Using Intra-Core Loop-Task Accelerators to Improve the Productivity and Performance of Task-Based Parallel Programs

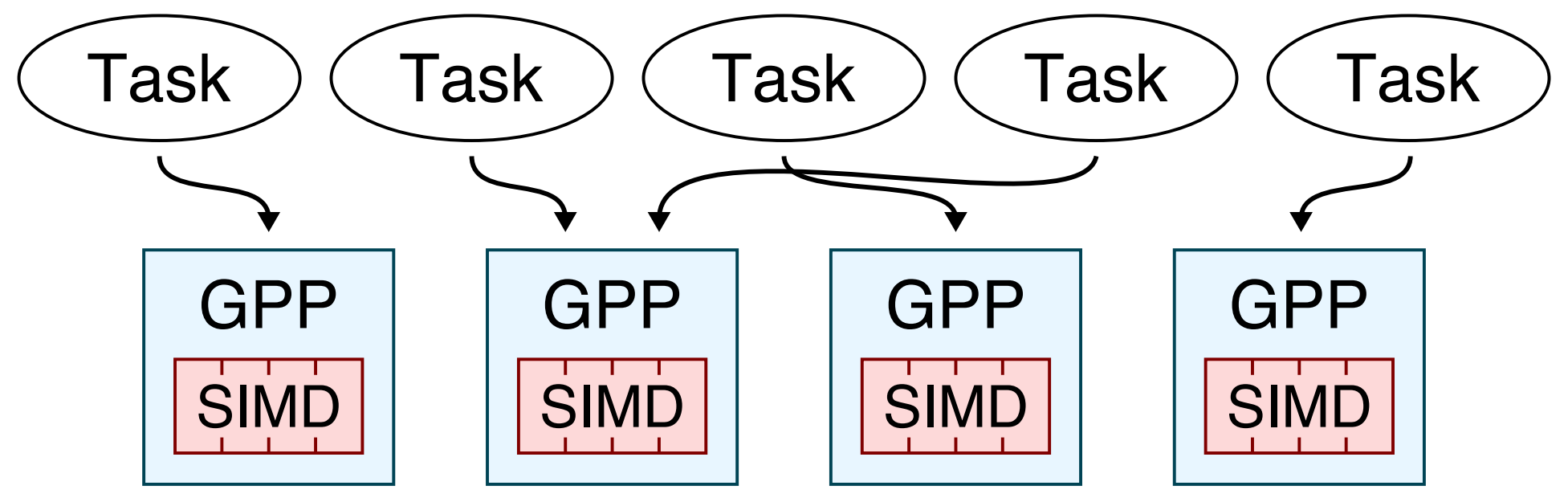
Ji Kim, Shunning Jiang, Christopher Torng
Moyang Wang, Shreesha Srinath, Berkin Ilbeyi
Khalid Al-Hawaj, Christopher Batten
Computer Systems Laboratory
School of Electrical and Computer Engineering, Cornell University

