining memory latencies encountered during thread spawns.
- The support required can be achieved with modest hardware overheads. For example, a 64-entry-per-core thread table sees only a 3% slowdown compared to assuming infinite hardware resources.

- Even in do-all parallelism, the variations of task lifetimes make dynamic thread scheduling important and useful. For our application suite, our simple distributed dynamic task scheduler achieves better performance than an elaborate offline static scheduler with profiled knowledge of thread lifetimes, instruction count, and full knowledge of the task graph. We plan further future work to develop more elaborate online schedulers in which cores share performance information to guide on-the-fly adjustments.
- In today's and future computer systems, good performance behavior is only a partial solution. Energy characteristics also play a fundamental role. For our system, we found that parallelization itself buys up to 85% savings in energy-delay-product, demonstrating the importance of leveraging parallel hardware. Fusing energy adjustments (power-gating or frequency scaling) with thread scheduling is expected to offer even further gains, due to the clean abstraction layers that NDP offers in this regard. From our simulations, dynamic voltage and frequency scaling can potentially save a further 50% of energy, improving the energy-delay-product up to over 93%.

Looking forward to future technologies and architectures, dynamic techniques for tolerating variations will only become more central. In addition to variations stemming from traditional sources such as dataset and memory-system variability, future systems will encounter a range of new sources of speed, energy, and reliability variability. Maintaining performance and parallelism on these high-variation chips will require techniques that can dynamically react to a range of events. The NDP architecture proposed here offers, we feel, a clean substrate for supporting aggressive parallelism in current systems, as well as variation-tolerant design in the future.

Acknowledgments

We thank David August and his group at Princeton for their support of our use of the Liberty Simulation Environment (LSE) and for extensive discussions on the NDP architecture. We also wish to thank Niraj Jha, Fei Sun and Anish Muttreja of Princeton for their pointers towards the MCP static scheduling algorithm as a comparison yardstick for NDP and performance macromodels. This work is supported in part by the MARCO Gigascale Systems Research Center, NSF grants CNS-0410937 and CNS-0305617.

7. References

[1] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," in *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, . 1998, pp. 119–129.

[2] G. J. Narlikar and G. E. Blelloch, "Pthreads for dynamic and irregular parallelism," in *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, . 1998, pp. 1–16.

[3] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August, "Decoupled software pipelining with the synchronization array," in *Proc. 13th International Conference on Parallel Architectures and Compilation Techniques*, 2004.

[4] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens, "A bandwidth-effi cient architecture for media processing," in *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society Press, 1998, pp. 3–13.

[5] M. B. Taylor *et al.*, "Evaulation of the RAW microprocessor: An exposed-wire-delay architecture for ILP and streams," in *Proc. International Symposium on Computer Architecture*, June 2004.

[6] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun, "Programming with transactional coherence and consistency," in *ASPLOS '04*, Oct. 2004, pp. 1–13.

[7] W. J. Dally, "Virtual channel flow control," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 2, pp. 194–205, Mar. 1992.

[8] e. a. Agrawal, A, "The mit alewife machine: A large-scale distributed-memory multiprocessor," Kluwer academic Publishers, 1991.

[9] M. Vachharajani *et al.*, "Microarchitectural exploration with Liberty," in *Proceedings of the 35th International Symposium on Microarchitecture (MICRO)*, November 2002, pp. 271–282.

[10] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architecture-Level Power Analysis and Optimizations," in *Proc. ISCA-27*, ISCA 2000.

[11] "International technology roadmap for semiconductors," http://public.itrs.net.

[12] M.-Y. Wu and D. D. Gajski, "Hypercool: a programming aid for message-passing systems," vol. 1, no. 3, 1990, pp. 330–343.

[13] Y.-K. Kwok and I. Ahmad, "Benchmarking and comparison of the task graph scheduling algorithms," vol. 59, 1999, pp. 381–421.

[14] P. Brucker, "Scheduling algorithms, 4th edition." Springer, 2004, ISBN 3540205241.

[15] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture," in *Proc. of the 30th International Symposium on Computer Architecture*, June 2003, pp. 422–433.

[16] K. Mai *et al.*, "Smart memories: A modular reconfi gurable architecture," in *Proc. Int. Symp. Computer Architecture*, Nov. 2000, pp. 161–171.

[17] S. Swanson *et al.*, "Wavescalar," in *Proc. MICRO*, November 2003.

[18] J. Oliver *et al.*, "Synchroscalar: A multiple clock domain, power-aware, tile-based embedded processor," in *Proceedings of the International Symposium on Computer Architecture*, 2004.

[19] K. Traub, M. Beckerle, G. Padadopoulous, J. Hicks, and J. Young, "Overview of the monsoon project," in *Proceedings of ICCD*, 1991.

[20] V. Grafe, G. Davidson, J. Hoch, and V. Holmes, "The epsilon dataflow processor," in *Proc. Int. Symp. Computer Architecture*, 1989.

[21] Y. Yamaguchi and S. Sakai, "An architectural design of a highly parallel dataflow machine," in *IFIP*, 1989.

[22] J. R. Gurd, C. C. Kirkham, and I. Watson, "The manchester prototype dataflow computer," *Communications of the ACM*, vol. 28, no. 1, Jan./Feb. 1985.

[23] Arvind and R. Nikhil, "Executing a program on the mit tagged-token dataflow architecture," *IEEE Transactions on Computers*, vol. 39, no. 3, 1990.

[24] D.E.Culler, K.E.Schauser, and T. von Eicken, "Two fundamental limits on dataflow multiprocessing," in *IFIP*, 1993.

[25] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken, "Tam - a compiler controlled threaded abstract machine," 1993.

[26] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proc. ISCA-22*, June 1995, pp. 392–403.

[27] H.-S. Kim and J. E. Smith, "An instruction set and microarchitecture for instruction level distributed processing," in *Proceedings of the 29th annual international symposium on Computer architecture*. IEEE Computer Society, 2002, pp. 71–81.

[28] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *Proceedings of the 24th annual international symposium on Computer architecture*. ACM Press, 1997, pp. 206–218.

[29] J.-M. Parcerisa *et al.*, "Effi cient interconnects for clustered microarchitectures," in *Proc. PACT*, 2002.

[30] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *Proceedings of the 22nd annual international symposium on Computer architecture*. ACM Press, 1995, pp. 414–425.

[31] M. D. Noakes, D. A. Wallach, and W. J. Dally, "The j-machine multicomputer: an architectural evaluation," in *Proceedings of the 20th annual international symposium on Computer architecture*. ACM Press, 1993, pp. 224–235.

[32] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee, "The m-machine multicomputer," in *Proceedings of the 28th annual international symposium on Microarchitecture*. IEEE Computer Society Press, 1995, pp. 146–156.

[33] J. Kuskin *et al.*, "The stanford flash multiprocessor," in *Proceedings of the 21ST annual international symposium on Computer architecture*. IEEE Computer Society Press, 1994, pp. 302–313.

[34] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. Morgan Kaufman Publishers, 2004.

[35] M. B. Taylor *et al.*, "Scalar operand networks: On-chip interconnect for ilp in partitioned architectures," in *Proc. HPCA*, February 2003.