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

School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
{ jyk46, sj634, clt67, mw828, ss2783, bi45, ka429, cbatten }@cornell.edu

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.

CCS CONCEPTS

• **Software and its engineering → Runtime environments**; • **Computer systems organization → Multicore architectures**; **Single instruction, multiple data**;

KEYWORDS

task-parallel programming frameworks; work-stealing run-times; programmable accelerators

ACM Reference Format:
J. Kim, S. Jiang, C. Torng, M. Wang, S. Srinath, B. Ilbeyi, K. Al-Hawaj, and C. Batten. 2017. Using Intra-Core Loop-Task Accelerators to Improve the Productivity and Performance of Task-Based Parallel Programs . In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 15 pages. https://doi.org/10.1145/3123939.3136952

MICRO-50, October 14–18, 2017, Cambridge, MA, USA

ACM ISBN 978-1-4503-4952-9/17/10…$15.00
https://doi.org/10.1145/3123939.3136952

1 INTRODUCTION

Task-based parallel programming frameworks are one of the most popular ways to exploit increasing thread counts in CMPs (e.g., Intel's Cilk Plus [29, 41], Intel's Threading Building Blocks (TBB) [30, 58], and others [5, 10, 36, 40, 56, 57]). Task-based frameworks use a software runtime to dynamically map many tasks to fewer threads. Programming with high-level tasks, as opposed to directly using low-level threads, offers many productivity and performance benefits including: an elegant encoding of fine-grain parallelism, implicit synchronization of serial and parallel regions, efficient load-balancing of tasks across threads, and portable performance across a wide range of CMPs.

Packed-SIMD extensions are commonly used in CMPs (e.g., AVX2 in Intel's Haswell [50], AVX512 in Intel's Xeon Phi [31], NEON in ARM's Cortex processors [26, 27], and MIPS's SIMD extensions [11]). In this work, we focus on a subset of task parallelism called *loop-task parallelism* that can potentially be mapped both across cores and to intra-core packed-SIMD extensions. 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. Loop-task parallelism is more flexible than fine-grain loop-level parallelism, but less general than coarse-grain (possibly nested/recursive) task-level parallelism. We argue there are two fundamental challenges that make using packed-SIMD units in this context particularly difficult.

Challenge #1: Intra-Core Parallel Abstraction Gap – Packed-SIMD extensions provide a low-level abstraction of operations on packed data elements exposed to programmers via compiler intrinsics or "auto-vectorization". Unfortunately, auto-vectorization does not always guarantee optimal vectorization in real applications [48]. Programmers are forced to use explicit vectorization [19, 24, 38], i.e., annotating vectorizable loops, explicit SIMD datatypes, SIMD-aligned memory accesses, converting branches into arithmetic, converting array-of-structs into struct-of-arrays, and annotating non-overlapping arrays. These optimizations are challenging to perform in loop-task parallel programs, since tasks can be arbitrarily complex and task sizes/alignments are not known at compile time. More importantly, this approach requires the programmer to use two fundamentally different parallel abstractions: tasks for inter-core parallelism and packed-SIMD for intra-core parallelism. Ultimately, this challenge reduces programmer productivity and can potentially prevent "multiplicative speedup" (i.e., the speedup of combining a task-based parallel runtime with packed-SIMD does not result in the product of each technique's speedup in isolation).

Challenge #2: 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

memory operations. Even assuming a programmer overcomes the first challenge and is able to map these irregular tasks to packed-SIMD extensions, the overall performance is likely to be disappointing for many reasons: 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. Ultimately, this challenge further reduces programmer productivity and usually prevents multiplicative speedup.

Section 2 explores these challenges using our experiences mapping regular and irregular applications to CMPs. 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. Although packed-SIMD applications can be rewritten to use an LTA, we still see benefit in including packed-SIMD units to manually exploit very fine-grain data parallelism with maximum efficiency.

Closing the Intra-Core Parallel Abstraction Gap – Section 3 describes the novel LTA software/hardware interface, which is based on adding a new hint in the instruction set that embeds the loop-task abstraction into the hardware for acceleration. This hint elegantly encodes loop-task execution by explicitly identifying indirect function calls with a predefined function signature. Leveraging the hint instruction requires only minor changes to a standard work-stealing runtime and enables a programmer to use a single parallel abstraction both across and within cores.

Efficiently Executing both Regular and Irregular Tasks – Section 4 describes how an LTA's execution of tasks can be coupled in either space or time (similar to the lock-step temporal and/or spatial execution used in SIMD microarchitectures), and we propose a task-coupling taxonomy to motivate our own design-space exploration. We describe a detailed LTA microarchitectural template which can be configured at design time for different amounts of spatial/temporal decoupling. Increased task decoupling can potentially enable better performance when executing irregular tasks, but also requires additional area and energy. One of the key research questions we seek to answer in this paper, is whether designers should favor spatial or temporal task decoupling when designing LTAs in a relatively resource constrained context.

Our approach is partly inspired by the success of general-purpose graphics processing units (GPGPUs) [8, 43]. GPGPUs mitigate the intra-core parallel abstraction gap by providing a fine-grain thread-based parallel abstraction in the programming model, instruction set, and SIMT microarchitecture. Section 7.1 crystallizes the fundamental differences between a GPGPU and CMP+LTA across the system architecture, intra-core parallel abstraction, and microarchitecture. However, this does raise the question: Are GPGPUs the right choice to improve the performance of task-based parallel programs? To answer this question, we must remember that GPGPUs are first and foremost designed to accelerate graphics rendering and as a consequence the entire GPGPU platform (e.g., massive temporal multithreading, lock-step SIMD execution, chip-level hardware scheduling, limited on-chip private/shared caching) is designed to excel when executing throughput-focused, regular programs with tremendous parallelism (i.e., tens of thousands of very similar threads). While some loop-task parallel programs may fall into this category, many do not since they include: more modest parallelism, a fine-grain mix of serial/parallel regions, and moderate to highly irregular control and memory access patterns. Mapping irregular loop-task workloads to GPGPUs requires significant software engineering effort [9, 25, 28, 47, 54, 68] and the overall performance can be mixed. At a high level, GPGPUs extend the intra-core SIMD abstraction across the entire chip, while CMP+LTAs push the chip-level task-parallel abstraction down into the core architecture.

Section 5 outlines our vertically integrated research methodology, and Section 6 presents cycle-level performance, area, and energy results which suggest designers should favor spatial over temporal task decoupling within resource constrained contexts. The primary contributions of this work are: (1) the LTA software/hardware abstraction based on a new hint instruction that explicitly encodes loop-task execution within standard software runtimes; (2) the LTA microarchitectural template with a novel taxonomy for understanding spatial/temporal task decoupling; and (3) a detailed design-space exploration of the area, energy, and performance implications of task decoupling and a comprehensive evaluation of CMP+LTA.

2 MOTIVATION

In this section, we describe our experiences traversing an application development flow to improve performance on two modern platforms: Intel Xeon E5-2620 v3 (Haswell) and Intel Xeon Phi 5110P in native mode (MIC). We develop both regular and irregular loop-task parallel applications and summarize our experiences with each platform to investigate both key challenges. Application development begins with a scalar reference implementation on both platforms before moving towards exploiting inter-core parallelism through parallel frameworks (e.g., Intel TBB), exploiting intra-core parallelism through packed-SIMD extensions (e.g., Intel AVX), or combining approaches for ideally multiplicative speedups. Table 1 lists the application kernels used (see Section 5.1 for details) and includes logical source lines of code (LOC; number of C++ statements) as a proxy for programmer productivity. Figure 1 compares the performance of the four different implementations on both the Haswell and MIC platforms.

The ***tbb*** bars show the performance of the multi-threaded implementation. For this study, we use TBB due to its productive task-based programming model, library-based implementation, and work-stealing runtime for fine-grain dynamic load balancing. On the Haswell platform with 12 threads (6 cores), TBB improves performance for both regular and irregular loop-task parallelism, as seen by the 2–11× speedup across all kernels. However, the benefits of multi-threading can be limited by memory bottlenecks; *bfs-nd* and *maxmatch* rely on atomic memory operations that exacerbate this issue. The MIC platform includes 60 relatively lightweight single-issue in-order cores, longer cache-to-cache latencies, and no shared last-level cache [7, 59]. Again, both regular and irregular kernels scale very well with speedups in the range of 46–153×. With four-way multithreading in each core, we observed a delicate performance tradeoff in the MIC platform between cache locality and hiding microarchitectural latencies. **Key Observation:** TBB is a productive framework. The LOC numbers are similar

compared to the scalar implementation, and only approximately one programmer-week was required to develop a relatively high-performance parallel implementation of the benchmark suite.

The ***avx*** bars show the performance of the *single-threaded* implementation with explicit vectorization using AVX. Vectorization only targets regular loop-task parallelism within a core, as both control and memory-access divergence can prevent lock-step execution. On the Haswell platform, kernels with regular loop-task parallelism (i.e., *sgemm*, *dct8x8m*, *mriq*, *rgb2cmyk*) see a speedup of at least 2.5×, while kernels with irregular loop-task parallelism (i.e., *bfs-nd*, *maxmatch*, *strsearch*) see negligible benefits. Similarly for the MIC platform, the regular kernels show speedups in the range of 1.4–32×, while irregular kernels have no speedup. Because vectorization primarily increases compute throughput, memory bottlenecks can still limit performance. Note that on the Haswell platform, naive vectorization with `#pragma simd` yielded speedups of less than 1.10× across all seven kernels. Maximally utilizing the SIMD units required numerous manual code changes to optimize/annotate alignment, the control flow, data structure layout, and memory aliasing [3, 19, 24]. **Key Observation:** Although AVX can improve performance for regular loop-tasks, we found the required manual optimizations greatly reduced productivity. Implementing, testing, and tuning kernels for AVX took multiple programmer-weeks for the full suite. No amount of manual optimization improved performance for *irregular* loop-tasks.

The ***tbb-avx*** bars show the performance of TBB combined with explicit vectorization using AVX. Regular loop-task parallel applications can *sometimes* combine the two approaches for multiplicative speedup. On the Haswell platform, *sgemm* achieves a close-to-ideal multiplicative speedup of 63×. However, combining TBB and AVX can sometimes worsen performance, as in *dct8x8m* and *mriq*, for several reasons. First, task partitioning with TBB can interfere with explicit vectorization with AVX: vectorization might fail even if SIMD-multiple task sizes are specified because TBB cannot guarantee exact task sizes at compile time. Second, vector-optimizations to enable AVX can limit load balancing with TBB: eliminating control divergence during vectorization may eliminate opportunities for load balancing by superficially equalizing the work across the SIMD width. MICs are designed to accelerate applications with immense regular loop-task parallelism, so performance depends heavily on maximally utilizing the SIMD units. Results for regular kernels on the MIC platform show much less than the expected multiplicative speedup and also some slowdown. For both platforms, irregular kernels show no benefit. **Key Observation:** Besides not guaranteeing better performance, combining TBB with AVX negates the productivity of the former. It took multiple programmer-weeks of manual optimization to vectorize our original TBB implementations for similar LOC as *avx*. In addition, despite using the same software framework, optimizing the MIC platform required non-trivial re-tuning for thread counts, task sizes, and algorithm restructuring.

This study provides evidence for the two key challenges involved in exploiting intra-core parallelism in loop-task parallel programs on the Haswell and MIC platforms. *The Intra-Core Parallel Abstraction Gap:* Using either TBB in isolation or packed-SIMD extensions in isolation can provide performance improvement depending on the application, but unfortunately, combining these two parallelization strategies is challenging, reduces programmer productivity, and results in less than the desired multiplicative speedup.

