AFRL-AFOSR-VA-TR-2019-0093

Exploiting Amorphous Data Parallelism Through Software and Architecture Co-Design

Christopher Batten
CORNELL UNIVERSITY
373 PINE TREE RD
ITHACA, NY 14850-2820

04/16/2019
Final Report

Air Force Research Laboratory
AF Office Of Scientific Research (AFOSR)/RTA2

DISTRIBUTION A: Distribution approved for public release.

Arlington, Virginia 22203
Air Force Materiel Command

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
16-04-2019	Final Performance	01 Sep 2015 to 31 Aug 2018

4. TITLE AND SUBTITLE
Exploiting Amorphous Data Parallelism Through Software and Architecture Co-Design

5a. CONTRACT NUMBER

5b. GRANT NUMBER
FA9550-15-1-0194

5c. PROGRAM ELEMENT NUMBER
61102F

6. AUTHOR(S)
Christopher Batten

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
CORNELL UNIVERSITY
373 PINE TREE RD
ITHACA, NY 14850-2820 US

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AF Office of Scientific Research
875 N. Randolph St. Room 3112
Arlington, VA 22203

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/AFOSR RTA2

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
AFRL-AFOSR-VA-TR-2019-0093

12. DISTRIBUTION/AVAILABILITY STATEMENT
A DISTRIBUTION UNLIMITED: PB Public Release

13. SUPPLEMENTARY NOTES

14. ABSTRACT
To address serious technology challenges, computer system designers are increasingly turning to a heterogeneous mix of general-purpose multicores and data-parallel accelerators integrated onto a single die. Unfortunately, this heterogeneity clearly increases complexity and costs at all levels of the computing stack. To complicate matters further, emerging application workloads are often exhibit amorphous data-parallelism with irregular control flow, unstructured data accesses, atomic tasks, and dynamic work generation. Significant breakthroughs to address these challenges will not happen by exploring either software or hardware in isolation. Therefore, this project used a cross-layer approach to rethink the software and hardware for heterogeneous systems with specific attention to efficiently supporting amorphous data parallelism. At the heart of our proposal was a new explicit-parallel-call (XPC) architectural design pattern that is based on the concept of explicitly encoding parallel function calls in the software/hardware interface.

15. SUBJECT TERMS
Parallelism, Amorphous Data, Software

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			NGUYEN, TRISTAN
Unclassified	Unclassified	Unclassified	UU		19b. TELEPHONE NUMBER *(Include area code)* 703-696-7796

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

AFOSR YIP Final Report

Grant Number:	FA9550-15-1-0194
Year:	Sep. 2015–Aug. 2018
Lead Organization:	Cornell University
Project Title:	Exploiting Amorphous Data Parallelism Through Software and Architecture Co-Design
Technical Point of Contact:	Dr. Christopher Batten Cornell University, ECE addr: 323 Rhodes Hall, Ithaca, NY 14853 phone: 607-255-2672 email: cbatten@cornell.edu
Administrative Point of Contact:	Columbia Warren Cornell University, Sponsored Projects Office addr: 373 Pine Tree Road, Ithaca, NY 14850 phone: 607-255-0655; fax: 607-255-5058 email: cjw43@cornell.edu
AFOSR Program Manager:	Dr. Erik Blasch AFOSR, Information Directorate email: erik.blasch.1@us.af.mil

1. Executive Summary

To address serious technology challenges, computer system designers are increasingly turning to a heterogeneous mix of general-purpose multicores and data-parallel accelerators integrated onto a single die. Unfortunately, this heterogeneity clearly increases complexity and costs at all levels of the computing stack. To complicate matters further, emerging application workloads are often exhibit amorphous data-parallelism with irregular control flow, unstructured data accesses, atomic tasks, and dynamic work generation. Significant breakthroughs to address these challenges will not happen by exploring either software or hardware in isolation. Therefore, this project used a cross-layer approach to rethink the software and hardware for heterogeneous systems with specific attention to efficiently supporting amorphous data parallelism. At the heart of our proposal was a new explicit-parallel-call (XPC) architectural design pattern that is based on the concept of explicitly encoding parallel function calls in the software/hardware interface.

