# Pressure-Driven Hardware Managed Thread Concurrency for Irregular Applications

John D. Leidel
Texas Tech University
Box 43104
Lubbock, Texas 79409-3104
john.leidel@ttu.edu

Xi Wang
Texas Tech University
Box 43104
Lubbock, Texas 79409-3104
xi.wang@ttu.edu

Yong Chen
Texas Tech University
Box 43104
Lubbock, Texas 79409-3104
yong.chen@ttu.edu

## ABSTRACT

Given the increasing importance of efficient data intensive computing, we find that modern processor designs are not well suited to the irregular memory access patterns found in these algorithms. This research focuses on mapping the compiler's instruction cost scheduling logic to hardware managed concurrency controls in order to minimize pipeline stalls. In this manner, the hardware modules managing the low-latency thread concurrency can be directly understood by modern compilers. We introduce a thread context switching method that is managed directly via a set of hardware-based mechanisms that are coupled to the compiler instruction scheduler. As individual instructions from a thread execute, their respective cost is accumulated into a control register. Once the register reaches a pre-determined saturation point, the thread is forced to context switch. We evaluate the performance benefits of our approach using a series of 24 benchmarks that exhibit performance acceleration of up to 14.6X.

## CCS CONCEPTS

• **Computer systems organization** → **Multiple instruction, multiple data**; *Multicore architectures*; • **Software and its engineering** → *Source code generation*; • **Computing methodologies** → Shared memory algorithms; Massively parallel algorithms;

## KEYWORDS

Data intensive computing; irregular algorithms; thread concurrency; context switching

**ACM Reference format:**
John D. Leidel, Xi Wang, and Yong Chen. 2017. Pressure-Driven Hardware Managed Thread Concurrency for Irregular Applications. In *Proceedings of IA$^3$'17: Seventh Workshop on Irregular Applications: Architectures and Algorithms, Denver, CO, USA, November 12–17, 2017 (IA$^3$'17),* 8 pages.
DOI: 10.1145/3149704.3149705

## 1 INTRODUCTION

In recent years, data intensive computing has garnered increased research and development attention in commercial and academic settings. The need for highly parallel algorithms and applications that support machine learning, graph theory, artificial intelligence and raw data analytics have spurred a flurry of research into novel hardware and software methodologies to accelerate these classes of applications.

One of the driving issues in coping with parallel, data intensive computing algorithms and applications is the inherent irregularity in the memory access patterns. These irregular memory access patterns that often fall outside of traditional multi-level caching hierarchies induce increased memory latencies that often stall the execution pipelines of the core. These pipeline stalls prevent efficient execution, and thus increase runtime of the algorithms.

Multi-threading and multi-tasking have been utilized in previous architectures [13] [4] [17] [32] to cope with these periods of latency by overlapping the memory accesses and execution of adjacent threads or tasks. However, the operating system and system software overhead required to manage this thread/task state, as well as to perform the fundamental context switch operations (context save and restore), consumes a significant number of execution cycles. As a result, these purely software-driven methods often conflict with the fundamental goal of efficiently filling the hardware's execution pipeline.

In this work, we present a novel methodology that couples an in-order, RISC execution pipeline to a low-latency context switching mechanism that stands to dramatically increase the efficiency and, subsequently, the throughput of thread or task-parallel data intensive computing algorithms and applications. At the core of our approach is a novel hardware-driven method that monitors the *pressure* a thread or task places on the execution pipeline by assigning costs to individual opcode classes early in the instruction decode stage. When threads or tasks exceed a maximum cost threshold, they are forced to yield the execution pipeline to adjacent threads, thus preventing significant pipeline stalls due to latent execution events.

The contributions of this research are as follows. We present a methodology for constructing in-order RISC execution pipelines that provides low latency context switching mechanisms between in-situ, hardware managed threads or tasks by monitoring the pressure placed on the pipeline by the executing instruction stack. The methodology is applicable to general architectures, and we demonstrate a sample implementation of the GoblinCore-64 methodology using a classic five-stage, in-order execution pipeline based upon the RISC-V Rocket micro architecture [31]. The end result exhibits up to a 14.6X performance increase on a set of twenty-four well-formed benchmark applications.


*IA$^3$'17, Denver, CO, USA*

DOI: 10.1145/3149704.3149705

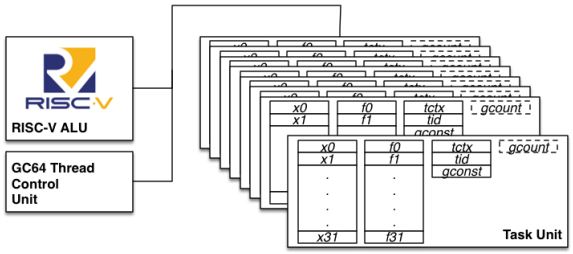


**Figure 1: GC64 Task Processor Architecture**

The remainder of this paper is organized as follows. Section 2 presents previous work in the area of hardware-driven context switching. Section 3 presents our methodology for performing hardware-driven, low latency context switching. Section 4 presents an evaluation of our methodology, including relevant application benchmarks and we conclude with a summary of this work in Section 5.

