International Journal of New Computer Architectures and their Applications (IJNCAA) 10(1): 10-22
The Society of Digital Information and Wireless Communications, 2020 ISSN 2220-9085 (Online); ISSN 2412-3587 (Print)

Hardware-based scheduling and synchronization for light-weight, fine-grained multi-threading

Samer Haddad and Jeanine Cook
New Mexico State University
Dept. of Electrical and Computer Engineering
1780 E University Ave, Las Cruces, NM 88003
sahaddad@nmsu.edu, jecook@nmsu.edu

ABSTRACT

Fine-grained, light-weight multi-threading enhances the performance and scalability of many applications due to its ability to hide memory and network latency and enhance load balance, however, it aggravates the software scheduling and synchronization overhead, which impairs the performance and scalability. In this research, we propose an efficient, dynamic hardware scheduler, which consumes much less time and bandwidth than software schedulers. The proposed scheduler replaces the software application programming interfaces (APIs) and *Workers* (operating-system-visible threads) with simple hardware that can be integrated on-chip with reasonable silicon area and power budgets. Simulation results show that our proposed hardware scheduling outperforms software scheduling by 22.76 times and it exhibits significant scalability improvement when compared with software scheduling.

KEYWORDS

Fine-grained, Light-weight, Multi-threading, Software scheduling, Hardware scheduling

1 INTRODUCTION

Multiple studies conclude that software scheduling and synchronization overhead plays a significant role in degrading the performance and scalability of applications exploiting fine-grained multi-threading. Software scheduling has its shortcomings: First, the scheduler's software consumes the system resources and significant time in making scheduling decisions and context switches. Second, the scheduler's software perturbs the cache sub-system degrading the executing application performance. Third, the scheduler's software may lead to page faults, which tangibly degrade the performance [1, 2, 3, 5, 6, 7]. Hardware acceleration has long been known to enhance performance, therefore, we propose a novel micro-architecture of hardware scheduling and synchronization to accelerate scheduling in contemporary computer systems. The scheduling comprises thread creation, selection, assignment, and blocking.

2 BACKGROUND AND MOTIVATION

Dividing a problem into fine-grained tasks is called fine-grained parallelism (FGP). FGP facilitates harnessing the computational power of multi-core CPUs and many-core GPUs to solve algorithms/applications more efficiently [4]. However, FGP aggravates the overhead of scheduling and synchronization [5], [8].

Much research is underway to develop efficient runtime systems to handle FGP (Qthreads, HPX, Cilk, Cilk+ and Cilk++ [9, 10, 11, 12, 13, 14, 15, 16]). These runtimes manage threads by software only, inflating the scheduling and synchronization overhead, thereby exacerbating network and memory contention. Significant improvements in the policies and algorithms of these runtimes have been achieved. However, these improvements are on the software side and they do not sufficiently resolve the performance and scalability challenges [2, 3].

Work done to address thread management by a combination of software and hardware techniques exists such as Carbon [5] and asynchronous direct messages (ADM) [1],

which claim reasonable performance improvement. However, these techniques still utilize a software-based *Worker* (operating-system-visible task) to manage the scheduling. We propose implementing the scheduling entirely in hardware eliminating the *Worker* task, which is expected to speed up scheduling and improve scalability. Also, the implementation of fine-grained synchronization (FGS) in these runtimes, if any, is handled by software and mainly depends on polling the status of memory lines, which burdens the cache subsystem and memory bandwidth. On the contrary, we propose using only the hardware to handle the FGS, thereby, enhancing performance.
The scheduling cost has to come down to help crossing the boundaries of exascale performance. This research is another contribution toward achieving this goal.

3 RELATED WORK

3.1 Carbon

Kumar et al. propose Carbon [5], a hardware technique to accelerate dynamic task scheduling, which comprises: 1) Local Task Unit (LTU), 2) Global Task Unit (GTU), 3) A hardware engine inside the GTU to execute task stealing. The local task unit (LTU) is a prefetch unit residing in each core (each core has an LTU). The LTU hosts a single-entry buffer for each hardware thread, which is used to prefetch light-weight tasks (task's context is a tuple of four 64-bit values) from the GTU. It is positioned between the cores and the GTU to alleviate the impact of latency between the cores and the GTU as the number of cores increases. The GTU is integrated with the cache sub-system as a separate unit so that both (GTU, cache sub-system) share the die interconnect with the cores and main memory. The GTU contains a set of double-ended queues (DEQUEs). There is one DEQUE for each hardware thread. Placing all DEQUEs in the same physical unit (GTU) facilitates task stealing among the DEQUEs. An on-chip network unit is used to connect the cores to the GTU and the cache sub-system. A *Worker* task (operating-system-visible task) is created per hardware thread, which handles the scheduling of light-weight tasks to the available hardware threads. A set of dedicated instructions issued by the *Worker* is used to enqueue and dequeue new tasks into the LTU and GTU. For example, an enqueue instruction is used to save a newly-created task to the LTU, and a dequeue instruction is used to assign a task from the LTU to a hardware thread to start executing. The LTU saves its task to the GTU if an enqueue instruction is issued and a valid task exists in the LTU's buffer. When the LTU's buffer becomes empty, the LTU fetches a new task from the GTU only if the number of tasks in the GTU's DEQUEs is higher than the number of hardware threads, otherwise, it does not fetch. A third instruction (called TQ_ENQUEUE_LOOP) can save new tasks in the GTU directly without writing to the prefetch buffer in the LTU. If the GTU becomes full, then a user-level software exception saves some of the tasks (e.g., a third of them, the actual number of tasks is heuristically determined) from the GTU to memory subsystem to create some space for newly-created tasks. Also, when the number of tasks in the GTU drops to a certain threshold, a user-level exception populates the GTU with new tasks from the memory subsystem if it has some. Work balance is achieved by a hardware-based work-stealing engine in the GTU. That is, when the LTU queue and its relevant GTU's DEQUEs become empty and when a dequeue instruction is issued, then a hardware-based engine steals tasks from the tail of another DEQUE chosen randomly.