Name	Suite	Input	scalar	tbb	avx	tbb-avx
sgemm	I	2K×2K float matrix	22	26	53	56
dct8x8m	I	518K 8×8 blocks	58	63	128	62†
mriq	I	262K-space 2K pnts	28	35	76	78
rgb2cmyk	I	7680×4320 image	20	17	70	69
bfs-nd	[65]	rMatG_J5_10M	27	48	29‡	51‡
maxmatch	[65], I	randLocG_J5_10M	22	36	24‡	38‡
strsearch	I	512 strs, 512 docs	35	42	38‡	45‡

Table 1: Application Kernels – I = in-house implementation. The last six columns show logical source lines of code (LOC). †Lower LOC because programming model does not support using the more efficient LLM algorithm [46]. ‡Only auto-vectorization, since explicit vectorization resulted in no improvement.

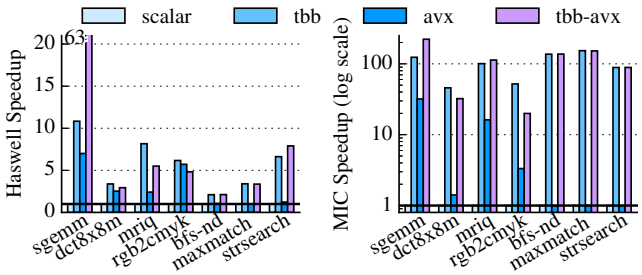


Figure 1: Performance Comparison of Various Implementations on Haswell and MIC Platforms – Normalized to each respective *scalar* (non-vectorized single-threaded) implementation. *Haswell* = Intel Xeon E5-2620 v3 (12 cores, AVX2/256b); *MIC* = Intel Xeon Phi 5110P (60 cores, AVX-512); *avx* = ICC v15.0.3 with explicit/auto-vectorization [48]; *tbb* = TBB v4.3.3.

Inefficient Execution of Irregular Tasks: Despite significant effort mapping irregular tasks to packed-SIMD extensions, irregular kernels show disappointing performance. These challenges motivate our interest in a new intra-core loop-task accelerator specifically designed to enable loop-task parallel applications to seamlessly exploit both inter- and intra-core parallel execution resources and produce truly multiplicative speedups.

3 USING LTAS TO CLOSE THE INTRA-CORE PARALLEL ABSTRACTION GAP

The LTA software/hardware interface closes the intra-core parallel abstraction gap by adding a lightweight hint in the instruction set that directly embeds the loop-task parallel abstraction into the hardware for accelerated execution. This section describes the minimal changes required for a standard work-stealing runtime to leverage the new interface.

This work uses the `parallel_for` primitive to express loop tasks, where a loop task is a functor applied across a range of loop iterations (see Figure 2(a)). More specifically, a loop task is a four-tuple of a function pointer, an argument pointer, and the start/end indices of the range. Figure 2(b) illustrates the loop-task function generated in this example, which is applied to the range $\langle 0, size \rangle$. The `step` argument is hidden from the application-level programmer but provides flexibility in the microarchitecture (see Section 4).

```
1 void vvadd( int dest[], int src0[], int src1[], int size ) {
2   LTA_PARALLEL_FOR( 0, size, (dest,src0,src1), ({
3     dest[i] = src0[i] + src1[i];
4   }));
5 }
```

(a) Element-Wise Vector-Vector Addition with Macro

```
1 void task_func( void* a, int start, int end, int step=1 ) {
2   args_t* args = static_cast<args_t*>(a);
3   int* dest = args->dest;
4   int* src0 = args->src0; int* src1 = args->src1;
5   for ( int i = start; i < end; i += step )
6     dest[i] = src0[i] + src1[i];
7 }
```

(b) Loop-Task Function Generated by Macro

Figure 2: LTA Programming API – A `parallel_for` construct is used to express loop-tasks that can be exploited across cores and within a core. We use a preprocessor macro in our current LTA runtime, since our cross-compiler does not yet support C++11 lambdas.

We have developed our own TBB-inspired task-based work-stealing runtime. Although we could have extended a commercial runtime such as TBB itself, developing our own runtime ensures we have detailed visibility and insight into all aspects of the software stack. Our runtime uses state-of-the-art techniques to efficiently distribute tasks across cores. It employs child-stealing, Chase-Lev task queues [12], and occupancy-based victim selection [16]. Section 5.2 validates that our runtime is comparable to both TBB and Cilk Plus for the specific features supported by LTAs. Figure 3 illustrates how a standard work-stealing runtime recursively partitions loop tasks into subtasks to facilitate load balancing. Tasks are partitioned until the range is less than the *core task size* at which point the subtask is called a *core task*. The runtime uses a core task size of $N/(k \times P)$, where N is the size of the initial range, k is a scaling factor, and P is the number of cores. Increasing k generates more core tasks with smaller ranges (better load balancing, higher overhead), whereas decreasing k generates fewer core tasks with larger ranges (worse load balancing, lower overhead). In this work, we found $k = 4$ to be a reasonable design point based on sensitivity studies.

The primary change required in an LTA-enabled work-stealing runtime is in how the runtime executes each core task. A traditional runtime uses an indirect function call (i.e., `jalr`) on the core task's function pointer with the given range and argument pointer, while an LTA-enabled runtime uses a new `jalr.lta` instruction. A `jalr.lta` is still an indirect function call with the same semantics as a `jalr`, except that it specifies a loop-task function pointer with the special signature in Figure 2(b). A `jalr.lta` hints that hardware can potentially use an LTA to further partition the core task into *micro-tasks* (μtasks), each responsible for a smaller range of iterations (see Figure 3). The LTA groups μtasks into task groups which execute on a set of *micro-threads* (μthreads) in lockstep (i.e., same instruction), exploiting structure for efficient execution. If an LTA is not available, a `jalr.lta` can be treated as a standard `jalr`. 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.

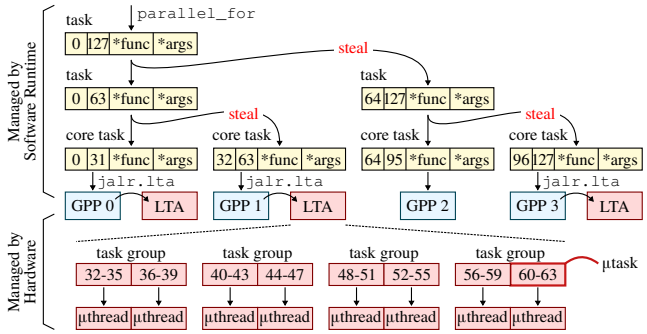


Figure 3: LTA Task Partitioning – Runtime partitions tasks into *core tasks* and dynamically distributes core tasks to GPPs. A core task is responsible for a range of loop iterations. The GPP sends the core task to an LTA using the `jalr.lta` instruction. The LTA further partitions a core task into *μtasks*. A μtask is responsible for a smaller range of loop iterations. The LTA groups μtasks into *task groups*. Task groups execute in lockstep on a set of *μthreads*.

4 USING LTAS TO EFFICIENTLY EXECUTE BOTH REGULAR AND IRREGULAR LOOP TASKS

In this section, we describe an LTA microarchitecture that implements the `jalr.lta` instruction to achieve true multiplicative speedups for *both* regular *and* irregular applications. While the LTA microarchitecture is partly inspired by the success of GPGPUs, a key LTA design decision will focus on *spatial/temporal task decoupling*. Coupled task execution means μtasks are executed in lockstep in space and/or time; decoupled task execution means μtasks can slip relative to each other in space and/or time.

4.1 LTA Task-Coupling Taxonomy

An LTA partitions a core task into μtasks which are then mapped to μthreads. We will use terminology from traditional vector processors, where μthreads may be organized spatially across *lanes* and temporally with *chimes*.

Figure 4(a) illustrates one approach for an 8-μthread LTA that *fully couples* μthread execution in both space and time. All eight μthreads must spatially execute in lockstep across the four lanes and also temporally execute in lockstep across the two chimes. Fully coupled execution enables the LTA to exploit arithmetic, control, and memory structure across μtasks to amortize various control area/energy overheads. At a high level, the task-management unit (TMU) receives information about a core task from the GPP, divides this core task into eight μtasks, and (compactly) sends the μtasks to the fetch/dispatch unit (F). The hardware is responsible for setting the argument registers for each μthread appropriately (see Section 3). To expose potential memory structure across μtasks, μthreads on neighboring lanes execute consecutive loop iterations. To enable this, the hardware sets the *range step value* mentioned in Section 3 so that μthreads execute iterations at a stride of eight. Figure 4(b) illustrates how the core task mapped to GPP 0 in Figure 3 might execute on this fully coupled LTA. The diagram shows how the fully coupled LTA struggles with divergence and may also force all μthreads to stall if any μthread stalls due to a RAW dependency

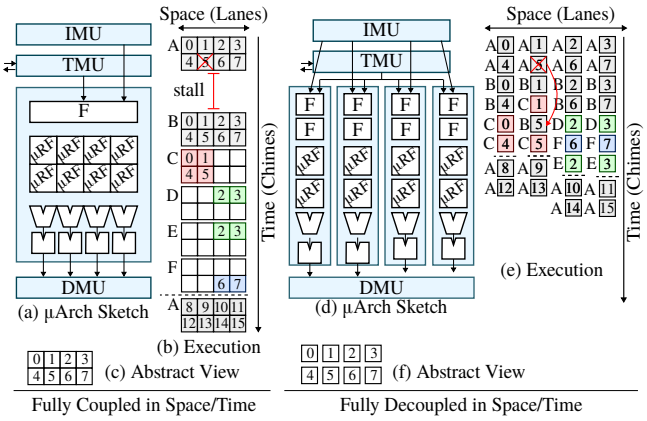


Figure 4: Coupled vs. Decoupled LTAs – Two 8-μthread LTAs each with 4 lanes, 2 chimes: (a–c) μthreads are fully coupled in space and time; (d–f) μthreads are fully decoupled in space and time. Letters denote instructions. In (b) and (e), μthreads 0, 1, 4, 5 execute instr A, B, C; μthreads 2, 3 execute instr A, B, D, E; μthreads 6, 7 execute A, B, F. Divergent control flows are shown in color. IMU = instr mgmt unit; TMU = task mgmt unit; DMU = data mgmt unit; F = fetch/dispatch unit; μRF = μthread regfile.

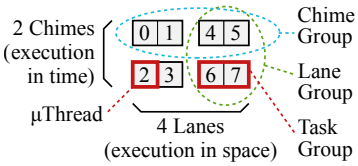


Figure 5: Terminology *LTA-4/2x2/2* configuration: 8-μthread LTA w/ 4 lanes, 2 chimes, 4 task groups, 2 lane groups, 2 chime groups.

or cache miss (marked with ×). Overall, fully coupled LTAs perform well on regular loop-task parallelism, but can perform poorly on irregular loop-task parallelism.

Figure 4(d) illustrates a different approach for an 8-μthread LTA that *fully decouples* μthread execution in both space and time. Figure 4(e) illustrates how the same core task from the previous example might execute on this fully decoupled LTA. All eight μthreads execute in a completely decoupled fashion in space (i.e., lanes can slip past other stalling lanes) and in time (i.e., chimes use fine-grain vertical multi-threading to execute whenever ready). Decoupled execution enables the LTA to better tolerate irregular control flow and memory latencies since each μthread can independently fetch, decode, dispatch, issue, and execute instructions. Unfortunately, this also means decoupled execution requires more front-end overhead.