## 2 PREVIOUS WORK

Several existing system architectures have explored numerous low-latency context switch mechanisms. One of the seminal architectures to employ a low-latency context switch technique was the Tera MTA-2, which evolved into the Cray XMT/XMT-2 [6] [25] [5]. The MTA/XMT series of system architectures employed a methodology to store the entire state of thread, or in MTA/XMT vernacular, a *stream*, in active processor state. For each execution core, the processor could store the register and state for up to 128 streams. Further, additional stream states could be stored in memory for highly divergent, parallel applications. A cooperation between the hardware infrastructure and the software runtime infrastructure maintained the state of the streams for parallel applications [5].

Additionally, the Convey MX-100 CHOMP instruction set and micro architecture employed overlapping thread state attached to a RISC pipeline as the basis for its high performance context switching [20]. Each RISC pipeline employs 64 *thread cache units* that represent the active thread state associated with a given core. Threads are context switched using a single cycle cadence and time division multiplexed with one another such that threads would receive ample execution time without starving the pipeline. Any time a thread cache unit flags a register hazard due to an outstanding memory operation or instruction cache fill, the context switch mechanisms would negate the associated thread from execution until the hazards are clear. The CHOMP ISA also employs a unique instruction format that permits the user and/or compiler to define any instruction to force a context switch based upon the inherent knowledge of the forthcoming instruction stream. In this manner, the MX-100 has a much more flexible scheduling mechanism than the MTA/XMT. However, given that the MX-100 was implemented using FPGA, it suffers from poor scalar performance.

The Sun UltraSPARC Niagara series of core processors employed a method near the intersection of the two aforementioned approaches [18] [26]. The Niagara series employed the SPARC instruction set architecture as the basis for server-class core processors for small and large shared memory platforms. The Niagara implemented a similar barrel processor technique as the original MTA/XMT, yet with four threads per core and a RISC pipeline. In addition, the Niagara employed a crossbar memory interface similar to that in the Convey MX-100 that connected all eight cores and two levels of cache to multiple memory controllers. The original Niagara-T1 was limited in its floating point throughput as a single floating point pipeline was shared amongst all eight cores. The Niagara-T2, however, implemented unique floating point pipelines for each core.

## 3 HARDWARE MANAGED THREAD CONCURRENCY

### 3.1 Thread Context Management

The proposed pressure-driven, hardware managed thread concurrency and context switching features are generally applicable, and are also designed to be coupled to the GoblinCore-64 micro architecture [22]. The GoblinCore-64 (GC64) micro and system architecture is designed to address data intensive computing algorithms and applications by providing unique extensions to simple, in-order RISC-style computing cores that promote concurrency and memory bandwidth utilization. In many cases, these workloads exhibit non-linear, random or irregular memory access patterns. As a result, traditional multi-level cache approaches are insufficient to prevent a high percentage of pipeline stalls when accessing memory in irregular patterns.

The GoblinCore-64 micro architecture provides a set of hierarchical hardware modules that couple 64-bit, in-order RISC-V cores [7] [31] [30] to a set of latency hiding and concurrency features. The hierarchical hardware modules are gathered into a system on chip (SoC) device and connected to one or more Hybrid Memory Cube (HMC) devices with a weakly ordered memory subsystem [3].

The GC64 micro architecture uniquely encapsulates threads or tasks into divisible units in hardware. These *Task Units* are designed to represent all of the necessary register and control states required to represent a thread or task in hardware. In this manner, a thread or task's context is saved to this internal storage following a context switch event. No memory operations are otherwise required to service a context switch event. As a result, context switch operations can be performed in a single cycle. Figure 1 depicts all of the RISC-V register states and unique GC64 register states encapsulated in a single task unit, including general purpose, floating point and all control register states.

The next level within the GC64 execution hierarchy is the *Task Processor*, or *Task Proc*. As shown in Figure 1, each task proc contains a single core execution unit, an associated thread/task control unit and a number of task units. Much like previous system architectures designed for data intensive computing [25] [6] [13] [4] [17] [20], GoblinCore-64 permits rapid context switching between the threads or tasks encapsulated in the task units directly attached to a given task processor. Task processors are not permitted to utilize task units from adjacent processors during context switch events. This permits us to minimize the on-chip routing required to build the additional control paths for the task unit and thread control unit management.

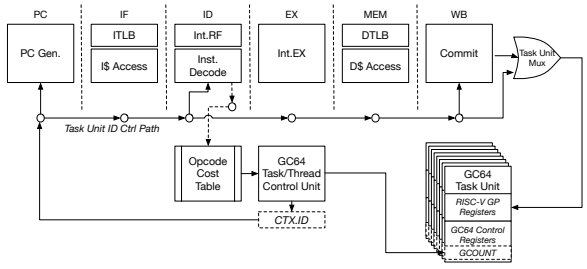


**Figure 2: Thread Context Pipeline**

## 3.2 Pressure-Driven Thread Context Switching

In order to support the unique, low-latency context switch infrastructure, additional control paths are designed to account for the replicated register state present in each task unit's storage, the thread/task control unit state management and the necessary facilities to maintain the identifiers for the active task in the pipeline. Figure 2 depicts these additional mechanisms as attached to the basic RISC-V in-order pipeline model. The pipeline and task control unit maintain an internal register, *CTX.ID*, to maintain the identifier of the task unit currently enabled for execution. During the first stage of the pipeline execution when the program counter is constructed for the target instruction, the CTX.ID is utilized in order to select the correct instruction from the target task's instruction stack. This identifier is then carried forth throughout the pipeline such that adjacent stages may perform register reads and writes to the correct task unit's register file. This selection is performed via a simple mux (Task Unit Mux) during the write-back stage of the pipeline.