1 Abstract

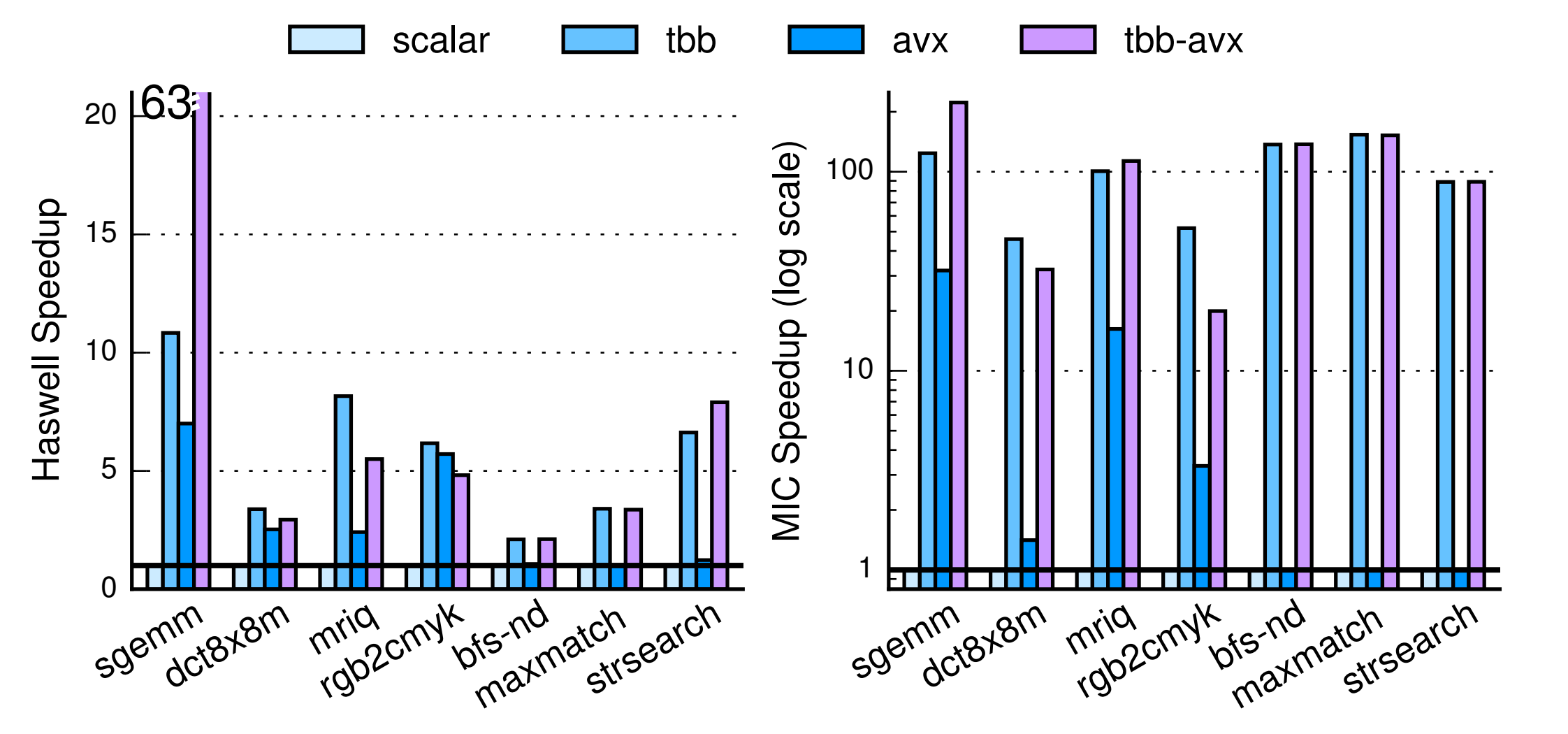
Task-based parallel programming frameworks offer compelling productivity and performance benefits for modern chip multi-processors (CMPs). At the same time, CMPs also provide packed-SIMD units to exploit fine-grain data parallelism. **Two fundamental challenges make using packed-SIMD units with task-parallel programs particularly difficult: (1) the intra-core parallel abstraction gap; and (2) inefficient execution of irregular tasks.** To address these challenges, we propose **augmenting CMPs with intra-core loop-task accelerators (LTAs)**. We introduce a lightweight hint in the instruction set to elegantly encode loop-task execution and an LTA microarchitectural template that can be configured at design time for different amounts of spatial/temporal decoupling to efficiently execute both regular and irregular loop tasks. **Compared to an in-order CMP baseline, CMP+LTA results in an average speedup of 4.2× (1.8× area normalized) and similar energy efficiency. Compared to an out-of-order CMP baseline, CMP+LTA results in an average speedup of 2.3× (1.5× area normalized) and also improves energy efficiency by 3.2×**. Our work suggests augmenting CMPs with lightweight LTAs can improve performance and efficiency on both regular and irregular loop-task parallel programs with minimal software changes.

2 Motivation

Loop-task parallelism is a common parallel pattern usually captured with the parallel for primitive, where a loop task functor is applied to a blocked range. There are two fundamental challenges that make using packed-SIMD units in this loop-task context particularly difficult.



- ▷ **Intra-Core Parallel Abstraction Gap** – Two fundamentally different parallel abstractions reduce productivity: tasks for inter-core parallelism (e.g., TBB) and packed-SIMD for intra-core parallelism (e.g., AVX). Auto-vectorization and explicit vectorization are challenging to perform since tasks can be arbitrarily complex and sizes/alignments are not known at compile time, potentially preventing ”multiplicative speedup”.
- ▷ **Inefficient Execution of Irregular Tasks** – Loop tasks are often complex with nested loops and function calls, data-dependent control flow, indirect memory accesses, and atomic operations compared to the scalar implementation. Converting branches into arithmetic results in wasted work, extra memory alignment and/or data transformations adds overhead, scatter/gather accesses often have much lower throughput, and a less efficient algorithm may be required for vectorization. All of these reasons derive from the fact that the microarchitecture for packed-SIMD extensions is fundamentally designed to excel at executing regular data parallelism as opposed to the more general loop-task parallelism.