The microarchitectures in Figure 4 are at two ends of a task-coupling spectrum. To simplify our discussion, Figures 4(c,f) and 5 illustrate abstract diagrams of how μthreads are coupled within an LTA. In Figure 5, the μthreads are divided into four *task groups*; the two μthreads within the task group execute in lockstep in both space and time. This example has two *lane groups* and two *chime groups*, which can be executed either spatially or temporally in a decoupled manner. For the remainder of this work, we abbreviate different LTA configurations with the following scheme: *num_lanes*/*num_lane_groups* x *num_chimes*/*num_chime_groups*. For example, Figure 5 represents an *LTA-4/2x2/2* configuration (four lanes organized into two lane groups, two chimes organized into two chime groups).

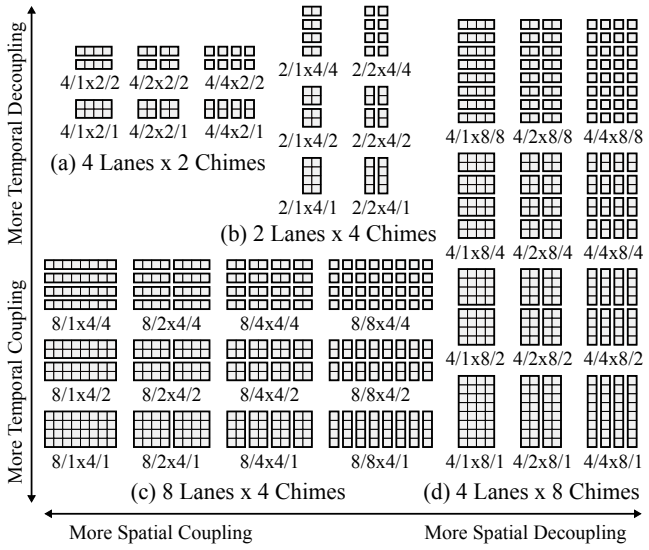


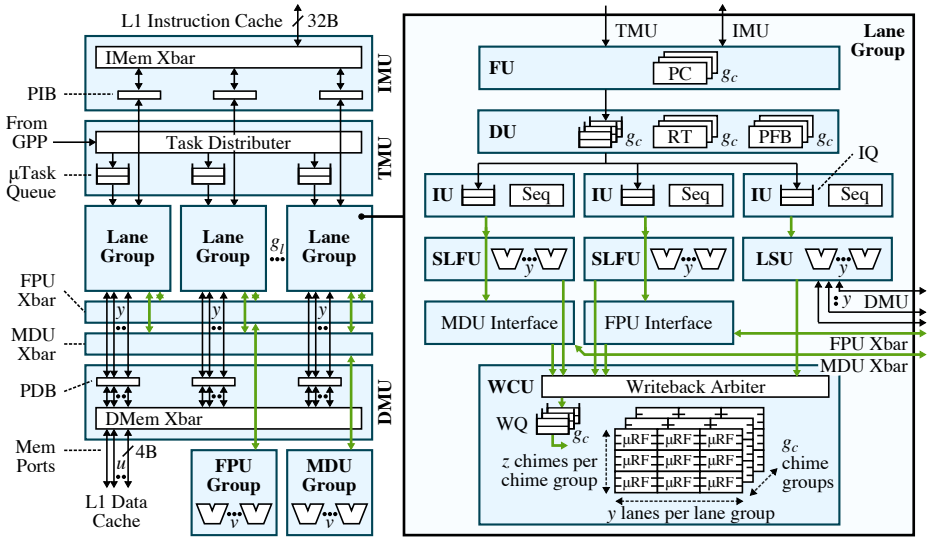
Figure 6: Task-Coupling Taxonomy – All possible spatial and temporal task-coupling configurations for: (a) 4 lanes, 2 chimes; (b) 2 lanes; 4 chimes; (c) 8 lanes, 4 chimes; (d) 4 lanes, 8 chimes. For a given subfigure, most-coupled configuration is bottom left and least-coupled configuration is top right.

Given this terminology, we can describe all possible spatial and temporal task-coupling configurations for a given number of lanes and chimes using a *task-coupling taxonomy* as shown in Figure 6. Figure 6(a) shows the six configurations for the 8-μthread LTA we have been discussing with four lanes and two chimes. Figure 6(b) presents an alternative 8-μthread LTA with two lanes and four chimes. Increasing the amount of temporal and/or spatial decoupling enables the microarchitecture to exploit more *thread-level parallelism* (TLP) across task-groups. However, increasing the amount of temporal and/or spatial decoupling also means the microarchitecture can exploit less *data-level parallelism* (DLP) within a task-group. Varying the amount of spatial and/or temporal decoupling also has a subtle impact on the ability of the microarchitecture to exploit instruction-level parallelism (ILP) within a task group.

4.2 LTA Microarchitectural Template

We describe an LTA microarchitectural template that can be configured at design time with any number of μthreads, lanes, lane groups, chimes, and chime groups (see Figure 7). The template enables design-space exploration of the task-coupling taxonomy described in the previous section.

The *task management unit* (TMU) is the interface between the GPP and LTA. The GPP sends a core task to the TMU via the `jalr.lta` instruction. The TMU then divides the core task into μtasks, groups these μtasks into task groups, and dynamically schedules the task groups across lane groups using per-lane-group queues. Upon receiving a new core task, the TMU initializes a pending task group counter. Lane groups assert a completion bit when they complete a task group. The TMU decrements the counter accordingly and sends a completion message to the GPP if the counter is zero. Currently, the GPP stalls until it receives this completion message.



(a) Top-Level LTA Microarchitecture (b) Detail of Lane Group

Figure 7: LTA Template – IMU = instr mngmt unit; TMU = task mngmt unit; DMU = data mngmt unit; PIB = pending instr buf; FPU = floating-point unit; MDU = int mult/div unit; PDB = pending data buf; FU = fetch unit; DU = dispatch unit; IU = issue unit; Seq = chime sequencer; SLFU = short-latency int functional unit; LSU = load-store unit; WCU = writeback/ commit unit; PC = program counter; RT = rename table; PFB = pending fragment buf; IQ = issue queue; WBQ = writeback queue; μRF = μthread regfile. l = tot num lanes; g_l = num lane groups; y = num lanes per lane group (l/g_l); c = tot num chimes; g_c = num chime groups; z = num chimes per chime group (c/g_c); u = num of dmem ports; v = tot scalar FPUs/MDUs. Thick green arrows = y worth of data per cycle.

A *lane group* executes the μtasks in a task group by using a set of μthreads. Task groups begin execution by jumping to the loop-task function pointer, but they must first initialize their argument registers: argument pointer in `a0`, start index in `a1`, end index in `a2`, and the range step value in `a3`. The range step value is set to be the number of μthreads in a task group, resulting in the μtask partitioning described in Section 4.1. Note that load balancing occurs naturally as faster lane groups obtain more task groups from the TMU. The level of *spatial* task coupling can be configured by organizing the lanes into different numbers of lane groups, each of which has an independent instruction stream and dynamically arbitrates for shared resources. The level of *temporal* task coupling can be configured by organizing the chimes into chime groups; one chime group per lane executes a task group. If a lane group supports multiple task groups, the execution of these task groups can be interleaved on a cycle-by-cycle basis.

Each lane group is further composed of a fetch unit (FU), a decode/dispatch unit (DU), issue units (IUs), short-latency functional units (SLFU), a load-store unit (LSU), and a writeback-commit unit (WCU). These units are connected by latency-insensitive interfaces, enabling a highly elastic pipeline. Recall that μthreads within a task group must execute in lockstep in both space and time. In this case, the frontend (e.g., FU, DU, IU) is amortized across the entire task group and each instruction operates at a task-group granularity. The FU has a program counter (PC) for each task group and an instruction from a different task group is fetched every cycle. The DU can temporally multiplex task groups by dispatching instructions from different task groups with round-robin arbitration. Note that task groups must stall on conditional branches until all μthreads in the task group have resolved the branch, but another independent task group can be dispatched to hide this latency. Instructions are dispatched in order within a chime group, but simple register renaming is used to allow out-of-order writeback. Dispatched instructions wait in the in-order issue queue (IQ) until its operands are ready to be bypassed or read from the register file. Operands are read for the entire task group from a 6r3w register file with per-μthread banks. The IU then sequences the chimes, which are executed by the appropriate functional unit; the μthreads within a chime are executed in parallel across the lanes. The SLFU handles integer operations and branches, while the LSU handles memory operations. The LSU can generate one memory request per lane per cycle. With three IUs, each lane group effectively has three issue slots. The WCU arbitrates writes from functional units to the writeback queue (WQ) at chime granularities. The register file is updated in order once the entire task group has written to the WQ.

Within a lane group, divergent branch resolutions within a task group are handled by executing the not-taken μthreads (active) first and pushing a *task group fragment* representing the taken μthreads (inactive) into a *pending fragment buffer* (PFB) to be executed later. Fragments in the PFB can reconverge with other fragments (including the active fragment) with matching PCs. We implement a two-stack PFB as described in [33, 39] that prioritizes fragments in current loop iterations.

At the top-level, the L1 instruction cache port is managed by the *instruction management unit* (IMU) and is always shared across the lane groups. The IMU includes per-lane-group pending instruction buffers (PIBs) that can store a cache-line-worth (32B) of instructions to amplify the fetch bandwidth, and a crossbar with round-robin arbitration. Provisioning an L1 data cache port, an integer multiply/divide unit (MDU), and a floating-point unit (FPU) for each lane can result in significant area overhead. In addition, highly multi-ported L1 data caches can significantly increase the energy per access. Thus, our template also enables sharing these expensive resources both within and across lane groups using an FPU crossbar, MDU crossbar, and a data-memory unit (DMU) at the top-level. Figure 8 illustrates the technique used to share two memory ports and a long-latency functional unit (LLFU = either an MDU or an FPU) with a scalar throughput of four operations/cycle across eight lanes. For the memory ports, the eight lanes are divided into two groups regardless of the amount of spatial decoupling, and