3.2 Asynchronous direct messages (ADM)

Sanchez et al. present Asynchronous Direct Messages (ADM) [1] to enhance performance of software-based schedulers. The ADM allows the *Workers* to communicate efficiently by using a dedicated hardware unit inside each core. The ADM shares the signaling interconnect with other units such as data cache (D$) and instruction cache (I$) inside each core, and it exploits the coher-

ence interconnect to secure inter ADM communication. The ADM provides direct exchange of asynchronous, short messages (e.g., task stealing messages) among the *Workers* in the chip-multi-processing (CMP) without going through the memory hierarchy, and thereby reducing the contention on memory subsystem. The ADM provides user-level support to send messages for control purposes from one *Worker* to another *Worker* through a scheduler's thread called the *Manager* thread. These messages are used to implement task-stealing and inter-schedulers coordination without using the cache subsystem that incurs coherence and synchronization overheads. These messages are used to achieve load balance among the work queues. The ADM is limited to achieving load balance by work stealing only using a dedicated hardware interconnect; the scheduling is executed entirely by software. This is contrary to our proposed scheduler which implements the scheduling, synchronization, and load balance in hardware.

3.3 Fine-grained synchronization (FGS)

Some computer systems attempted to harness the performance merits of fine-grained synchronization (FGS) typified by the XMT [17, 18, 19], Alewife [20, 21] and M-machine systems [22, 23]. Table 1 shows how the FGS is implemented in these machines.

The Cray's XMT is a scalable multi-threaded computer built with the Cray's Threadstorm processor. This custom processor is designed to exploit parallelism that is only available through its unique ability to rapidly context-switch among many independent hardware execution streams. Cray Inc. launched the XMT computer system in 2006 [24, 25].

The Alewife, developed at MIT in the early 1990s, is a large-scale multiprocessor that integrates both cache-coherent, distributed shared memory, and user-level message-passing in a single integrated hardware framework. Each Alewife node consists of a 33 MHz Sparcle integer unit, an off-the-shelf FPU, 64 Kbytes of direct-mapped cache, and 4 Mbytes of globally-shared main memory. The nodes communicate via messages on a two-dimensional mesh network. The current implementation scales directly to 512 nodes. The Alewife is a research computer system that was not launched commercially [21, 26, 27].

The M-machine computer system is a non-commercial, research multi-computer developed at Stanford and MIT to test architectural concepts motivated by the constraints of modern semiconductor technology and the demand of programming systems. The M-machine nodes are connected with a 3-D mesh network; each node is a multi-threaded processor incorporating 12 functional units, an on-chip cache, and a local memory [22].

Table 1. FGS summary

Conditional load/store instructions examine the Full/Empty (F/E) bit. If the F/E bit matches an expected value, then a state hit is detected, otherwise, a state miss is detected.	
Machine	Synchronization
XMT	Full/Empty (F/E) bit per double-word in main memory. On a state miss, hardware retries the conditional load/store, and if a retry limit expires, a trap handler blocks the thread and saves it to main memory. On a state hit, the conditional load/store commits.
Alewife	Full/Empty (F/E) bit per word in main memory. On a state miss, a software trap handlers deal with retrying the conditional load/store, thread blocking, queueing and scheduling. On a state hit, the conditional load/store commits.
M-Machine	Full/Empty (F/E) bit per register in the register file and F/E bit per double-word in main memory. On register F/E bit access, the operation stalls on a state miss and issues when the F/E bit takes expected value. On main memory access, a software loop polls the memory F/E bit until it takes expected value.

4 PROPOSED ARCHITECTURE

4.1 Scheduler's functionality

Figure 1 shows a top-level diagram of the proposed hardware scheduler. Threads are created by a spawn instruction. When the spawn instruction is executed, the hardware reads the thread's context registers (a tuple of 8 registers where each register is 8 bytes) to sample its content and then saves it in a Last-In-First-Out (LIFO) residing in the Runnable Thread Unit (RTU). This LIFO is called the RTU_LIFO. There is one RTU per core. New threads are assigned to a LIFO and not to a

FIFO so that they benefit from the cache locality established by their ancestors. If the RTU_LIFO is full, then a software trap, called the RTU overflow trap, is invoked to save the thread's context to a buffer in memory subsystem called the Runnable Thread Memory Buffer (RTMB). A running thread with a state miss is blocked from the hardware thread and is saved in the Deferred Thread Buffer (DTB) in an inactive state. The DTB resides in the deferred thread unit (DTU), which is shared among the cores. A state miss occurs whenever the Full/Empty (F/E) bit does not match an expected value. The F/E bit is examined by conditional load/store instructions.

Normal load/store instructions do not examine the F/E bit. When a thread is blocked on a state miss, it is blocked by hardware only via a signal from the cache sub-system (e.g., L2$ as shown in Figure 1), where the state hit/miss on the line is resolved. This signal triggers the pipeline to evict the thread and store it in the DTB, leaving behind an available hardware thread that can be used by another thread. If the DTB is full on a state miss, then a software trap called the Deferred Thread Buffer overflow trap is invoked to evict the thread and store it in a buffer in memory subsystem called the Deferred Thread Memory Buffer (DTMB).

When the line associated with the blocked thread becomes ready, the corresponding thread in the DTB is activated (through a signal from the cache subsystem where line state as hit/miss is resolved) so that it arbitrates through the thread selection and assignment unit (TSAU), as shown in Figure 1, to gain access to any available hardware thread. More than one thread can be blocked on access to the same line. These threads are linked together as a LIFO in the DTB and are issued accordingly. Only the threads whose lines become ready are scheduled to gain access to the hardware threads, others wait in the DTB until their data is ready. This enhances the performance since the cores will only run the threads that are potentially able to achieve progress.