The project was organized around two research thrusts. In *Research Thrust #1: XPC Hardware*, we designed a new XPC instruction set and investigated three microarchitectures: tightly coupled lanes where parallel function calls are executed in lock step, loosely coupled lanes where parallel function calls are executed using dynamic load balancing, and cooperative multicore where traditional in-order cores work together with hardware support for work redistribution. In *Research Thrust #2: XPC Software*, we explored a productive XPC programming framework; ported interesting amorphous data-parallel applications to this framework; investigated dynamic binary translation to enable XPC binaries to run on legacy systems; and explored new runtime systems to adaptively schedule XPC binaries on various heterogeneous tiles. This work will hopefully benefit the Air Force by contributing fundamental research on techniques for maintaining portability across heterogeneous systems, and by demonstrating the potential for XPC in computational contexts relevant to the Air Force mission, e.g., high-performance computing for modeling and simulation, and embedded systems for unmanned aerial vehicles.

We began by mapping a set of benchmarks to a traditional chip-multiprocessor (CMP) and two programmable accelerators: an Intel many-integrated-cores (MIC) and an NVIDIA general-purpose graphics

processing unit (GPGPU). We implemented these applications using various combinations of existing software frameworks and hardware architectures. Our experiences gave us deep insight into the fundamental trade-offs between productivity, portability, and performance (the 3Ps) and helped provide additional motivation for the XPC vision. We then developed the new XPC instruction set which involves adding a new `jalr.xpc` instruction to a standard RISC instruction set. The `jalr.xpc` instruction is an instantiation of the explicit-parallel-call design pattern discussed in the proposal. This instruction elegantly encodes loop-task execution by explicitly identifying the specific indirect function call used to execute loop-tasks in a software runtime. We developed an XPC programming framework which enables programmers to express both loop-level, fork-join, and nested parallelism, and we also developed a state-of-the-art work-stealing runtime which includes support for child-stealing, occupancy-based victim selection, and Chase-Lev task queues. The runtime is able to take advantage of the new XPC instruction set. We created a single, unified microarchitectural template that can be configured at design time to use either tightly coupled lanes, loosely coupled lanes, or a hybrid of both tightly and loosely coupled lanes. We developed a novel taxonomy that helps explain the space of these accelerators, and we conducted a detailed design-space exploration of the XPC microarchitecture. This design-space exploration spans application development, runtime development, instruction set design, microarchitectural design, and VLSI implementation. Compared to an in-order (out-of-order) CMP baseline, XPC yields average improvements of 5.5× (3.0×) in raw performance, 2.5× (1.7×) in performance per area, and 1.2× (2.5×) in energy efficiency. Our results suggest that augmenting CMPs with lightweight XPC engines improves performance and energy efficiency on both regular and irregular loop-task parallel programs with minimal software changes.

This project led to the publication of the following four top-tier conference publications, two Ph.D. theses, and one workshop paper:

- Christopher Torng, Moyang Wang, and Christopher Batten. "Asymmetry-Aware Work-Stealing Schedulers." *43rd ACM/IEEE Int'l Conf. on Computer Architecture (ISCA)*, June 2016.
- Ji Kim, "Software/Hardware Co-Design to Improve Productivity, Portability, and Performance of Loop-Task Parallel Applications," *Cornell Ph.D. Thesis*, Feb. 2017.
- Ji Kim, Shunning Jiang, Christopher Torng, Moyang Wang, Shreesha Srinath, Berkin Ilbeyi, Khalid Al-Hawaj, and Christopher Batten. "Using Intra-Core Loop-Task Accelerators to Improve the Productivity and Performance of Task-Based Parallel Programs." *50th ACM/IEEE Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2017.
- Shreesha Srinath, "Lane-Based Hardware Specialization for Loop- and Fork-Join-Centric Parallelization and Scheduling Strategies," *Cornell Ph.D. Thesis*, May 2018.
- Tuan Ta, Lin Cheng, and Christopher Batten. "Simulating Multi-Core RISC-V Systems in gem5." *2nd Workshop on Computer Architecture Research with RISC-V (CARRV)*, June 2018.
- Shunning Jiang, Berkin Ilbeyi, and Christopher Batten. "Mamba: Closing the Performance Gap in Productive Hardware Development Frameworks." *55th ACM/IEEE Design Automation Conf. (DAC)*, June 2018.
- Tao Chen, Shreesha Srinath, Christopher Batten, and Edward Suh. "An Architectural Framework for Accelerating Dynamic Parallel Algorithms on Reconfigurable Hardware." *51st ACM/IEEE Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2018.