Name	Suite	Input	loops	func	cond	ind	amos	DynInst (M) S	DynInst (M) P	DynInst (M) T%	Avg Size Loop	Avg Size Iter	Intensity slfu	Intensity llfu	Intensity mem	CMP IO	CMP IO	CMP O3	LTA-8/1x4/1 IF	LTA-8/1x4/1 A	LTA-8/1x4/1 IU	LTA-8/1x4/1 M	LTA-8/4x4/1 IF	LTA-8/4x4/1 A	LTA-8/4x4/1 IU	LTA-8/4x4/1 M	LTA-8/1x4/4 IF	LTA-8/1x4/4 A	LTA-8/1x4/4 IU	LTA-8/1x4/4 M
mriq	C	100-space, 256 pts	F	X				11	11	99%	256	23K	53%	20%	21%	13.9	4.0	4.1	0.04	32	0.24	0.0	0.14	32	0.31	0.0	0.13	32	0.22	0.0
sgemm	C	256×256 fp matrix	F					75	76	99%	576	131K	47%	19%	21%	110.7	3.7	4.0	0.03	32	0.21	3.5	0.13	32	0.31	0.2	0.13	32	0.21	3.5
bilateral	C	256×256 image	F					26	27	99%	66K	409	25%	51%	16%	60.0	4.0	3.6	0.03	31	0.20	0.6	0.14	31	0.19	1.3	0.13	31	0.17	1.4
dct8x8m	C	782 8x8 blocks	F	X				55	55	99%	50K	1096	4%	64%	30%	58.6	3.9	3.8	0.03	31	0.09	26.6	0.13	32	0.15	18.6	0.13	32	0.09	27.8
nbody	P	3DinCube_1000	F	R	X			92	93	99%	1000	31K	18%	43%	33%	224.2	3.7	3.7	0.04	30	0.16	0.3	0.14	31	0.20	0.4	0.13	30	0.16	0.8
radix-2	P	exptSeq_500K_int	F		X	X		57	69	81%	46	92K	59%	0%	33%	84.0	1.2	1.0	0.04	30	0.06	37.8	0.14	30	0.09	37.4	0.13	30	0.06	37.6
rgb2cmyk	C	1380×1080 image	F		X			43	43	99%	1380	31K	47%	0%	39%	52.4	3.4	2.6	0.04	29	0.13	8.4	0.15	30	0.19	8.8	0.13	30	0.11	16.4
radix-1	P	randomSeq_1M_int	F		X	X		93	104	94%	229	74K	57%	0%	33%	140.4	2.3	2.1	0.05	26	0.07	33.3	0.18	28	0.11	32.4	0.15	27	0.06	33.1
maxmatch	P	randLocG_E5_400K	W		X	X	X	23	49	94%	1.7M	19	58%	0%	19%	184.4	4.3	1.6	0.05	25	0.06	9.6	0.18	27	0.10	12.0	0.15	27	0.05	12.0
dict	P	exptSeq_1M_int	W		X	X	X	39	51	99%	451K	25	66%	0%	19%	90.9	3.5	1.8	0.06	20	0.13	6.6	0.19	26	0.13	13.7	0.16	26	0.08	10.5
bfs-nd	P	randLocG_J5_150K	F		X	X	X	23	55	81%	36K	99	56%	0%	26%	115.8	1.8	1.4	0.07	17	0.05	13.9	0.23	21	0.09	11.8	0.19	21	0.05	13.5
rdups	P	trigrSeq_300K_int	W		X	X	X	36	56	99%	508K	23	56%	0%	21%	66.4	2.8	1.9	0.12	12	0.07	17.5	0.32	17	0.10	19.3	0.25	16	0.06	19.4
sarray	P	trigrString_200K	F		X	X		68	75	86%	76K	50	56%	0%	29%	205.0	3.6	2.2	0.09	12	0.07	45.7	0.24	19	0.10	41.3	0.21	19	0.07	45.4
bfs-d	P	randLocG_J5_150K	F		X	X	X	23	35	95%	50K	75	56%	0%	26%	115.8	2.4	1.6	0.11	11	0.04	12.5	0.32	15	0.06	10.9	0.25	17	0.03	12.8
strsearch	C	210 strs, 210 docs	W		X	X		20	20	99%	210	49K	57%	0%	19%	29.9	3.3	3.4	0.20	6	0.08	0.7	0.55	10	0.10	0.7	0.27	15	0.13	0.7
mis	P	randLocG_J5_400K	W		X	X	X	14	32	99%	400K	27	52%	0%	25%	56.0	2.2	1.1	0.25	5	0.03	20.1	0.58	9	0.05	18.5	0.48	8	0.03	11.8
knn	P	2DinCube_10K	F	R	X	X		35	43	33%	9867	716	17%	32%	37%	57.0	1.5	1.3	0.21	5	0.03	9.3	0.42	11	0.06	9.5	0.43	9	0.05	11.8

Table 2: Application Kernels – Apps roughly organized from more regular to more irregular. Suite: P = PBBS [65], C = custom; Loop-task contains: inner loops (F = for loop in addition to loop due to line-5 in Figure 2(b), W = while loop), func = function calls (R = recursive), cond = data-dependent conditionals, ind = indirect memory accesses, amos = atomic memory operations; DynInsts = dyn. insts in millions (S for serial impl, P for parallel impl); T% = percent of total dyn. insts in tasks; Avg Loop Size = avg num of iterations/loop; Avg Iter Size = avg num of dyn. insts/iteration; { slfu, llfu, mem } Intensity = percent of total dyn. insts that are {short-latency arith, long-latency arith, mem operation}; IO = num of cycles (in millions) of optimized single-threaded impl on in-order core; CMP-IO = speedup of multi-threaded impl on 4 in-order cores over single in-order core; -O3 = speedup of multi-threaded implementation on 4 out-of-order cores over single O3 core; IF = ratio of total inst fetches to total dyn. insts; A = avg active μthreads in LTA per dyn. inst; IU = effective issue slot utilization; M = misses in L1 D$ per thousand dyn. insts; *LTA-8/1x4/1*: fully coupled in space and time; *LTA-8/4x4/1*: decoupled in space; *LTA-8/1x4/4*: decoupled in time.

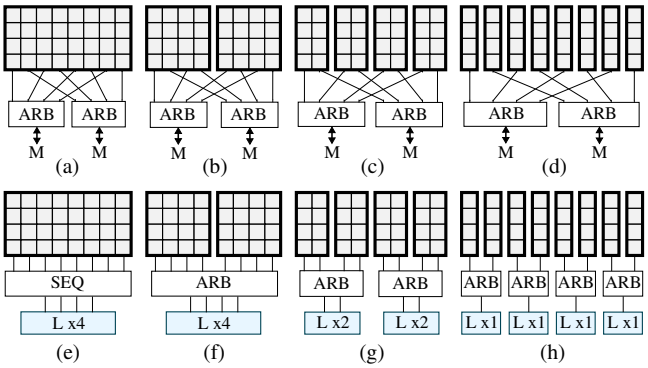


Figure 8: Connecting the LTA to Memory Ports and LLFUs Illustrates how eight lanes are connected to two memory ports and four long-latency functional units (LLFUs) for entire range of spatial decoupling. ARB = Arbiter; SEQ = Sequencer; M = data cache port; L x*N* = *N*-wide LLFU.

the four lanes within each group dynamically arbitrate for one of the two memory ports (see Figure 8(a–d)). For the LLFUs, the fully coupled configuration uses a sequencer to serialize an eight-wide chime across a four-wide LLFU (see Figure 8(e)). The decoupled configurations use limited dynamic arbitration for an LLFU where the width of the LLFU matches the number of lanes per lane group (see Figure 8(f–h)). We use grant-and-hold arbitration for memory ports and LLFUs to ensure an entire task group is processed together. The DMU includes pending data buffers (PDBs) that can store a task-group-worth of 4B words to facilitate access/execute decoupling.

5 EXPERIMENTAL METHODOLOGY

In this section, we describe the details of our vertically integrated research methodology spanning applications, runtime, architecture, and VLSI. The 24 LTA configurations primarily used in this study are shown in Figure 6(c–d).

5.1 Application Kernels

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 PBBS [65] or developed in-house to create a suite with diverse task-level and instruction-level characteristics (see Table 2). We include two datasets for *radix*, since it exhibits strong data-dependent variability. See [65] for more detailed descriptions of the PBBS kernels. *bilateral* performs a bilateral image filter with a lookup table for the distance function and an optimized Taylor series expansion for calculating the intensity weight; we parallelize across output pixels. *dct8x8m* calculates the 8x8 discrete cosine transform on an image using the LLM algorithm [46]; we parallelize across 8x8 blocks. *mriq* is an image reconstruction algorithm for MRI scanning; we parallelize across the output magnetic field gradient vector. *rgb2cmyk* performs color space conversion on an image and is parallelized across the rows. *strsearch* implements the Knuth-Morris-Pratt algorithm with a deterministic finite automata to search a collection of byte streams for a set of substrings. The search is parallelized by having all threads search for the same substrings in different streams. The deterministic finite automata are also generated in parallel. *sgemm* performs a single-precision matrix multiplication for square matrices using a standard blocking algorithm; we parallelize across blocks.

	sgemm	*dct8x8m*	*mriq*	*bfs-nd*	*maxmatch*	*strsearch*
Cilk+	10.42	3.32	7.53	2.29	1.61	11.18
TBB	11.76	3.33	8.83	1.77	1.73	9.97
LTA	10.32	3.38	9.54	1.77	2.05	11.22

Table 3: Comparison of Various Runtimes on x86 – Speedup over optimized single-thread implementation using 12 threads on Linux server with 2 Intel Xeon E5-2620 v3 processors. Cilk+ = uses Cilk Plus's `cilk_for`; TBB = uses TBB's `parallel_for`; LTA = x86 port of our runtime. All apps compiled with Intel ICC 15.0.3.

5.2 Validating the LTA Runtime

To show that our runtime is competitive with other popular task-based work-stealing runtimes, Table 3 compares the x86 port of our runtime to Intel Cilk Plus and Intel TBB using the same setup as in Section 2. The results verify that the LTA runtime has comparable performance to Intel TBB and is slightly faster in some cases because it is lighter weight (e.g., no C++ exceptions or task cancellation).

5.3 Cycle-Level Methodology

We utilize a co-simulation framework with gem5 [6] and PyMTL [45], a Python-based hardware modeling framework. Table 4 lists the corresponding gem5 configurations. *IO* describes the baseline scalar in-order processor, and *O3* describes the baseline four-way superscalar out-of-order processor. Multi-core configurations have four cores. 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. In Table 2, we supplement our evaluation with detailed information from our cycle-level simulation regarding fetch, μthread activity, issue slot utilization, and memory system sensitivity for three specific LTA configurations.

5.4 Area/Energy Modeling

Area/energy are estimated using component/event-based modeling based on Verilog RTL implementations of previously developed designs with structures comparable to those used in LTAs, FG-SIMT [33] and XLOOPS [67], which we synthesize and place-and-route using Synopsys DesignCompiler and IC Compiler with a TSMC 40 nm LP standard-cell library characterized at 1 V. We identify the dominant contributors to inform our component- and event-based models. We model SRAMs with CACTI [53].

We use FG-SIMT to model the area of a lane-group, DMU, and D$ crossbar network; and we use XLOOPS to model the area of the IMU and TMU. The dominant contributors are the L1 caches (33%), regfiles (26%), LLFUs (20%), SLFUs (10%), and assorted queues (7%). We do not have RTL for the rename table, reorder buffer, and arbitration logic, so we model them using flop-based 1r1w regfiles, integer ALUs, and muxes. We assume the GPP and LTA can share LLFUs and memory ports. Lacking *O3* RTL, we determine a reasonable scaling factor for *O3* vs. *IO without* L1 caches (≈5×) based on McPAT [42] and publicly available VLSI comparisons from ARM (A15 vs. A7 [13]), RISC-V (BOOM vs. Rocket [4]), and Alpha (21064 vs. 21164 vs. 21264 [35]). After including the same L1 memory system for both *IO* and *O3*, we estimate that *O3* increases area by roughly 50% which is likely conservative. Detailed estimates are shown in Table 5.