A hardware-generated thread called the Recruiting Thread (RT) fetches in LIFO order for increased locality the threads saved in memory subsystem due to RTU_LIFO full or DTB full (these are the threads saved in RTMB or DTMB) and saves them in the RTU_FILLQ. The RT is generated when the number of threads in the RTU_LIFO drops below a programmable threshold and the RTU_FILLQ is empty and when a flag stored in a register in the RTU denotes the presence of threads saved in memory subsystem due to either RTU_LIFO full or DTB full. This flag is set by either the RTU overflow trap or the DTB overflow trap. The RT must run on the core that invoked it to ensure that the threads fill in an empty RTU_FILLQ and run in a core whose hardware threads are less likely busy running earlier threads. If a hardware thread is not available on the core that invoked the RT (status of each hardware thread whether idle or busy is signaled to the TSAU by the cores) then the RT is not allowed to arbitrate to occupy a hardware thread until it becomes available.

The TSAU, as shown in Figure 1, arbitrates the requests from the RTU_LIFO, RTU_FILLQ, and DTB using a round-robin scheme. The requests from the recruiting threads are given higher priority than RTU_LIFO, RTU_FILLQ, and DTB requests and they are only assigned to the core that issues them. The newly-created threads are assigned to the hardware threads according to a fixed priority scheme where core0 hardware threads have higher priority than core1 hardware threads, core1 hardware threads higher than core2 hardware threads and so forth. The TSAU assigns one thread per CPU cycle to an available hardware thread.

It takes a small number of CPU cycles to select a thread and assign it to a hardware thread, therefore, hardware thread starvation becomes unlikely as long as some threads exist in the queues. A thread awaiting its turn to be assigned will not wait so long due to the fast rate by which the hardware assigns threads to the available hardware threads.

As the sizes of the RTU_LIFO and DTB increase, they become less likely to fill up, reducing the call on the RTU and DTB overflow traps resulting in less RT invocation,

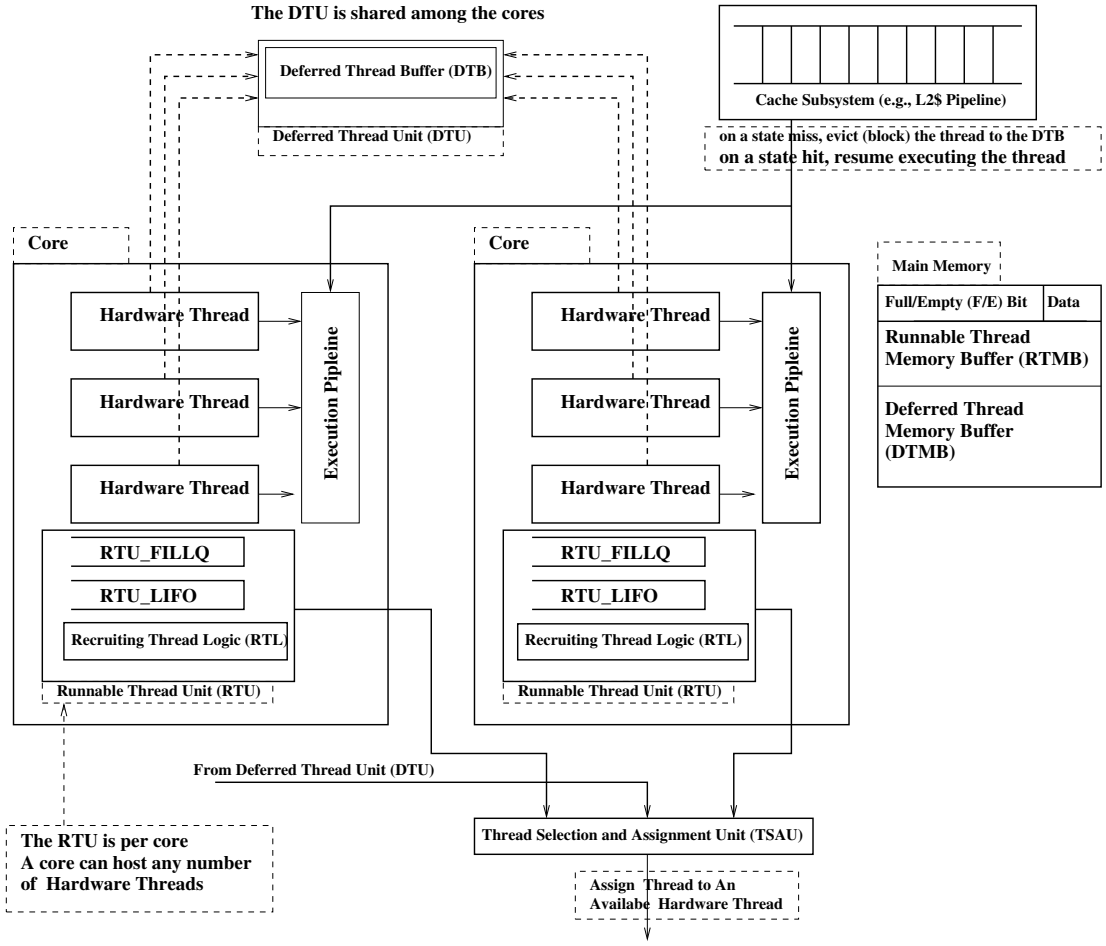

Figure 1. Proposed hardware scheduler

which is expected to enhance the performance. The maximum sizes of the RTU_LIFO, RTU_FILLQ, and DTB are dictated by the silicon and power budgets. For example, choosing RTU_LIFO, RTU_FILLQ and DTB with a 1024-entry each demands 64 Kbytes (this is double the size of a typical Icache or Dcache size in commercial processors) of silicon area per unit (RTU_LIFO, RTU_FILLQ, DTB) since each thread context is 64 Bytes. More on the scheduler's silicon area is provided in subsection 4.5. On the other hand, choosing 64-entry demands 4 Kbytes of silicon area per unit (RTU_LIFO, RTU_FILLQ, DTB).