In addition to the basic control state in the pipeline, the instruction decode (ID) phase utilizes the task unit identifier to calculate the pressure from the respective task unit's execution. Once the ID phase cracks the instruction payload's opcode, the opcode value is used to drive a table lookup in the Opcode Cost Table. The values stored in this table, one per opcode, represent the relative cost of the respective instruction toward the total execution of the task unit. The task control unit then utilizes this value and the CTX.ID value to perform a summation of the respective task unit's pressure for this cycle. This value is subsequently stored in the *GCOUNT* register for the target task unit.

At this point, the task control unit permits the instruction to continue its execution through the pipeline providing that it has not exceeded a pre-determined overflow value. However, if the *GCOUNT* value exceeds the maximum pressure threshold, the task control unit forces the task unit to yield its state to an adjacent task unit. The yield operation stops the execution of the current instruction and rewinds the program counter back to the value of the offending instruction. The task control unit also selects the next available task unit for execution in a round-robin manner and begins its execution at the next value stored in the target task unit's program counter. The value of the CTX.ID internal register is updated with the new value. This process of yielding and selecting a new task requires a single cycle as no memory operations are required to perform context save or restores to/from memory. Further, the thread inducing the context switch sacrifices only three cycles of unused execution; two for the initial IF and ID pipeline stages that are rewound and one for the context switch operation.

## 3.3 Thread Pressure Summation

One interesting aspect of the newly proposed context switching approach is choosing an appropriate cost table to drive the necessary opcode cost summation. Given that the cost table is effectively static, the values stored therein will never be perfectly accurate to the resolution of the target micro architectural implementation. However, we may heuristically derive reasonable values for each of the opcode classes as described below. Our evaluation has confirmed that these values are properly chosen. We present the rationale of these choices below too.

We utilize classic compiler scheduling methodologies as the basis for the cost table derivation [28] [10] [27] [23] [19] [11] [12]. Operations considered to be especially latent such as memory loads are represented with large cost values. Simple operations such as integer arithmetic and combinatorial logic are given values equivalent to the pipeline costs required to execute the respective operation. Intermediate operations such as floating point arithmetic are also given costs equivalent to their respective pipeline depth. In this manner, the hardware scheduling methods can be directly expressed in the compiler code generator. Table 1 presents the full set of values for each of the opcode classes in the standard set of RISC-V extensions.

# 4 EVALUATION

## 4.1 Methodology

In order to evaluate the newly proposed context switching methodology, we utilized the existing RISC-V *Spike* golden-model simulation environment as the basis for our exploration. The Spike simulator was modified to utilize the aforementioned context switching algorithm in a manner that permitted us to define the maximum pressure threshold at simulator boot time. Each of the benchmarks was executed using five configurations. First, we execute a *base* configuration with no hardware-managed context switching. Next, we execute four configurations that represent two and eight threads per core with 64 and 128 maximum pressure thresholds, respectively. The former was derived using a simple dyadic operation and the latter was derived from triadic operations.

For each of the aforementioned configurations, we record the context switch events for each thread and each respective opcode code. This data is later utilized to evaluate the efficacy and distribution of context events. We also record each application's wallclock runtime in order to analyze the relative performance impact of the target configuration.

## 4.2 Workloads

For our evaluation, we utilize a mixture of application workloads and benchmarks the are historically known to make use of dense memory operations and sparse memory operations. These representative applications also span a variety of traditional scientific (physical) simulation, biological simulation, core numerical solvers and analytics workloads. All of the workloads are compiled with optimization using the RISC-V GCC 5.3.0 toolchain. We classify

**Table 1: RISC-V Opcode Cost Table**

| Opcode | Short Name | Cost | Description | Opcode | Short Name | Cost | Description |
|---|---|---|---|---|---|---|---|
| 0x37 | LUI | 2 | Load upper imm | 0x17 | AUIPC | 2 | Add upper imm-PC |
| 0x6F | JAL | 15 | Jump and link | 0x67 | JALR | 15 | Jump&link reg |
| 0x63 | Branches | 10 | Cond branches | 0x7 | Float Loads | 30 | Float loads |
| 0x3 | Loads | 30 | Integer loads | 0x27 | Float Stores | 20 | Float stores |
| 0x23 | Stores | 20 | Integer memory stores | 0x13 | Integer Arith | 5 | Integer arith. |
| 0x33 | Integer Arith | 5 | Reg-reg int arith | 0x1B | Integer Arith | 5 | SEXT int-imm arith |
| 0x3B | Integer Arith | 5 | CSR Arith | 0xF | Fence | 20 | Memory fence |
| 0x73 | System | 30 | System insns | 0x2F | Atomics | 30 | Atomics |
| 0x43 | FMAdd | 20 | FMAdd | 0x47 | FMSub | 20 | FMSub |
| 0x4B | FNMSub | 20 | FMSub & Negate | 0x4F | FNMAdd | 20 | FMAdd & Negate |
| 0x53 | Float Conversion | 15 | Float-Conv | Unknown | Misc | 30 | Misc |

these workloads into four categories, *BOTS*, *GAPBS*, *NASPB* and *MISC*.