Tech	TSMC 40nm LP, 500MHz nominal frequency
ALU	4/10-cyc int mul/div, 6/6-cyc FP mul/div, 4/4-cyc FP add/sub
IO	1-way, 5-stage in-order, 32 phys regs
O3	4-way, out-of-order, 128 phys regs, 32-entry IQ and LSQ, 96-entry ROB, tournament branch pred
Caches	1-cycle, 2-way, 32KB L1I, 1-cycle 4-way 32KB L1D per core with 16-entry MSHRs; 20-cycle, 8-way, shared 1MB L2; MESI protocol
OCN	crossbar topology, 2-cycle fixed latency
DRAM	200ns fixed lat, 12.8GB/s bandwidth SimpleMemory model

Table 4: Cycle-Level System Configuration

	Area	**LTA**	**Area**	**LTA**	**Area**	**LTA**	**Area**	**LTA**	**Area**
IO	0.61	*4/1x8/1*	1.32	*4/1x8/2*	1.33	*4/1x8/4*	1.35	*4/1x8/8*	1.40
O3	0.91	*4/2x8/1*	1.35	*4/2x8/2*	1.36	*4/2x8/4*	1.38	*4/2x8/8*	1.42
		4/4x8/1	1.35	*4/4x8/2*	1.36	*4/4x8/4*	1.38	*4/4x8/8*	1.42
		8/1x4/1	1.41	*8/2x4/1*	1.44	*8/4x4/1*	1.44	*8/8x4/1*	1.44
		8/1x4/2	1.42	*8/2x4/2*	1.45	*8/4x4/2*	1.45	*8/8x4/2*	1.45
		8/1x4/4	1.45	*8/2x4/4*	1.47	*8/4x4/4*	1.47	*8/8x4/4*	1.48

Table 5: LTA Single-Core Area Estimates All area numbers are in mm^2 and include the L1 I$ and D$. See Section 5.4 for details.

We developed 70+ energy microbenchmarks to measure per-access energies for the dominant contributors (e.g., caches, regfiles, SLFUs/LLFUs) using gate-level simulation. Net activity factors are combined with post-PAR layout using Synopsys PrimeTime PX for power estimates. We built an event/component-based modeling tool that parses event traces from cycle-level simulations to estimate energy. Events in the LTA and O3 with no corresponding RTL were estimated using carefully tuned McPAT component-level models.

6 EVALUATION

In this section, we first explore the effect of spatial/temporal task decoupling on LTA performance under gradually reduced hardware resources within the context of a single core. We then evaluate the performance, area, and energy of realistic 32-μthread single-core LTA configurations, and we use the most promising single-core LTAs to show the potential for multiplicative speedup in CMP+LTA.

6.1 General Trends for Performance vs. Hardware Resources

Each subplot in Figure 9 shows the speedup of a single core augmented with various LTA configurations over the serial version of the benchmark executing on *IO*. The first row is for a regular benchmark (*mriq*), the second row is for an irregular benchmark (*sarray*), and the third row is the geometric mean speedup across all benchmarks. Each column makes a different assumption about the available hardware resources (left has more resources, right is more resource constrained). The x-axis in each subplot indicates the number of task groups. Different LTA configurations can provide the same number of task groups in different ways (e.g., *LTA-8/4x4/1* has four task groups via spatial decoupling, while *LTA-8/1x4/4* has four task groups via temporal decoupling). The black line (circles) indicates the general trend for spatial decoupling in isolation, the

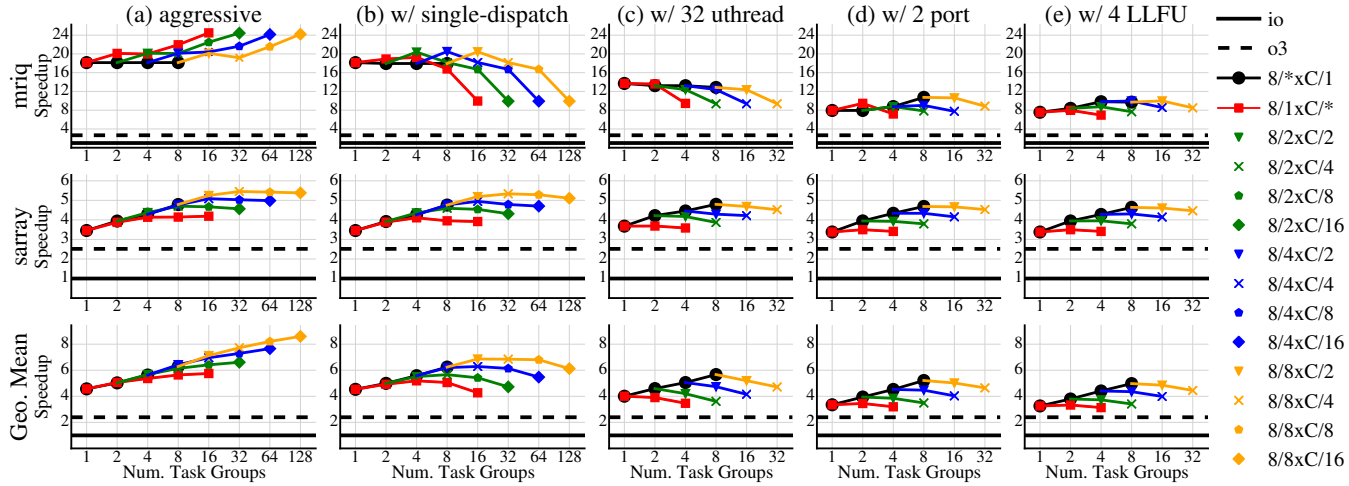


Figure 9: Performance vs. Available Hardware Resources – *mriq* (regular) and *sarray* (irregular) are shown along with the geometric mean of speedup across all apps. (a) has 128 µthreads, with an unrealistic eight memory ports, eight LLFUs, and aggressive multi-dispatch task group scheduler; (b) is the same as (a) except with a conservative single-dispatch frontend; (c) is the same as (b) except with 32 µthreads; (d) is the same as (c) except with two memory ports; (e) is the same as (d) except with four LLFUs; C = 16 for (a–b); C = 4 for (c–e).

red line (squares) indicates the general trend for temporal decoupling in isolation, and the remaining lines indicate the trends for combining both spatial and temporal decoupling.

128-µthread LTA with Aggressive Front-End – The results in Figure 9(a) assume a configuration with heavily over-provisioned resources. The LTA includes 128 µthreads across eight lanes, eight L1 data cache ports, eight LLFUs, and an aggressive front-end meaning each task group has its own 1 KB PIB (very unrealistic) and *each* lane-group pipeline supports: fetching from three PIBs per cycle, decoding three instructions per cycle, dispatching three instructions from different task groups per cycle, and increased writeback and commit bandwidth. The fully coupled *LTA-8/1x16/1* is able to achieve 18× speedup on *mriq* and 3.5× speedup on *sarray* over *IO* for several reasons. *LTA-8/1x16/1* has 8× more memory ports and LLFUs enabling *LTA-8/1x16/1* to exploit DLP in space (i.e. executing the same instruction across all eight lanes on different data). Critically, *LTA-8/1x16/1* is able to keep those resources busy even with a single task group. A single instruction can keep an issue unit busy for 16 cycles, giving the dispatch unit ample opportunity to issue an independent instruction to a different issue unit.

For very regular benchmarks like *mriq*, increased spatial decoupling has little impact on performance compared to *LTA-8/1x16/1*. The dispatch unit chooses the same schedule regardless of how the lanes are grouped spatially. Interestingly, increased temporal decoupling *does* improve performance compared to *LTA-8/1x16/1*. The aggressive front-end exploits ILP *and* TLP across task groups to better utilize the three issue units, and scheduling flexibility is enhanced because fewer chimes per task group means issues units are occupied less avoiding structural hazards. For more irregular benchmarks like *sarray* with both control and memory divergence, spatial and/or temporal decoupling can increase performance significantly since each task group can slip relative to the other task groups. Structural hazards due to the large chime in spatially decoupled configurations has less of an impact owing to fewer LLFU operations in irregular benchmarks. The overall results suggest that in a resource unconstrained environment, designers should favor temporal over spatial decoupling, especially if a workload is dominated by regular applications.

128-µthread LTA with Realistic Front-End – The results in Figure 9(b) assume the same configuration as in Figure 9(a) except with a more realistic front-end. Each task group has a single-cache-line PIB (32B) and each lane-group pipeline only supports fetching, decoding, and dispatching one instruction per cycle. Again, *LTA-8/1x16/1* can keep multiple issue units busy even with a single-instruction dispatch unit because it exploits DLP in time and ILP.

For very regular benchmarks like *mriq*, increased spatial decoupling has a slight negative impact on performance due to increased L1 instruction cache bandwidth pressure. A modest increase in temporal decoupling marginally improves performance; we can exploit TLP across task groups, but we also have 4–8 chimes per task group to exploit DLP in time. More extreme temporal decoupling actually reduces performance; while we can still exploit TLP across task groups to interleave independent instructions, this comes at the expense of decreasing issue-unit utilization. The results for *sarray* clearly demonstrates the benefit of decoupling to improve the performance on irregular benchmarks even with a more realistic front-end. The overall results suggest that with a more realistic front-end, designers should favor spatial decoupling or a moderate amount of temporal decoupling.

32-µthread LTA with Realistic Front-End – The results in Figure 9(c) assume the same configuration as in Figure 9(b) except with 32 µthreads instead of 128 µthreads across eight lanes. While the speedups for the 128-µthread configurations are impressive, the area overhead due to a large register file, rename table, and other data structures means the area normalized performance would likely be more modest. For the fully coupled configuration, fewer µthreads reduce the performance on *mriq* from 18× to 14×, but the performance on *sarray* is largely unchanged. While 128 µthreads can effectively exploit DLP on regular benchmarks, many of these µthreads are actually inactive on irregular benchmarks.

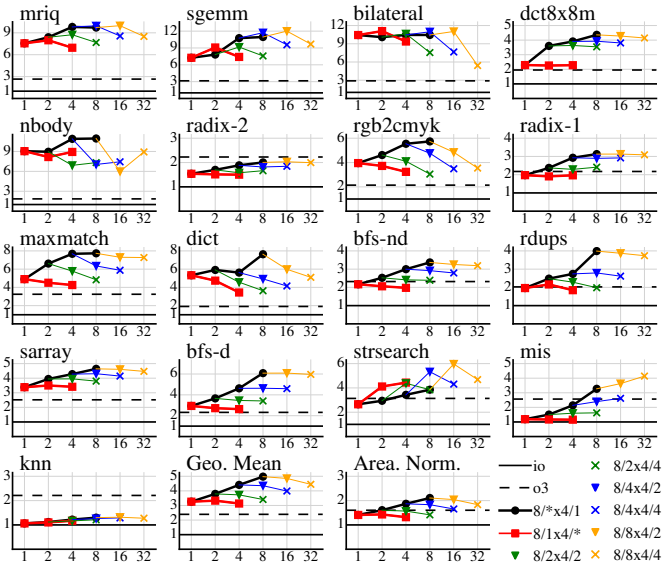


Figure 10: Performance of Single 8-lane 32-µthreads LTA – All results are normalized against scalar in-order core. Apps roughly organized from regular to irregular.

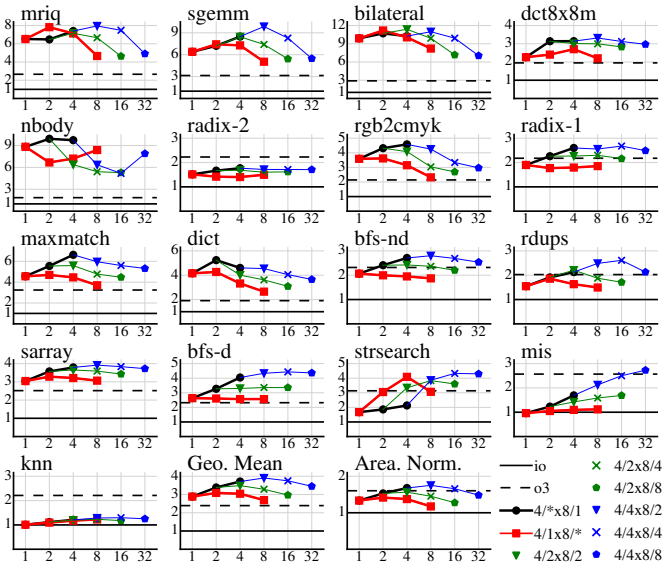


Figure 11: Performance of Single 4-lane 32-µthreads LTA – All results are normalized against scalar in-order core. Apps roughly organized from regular to irregular.