The core work on XPC was published in MICRO'17 and formed the heart of the Ph.D. thesis for Ji Kim, one of the students funded through this project. The thesis is titled "Software/Hardware Co-Design to Improve Productivity, Portability, and Performance of Loop-Task Parallel Applications". Ji Kim's thesis was awarded the prestigious Cornell ECE Outstanding Ph.D. Thesis Award, and this was the first time this award was given to a Ph.D. thesis in computer architecture!

Section 2 provides a detailed summary of the research conducted throughout this project, Section 3 briefly reviews the tasks for the project as outlined in the proposal, and Section 4 discusses the project finances.

2. Detailed Research Summary

In this section, we first describe how our thinking has evolved on the motivation for XPC based on the need for creating new platforms that can improve productivity, portability, and performance (3Ps). Section 2.2 describes our experiences traversing a modern development flow for high-performance applications which helps motivate why current solutions struggle to excel at the 3Ps. Section 2.3 describes our XPC software stack, and Section 2.4 describes our progress on the XPC hardware. Section 2.5 describes our evaluation methodology including the application kernels, validating the XPC runtime, cycle-level modeling, and area/energy modeling. Section 2.6 provides a detailed design-space exploration of our new XPC engine microarchitecture including a qualitative and quantitative evaluation of the 3P's for an XPC platform (productivity, portability, and performance). Finally, Section 2.7 describes our related work on extending XPC to handle asymmetric multicores, fork-join parallelism, and application-specific accelerators. We also describe simulator infrastructure research that was a key enabler for this project.

2.1. XPC Motivation

Fine-grain loop-level parallelism has traditionally been exploited in chip multi-processors (CMPs) at the intra-core level using packed-SIMD extensions (e.g., Intel SSE/AVX and others). Packed-SIMD extensions use a relatively low-level abstraction of operations on packed data elements. At the same time, many applications exploit *coarse-grain task-level parallelism* in CMPs at the inter-core level using task-based parallel programming frameworks (e.g., Intel's C++ Threading Building Blocks (TBB) and others). Task-based parallel programming frameworks unsurprisingly use tasks as the primary abstraction, and these tasks are often dynamically scheduled using a common thread pool. Ideally, programmers would be able to productively and portably exploit both intra- and inter-core mechanisms at the same time, but the significantly different abstractions make it challenging to elegantly compose these intra- and inter-core mechanisms.

Efficiently exploiting both loop- and task-level parallelism becomes even more important on modern accelerators that have significantly more intra- and inter-core resources. For example, Intel's many integrated cores (MICs) include dozens of lightweight cores containing wide packed-SIMD units. While MICs support the same task-based parallel programming frameworks as CMPs, porting applications written for CMPs to MICs is rarely trivial and often requires MIC-specific optimizations to achieve the highest performance. NVIDIA/AMD's general-purpose graphics processing units (GPGPUs) also have tens of cores, each with dozens of tightly coupled compute lanes. GPGPUs use low-level CUDA threads as the common intra- and inter-core parallel abstraction. GPGPUs harden the thread scheduling logic and require a detailed understanding of the microarchitecture to achieve the highest performance. For MICs and GPGPUs, resource-proportional performance is difficult for applications with irregular control and/or memory accesses.

These challenges suggest that software programmers and hardware architects should focus on "the 3P's" of productivity, portability, and performance as the primary metrics when developing and evaluating computing platforms. A 3P platform would achieve the goals of our original proposal by enabling programmers to quickly develop parallel applications, map these applications to both traditional CMPs as well as accelerators, and achieve resource-proportional performance across a wide range of applications. Existing approaches to 3P platforms, such as unified offload programming frameworks (e.g., OpenCL, C++ AMP), virtual ISAs (e.g., AMD HSA), and domain-specific languages (e.g., Halide, Delite), partially achieve this goal, but the software-centric focus of these approaches result in a loss of efficiency and/or require focusing on a limited application domain. **We argue the key to achieving a 3P platform is raising the level of abstraction in the instruction set and microarchitecture to better match programming abstractions that have been proven to be productive.**

To narrow the scope of the first year of the project, we focus on a very common parallel pattern we call *loop-task parallelism* usually captured with the ubiquitous `parallel_for` primitive, where a *loop-task* is a functor that is applied across a range of loop iterations. Loop-task parallelism is more flexible than fine-grain loop-level parallelism, but less general than coarse-grain task-level parallelism. Loop-task parallelism is essentially a specific example of the amorphous data parallelism described in the proposal. **In the begin-**