The RTU and DTU can be connected to the load/store pipeline through their own address space so that they can be accessed by load/store instructions just like any other resource on the chip.

4.2 Hardware-based fine-grained synchronization (FGS)

A set of conditional load/store instructions is used to access a data line that is tagged by a Full/Empty (F/E) bit. These instructions can execute in the cache sub-system (typically this is the cache closest to the main memory called the first-level cache) as a read-modify-write (RMW) to resolve the state of the line as a hit or miss. Resolving the state of the line in the cache sub-system instead of the main memory saves the memory bandwidth. On a state hit, the cache sub-system signals the execution pipeline to resume executing the thread, and on a state miss, the cache sub-system signals the execution pipeline to evict (block) the thread and store it in the DTB. If the DTB is full on a state miss, then the cache sub-system asserts a request to the execution pipeline to in-

voke the DTB overflow trap to save the thread in the DTMB shown in Figure 1. Compared with the XMT, Alewife, M-Machine FGS implementations, our proposed hardware-based FGS does not waste the memory bandwidth by polling the line F/E bit on a state miss. On the contrary, it blocks the thread entirely by hardware in a few CPU clock cycles by saving it to the DTB rather than using the software as in the XMT, Alewife, and M-Machine. It allows the blocked thread to arbitrate for a hardware thread only when its respective line state represented by the F/E bit matches an expected value. Therefore, when the thread runs again, it is guaranteed to achieve progress.

4.3 Load balance

All cores and hardware threads in a multi-core processor must be kept busy doing useful work to achieve load balance. The hardware scheduler uses the following techniques to balance the load: 1) The TSAU implements the randomized distributed task-stealing technique in hardware, which has been shown as efficient in execution time [5]. The scheduler's runnable thread unit (RTU) is allocated per core. The deferred thread unit (DTU) and the thread selection and assignment unit (TSAU) are shared among the cores. The TSAU receives requests from the RTUs and DTU to schedule threads. The TSAU detects the availability of hardware threads in all cores via a set of signals from each core to the TSAU signaling the status of the core's hardware threads as running or idle. The TSAU assigns a thread fast (from any RTU or DTU) to any available hardware thread in any of the cores reducing the hardware thread idle time, thereby enhancing load balance. And 2) The RT examines the RTMB and DTMB to see if there are any threads, and if so, it loads the threads to the empty scheduler's RTU_FILLQ in the core where the RT is running, which reduces the TSAU starvation to new requests. This helps distribute the work fast to all cores.

4.4 Scheduler's building units

The hardware scheduler comprises the RTU, DTU, and TSAU. The RTU comprises the RTU_LIFO and RTU_FILLQ typically implemented using an SRAM-based memory, a read/write controller, and a data path unit used to process input/output packets (a packet comprises thread's context bits and some status bits such as packet's valid bit) through latching/multiplexing/Boolean operations. The DTU comprises the deferred thread buffer (DTB), typically implemented using SRAM-based memory, a controller used to control read/write operations to the DTB, and a data path. The TSAU comprises a controller and a data path. The TSAU controller implements round-robin and fixed arbitration schemes while the data path processes data via latching/multiplexing/Boolean operations.

4.5 Scheduler's silicon area and power

The scheduler silicon area and power are dominated by the sizes of RTU_LIFO, RTU_FILLQ, and DTB, which are determined by their number of entries (each entry is a single task's context). As the number of entries increases, the performance improves due to the access count reduction to the memory subsystem. However, the required silicon area and power increase. Using 1024 entries for each of these units dictates utilizing a 1.0625 Mbytes of silicon area for an 8-core processor. In a processor with 64 Mbytes of cache similar to the M8 processor from Oracle, the 1.0625 Mbytes occupies around 1.66% of the cache silicon area. On the other hand, using 64 entries cuts down the silicon area from 1.0625 Mbytes to 68 Kbytes. The TSAU controller is an arbiter (using round-robin and fixed arbitration scheme) whose silicon and power requirements are small. The TSAU data path comprises flops and multiplexers whose role is to select and latch a task then assign it to an available hardware thread. A total of 512 flops (used as a flop station by the TSAU data path output bus to the cores), 512 9x1 multiplexers, 512 8x1 multiplexers, and 512 2x1 multiplexers are used to

implement the TSAU data path. More flops may be needed if the clock timing to transmit the signals from the TSAU to the cores is not met. The TSAU is realizable with reasonable silicon area and power budgets. The TSAU occupies in silicon what is equivalent to 8 Kbytes of SRAM-based memory based on worst-case estimation by counting the number of transistors in the TSAU and comparing it to the number of transistors in a 6T-SRAM-based 64 Kbytes memory (6T: 6-transistor SRAM memory cell).

Carbon does not provide information about the size, silicon, and power budgets of the GTU. Therefore, we could not numerically compare our proposed architecture with Carbon. However, we expect the silicon and power budgets of RTU_LIFO, RTU_FILLQ, and DTB to be close to carbon's GTU due to the similarity in structure and operation. However, our proposed scheduler adds the TSAU, whose silicon and power budgets may be offset by that of the LTU in Carbon.