With fewer µthreads, moderate temporal decoupling no longer has any benefit because the chimes per task group quickly becomes too short to effectively utilize the issue units. The results for *sarray* show that moderate temporal decoupling can no longer even improve the performance of irregular benchmarks. The overall results suggest that with a more realistic front-end and number of µthreads, designers should favor spatial over temporal decoupling.

32-µthread LTA with Fewer Memory Ports and LLFUs – The results in Figure 9(d) assume the same configuration as in Figure 9(c) except with two instead of eight L1 data cache ports, and the results in Figure 9(e) further assume only four LLFUs. Figure 8 illustrates how the memory ports and LLFUs are shared across the eight lanes. An eight-ported L1 data cache would add significant area and energy overhead, and LLFUs are one of the larger subsystems in the LTA, so these final two configurations are more realistic. For *LTA-8/1x4/1* fewer memory ports and LLFUs reduce the performance on *mriq* from 14× to 8×, but the performance on *sarray* is only reduced from 4× to 3×. The data memory ports and LLFUs are usually less utilized in the more irregular benchmarks.

Increased spatial decoupling now improves performance even on regular benchmarks like *mriq* because all lanes arbitrate for memory ports causing latency divergence. In *LTA-8/1x4/1*, the LSU or LLFU interface must wait for the entire task group to finish to preserve lock-step execution. Spatial decoupling can better tolerate these dynamic latencies. Temporal decoupling continues to produce no meaningful performance improvement. Any improved tolerance to control or memory divergence is mitigated by the inability to keep multiple issue units busy with just 1–2 chimes per task group. Notice that the results in Figure 9(d) and (e) are similar. With a realistic number of memory ports we can no longer keep eight LLFUs busy, motivating our final hardware configuration. The overall results continue to suggest that with more realistic hardware resources, designers should favor spatial over temporal decoupling.

6.2 Detailed Per-Benchmark Performance

Figure 10 shows the performance of all 16 benchmarks across all 12 configurations in Figure 6(c) assuming a single-instruction dispatch unit, 32 µthreads, two memory ports, and four LLFUs. The benchmarks are sorted from more regular to less regular based on the average number of active µthreads in the fully coupled *LTA-8/1x4/1* configuration (see Table 2). For regular applications and two task groups (e.g., *LTA-8/2x4/1*, *LTA-8/1x4/2*) temporal and spatial decoupling can have similar performance (e.g., *mriq*, *sgemm*, *bilateral*, *nbody*, *radix*). However, once we increase the number of task groups to four or more, spatial decoupling almost always provides similar or better performance compared to temporal decoupling alone. The effective issue unit utilization data in Table 2 (see IU columns) helps explain one of the primary benefits of spatial vs. temporal decoupling. Across most of the benchmarks, *LTA-8/4x4/1* is able to achieve higher issue unit utilization compared to *LTA-8/1x4/4*. More generally, one of the key costs of decoupling in either space or time is an increase in the number of instruction fetches. Table 2 shows the number of instruction fetches is approximately 3× for *LTA-8/4x4/1* and *LTA-8/1x4/4* compared to *LTA-8/1x4/1*. This motivates the need for PIBs in the IMU to provide instruction fetch bandwidth amplification; simply having each lane group fetch scalar instructions from the L1 instruction cache would not be effective. Configurations which combine both spatial and temporal decoupling achieve intermediate performance, and only rarely exceed spatial decoupling alone (e.g., *strsearch*, *mis*).

Figure 11 shows the performance across all 12 configurations in Figure 6(d) with four lanes. These configurations still have two memory ports and four LLFUs. While reducing the number of lanes does decrease performance in some benchmarks, the impact is not as great as one might expect owing to the already reduced number of memory ports and LLFUs. The overall trends are similar, although there are occasions where combining a fully spatially

decoupled configuration with moderate temporal decoupling (e.g., *LTA-4/4x8/2*) can further improve performance on some irregular benchmarks. It is interesting to note that eight task groups (i.e., four µthreads per task group) achieves the peak average speedup in both the eight- and four-lane configurations.

Based on these results, it seems clear that designers should favor spatial over temporal decoupling *assuming limited resources*. Our conclusions might vary if we assumed support for superscalar execution, high-throughput data caches, and plenty of area for LLFUs. However, LTA is as an intra-core accelerator that can augment traditional CMPs which must prioritize many different workloads and usage scenarios. This motivates our focus on complexity-effective designs which can achieve high performance with limited resources.

6.3 Area and Energy Analysis

Figure 10 and Figure 11 also show the geometric mean of the performance across all benchmarks normalized by the area of each configuration. The highest performing LTA configurations are able to achieve approximately a 2× improvement in area-normalized performance compared to *IO* and 1.3× improvement in area-normalized performance compared to *O3*. Note that this is a conservative analysis, since these results are relative to the *serial* version of the benchmarks running on *IO* and *O3*. As we shall see in the next section, we would need to use the parallel versions to scale across multiple *IO* and *O3* cores.

Figure 12 shows the absolute energy breakdowns for the *IO* and *O3* baselines and the most promising LTA configurations for *mriq* and *sarray*. The fully coupled *LTA-8/1x4/1* configuration is able to amortize front-end energy across many µthreads on *mriq* but at the cost of increased register file energy. Spatial decoupling reduces some of this amortization by increasing the number of instruction cache accesses and adding additional energy overhead for PIBs. The LTA energy for *sarray* is higher than *IO* since control divergence results in more instruction fetch accesses and PIB management overhead across all configurations. The LTA data cache energy is higher since the parallel runtime (used even on single-core LTA) results in an increased number of memory accesses. Figure 13 shows the energy efficiency vs. performance for all benchmarks normalized to both *IO* and *O3*. LTA is less energy efficient compared to a single *IO* partly because the serial version simple does less work compared to the parallel version used with the LTA configurations. Regardless, even in the single-core context, LTA is able to achieve higher raw performance and area-normalized performance compared to *IO*, and is able to achieve higher raw performance, area-normalized performance, and energy efficiency compared to *O3*.

6.4 CMP vs. CMP+LTA

Figure 14 shows performance of a quad-core system with an LTA per core. The results confirm that *CMP+LTA* achieves multiplicative speedups by exploiting loop-task parallelism both across and within cores. Referring back to Figure 10 and Figure 11, the geometric mean speedup of the most promising LTAs is 4.0–5.0× over *IO*, and using the runtime on *CMP-IO* yields an average speedup of 2.85× (see Table 2). We see that all LTAs are able to achieve close to multiplicative speedups of 10–12×. 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×.

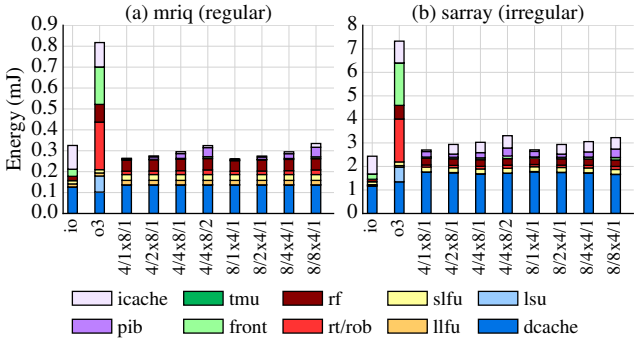


Figure 12: Energy Breakdown

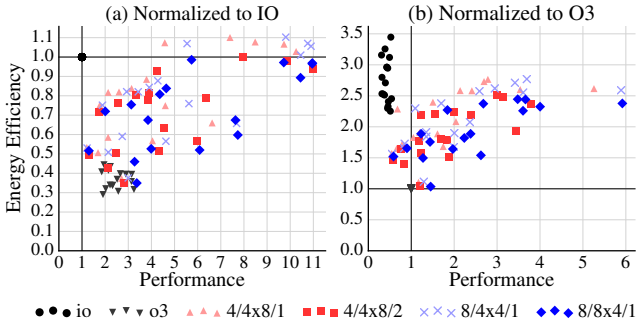


Figure 13: Energy Efficiency vs. Performance – Each point represents an application executing on a certain LTA configuration normalized to the serial version.

Compared to a more aggressive *CMP-O3*, *CMP+LTA* improves raw performance by 2.3×, performance per area by 1.5× (excluding the L2), and energy efficiency by 3.2×. Note that these *CMP-IO* and *CMP-O3* energy results are based on the parallel benchmarks running on the multicore configurations as opposed to the serial versions discussed in the previous section. In terms of productivity, 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 of the kernel can be written and compiled once, then executed on a system with any combination of GPPs and homogeneous or heterogeneous LTAs.

7 RELATED WORK

In this section, we first compare GPGPUs and CMP+LTAs before discussing recent research proposals that have attempted to address the two key challenges discussed throughout this paper. To our knowledge, LTA is one of the first proposals to address both of these challenges for CMPs through a combination of instruction set and microarchitectural design.

7.1 GPGPUs versus CMP+LTAs

Our approach is partly inspired by GPGPUs. GPGPUs use threads as a unifying abstraction in the programming model, instruction set, and microarchitecture. Recently, persistent threads (PT) has been proposed to map task-based frameworks to GPGPUs. In this scheme, the number of GPGPU threads is configured to equal the number of hardware threads, each warp is treated as a single worker,

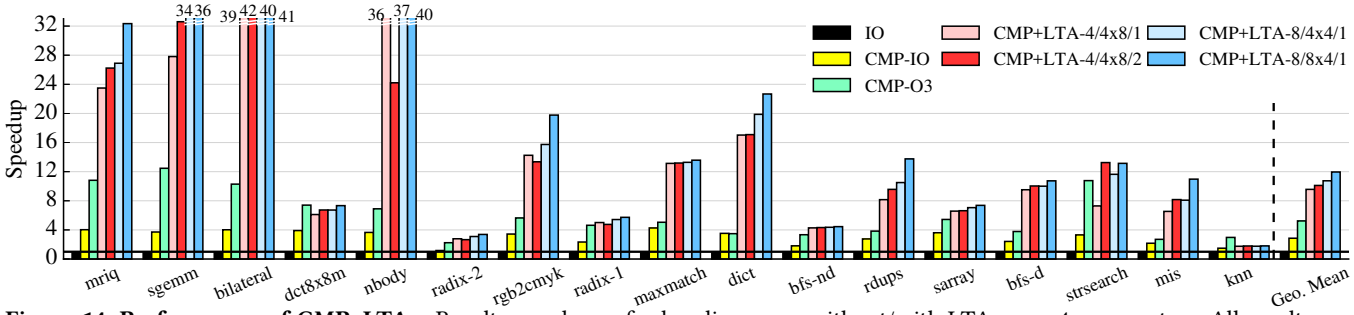


Figure 14: Performance of CMP+LTA – Results are shown for baseline cores without/with LTAs on a 4-core system. All results are normalized against the performance of a single in-order core for each kernel.

and workers implement a software runtime instead of using the GPGPU's hardware scheduler [25, 68]. However, PT re-introduces the intra-core parallel abstraction gap since the programmer explicitly manages tasks between warps as well as data-parallelism within a warp, and PT still struggles to efficiently execute irregular tasks using SIMD resources. It is simply challenging to use GPGPUs to accelerate irregular task-based parallel programs.