The first set of workloads is built using the Barcelona OpenMP Tasks Suite, or BOTS [16]. BOTS utilizes the tasking features from the OpenMP [2] shared memory programming model to demonstrate a series of pathological workloads from a variety of different scientific disciplines. The second set of workloads is provided by the GAP Benchmark Suite [9]. This suite of benchmarks is designed to provide a standardized set of workloads written in C++11 and OpenMP to evaluate a target platform's efficacy on large scale graph processing. The third set of workloads is provided by the NAS Parallel Benchmarks, or NASPB [8]. The OpenMP-C version of the benchmark source is utilized for our tests. We make use of the *A* problem size across all kernels and pseudo-applications. The final set of workloads include the HPCG benchmark [14] [15], LULESH [1] [29] and STREAM [24].

## 4.3 Results

*4.3.1 Context Switch Event Analysis.* Our analysis of the resulting benchmark data consists of three major categories of data. First, we analyze the raw context switch data from each of the aforementioned configurations and respective benchmarks such that we may fully understand where and how the applications induce pressure on the execution pipelines. Next, we analyze the raw performance of each of the benchmarks and benchmark configurations and examine how each performs as we scale the performance and the maximum pressure threshold. Finally, using the scaled performance data, we analyze the peak and average speedup of each application and attempt to make a determination which configuration provides the best general performance against our pathological set of applications.

The first portion of our analysis focuses on understanding the nature of each benchmark's context switch pattern such that we may tune our methodology for a wider array of applications. Figure 3 presents the total number of context switch events for each benchmark and each respective configuration. As expected, when the number of threads is increased, the number of context switch events generally also increases. The only tests that do not follow this trend are the GAPBS BFS tests using 2 threads and the 64 and 128 maximum pressure thresholds, respectively. These two thread per core tests actually exhibit a lower number of context switch events than the respective baseline configurations.