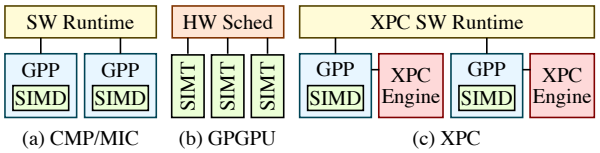


Figure 1: Vision for XPC Platform

Name	Suite	Input	CS	CA	CT	CTA	MTA	GC
sgemm	I, C	2K×2K float matrix	22	53	26	56	56	76
dct8x8m	I, C	518K 8×8 blocks	58	128	63	62†	62†	146
mriq	I	262K-space 2K pnts	28	76	35	78	78	133
rgb2cmyk	I	7680×4320 image	20	70	17	69	85	65
bfs-nd	P	rMatG_J5_10M	27	29‡	48	51‡	51‡	94
maxmatch	I, P	randLocG_J5_10M	22	24‡	36	38‡	38‡	73
strsearch	I	512 strs, 512 docs	35	38‡	42	45‡	45‡	95

Table 1: Application Kernels – I = in-house implementation; C = CUDA SDK; P = PBBS. The last six columns show logical source lines of code (LOC): CS = cmp-scalar; CA = cmp-avx; CT = cmp-tbb; CTA = cmp-tbb-avx; MTA = mic-tbb-avx; GC = gpgpu-cuda. †Lower LOC because programming model does not support using the more efficient LLM algorithm. ‡Used only basic auto-vectorization, since manual optimizations resulted in no improvement.

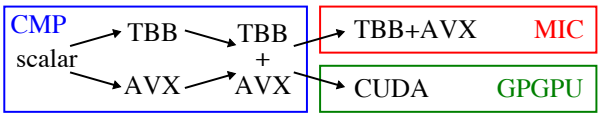


Figure 2: High-Performance App Development Flow

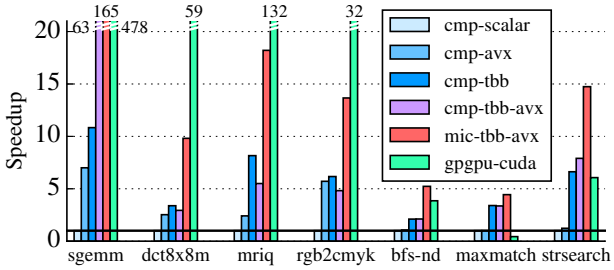


Figure 3: Performance Comparison of Selected SW/HW Platforms – Normalized to *cmp-scalar*, the non-vectorized single-threaded implementation. *cmp* = Intel Xeon E5-2620 v3 (12 cores, AVX2/256b); *mic* = Intel Xeon Phi 5110P (60 cores, AVX-512); *gpgpu* = NVIDIA Tesla C2075 (14 SMs); *avx* = ICC v15.0.3 with auto-vectorization; *tbb* = TBB v4.3.3; *cuda* = CUDA v7.5.17.

ning of this project, we are exploring a new XPC platform that uses loop-tasks as the common parallel abstraction in the programming model, runtime, instruction set, and microarchitecture. An XPC platform includes a traditional task-based parallel programming framework with `parallel_for` primitives, an XPC work-stealing runtime, a new XPC hint in the instruction set to explicitly identify loop-task execution, and a new XPC engine template that can be configured at design time to achieve resource-proportional performance on both regular and irregular applications (see Figure 1).

2.2. Traditional Software/Hardware Platforms

In this section, we describe our experiences traversing a development flow for applications that employ various combinations of existing SW frameworks and HW architectures to improve performance. We develop both regular and irregular loop-task parallel applications and summarize our experiences for each platform with respect to the 3P's. Figure 2 illustrates how application development begins with a scalar reference implementation before moving towards exploiting intra-core parallelism through packed-SIMD extensions (e.g., Intel AVX) or towards exploiting inter-core parallelism through parallel frameworks (e.g., Intel TBB). More performance is possible with dedicated accelerators like MICs or GPGPUs. Table 1 lists the application kernels used in this study and logical source lines of code (LOC; number of C++ statements). Figure 3 compares the performance of six different SW/HW platforms.