4.6 Proposed scheduler vs. carbon

Comparing our proposed scheduler with Carbon, we find: 1) Carbon blocks threads by software, whereas our proposed scheduler blocks threads by hardware when FGS state miss (Full/Empty bit does not match an expected value) is detected, 2) Carbon assigns new threads to hardware threads using a dedicated instruction (dequeue instruction) whereas our proposed scheduler does the thread assignment to the hardware threads using the TSAU (hardware only), 3) Carbon is a hybrid scheduler that makes use of software and hardware to do the scheduling; it utilizes a *Worker* to manage threads whereas our proposed scheduler utilizes mainly the hardware to manage threads; the software is only used when the hardware-based queues require data backup/restore to/from memory subsystem, 4) our proposed scheduler implements the FGS in hardware contrary to Carbon, which does not dedicate any hardware structure for FGS.

5 EVALUATION FRAMEWORK

We use the register transfer language (RTL) to simulate our proposed architecture since it is more accurate than software simulators. At first, we planned to emulate the RTL on a Xilinx FPGA board. However, due to unanticipated resource constraints, we chose to harness performance results based on RTL simulations rather than executing standard benchmarks on Xilinx FPGA. We chose to modify Oracle's open-source T1 RTL to make it support hardware scheduling and synchronization. The T1 processor is an in-order execution processor comprising 8 cores, and a cross-bar unit that links the cores to an on-chip L2$ (based on T1 Terminology, this is the cache higher than D$ and I$). The T1 supports up to 8 cores with each core comprising 4 hardware threads (total of 8x4 = 32 hardware threads on processor), which are sufficient to investigate the performance and scalability improvements, if any, of hardware scheduling and synchronization. Each 4 hardware threads share one execution pipeline that executes an instruction from each thread every cycle. A thread that is waiting for a long-latency instruction (a load instruction, for example) is switched off (does not issue instructions to the shared pipeline) until the instruction completes. In the meantime, the pipeline continues issuing instructions from other threads. The T1 architecture exemplifies the use of fine-grained parallelism to tolerate the main memory latency by running multiple threads simultaneously keeping the pipeline busy even when some threads are awaiting long latency instructions. The T1 processor supports a register file comprising 256, 8-byte registers that form 8-SPARC windows and a set of 8 global registers (g0 to g7). Since we are only concerned with light-weight threads, we only use the global registers (g0 to g7) to a hold thread's context. Register g0 is always 0, so on a context switch, it is not saved. The thread context comprises g1 to g7 (each register is 8 bytes) and another 8-byte register representing the program counter and condition code register. The simulation clock frequency used is 1 GHz. We quantify the execu-

tion time based on the number of clock cycles, therefore, the clock frequency does not matter.

6 BENCHMARKS

The required simulation time to execute a set of standard benchmarks on the RTL is very high. Therefore, we developed a set of T1-SPARC-Assembly based benchmarks to quantify the performance and scalability of hardware and software scheduling. Following is a description of these benchmarks: 1] Hardware thread creation and scheduling benchmark (HARD-TCSB). The spawned threads are the scheduler's workload, so we will refer to the spawned threads as workloads. A single thread in the HARD-TCSB benchmark is used to spawn the workloads. The HARD-TCSB supports these Workloads: a] Workload-A: each thread comprises multiple setx instructions (used to set any register in the register file to a chosen value), arithmetic/logical instructions, thread terminate instruction (devised to support the proposed scheduler, it sets the hardware thread to idle at the end of each thread to make it available for use by other threads) and a branch instruction. We define the work unit by how many times these instructions are executed inside a single thread. A work unit at 1 executes these instructions only one time; a work unit at 10 executes these instructions 10 times and so forth. We use work units 1, 10, and 50 to collect performance data. The instruction counts for these works units are 10, 100, and 500 instructions, respectively. A fine-grained thread is a thread with a number of instructions less than 1000 instructions [23]. Therefore, our choice of the thread's instruction count conservatively meets the definition of a fine-grained thread. b] Workload-B: each thread comprises two conditional instructions (load conditional (ldxcfe), store conditional (stxcef)) accessing the same address, arithmetic/logical instructions, a branch instruction, and a thread terminate instruction. Only work unit 50 is used, which generates 500 instructions in each spawned thread. Workload-B is used to study the impact of thread blocking on scalability. c] Workload-C: Workload-C is similar to Workload-B except we replace the conditional load/store instructions with non-conditional (normal) load/store instructions. Workload-C is used for scalability comparison against Workload-B. 2] Software thread creation and scheduling benchmark (SOFT-TCSB). This benchmark executes the same instructions as in the HARD-TCSB Workload-A benchmark except software scheduling rather than hardware scheduling is utilized. The SOFT-TCSB models the multi-threaded shepherds (MTS) scheduling policy since it outperformed other scheduling policies [11]. We ran the MTS on the modified T1 processor supporting 8 cores, with 4 hardware threads per core. The MTS implements the Shared Queue Policy and it allocates a LIFO for all the cores in a processor. A *Worker* is mapped to each hardware thread in each core. The *Worker*, a POSIX thread (Pthread), processes the Qthreads waiting in the LIFO; the Qthreads API is a portable abstraction that provides basic light-weight thread control and synchronization primitives that enables the development of large-scale multi-threading applications on commodity architectures developed at Sandia National Labs [9]. Using a LIFO enhances the cache locality. That is, parent threads share the data with their children, so allowing the children to run first using a LIFO policy increases the chance that the data, pertaining to the parent threads, is still in the local caches when the child threads access it. Utilizing one *Worker* per hardware thread in each core enhances load balance since they fetch the Qthreads from the LIFO on demand. Accessing the LIFO is controlled by a coarse-grained (CGS) lock (e.g., semaphore); the *Worker* can either succeed or fail when trying to acquire the lock. On a success, the *Worker* continues executing and on a failure, the *Worker* is blocked awaiting the CGS lock to become available. The task placement is local, therefore, each processor assigns its newly-created Qthreads to the corresponding LIFO. Load balance among processors is achieved by work stealing from another LIFO. When a *Worker* detects that no work is left in its respective

LIFO, it probes the other LIFOs for work starting with the LIFO that has the nearest ID to its corresponding LIFO. Work is stolen using a First-In-First-Out (FIFO) policy from the victim LIFO. It is expected that the oldest threads benefit the least from the cache locality in the relevant processor. Therefore, stealing the oldest threads from the processor's LIFO allows the newer threads to run on the processor that created them, thereby enhancing cache locality. If a local processor's *Worker* accesses the LIFO and does not find any work, then it continues polling its LIFO until work becomes available. The *Worker* releases the CGS lock before consecutive accesses to the LIFO to give a turn to other *Workers* to access it.