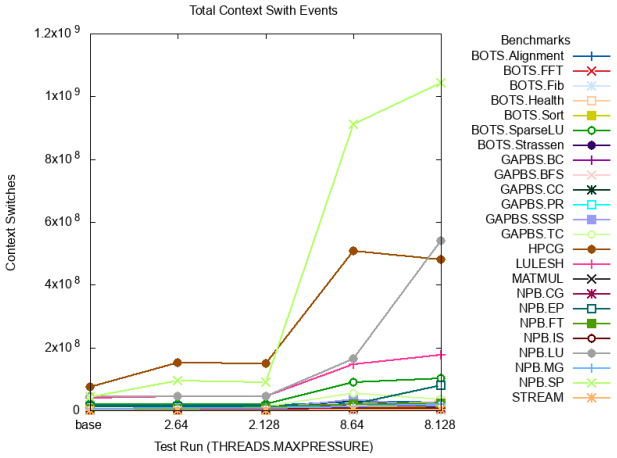


**Figure 3: Total Context Switch Events**

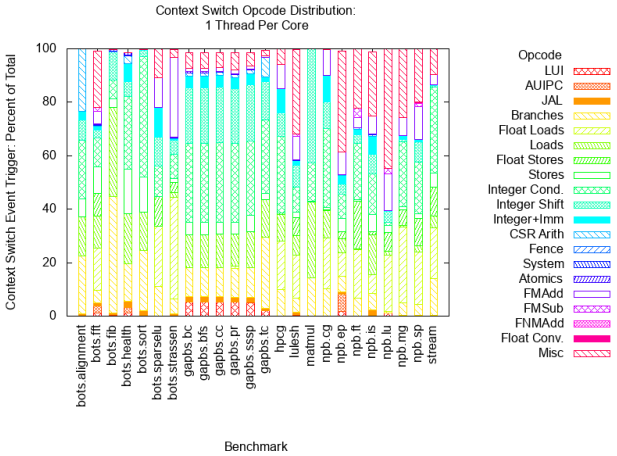


**Figure 4: Baseline Opcode Distribution**

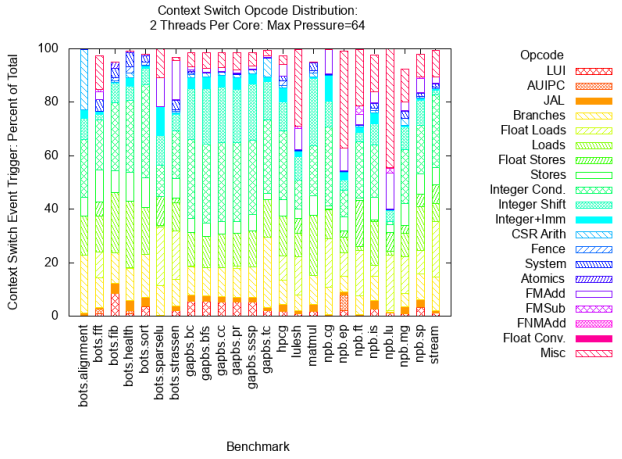


**Figure 5: Two Thread, 64 Max Pressure Opcode Distribution**

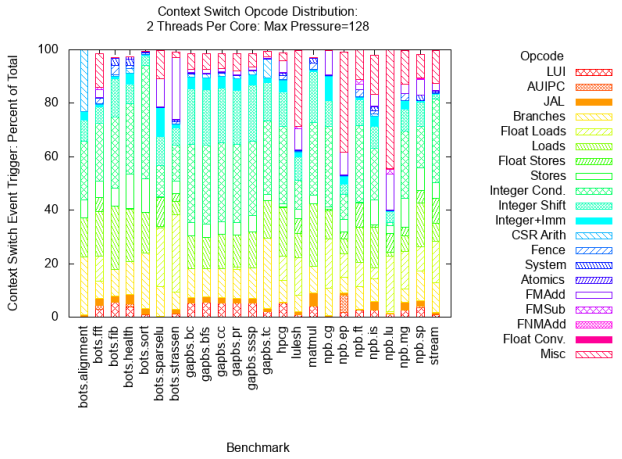


**Figure 6: Two Thread, 128 Max Pressure Opcode Distribution**

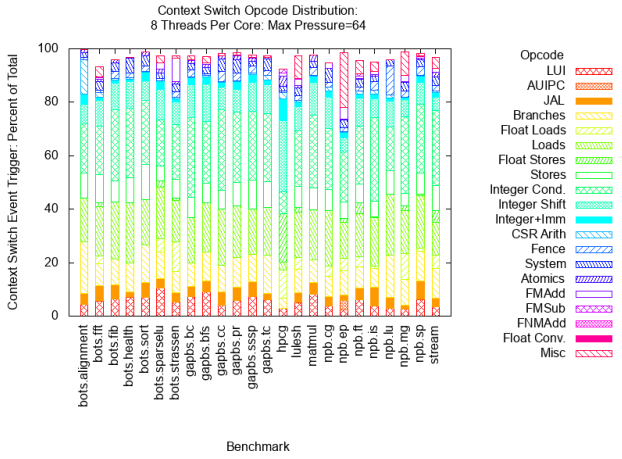


**Figure 7: Eight Thread, 64 Max Pressure Opcode Distribution**

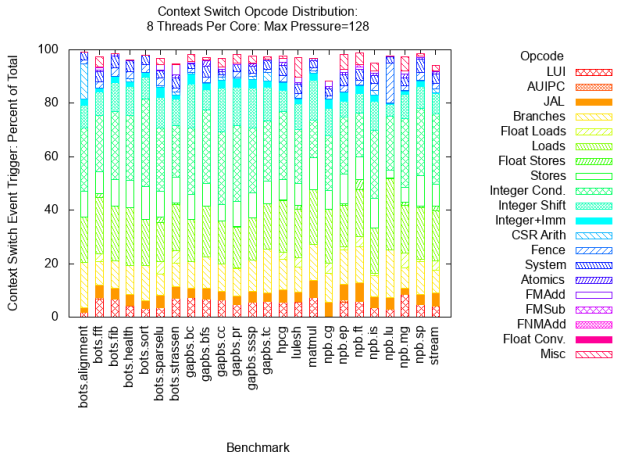


**Figure 8: Eight Thread, 128 Max Pressure Opcode Distribution**

The next stage of our analysis considers the distribution of the various opcode classes as a function of the total context switch events for each respective configuration and benchmark. Figure 4 presents the data for each benchmark when executed using the baseline configuration. We compiled statistics based upon the opcode class that induced the respective context switch event. We can clearly see that for the baseline configuration, a large degree of all context switch events are induced by integer arithmetic opcodes in the 0x13 and 0x33 classes. Given that these context switch events are driven by the operating system kernel, the baseline data cannot be used to derive any specific conclusions on the various evaluated workloads.

Figure 5 and Figure 6 display the opcode context switch statistics for the two thread tests using 64 and 128 as the maximum pressure threshold, respectively. There are a number of interesting trends we find in the two thread results. First, both of the two thread result sets contain relatively similar context switch result statistics as the baseline tests. In this manner, we can observe that, despite having a reasonable performance impact as noted below, utilizing two threads per core does not dramatically change the execution flow of the core. Much like the baseline statistics, we find that many of the tests depict the highest output of context switches falling within one or more integer arithmetic opcode variants. We may theorize that this is due to the cores still executing with a significant number of pipeline stalls due to latent memory requests not returning in time for basic integer arithmetic. The implication being that, for most tests, by scaling the number of threads/tasks per core we should observe a higher degree of pipeline utilization due to higher concurrent overlapping of memory requests and compute opcode classes.

We can also observe several interesting phenomena for the various benchmark applications. Note the significant number of context switch events for the Lulesh, NASPB EP, NASPB FT and NASPB LU codes in the *Float conversion* opcode class. These opcodes represent conversion operations between various precisions of floating point