In this paper, we propose intra-core loop-task accelerators (LTAs) to address these challenges. A standard runtime schedules tasks across general purpose processors (GPPs) and small software changes enable a GPP to use an LTA to accelerate loop-task execution.

3 LTA Software

Programming Interface

```
// Element-Wise Vector-Vector Addition
// with LTA_PARALLEL_FOR Macro

void vvadd( int dest[], int src0[], int src1[], int size )
{
  LTA_PARALLEL_FOR( 0, size, (dest,src0,src1), ({
    dest[i] = src0[i] + src1[i];
  }));
}
```

The LTA_PARALLEL_FOR macro generates an indirect function call in-place with runtime management code around it. The destination of the indirect function call is the following loop task function.

```
// The Loop-Task Function Generated by the Macro

void task_func( void* a, int start, int end, int step=1 )
{
  args_t* args = static_cast<args_t*>(a);
  int* dest = args->dest;
  int* src0 = args->src0; int* src1 = args->src1;
  for ( int i = start; i < end; i += step )
    dest[i] = src0[i] + src1[i];
}
```

In general, a loop task is a four-tuple of a function pointer, an argument pointer, and the start/end indices of the range.

ISA Extension: jalr.lta

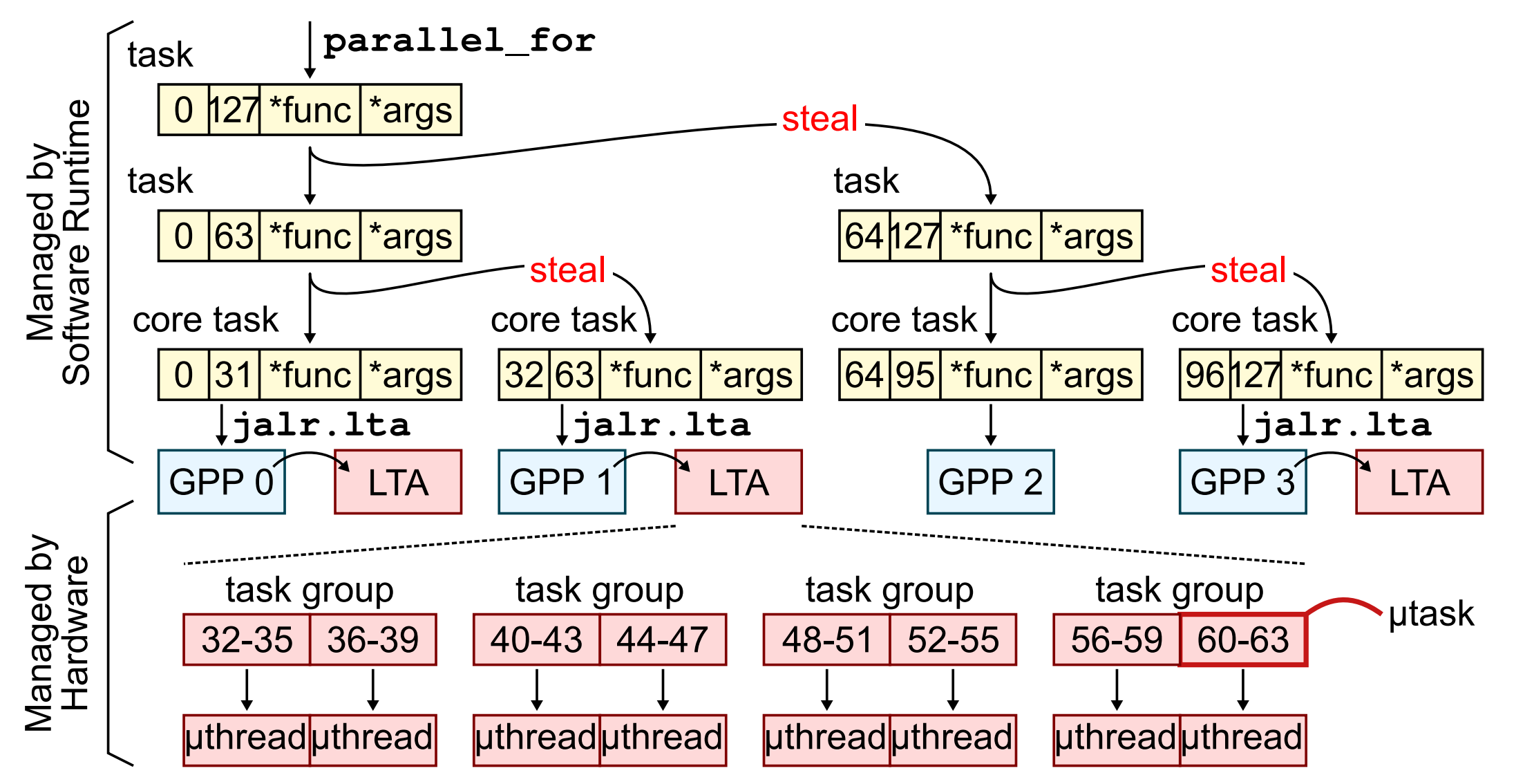
`jalr.lta $rd, $rs`

$rs	$a0	$a1	$a2	$a3
*loop_task_func	*args	0	N	step

We propose a new **jalr.lta** instruction that has the same semantics as normal indirect function call jalr, but serves as a hint to the underlying hardware that the function has the special signature of loop task.

LTA-Enabled Work-Stealing Runtime

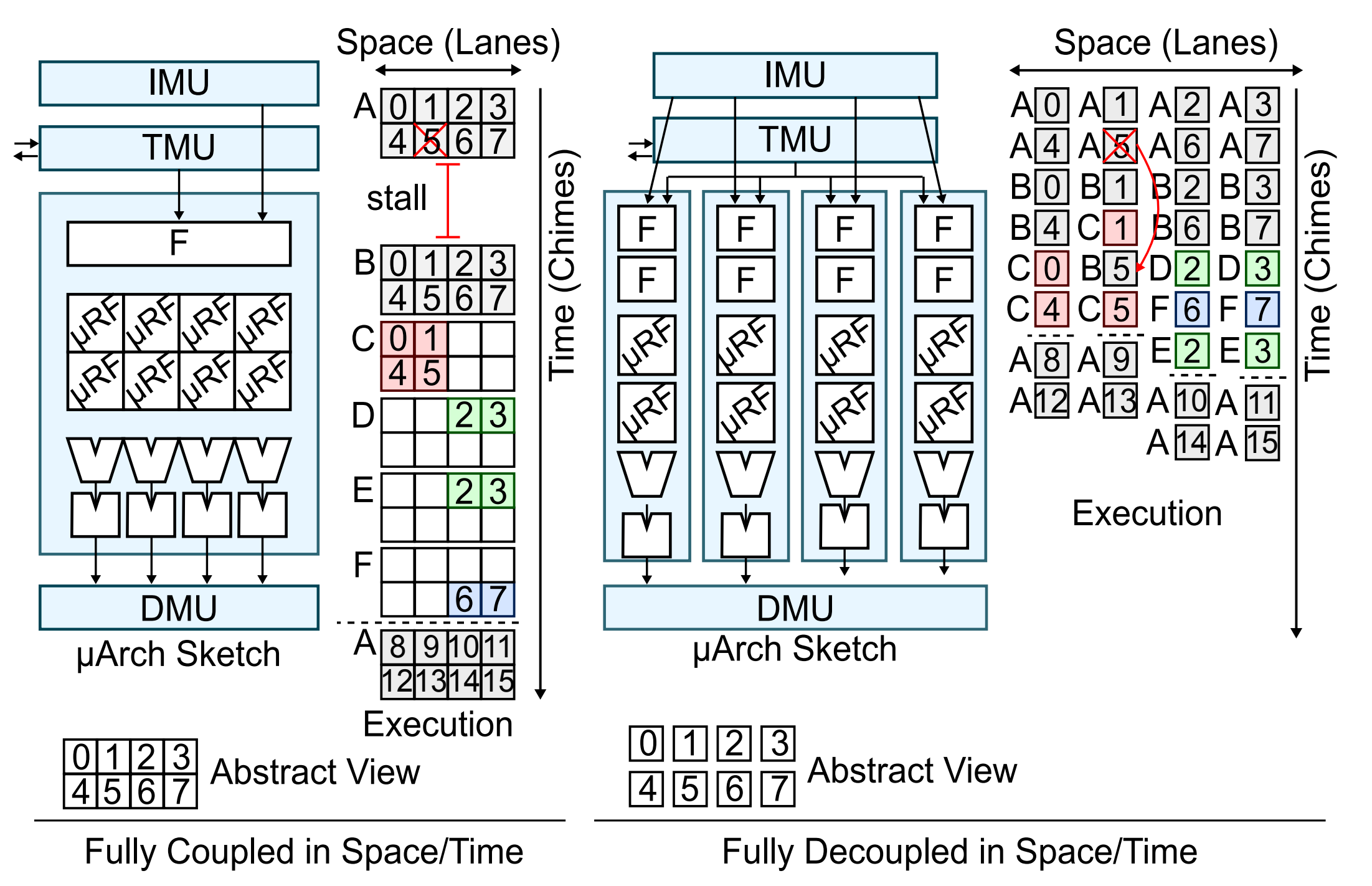
The **LTA-enabled work-stealing runtime** still recursively partitions loop tasks into subtasks to facilitate load balancing until the range is less than the core task size, but then uses the jalr.lta instruction. If an LTA is available, the GPP can potentially use the LTA to further partition the core task into µtasks, each responsible for a smaller range of iterations. The LTA groups µtasks into task groups which execute on a set of µthreads in lockstep (i.e., same instruction), exploiting structure for efficient execution.