7 RESULTS

7.1 HARD-TCSB benchmark results

Figure 2 shows the results of hardware scheduling when executing the HARD-TCSB benchmark, Workload-A, the number of spawned threads is 1000 (1000 threads are enough to overflow the RTU's queue), the size of the DTB, RTU_LIFO, and RTU_FILLQ is 64 entries each (see subsection 4.5), work units 1, 10 and 50 are used. For work unit 1, the lack of scalability beyond 8 hardware threads is due to the parent thread inability to create threads fast enough (it takes time to compute and populate the thread's context registers due to the load instructions long memory sub-system access latency) to utilize all hardware threads simultaneously and keep up with the low average execution time of each thread (120 CPU clock cycles since it executes 10 instructions only) and the fast scheduler. Therefore, a load balance issue shows up since most of the threads run only on core0 and core1 while other cores remain idle most of the execution time. The graph shows clear scalability for all workloads between 4 (core0 only) and 8 (core0 and core1) hardware threads because core0 and core1 run most of the threads (768 threads run on core0 and core1, the remaining 232 threads run on the other cores). As the work unit increases to 10 units, the scalability improves significantly due to distributing the workload almost evenly across all cores (better load balance). The average execution time of a spawned thread is around 1000 CPU clock cycles at work unit 10 and each thread executes 100 instructions. The scalability is visible at work unit 50 where the average execution time of a spawned thread is around 4500 CPU clock cycles and each thread executes 500 instructions. The scalability at work unit 50 is better than work unit 10 based on slope comparison.

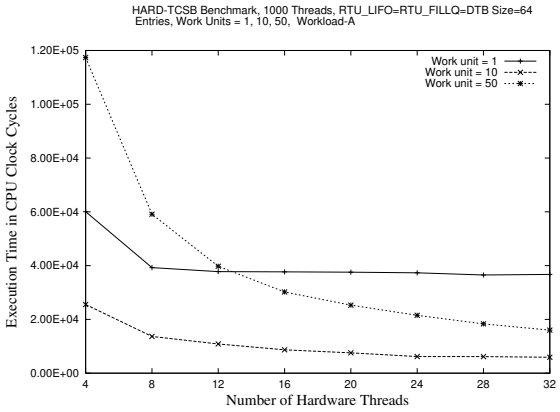

Figure 2. Scalability of HARD-TCSB benchmark, Workload-A, 1000 threads spawned, RTU_LIFO = RTU_FILLQ = 64 entries, DTB = 64 entries, work units = 1, 10 and 50. The Y-axis is scaled down by a factor of 10 for work units 10 and 50.

As the work unit increases, the number of times the RTU_LIFO becomes full also increases, since the hardware threads become less idle, which causes the newly-created threads to wait longer in the RTU_LIFO causing its overflow. The scalability is impacted at work units 10, 50 due to calling the recruiting thread (RT) 7, 13 times, and RTU overflow trap 7, 13 times, respectively. Also, the number of blocked threads is 0 since this benchmark does not utilize conditional instructions, therefore, thread blocking (deferring) does not occur. Also, the deferred thread buffer full count is 0 since no threads are blocked (deferred).
Figure 3 shows the results of hardware scheduling when executing HARD-TCSB benchmark, Workload-B, Workload-C, the

number of spawned threads is 1000 threads (enough to overflow the RTU and cause thread blocking on conditional instructions).

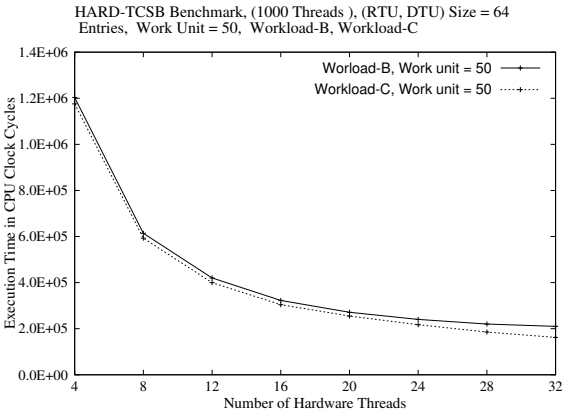

Figure 3. Scalability of HARD-TCSB benchmark, Workload-B, Workload-C, 1000 threads spawned, RTU_LIFO = RTU_FILLQ = 64 entries, DTB = 64 entries, work units = 50.

The size of the DTB is 64 entries, the size of RTU_LIFO and RTU_FILLQ is 64 entries each (see subsection 4.5). Work unit 50 is used since it shows better scalability than work units 1, 10. The scalability persists though thread blocking occurs 8, 19, 29, 39, 68, 71, 77, 83 times for hardware threads 4, 8, 12, 16, 20, 24, 28, 32, consecutively, and 1 call to the recruiting thread (RT) occurs at hardware threads 28, 32. This shows that hardware thread blocking can mitigate scalability degradation due to the fast rate by which the hardware scheduler can block a thread and save it in the DTB. Workload-B scalability is close to Workload-C as shown in Figure 3 though it becomes weaker between hardware threads 24, 32 due to calling the RT.