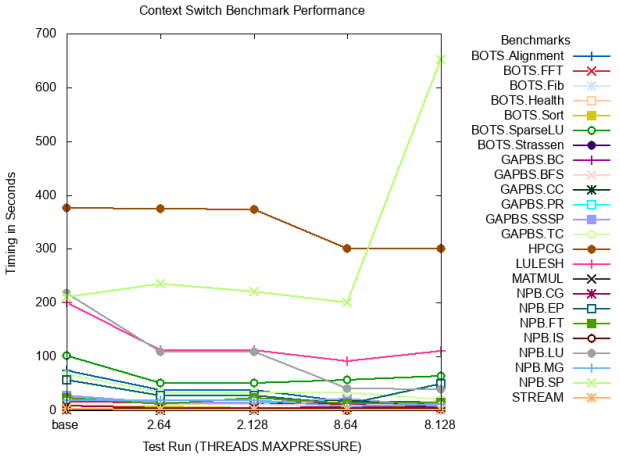


**Figure 9: Application Performance**

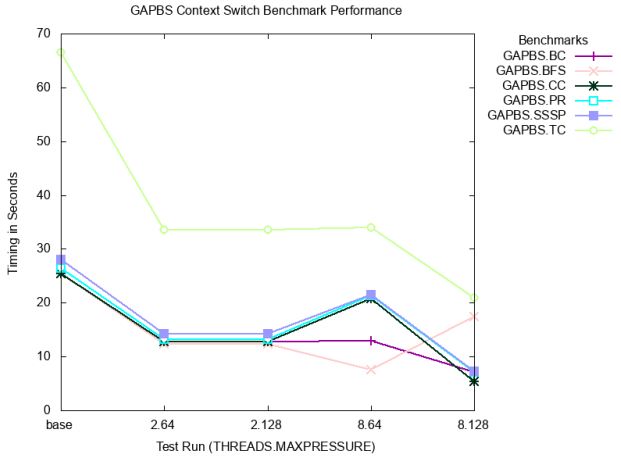


**Figure 10: GAPBS Application Performance**

values and integer values. These operations represent a statistically significant portion of the context switch events and, potentially, the overall runtime of the respective benchmark. Future examination of these applications may conclude that operating in a homogenous precision may serve to improve the overall performance and concurrent throughput of the solvers.

Figure 7 and Figure 8 display the opcode context switch statistics for the eight thread tests using 64 and 128 as the maximum pressure threshold. These context switch statistics drastically depart from those recorded in the baseline and two thread tests. The most noticeable results are found in the NASPB EP benchmark when the maximum pressure threshold is 64. In these results, we measure a stastistically significant number of context switches due to floating point conversion events. Despite the extensive use of floating point arithmetic across many of our target applications, we still see that the vast majority of context switch events are induced via integer-related opcodes. This may imply that as the concurrency per core is scaled, the overall effect of floating point pipeline latency is minimized to a statistically insignificant amount.

*4.3.2 Application Timing Analysis.* In addition to the raw context switch event statistics, we can also analyze the effect of our approach on the actual application timing. Figure 9 presents the application timing data (in seconds) scaled by the individual test configurations (x-axis). The runtime is generally reduced as the number of concurrent threads per core is scaled. However, there are several outliers. The HPCG application dominates the chart with the highest runtime on four of the five test configurations. It does not scale well at two threads per core, but does improve for eight threads per core. Further, the NASPB SP benchmark displays a rather interesting behavior. As the number of threads per core is scaled to two and eight, respectively, the performance undulates over and under the baseline recorded timing. However, the eight thread test using 128 as the maximum pressure threshold is significantly worse. We see that the performance scales from 200.630 seconds for the eight thread, 64 maximum pressure threshold to 651.873 seconds for the final configuration, or a 3.2X increase in timing.

In addition to the aforementioned timing graph, we can also analyze the individual classes of benchmark in order to draw more relevant conclusions on classes of algorithmic constructs. Figure 10 presents the scaled runtime of all the GAPBS benchmark applications. In the GAPBS application runtime results, we can clearly see that our approach as a positive effect on the overall application performance as we scale the thread concurrency per core. For all these graph-related benchmarks, the performance of both two thread per core results exceed the baseline results. However, for the PageRank (PR), Single Source Shortest Path (SSSP) and Connected Components (CC), we see a slight knee in performance for the eight thread per core results when using a maximum pressure threshold of 64. Given the rather irregular data access patterns exhibited in traditional graph algorithms, the maximum pressure threshold is likely not permitting a sufficient number of outstanding memory requests to overlap the latency of the requests with adjacent arithmetic operations. However, with the higher maximum pressure threshold of 128, the benchmark performance returns to a positive scale of application speedup.

In addition, we have also evaluated the classes of applications present in the NASPB suite. Figure 11 presents the application runtime from all the NASPB benchmarks. As mentioned above, we see the obvious outlier in the SP benchmark results. However, the remainder of the NASPB applications exhibit positive application speedups when scaling the number of concurrent threads per core. The NASPB LU benchmark displays the best performance speedup with a 5.53X increase using the eight thread configuration with a maximum pressure threshold of 128. Of the maximum recorded speedups, the SP benchmark displays the lowest with a 1.05X speedup using the eight thread per core configuration with 64 as the maximum pressure threshold. Across all the NASPB applications, we see an average maximum application speedup of 2.722X.

*4.3.3 Application Speedup Analysis.* The final set of evaluations we made is to determine which configuration is ideal for each benchmark as well as the total speedup available. Figure 12 plots each respective benchmark and its best recorded speedup. We can