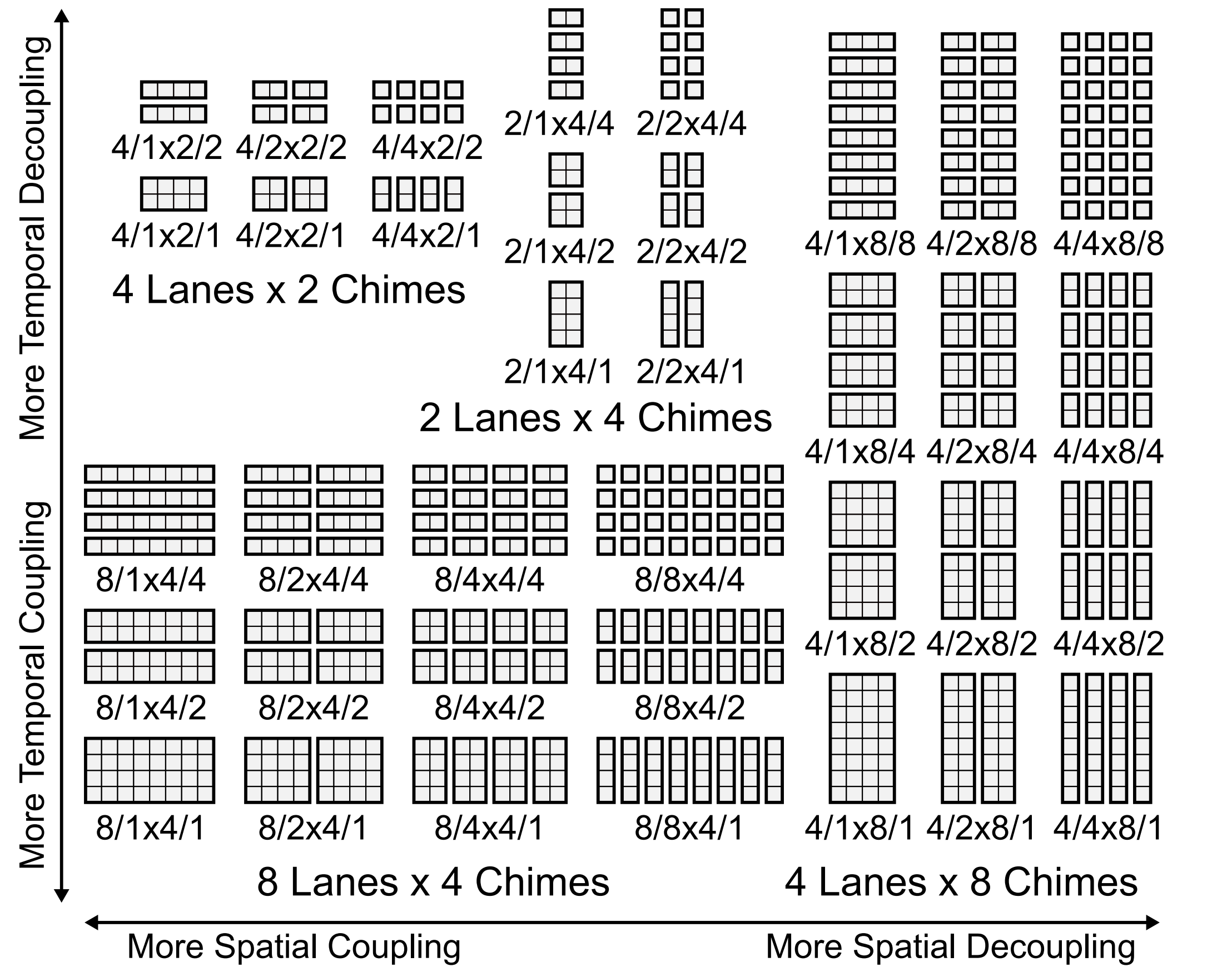
If an LTA is not available (GPP 2 in the above diagram), a jalr.lta can be treated as a standard jalr executing on the GPP. This approach requires minimal changes to a standard work-stealing runtime and practically no changes to the parallel program. Compare this to the significant software changes required to combine task-parallel programming and packed-SIMD extensions.