System Architecture – GPGPUs are designed from the ground up to prioritize high-throughput execution of massively parallel applications with tens of thousands of very similar threads. Both discrete and integrated GPGPUs use an offload model, which increases the minimum reasonable problem size. GPGPUs use hardware scheduling of thread blocks between cores and a memory system tuned for high-throughput rendering (e.g., tiny L1 caches, potentially no shared last-level cache or chip-wide cache coherence, large specialized scratchpad and texture memories). CMP+LTA is fundamentally a CMP. It supports: hosted execution, intermingling general-purpose serial code with parallel execution, diverse multiprogramming, and different software schedulers/runtimes. A CMP+LTA memory system is tuned for general-purpose software with deeper, coherent cache hierarchies. CMP+LTA offers a balance between low-latency and high-throughput execution.

Intra-Core Parallel Abstraction – GPGPUs use threads within a core, while LTAs use loop tasks. This is a subtle, yet important difference. The `jalr.lta` instruction enables "serial execution," meaning a microarchitecture without an LTA can execute a `jalr.lta` as a standard indirect function call. GPGPU's thread model has no such equivalent serial execution (at least at the instruction-set level) since thread state is exposed in the architecture. GPGPU just-in-time compilation can help, but also serves to reinforce the differences between these approaches. An LTA supports a subset of the GPP instruction set; a function can be compiled once and called from either the GPP or the LTA. In addition, using loop tasks enables the hardware to manage partitioning core tasks into μtasks and to control the mapping of the iteration space.

Intra-Core Microarchitecture – GPGPUs include support for managing thread divergence/reconvergence. LTAs also include such support, but there are important microarchitectural differences. GPGPUs are optimized for high-throughput and regular workloads; each SM includes a large number of lanes with tens of warps with 1–2 chimes/warp, huge SRAM-based minimally ported register files, dynamic operand collection [44], very deep execution pipelines, and stack-based reconvergence [17]. LTAs are explicitly designed to be tightly integrated with a GPP and include just a few task groups to ensure small area overhead. This results in a very different microarchitecture with a small highly ported register file, shallow execution pipelines, vector chaining, and PC-based reconvergence via a PFB. LTA uses spatial decoupling, which has no equivalent in commercial GPGPUs (although there have been research proposals [64]), and LTAs use more temporal coupling than GPGPUs, which enables hiding execution latencies with chimes instead of interleaving warps.

In summary, while GPGPUs and CMP+LTAs have some high-level similarities, each platform targets a very different application domain. GPGPUs target massively parallel, regular applications while CMP+LTAs target more general-purpose computing workloads. To reinforce this point, we ported the applications in Table 1 to CUDA, and we evaluated their performance on a NVIDIA Tesla C2075 GPGPU. This required non-trivial programmer effort to manually optimize control and memory divergence. The speedup relative to the CMP baseline from Figure 1 was: *sgemm* = 477×, *dct8x8* = 59×, *mriq* = 132×, *rgb2cmyk* = 32×, *bfs-nd* = 4×, *maxmatch* = 1×, *strsearch* = 5×. While we saw impressive results for regular kernels (*sgemm*), irregular kernels saw very little speedup even with significant manual optimization (*strsearch*). This is not too surprising given a GPGPUs primary focus on exploiting massively regular data parallelism.

7.2 Intra-Core Parallel Abstraction Gap

Most of the prior work on the first challenge has focused on software-only techniques. There has, of course, been significant work on generally improving work-stealing runtimes [1, 2, 16, 20, 37], but much of the work on leveraging packed-SIMD extensions in work-stealing runtimes has required the programmer to use a task-based abstraction across cores and then use explicit intra-core vectorization. Intel Cilk Plus provides explicit array notation [29] and Intel ISPC supports a SPMD programming model [15]. Both can be compiled to packed-SIMD extensions but require the programmer to explicitly manage two separate parallel abstractions. One can also view work on frameworks that attempt to unify CPU and GPGPU execution (e.g, OpenCL [55], OpenMP [5], C++ AMP [66]) as making progress in closing the abstraction gap, but these frameworks use offload models more suitable to GPGPUs and their ability to leverage packed-SIMD extensions is somewhat limited (especially for irregular tasks). Indeed, case studies using OpenMP and OpenCL illustrate the need to focus on regular loops and/or use explicit

vectorization [49, 51]. Recent work by Ren et al. uses a separate specification language to enable Cilk programs to take advantage of packed-SIMD extensions in both the base case and recursive steps in some limited instances [60, 61], but this work still requires explicit vectorization to deal with the irregularity inherent in these programs. Unlike LTA, binaries with packed-SIMD extensions cannot be transparently executed serially, nor can they be executed on architectures with a different packed-SIMD width. While we have no doubt that the ingenuity of software researchers will continue to close this abstraction gap using software-only solutions, in this work we are essentially answering a different question: Can lightweight changes to the instruction set and microarchitecture enable a fundamental change in how we write loop-task parallel programs such that they can seamlessly exploit both inter- and intra-core parallel execution resources?

7.3 Inefficient Execution of Irregular Tasks

Figure 15 illustrates how we might generally position other approaches in our task-coupling taxonomy. There has been a tremendous amount of work on enabling GPGPUs to better tolerate control and memory divergence [14, 18, 22, 23, 52, 62]. Variable warp sizing (VWS) explores spatial task decoupling by splitting a 32-thread warp into 4-lane slices that can gang together [64]. Essentially, VWS supports dynamically moving in our task-coupling taxonomy at runtime. VWS does not explore the trade-offs involved with temporal task coupling. While VWS provides further evidence for the benefit of spatial task decoupling when executing irregular applications, it is still fundamentally a GPGPU technique. LTA is focused on integration with CMPs, which leads to a very different system architecture, parallel abstraction, and microarchitecture. Temporal SIMT proposes single-lane lane groups with tight temporal coupling in the context of a GPGPU, although this proposal has yet to be fully evaluated [32]. There is less related work in efficiently executing irregular tasks in the context of task-parallel runtimes and packed-SIMD extensions. Other decoupled lane approaches do not explore temporal task coupling nor how to integrate such accelerators into a standard work stealing runtime [67]. Other coupled lane approaches [21, 33] struggle to achieve high performance on irregular tasks, and do not address how to integrate such accelerators into a work stealing runtime.

Although vector-threading (VT) looks to seamlessly intermingle the vector and multithreaded architectural design patterns [34, 39], VT still struggles with the intra-core parallel abstraction gap. High-performance VT codes require significant use of vector memory operations interleaved with vector-fetched blocks. [39] uses two very different abstractions for inter-core (very simplistic thread library) and intra-core (explicit vector memory operations and vector-fetched blocks). Unlike `jalr.lta` which is just a hint, the `vfetch` instruction *requires* an accelerator complicating application porting. [39] shows VT performs quite poorly for "SIMT-like" programs because each vector-fetched block is for a single iteration, while LTA focuses on hardware support for loop-tasks where each µthread processes many iterations (and thus address computation, shared constant loads are refactored out of the inner task loop). [34] is decoupled in space and time (owing to its very different AIB execution model), does not include any support for reconvergence or memory coalescing, and requires a brand new compiler owing to a completely new "AIB-with-clusters" ISA in vector-fetched blocks.

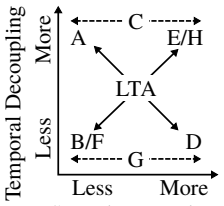


Figure 15: Positioning Related Work in Taxonomy – A = GPGPU/MICs use spatial coupling w/ temporal decoupled warps or control threads; B = traditional vector (e.g., Tarantula [21]); C = VWS [64]; D = temporal SIMT [32]; E = VT [34]; F = VT [39]; G = VLT [63]; H = XLOOPS [67].

Prior work on vector-lane threading (VLT) explores spatial decoupling (although not temporal decoupling) in traditional vector architectures [63]. VLT also observes the trade-off between tight coupling for regular codes vs. loose coupling for irregular codes. For a fully spatially decoupled configuration, VLT is able to use each lane to execute a scalar thread, but requires a relatively large 4KB per-lane instruction cache. While LTA draws upon VLT's insights, the actual LTA microarchitecture is different in order to support the new `jalr.lta` instruction which is a key contribution. VLT is fundamentally a multithreaded vector architecture, and suffers from the intra-core parallel abstraction gap. [63] uses the Cray vectorizing compiler, but our experiences with Intel ICC auto-vectorizion have been disappointing, probably due to our more "task-like" applications with many nested for/while loops, (recursive) function calls, data-dependent conditionals, unstructured memory accesses, and AMOs (see Table 2).

The LTA microarchitectural template leverages the best of prior work on vector, SIMT, VT, and VLT to support a novel instruction set with minimal software changes and enables broader design-space exploration than prior work.

8 CONCLUSIONS

Augmenting a CMP with LTAs is a promising direction for improving the productivity (i.e., minimal software changes) and performance (i.e., multiplicative speedup) of loop-task parallel programs. The novel `jalr.lta` instruction illustrates the potential for directly encoding task execution in the software/hardware interface to enable both traditional execution on GPPs and specialized execution on LTAs. Our task-coupling taxonomy provides a simple way to conceptualize the various approaches for decoupling task execution to improve performance on irregular programs. One of the key conclusions of the work is that designers should consider favoring spatial decoupling over temporal decoupling within resource constrained contexts (e.g., limited number of µthreads, memory ports, and LLFUs). We see LTAs as a first step towards accelerating even more general parallel patterns in task-based frameworks (e.g., nested, recursive, and dynamic tasks).

ACKNOWLEDGMENTS

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 coprocessor. We thank Dave Albonesi and Adrian Sampson for their valuable feedback.

REFERENCES

[1] Umut A. Acar, Arthur Chargeéraud, and Mike Rainey. 2013. Scheduling Parallel Programs by Work Stealing with Private Deques. *Symp. on Principles and practice of Parallel Programming (PPoPP)* (Feb 2013).

[2] Kunal Agrawal, Charles E. Leiserson, Yuxiong He, and Wen Jing Hsu. 2008. Adaptive Work-stealing with Parallelism Feedback. *ACM Trans. on Computer Systems (TOCS)* 26, 3 (Sep 2008), 7.

[3] Randy Allen and Steve Johnson. 1988. Compiling C for Vectorization, Parallelization, and Inline Expansion. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)* (Jun 1988).

[4] Krste Asanovic, David A. Patterson, and Christopher Celio. 2015. *The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor.* Technical Report. UCB/EECS-2015-167.

[5] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. 2009. The Design of OpenMP Tasks. *IEEE Trans. on Parallel and Distributed Systems (TPDS)* 20, 3 (Mar 2009), 404–418.

[6] Nathan Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 Simulator. *SIGARCH Computer Architecture News (CAN)* 39, 2 (Aug 2011), 1–7.

[7] J. Bolaria. 2012. Xeon Phi Targets Supercomputers. *Microprocessor Report* (Sep 2012).

[8] Alexandar Branover, Denis Foley, and Maurice Steinman. 2012. AMD Fusion APU: Llano. *IEEE Micro* 32, 2 (Mar/Apr 2012), 28–37.

[9] M. Burtscher, R. Nasre, and K. Pingali. 2012. A Quantitative Study of Irregular Programs on GPUs. *Int'l Symp. on Workload Characterization (IISWC)* (Oct 2012).

[10] Colin Campbell, Ralph Johnson, Ade Miller, and Stephen Toub. 2010. *Parallel Programming with Microsoft .NET: Design Patterns for Decomposition and Coordination on Multicore Architectures (Patterns & Practices).* Microsoft Press.

[11] Loyd Case. 2015. MIPS Broadens Reach with New Cores. *Microprocessor Report* (Dec 2015).