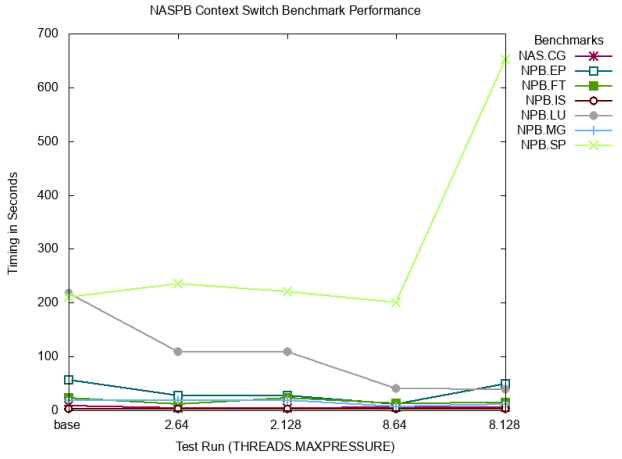


**Figure 11: NASPB Application Performance**

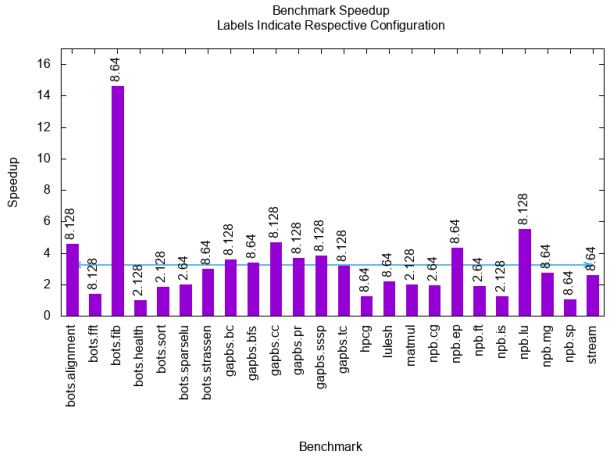


**Figure 12: Application Speedup**

quickly see that the best speedup observed during our tests was the BOTS Fibonacci benchmark with a recorded speedup of 14.6X over the baseline configuration using eight threads and a maximum thread pressure of 64 (8.64). Despite the lowest runtime of any of the target applications, the Fibonacci benchmark exhibited the best performance speedup. This may be due to, in part, the relatively small algorithmic kernel in terms of the number of instructions. This provides excellent instruction cache locality, whereby only the initial kernel entry points pay a penalty to fill. As a result, the total parallel application throughput easily amortizes any potential latency due to instruction cache misses.

The second highest recorded speedup value was for the NASPB LU benchmark with a 5.53X speedup. This was recorded for eight threads using 128 as the maximum thread pressure. Conversely, the lowest speedup was recorded using the BOTS Health benchmark with a maximum speedup of 1.001X. This was recorded using 2 threads and 128 for the maximum thread pressure. Over all the benchmark applications, we observe an average speedup of 3.23X over the baseline configuration depicted as a blue horizontal line in Figure 12.

## 5 CONCLUSION

In this work, we have described a novel hardware managed context switching methodology that couples an in-order, RISC execution pipeline to a low-latency context switching mechanism designed to dramatically increase the efficiency and throughput of thread or task-parallel data intensive computing algorithms and applications. This approach, implemented and evaluated as a part of the RISC-V extension and micro architecture, utilizes a modified 5-stage, in-order pipeline to monitor the decoded instruction opcodes. These opcode values are subsequently utilized to capture the relative pressure a thread or task places on the associated execution unit.

When the thread/task exceeds a pre-determined maximum threshold parameter, the thread or task control unit forces the thread to yield its state and selects an adjacent thread/task for execution. The goal of this newly proposed approach is to utilize the thread/task concurrency to hide the latency to memory across disparate parallel execution units, and thus, garner a high utilization of the hardware's execution pipeline.

In order to evaluate this new approach, we utilized a modified RISC-V simulation infrastructure coupled to a unique data collection mechanism to monitor the context switch statistics and associated performance values (please check out an open-source release under a BSD-style license [21]). We executed 24 unique benchmarks and applications that span a wide array of traditional numerical solvers, scientific computing applications and data intensive algorithms. We evaluated all of the aforementioned benchmark applications using a baseline configuration with one thread per core as well as four configurations that consisted of two and eight thread configurations and utilized 64 and 128 as their maximum thread pressure values, respectively. For these analyses, we decomposed the individual configuration context switch event data into statistical histograms such that we may understand how individual algorithmic constructs induce pressure on a given core.

Finally, we analyzed the results of our execution matrix and found that the eight thread per core configuration, on average, outperformed the two thread and baseline configurations. Further, we recorded a maximum statistical speedup of **14.6X** and an average speedup of **3.2X** over the baseline configuration. This was achieved without increasing the theoretical peak computational resources present in the simulated micro architecture. As a result, we can conclude that our approach has the potential to dramatically increase the potential throughput of data intensive algorithms and applications without additional processing resources.

## REFERENCES


[1] *Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory.* Technical Report LLNL-TR-490254. 1–17 pages.

[2] 2013. *OpenMP Application Program Interface Version 4.0.* Technical Report. OpenMP Architecture Review Board. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[3] 2015. *Hybrid Memory Cube Specification 2.0.* Technical Report. Hybrid Memory Cube Consortium. http://www.hybridmemorycube.org/files/SiteDownloads/HMC-30G-VSR_HMCC_Specification_Rev2.0_Public.pdf

[4] George Almási, Călin Caşcaval, José G. Castaños, Monty Denneau, Derek Lieber, José E. Moreira, and Henry S. Warren, Jr. 2003. Dissecting Cyclops: A Detailed Analysis of a Multithreaded Architecture. *SIGARCH Comput. Archit. News* 31, 1 (March 2003), 26–38. DOI: https://doi.org/10.1145/773365.773369

[5] Gail Alverson, Preston Briggs, Susan Coatney, Simon Kahan, and Richard Korry. 1997. Tera Hardware-software Cooperation. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing (SC '97)*. ACM, New York, NY, USA, 1–16. DOI: https://doi.org/10.1145/509593.509631

[6] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. 1990. The Tera Computer System. *SIGARCH Comput. Archit. News* 18, 3b (June 1990), 1–6. DOI: https://doi.org/10.1145/255129.255132

[7] Krste Asanovic and David A. Patterson. 2014. *Instruction Sets Should Be Free: The Case For RISC-V.* Technical Report UCB/EECS-2014-146. EECS Department, University of California, Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html

[8] D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon. 1992. NAS Parallel Benchmark Results. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing (Supercomputing '92)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 386–393. http://dl.acm.org/citation.cfm?id=147877.148032

[9] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). http://arxiv.org/abs/1508.03619

[10] David Gordon Bradlee. 1991. *Retargetable Instruction Scheduling for Pipelined Processors.* Ph.D. Dissertation. Seattle, WA, USA. UMI Order No. GAX91-31611.