4 LTA Hardware

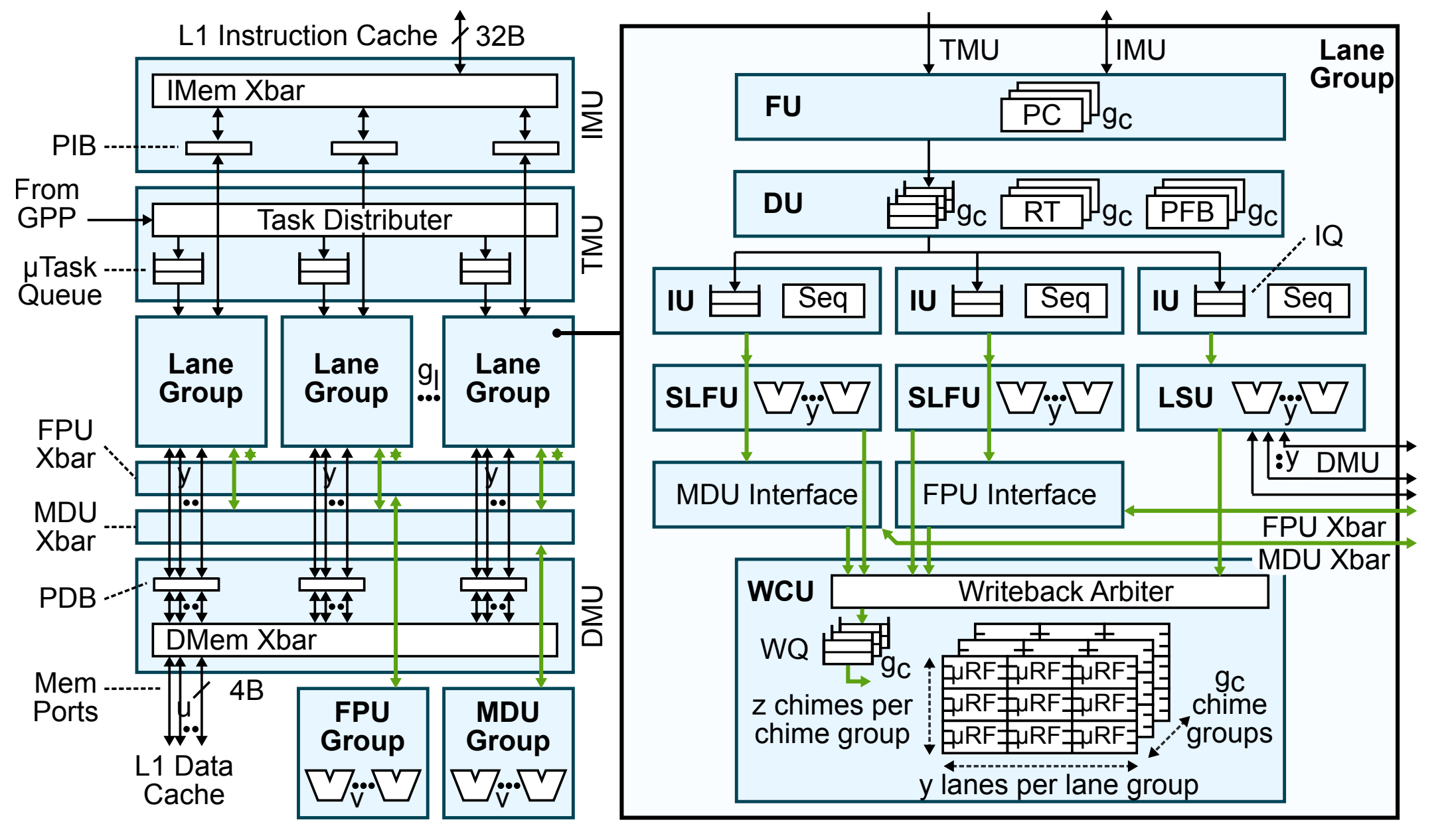
Spatial and Temporal Task Coupling



Task Coupling Taxonomy



Microarchitecture Template

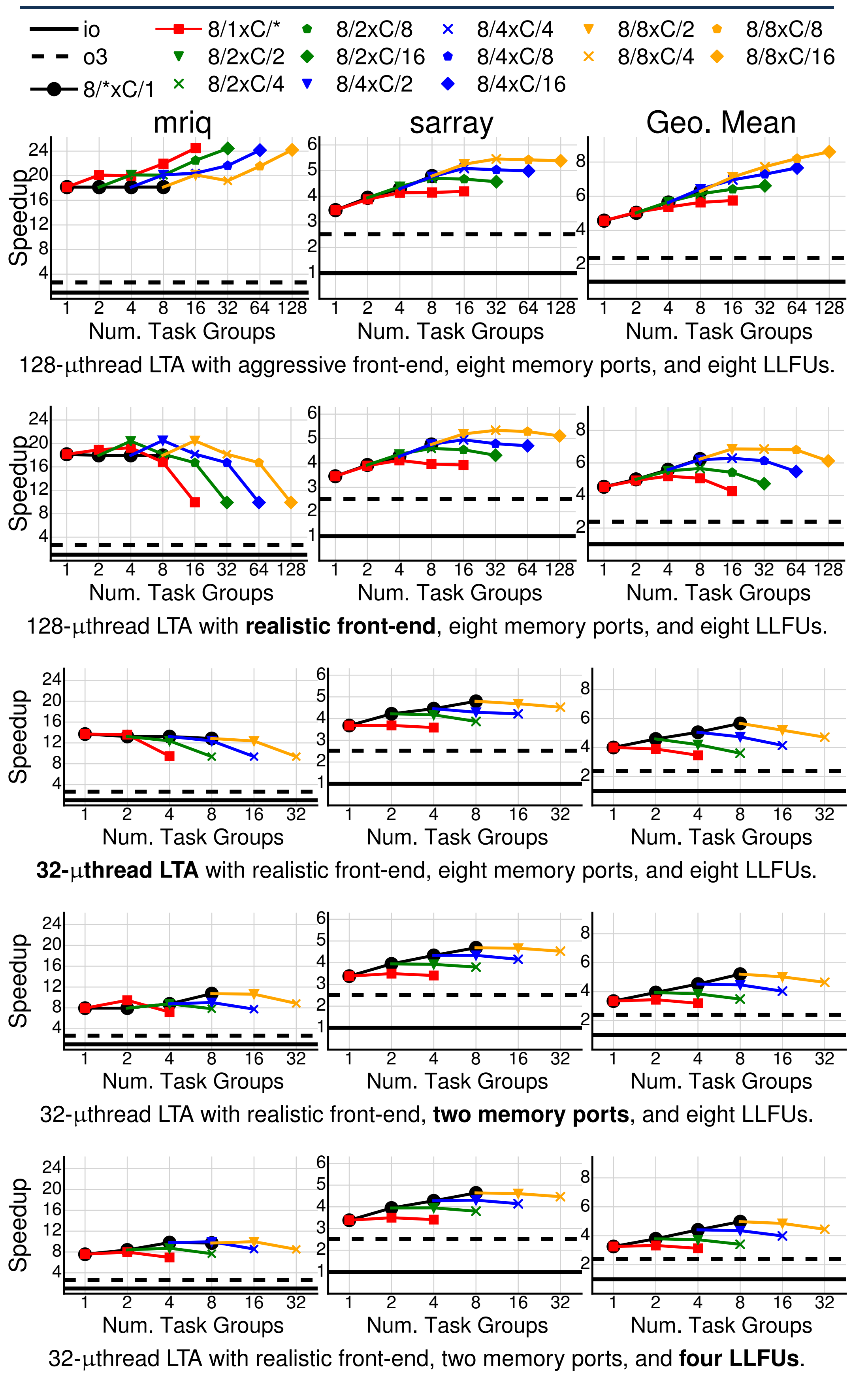


This work was supported in part by NSF CAREER Award #1149464, NSF XPS Award #1337240, NSF CRI Award #1512937, NSF SHF Award #1527065, AFOSR YIP Award #FA9550-15-1-0194, and donations from Intel, NVIDIA, and Synopsys. The authors acknowledge and thank Scott McKenzie and Alvin Wijaya for their early work on LTA RTL modeling, Jason Setter and Wei Geng for their work on scalar processor PyMTL modeling, and David Bindel for access to an Intel Xeon Phi 5110P.