[12] David Chase and Yossi Lev. 2005. Dynamic Circular Work-stealing Deque. *Symp. on Parallel Algorithms and Architectures (SPAA)* (Jun 2005).

[13] Peter Clarke. 2013. How ARM's Cortex-A7 Beats the A15. *EE Times* (Jul 2013). `http://www.eetimes.com/author.asp?section_id=36&doc_id=1318968`.

[14] Sylvian Collange. 2011. *Stack-less SIMT Reconvergence at Low Cost.* Technical Report HAL-00622654. ARENAIRE.

[15] Intel SPMD Program Compiler. 2015. DesignWare ARC Processor Cores. Online Webpage. (2015). `https://ispc.github.io`.

[16] Gilberto Contreras and Margaret Martonosi. 2008. Characterizing and Improving the Performance of Intel Threading Building Blocks. *Int'l Symp. on Workload Characterization (IISWC)* (Sep 2008).

[17] Brett W. Coon and John Erik Lindholm. 2008. System and Method for Managing Divergent Threads in a SIMD Architecture. US Patent 7353369. (Apr 2008).

[18] Gregory Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Andrew Keer, Haicheng Wu, and Sudhakar Yalamanchili. 2011. SIMD Re-Convergence at Thread Frontiers. *Int'l Symp. on Microarchitecture (MICRO)* (Dec 2011).

[19] Neil Dickson, Kamran Karimi, and Firas Hamze. 2011. Importance of Explicit Vectorization for CPU and GPU Software Performance. *Journal of Computational Physics (JCP)* 230 (Jun 2011), 5383–5398. Issue 13.

[20] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. 2009. Scalable Work Stealing. *Int'l Conf. on High Performance Networking and Computing (Supercomputing)* (Nov 2009).

[21] Roger Espasa, Federico Ardanaz, Joel Emer, Stephen Felix, Julio Gago, Roger Gramunt, Isaac Hernandez, Toni Juan, Geoff Lowney, Matthew Mattina, and André Seznec. 2002. Tarantula: A Vector Extension to the Alpha Architecture. *Int'l Symp. on Computer Architecture (ISCA)* (Jun 2002).

[22] Wilson W.L. Fung and Tor M. Aamodt. 2011. Thread Block Compaction for Efficient SIMT Control Flow. *Int'l Symp. on High-Performance Computer Architecture (HPCA)* (Feb 2011).

[23] Wilson W.L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. 2009. Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware. *ACM Trans. on Architecture and Code Optimization (TACO)* 6, 2 (Jun 2009), 1–35.

[24] Vekatraman Govindaraju, Tony Nowatzki, and Karthikeyan Sankaralingam. 2013. Breaking SIMD Shackles with an Exposed Flexible Microarchitecture and the Access Execute PDG. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)* (Sep 2013).

[25] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. 2012. A Study of Persistent Threads Style GPU Programming for GPGPU Workloads. *Innovative Parallel Computing (InPar)* (2012).

[26] Linley Gwennap. 2015. Cortex-A35 Extends Low End. *Microprocessor Report* (Nov 2015).

[27] Linley Gwennap. 2015. Cortex-A57 is Most Efficient CPU. *Microprocessor Report* (Feb 2015).

[28] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. 2011. Accelerating CUDA Graph Algorithms at Maximum Warp. *Symp. on Principles and practice of Parallel Programming (PPoPP)* (Feb 2011).

[29] Intel. 2013. Intel Cilk Plus Language Extension Specification, Version 1.2. Intel Reference Manual. (Sep 2013). `https://www.cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_1.2.htm`.

[30] Intel. 2015. Intel Threading Building Blocks. Online Webpage. (2015). `https://software.intel.com/en-us/intel-tbb`.

[31] David Kanter. 2015. Knights Landing Reshapes HPC. (Sep 2015).

[32] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. 2011. GPUs and the Future of Parallel Computing. *IEEE Micro* 31, 5 (Sep/Oct 2011), 7–17.

[33] Ji Kim, Christopher Torng, Shreesha Srinath, Derek Lockhart, and Christopher Batten. 2013. Microarchitectural Mechanisms to Exploit Value Structure in Fine-Grain SIMT Architectures. *Int'l Symp. on Computer Architecture (ISCA)* (Jun 2013).

[34] Ronny Krashinsky, Christopher Batten, Mark Hampton, Steve Gerding, Brian Pharris, Jared Casper, and Krste Asanović. 2004. The Vector-Thread Architecture. *Int'l Symp. on Computer Architecture (ISCA)* (Jun 2004).

[35] R. Kumar, K.I. Farkas, N.P. Jouppi, P. Ranganathan, and D.M. Tullsen. 2003. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. *Int'l Symp. on Microarchitecture (MICRO)* (Dec 2003).

[36] Doug Lea. 2000. A Java Fork/Join Framework. *Java Grade Conference* (Jun 2000).

[37] I-Ting Angelina Lee, Aamir Shafi, and Charles E. Leiserson. 2012. Memory-Mapping Support for Reducer Hyperobjects. *Symp. on Parallel Algorithms and Architectures (SPAA)* (Jun 2012).

[38] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. 2010. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. *Int'l Symp. on Computer Architecture (ISCA)* (Jun 2010).

[39] Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. 2011. Exploring the Tradeoffs between Programmability and Efficiency in Data-Parallel Accelerator Cores. *Int'l Symp. on Computer Architecture (ISCA)* (Jun 2011).

[40] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. 2009. The Design of a Task Parallel Library. *Conf. on Object-Oriented Programming Systems Languages and Applications (OOPSLA)* (Oct 2009).

[41] Charles E. Leiserson. 2009. The Cilk++ Concurrency Platform. *Design Automation Conf. (DAC)* (Jul 2009).

[42] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. *Int'l Symp. on Microarchitecture (MICRO)* (Dec 2009).

[43] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. 2008. NVIDIA Tesla: A Unified Graphics and Computer Architecture. *IEEE Micro* 28, 2 (Mar/Apr 2008), 39–55.

[44] Samuel Liu, John Erik Lindholm, Ming Y Siu, Brett W. Coon, and Stuart F. Oberman. 2010. Operand Collector Architecture. US Patent US7834881 B2. (Nov 2010).

[45] Derek Lockhart, Gary Zibrat, and Christopher Batten. 2014. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. *Int'l Symp. on Microarchitecture (MICRO)* (Dec 2014).

[46] C. Loeffler, A. Ligtenberg, and G. S. Moschytz. 1989. Practical Fast 1-D DCT Algorithms with 11 Multiplications. *Int'l Conf. on Acoustics Speech and Signal Processing* (May 1989).

[47] L. Luo, M. Wong, and W. Hwu. 2010. An Effective GPU Implementation of Breadth-First Search. *Design Automation Conf. (DAC)* (Jun 2010).

[48] Saeed Maleki, Yaoqing Gao, Maria Garzaran, Tommy Wong, and David Padua. 2011. An Evaluation of Vectorizing Compilers. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)* (Oct 2011).

[49] Matt Martineau, James Price, Simon McIntosh-Smith, and Wayne Gaudin. 2016. Pragmatic Performance Portability with OpenMP 4.x. *Int'l Workshop on OpenMP* (Sep 2016).

[50] Alberto J. Martinez, Atiq A. Bajwa, David L. Hill, Erik Hallnor, Hong Jiang, Martin Dixon, Michael Derr, Mikal Hunsaker, Rajesh Kumar, Randy B. Osborne, Ravi Rajwar, Ronak Singhal, Reynold D'Sa, Robert Chappell, Shiv Kaushik, Srinivas Chennupaty, Stephan Jourdan, Steve Gunther, Tom Piazza, and Ted D. Kanter Burton. 2014. Haswell: The Fourth-Generation Intel Core Processor. *IEEE Micro* 34, 2 (Mar/Apr 2014), 6–20.

[51] Simon McIntosh-Smith, Michael Boulton, Dan Curran, and James Price. 2014. On the Performance Portability of Structured Grid Codes on Many-Core Computer Architectures. *Int'l Symp. on Supercomputing (ICS)* (Jun 2014).

[52] Jiayuan Meng, David Tarjan, and Kevin Skadron. 2010. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. *Int'l Symp. on Computer Architecture (ISCA)* (Jun 2010).

[53] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. 2009. CACTI 6.0: A Tool to Model Large Caches. (2009).

[54] R. Nasre, M. Burtscher, and K. Pingali. 2013. Morph Algorithms on GPUs. *Symp. on Principles and practice of Parallel Programming (PPoPP)* (Feb 2013).

[55] OpenCL. 2011. OpenCL Specification, v1.2. Khronos Working Group. (2011). `http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`.

[56] OpenMP. 2013. OpenMP Application Program Interface, Version 4.0. OpenMP Architecture Review Board. (Jul 2013). http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[57] Oracle. 2015. Java API: ForkJoinPool. Online API Documentation. (2015). http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinPool.html.

[58] James Reinders. 2007. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism.* O'Reilly.

[59] James Reinders. 2012. An Overview of Programming for Intel Xeon Processors and Intel Xeon Phi Coprocessors. Intel White Paper. (2012). https://software.intel.com/sites/default/files/article/330164/an-overview-of-programming-for-intel-xeon-processors-and-intel-xeon-phi-coprocessors_1.pdf.

[60] Bin Ren, Youngjoon Jo, Sriram Krishnamoorthy, Kunal Agrawal, and Milind Kulkarni. 2015. Efficient Execution of Recursive Programs on Commodity Vector Hardware. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)* (Jun 2015).

[61] Bin Ren, Sriaram Krishnamoorthy, Kunal Agrawal, and Milind Kulkarni. 2017. Exploiting Vector and Multicore Parallelism for Recursive, Data- and Task-Parallel Programs. *Symp. on Principles and practice of Parallel Programming (PPoPP)* (Feb 2017).

[62] Minsoo Rhu and Mattan Erez. 2012. CAPRI: Prediction of Compaction-Adequacy for Handling Control-Divergence in GPGPU Architectures. *Int'l Symp. on Computer Architecture (ISCA)* (Jun 2012).

[63] Suzanne Rivoire, Rebecca Schultz, Tomofumi Okuda, and Christos Kozyrakis. 2006. Vector Lane Threading. *Int'l Conf. on Parallel Processing (ICPP)* (Aug 2006).

[64] Timothy G. Rogers, Daniel R. Johnson, Mike O'Connor, and Stephen W. Keckler. 2015. A Variable Warp Size Architecture. *Int'l Symp. on Computer Architecture (ISCA)* (2015).

[65] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, and Harsha Vardhan Simhadri. 2012. Brief Announcement: The Problem Based Benchmark Suite. *Symp. on Parallel Algorithms and Architectures (SPAA)* (Jun 2012).

[66] S. Somasegar. 2011. Targeting Heterogeneity with C++ AMP and PPL. MSDN Blog. (Jun 2011). http://blogs.msdn.com/b/somasegar/archive/2011/06/15/targeting-heterogeneity-with- c-amp-and-ppl.aspx.

[67] Shreesha Srinath, Berkin Ilbeyi, Mingxing Tan, Gai Liu, Zhiru Zhang, and Christopher Batten. 2014. Architectural Specialization for Inter-Iteration Loop Dependence Patterns. *Int'l Symp. on Microarchitecture (MICRO)* (Dec 2014).

[68] Stanley Tzeng, Brandon Lloyd, and John D. Owens. 2012. A GPU Task-Parallel Model with Dependency Resolution. *IEEE Computer* 45, 8 (Aug 2012), 34–41.