[11] Preston Briggs, Keith D. Cooper, and Linda Torczon. 1994. Improvements to Graph Coloring Register Allocation. *ACM Trans. Program. Lang. Syst.* 16, 3 (May 1994), 428–455. DOI: https://doi.org/10.1145/177492.177575

[12] Keith D. Cooper and Anshuman Dasgupta. 2006. Tailoring Graph-coloring Register Allocation For Runtime Compilation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '06)*. IEEE Computer Society, Washington, DC, USA, 39–49. DOI: https://doi.org/10.1109/CGO.2006.35

[13] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. 2006. Toward a Software Infrastructure for the Cyclops-64 Cellular Architecture. In *Proceedings of the 20th International Symposium on High-Performance Computing in an Advanced Collaborative Environment (HPCS '06)*. IEEE Computer Society, Washington, DC, USA, 9–. DOI: https://doi.org/10.1109/HPCS.2006.48

[14] Jack Dongarra and Michael A. Heroux. 2015. Toward a New Metric for Ranking High Performance Computing Systems. (Mar. 2015).

[15] Jack Dongarra, Michael A. Heroux, and Piotr Luszczek. 2015. *HPCG Benchmark: a New Metric for Ranking High Performance Computing Systems.* Technical Report. University of Tennessee, Sandia National Laboratories.

[16] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. 2009. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *Proceedings of the 2009 International Conference on Parallel Processing (ICPP '09)*. IEEE Computer Society, Washington, DC, USA, 124–131. DOI: https://doi.org/10.1109/ICPP.2009.64

[17] Ge Gan, Xu Wang, Joseph Manzano, and Guang R. Gao. 2009. Tile Percolation: An OpenMP Tile Aware Parallelization Technique for the Cyclops-64 Multicore Processor. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing (Euro-Par '09)*. Springer-Verlag, Berlin, Heidelberg, 839–850. DOI: https://doi.org/10.1007/978-3-642-03869-3_78

[18] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. 2005. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro* 25, 2 (March 2005), 21–29. DOI: https://doi.org/10.1109/MM.2005.35

[19] Chris Lattner and Vikram Adve. 2002. The LLVM instruction set and compilation strategy. *CS Dept., Univ. of Illinois at Urbana-Champaign, Tech. Report UIUCDCS* (2002).

[20] John D Leidel, Kevin Wadleigh, Joe Bolding, Tony Brewer, and Dean Walker. 2012. CHOMP: a framework and instruction set for latency tolerant, massively multithreaded processors. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE, 232–239.

[21] John D. Leidel, Xi Wang, and Yong Chen. GC64-Ctxdata Source Code. http://discl.cs.ttu.edu/gitlab/gc64/gc64-ctxdata. (????). Accessed: 2017-01-17.

[22] John D. Leidel, Xi Wang, and Yong Chen. 2015. *GoblinCore-64: Architectural Specification.* Technical Report. Texas Tech University. http://gc64.org/wp-content/uploads/2015/09/gc64-arch-spec.pdf

[23] Roberto Castañeda Lozano, Mats Carlsson, Frej Drejhammar, and Christian Schulte. 2012. Constraint-based register allocation and instruction scheduling. In *Principles and Practice of Constraint Programming*. Springer, 750–766.

[24] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25.

[25] David Mizell and Kristyn Maschhoff. 2009. Early Experiences with Large-scale Cray XMT Systems. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing (IPDPS '09)*. IEEE Computer Society, Washington, DC, USA, 1–9. DOI: https://doi.org/10.1109/IPDPS.2009.5161108

[26] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. 1996. The Case for a Single-chip Multiprocessor. *SIGOPS Oper. Syst. Rev.* 30, 5 (Sept. 1996), 2–11. DOI: https://doi.org/10.1145/248208.237140

[27] Todd Alan Proebsting. 1992. *Code Generation Techniques.* Ph.D. Dissertation. Madison, WI, USA. UMI Order No. GAX92-31217.

[28] Philip John Schielke. 2000. *Stochastic Instruction Scheduling.* Ph.D. Dissertation. Houston, TX, USA. Advisor(s) Cooper, Keith D. AAI9969315.

[29] Geoffrey Taylor. 1950. The Formation of a Blast Wave by a Very Intense Explosion. II. The Atomic Explosion of 1945. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 201, 1065 (1950), 175–186. DOI: https://doi.org/10.1098/rspa.1950.0050 arXiv:http://rspa.royalsocietypublishing.org/content/201/1065/175.full.pdf

[30] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A. Patterson, and Krste Asanovic. 2015. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.7.* Technical Report UCB/EECS-2015-49. EECS Department, Univ. of California, Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-49.html

[31] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovic. 2014. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0.* Technical Report UCB/EECS-2014-54. EECS Department, University of California, Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html

[32] Kyle Wheeler, Richard Murphy, and Douglas Thain. 2008. Qthreads: an API for Programming with Millions of Lightweight Threads. In *Workshop on Multithreaded Architectures and Applications*. Miami, Florida, USA.