5 Cycle-Level Evaluation

We utilize a co-simulation framework with gem5 and PyMTL, a Python-based hardware modeling framework. The cycle-level models for LTAs were implemented in PyMTL. Each core and its LTA share the L1 caches and all cores share the L2 cache. We simulate a bare-metal system with system call emulation. We ported 16 C++ application kernels to a MIPS-like architecture. We used a cross-compiler based on GCC-4.4.1, Newlib-1.17.0, and the GNU standard C++ library. Application kernels were either ported from Problem Based Benchmark Suite (PBBS) or developed in-house to create a suite with diverse task-level and instruction-level characteristics.

Performance vs. HW Resource



128-µthread LTA with aggressive front-end, eight memory ports, and eight LLFUs.

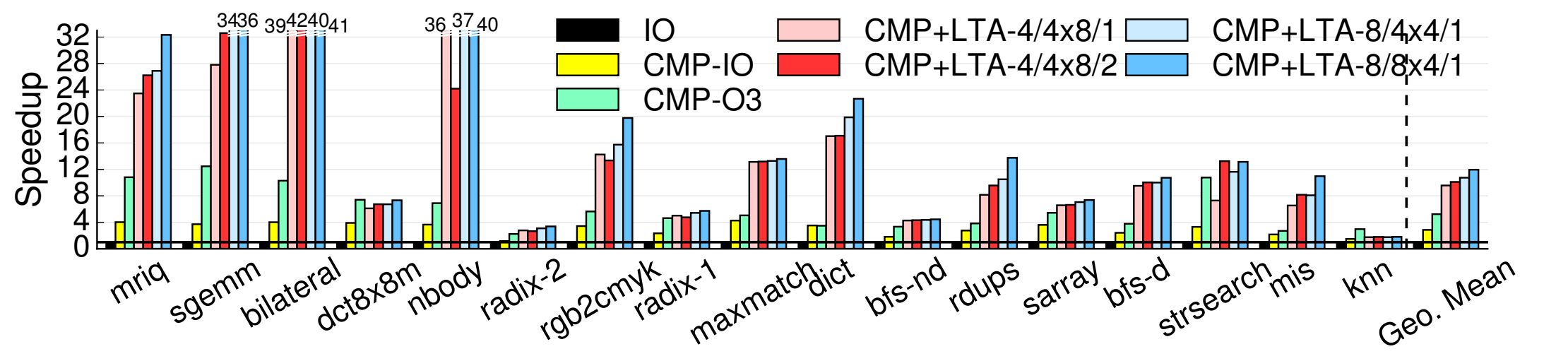
128-µthread LTA with **realistic front-end**, eight memory ports, and eight LLFUs.

32-µthread LTA with realistic front-end, eight memory ports, and eight LLFUs.

32-µthread LTA with realistic front-end, **two memory ports**, and eight LLFUs.

32-µthread LTA with realistic front-end, two memory ports, and **four LLFUs**.

Chip-Multiprocessor(CMP) with LTA



Compared to CMP-IO, CMP+LTA improves average raw performance by up to 4.2×, performance per area by 1.8× (excluding the L2), and energy efficiency by 1.1×. Compared to a more aggressive CMP-O3, CMP+LTA improves performance by 2.3×, performance per area by 1.5× (excluding the L2), and energy efficiency by 3.2×.

Using the jalr.lta instruction closes the intra-core parallel abstraction gap, and allows porting the kernels from TBB implementations with minimal changes. A single implementation is written and compiled once, then executed on a system with any combination of GPPs and homogeneous or heterogeneous LTAs.

Publication: Appears in the Proceedings of the 50th Int'l Symp. on Microarchitecture (MICRO-50), Oct. 2017. **URL:** `http://www.csl.cornell.edu/~cbatten/pdfs/kim-lta-micro2017.pdf`

Contact Author: Shunning Jiang, 471 Rhodes Hall, Ithaca, NY 14853, `sj634@cornell.edu`