7.2 SOFT-TCSB benchmark results

Figures 4 and 5 show the results from software scheduling when executing the SOFT-TCSB benchmark, Workload-A, the number of spawned threads is 1000, work units 1, 10, 50, 400 and 600 are used (scalability shows up at around work unit 400). For work units 1, 10 and 50, there is not any scalability. Most of the execution time is consumed by the *Workers* rather than the threads whose instruction count is small, 10, 100, and 500 instructions for work units 1, 10, and 50, respectively. The *Workers* issue a high number of loads/stores to the memory sub-system to manage the work LIFO increasing the scheduling time, thereby weakening the scalability. The scalability starts to show up at around work unit 400 though it is weak after 20 hardware threads (thread executes 4000 instructions, average thread execution time 38001 CPU clock cycles). We also study the scalability at work unit 600 (thread executes 6000 instructions, average thread execution time 58859 CPU clock cycles) to see how far the scalability improves as the work unit is increased. The scalability becomes visible at work unit 600.

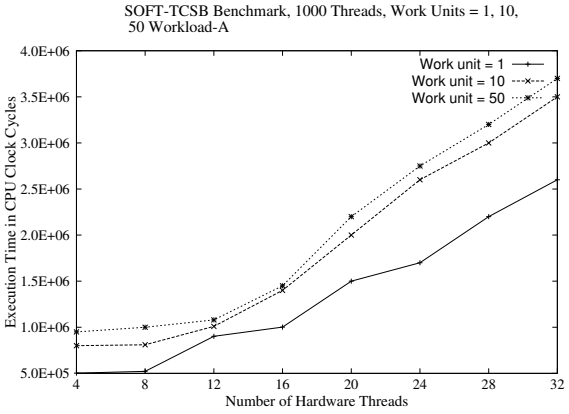

Figure 4. Scalability of SOFT-TCSB benchmark, Workload-A, 1000 threads spawned, work units 1, 10 and 50.

7.3 Execution time comparison

To estimate the average speedup of hardware scheduling when compared with software scheduling, we chose the execution time results of HARD-TCSB benchmark (1000 Threads, RTU_FILLQ = 64 Entries, RTU_LIFO = 64 Entries, DTB Size = 64 Entries, Work Unit 50, Workload-A) and SOFT-TCSB benchmark (1000 Threads, Work Units = 50, Workload-A). The simulation results based on 8 cores

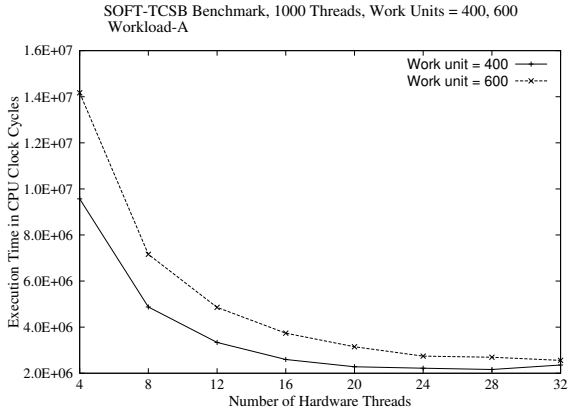

Figure 5. Scalability of SOFT-TCSB benchmark, Workload-A, 1000 threads spawned, work units 400 and 600

(32 hardware threads) are used since the 8 cores produce the maximum computational capacity of the processor. Based on the comparing execution times, hardware scheduling is 22.76 times faster than software scheduling. The performance of our proposed scheduler is expected to be higher than Carbon since the scheduling is fully managed by hardware. The software *Worker* that typically handles thread scheduling in Carbon is replaced by hardware in our scheduler. Also, we introduce hardware-based FGS that is not supported by Carbon. Carbon was compared with software scheduling and we did the same since extracting any performance data from a modeled Carbon would have been implementation dependent rendering the comparison inaccurate. Intel used an in-house simulator to model Carbon which we could not access.

7.4 Results summary

The hardware scheduler's scalability of fine-grained threads (instruction count is 100 and above) is clear and it persists even when FGS thread blocking occurs, which is due to the scheduler's ability to block threads fast. The software scheduler exhibits no scalability when executing fine-grained threads compared with hardware scheduling, the scalability starts showing up when the count of the thread's instructions is increased to 4000 instructions (coarse-grained thread). The proposed hardware scheduling outperforms software scheduling by 22.76 times.

8 CONCLUSIONS

In this work, we propose the use of hardware scheduling and synchronization to overcome the scalability challenge of light-weight, fine-grained multi-threading. The proposed hardware scheduler is capable of scheduling fine-grained threads whose instruction count is 100 instructions (thread average execution time is 1000 CPU clock cycles) with tangible scalability though weakened by the calls to the RT and RTU/DTU overflow traps. Our proposed hardware scheduler performs 22.76 times faster than software scheduling. The silicon area occupied by the hardware scheduler depends on the size of its queues. As the size increases, it becomes less likely to resort to software to backup/restore threads to/from memory subsystem, therefore, the performance becomes better. The hardware scheduler's RTU and DTU require 68 Kbytes of silicon area to accommodate 64 threads and 1.0625 Mbytes to accommodate 1000 threads. The TSAU requires what is equivalent to 8 Kbytes of memory at worst-case estimation, which can be easily contained in terms of silicon area and power budgets. The size of the queues in the RTU and DTU must be selected based on implementation choices that balance performance against silicon and power budgets.

REFERENCES

[1] D. Sanchez, R. M. Yoo, and C. Kozyrakis, "Flexible architectural support for fine-grain scheduling," in Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems, ASPLOS XV, (New York, NY, USA), pp. 311–322, ACM, 2010.

[2] T. Gilmanov, M. Anderson, M. Brodowicz, and T. Sterling, "Application characteristics of many-tasking execution models," in the 19th International Conference on Parallel and Distributed Processing Techniques and Applications, (Las Vegas, USA), July, 2013.

[3] M. Anderson, M. Brodowicz, A. Kulkarni, and T. Sterling, "Performance modeling of gyrokinetic toroidal simulations for a many-tasking runtime system," in the 4th international workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems at SC13, (Denver, CO USA), Nov, 2013.

[4] H. Fadhil, Y. Mohammed, "Parallelizing RSA Algorithm on Multicore CPU and GPU," International Journal of Computer Applications, vol. 87, 02 2014.

[5] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: Architectural support for fine-grained parallelism on chip multiprocessors," in Proceedings of the 34th Annual International Symposium on Computer Architecture, ISCA '07, (New York, NY, USA), pp. 162–173, ACM, 2007.

[6] D. K. Gauba, "Hardware implementation of real time operating system's thread context switch," Master's thesis, Boise State University, 08/2010.

[7] F. M. David, J. C. Carlyle, and R. H. Campbell, "Context switch overheads for linux on arm platforms," in Proceedings of the 2007 Workshop on Experimental Computer Science, ExpCS '07, (New York, NY, USA), ACM, 2007.

[8] D. S. Martin, Hardware and Software Techniques for Scalable Thousand-core Systems. PhD thesis, Stanford University, August 2012.

[9] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An api for programming with millions of lightweight threads," in IPDPS, pp. 1–8, IEEE, 2008.

[10] B. W. Barrett, J. W. Berry, R. C. Murphy, and K. B. Wheeler, "Implementing a portable multi-threaded graph library: The mtgl on qthreads," in Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing, IPDPS '09, (Washington, DC, USA), pp. 1–8, IEEE Computer Society, 2009.

[11] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, and J. F. Prins, "Scheduling task parallelism on multi-socket multicore systems," in Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers, ROSS '11, (New York, NY, USA), pp. 49–56, ACM, 2011.

[12] H. Kaiser, M. Brodowicz, and T. Sterling, "Parallex an advanced parallel execution model for scaling-impaired applications," in Proceedings of the 2009 International Conference on Parallel Processing Workshops, ICPPW '09, (Washington, DC, USA), pp. 394–401, IEEE Computer Society, 2009.

[13] HPX. http://stellar-group.github.io/hpx/docs/schedulers.html, 2018/04/04.

[14] R. D. Blumofe, C. F. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," Journal of Parallel and Distributed Computing, vol. 37, 02 1999.

[15] M. A. Bender and M. O. Rabin, "Scheduling cilk multithreaded parallel programs on processors of different speeds," in Proceedings of the 12th Annual Symposium on Parallel Algorithms and Architectures, pp. 13–21, 2000.

[16] Intel Corporation, "Intel Cilk Plus." https://software.intel.com/en-us/cpp-compiler-18.0-developer-guide-and-reference-intel-cilk-plus, 2018/04/04.

[17] R. Alverson, D. Callahan, D. Cummings, A. Porterfield, and B. Smith, "The tera computer system," in International Conference on Supercomputing, pp. 1–6, June 1990.

[18] G. Alverson, R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. Smith, "Exploiting heterogeneous parallelism on a multithreaded multiprocessor," in Proceedings of the 6th International Conference on Supercomputing, ICS '92, (New York, NY, USA), pp. 188–197, ACM, 1992.

[19] A. Snavely, L. Carter, J. Boisseau, A. Majumdar, K. S. Gatlin, N. Mitchell, J. Feo, and B. Koblenz, "Multi-processor performance on the tera mta," in Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, SC '98, (Washington, DC, USA), pp. 1–8, IEEE Computer Society, 1998.

[20] D. Kranz, B. hong Lim, D. Yeung, and A. Agarwal, "Low-cost support for fine-grain synchronization in multiprocessors," Multithreaded Computer Architecture: A Summary of the State of the Art, chapter 7, pp. 139–166, 1994.

[21] A. Agarwal, D. Chaiken, D. Kranz, J. Kubiatowicz, K. Kurihara, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung, "The mit alewife machine: A large-scale distributed-memory multiprocessor," in Proceedings of Workshop on Scalable Shared Memory Multiprocessors, Kluwer Academic Publishers, 1991.

[22] M. Fillo, S. Keckler, W. Dally, N. Carter, A. Chang, Y. Gurevich, and W. Lee, "The m-machine multicomputer," in Microarchitecture,

1995., Proceedings of the 28th Annual International Symposium on Microarchitecture, pp. 146–156, Nov 1995.

[23] S. W. Keckler, W. J. Dally, D. Maskit, N. P. Carter, A. Chang, and W. S. Lee, "Exploiting fine-grain thread level parallelism on the mit multi-alu processor," in 25TH Annual International Symposium on Computer Architecture, pp. 306–317, 1998.

[24] Cray Inc., "Poduct timeline." https://www.cray.com/sites/default/files/resources/CrayTimeline.pdf, 2018/04/04.

[25] A. Kopser and D. Vollrath, "Overview of the next generation cray xmt," in Cray User Group 2011 Proceedings 1 of 10, Cray Inc., 2011.

[26] A. Agarwal, R. Bianchini, D. Chaikeny, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Limz, K. Mackenzie, and D. Yeung, "The mit alewife machine: Architecture and performance," in ISCA '95 Proceedings of the 22nd annual international symposium on Computer architecture Pages 2-13, S. Margherita Ligure, Italy June 22-24, 1995.

[27] MIT, "Alewife machine." http://groups.csail.mit.edu/cag/alewife/, 2018/04/04.