cmp-avx shows the performance of the single-threaded implementation with auto-vectorization using AVX on a CMP. Vectorization only targets regular loop-task parallelism within a core, as both control and memory-access divergence can prevent lock-step execution. 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. Because vectorization primarily increases compute throughput, memory bottlenecks can still limit performance. Note that "auto-vectorization" is a misnomer, since naive attempts at auto-vectorization by only annotating loops with `#pragma simd` yielded speedups of less than 1.10× across all seven kernels. Maximally utilizing the SIMD units required numerous manual optimizations: SIMD-aligning memory accesses, converting branches into arithmetic, converting array of structs into struct of arrays, annotating non-overlapping arrays with the `__restrict__`

keyword, and of course annotating vectorizable loops with `#pragma ivdep/simd`. **Although AVX can improve performance for *regular* loop-task parallelism, we found that the required manual optimizations greatly reduced productivity; implementing, testing, and tuning kernels for AVX took us multiple programmer-weeks for the full suite (≈2× increase in LOC), and no amount of manual optimization significantly improved performance for *irregular* loop-task parallelism.**

cmp-tbb shows the performance of the multi-threaded implementation running on 12 threads with no vectorization on a CMP. 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. Unlike vectorization, multi-threading can improve 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 can exacerbate this issue. **TBB was the most productive framework in this study; the LOC were similar to scalar implementations and approximately one programmer-week was required to develop a relatively high-performance parallel implementation of the benchmark suite. Unfortunately, TBB is limited to exploiting loop-task parallelism across cores as opposed to within a core.** Note that we also experimented with porting our benchmark suite to OpenMP, and we found similar trends.

cmp-tbb-avx shows the performance of TBB running on 12 threads on a CMP combined with auto-vectorization using AVX. Regular loop-task parallel applications can combine multi-threading across cores and vectorization within a core to achieve multiplicative benefits in performance; *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 two key reasons. First, task partitioning with TBB can interfere with auto-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. Specifically, eliminating control divergence during vectorization may also eliminate opportunities for load balancing by superficially equalizing the work across the SIMD width. **Aside from not guaranteeing better performance, combining TBB with AVX negates the productivity of the former; it took multiple programmer-weeks of manual optimization to add auto-vectorization to our original TBB implementations with similar LOC as cmp-avx.**

mic-tbb-avx shows the performance of TBB implementations with vectorization running on 60–240 threads on a 60-core MIC accelerator in native mode. MICs have relatively lightweight single-issue in-order cores, longer cache-to-cache latencies, and no shared L3 cache. Kernels with regular loop-task parallelism demonstrate the greatest improvement compared to *cmp-tbb-avx* due to the increased number of cores and wider SIMD units (512b vs. 256b). MICs are designed to accelerate applications with immense regular loop-task parallelism, thus performance heavily depends on maximally utilizing the SIMD units. Despite using the same SW framework as the CMP, the MIC required re-tuning: optimal thread counts change, task sizes change, and optimal load balancing across 60 cores may necessitate algorithm restructuring. **Overall, porting and optimizing CMP implementations for the MIC was a non-trivial process, and MICs do not achieve resource-proportional performance for irregular loop-task parallelism.**

gpgpu-cuda shows the performance of CUDA implementations running on 448 threads (maximum) on a GPGPU. GPGPUs have higher computational throughput than CMPs/MICs, as evident by the 60–165× speedups on *regular* kernels. Irregular kernels struggle to achieve resource-proportional performance due to serialized execution and inefficient scatters/gathers. Productivity-wise, CUDA offers a unified approach to exploiting loop-task parallelism across and within cores, but its offload programming model requires several substantial changes: explicit allocation/copying of device memory, manual work partitioning into blocks/grids (due to HW scheduler), effective utilization of texture/shared memory, and limitations on programming features available from within a kernel. Inter-thread communication, like in *bfs-nd* and *maxmatch*, is especially difficult to express efficiently, and may require a different algorithm to avoid barriers that can cause deadlock. Though not explored here, OpenCL and C++ AMP are alternative frameworks that face similar challenges due to their offload programming models. **Unsurprisingly, porting CMP implementations to the GPGPU was a heavily involved process requiring multiple programmer-weeks for**

```
1 void vvadd( int dest[], int src0[],
2             int src1[], int size )
3 {
4   XPC_PARALLEL_FOR( 0, size, (dest,src0,src1),
5   ({
6     dest[i] = src0[i] + src1[i];
7   }));
8 }
```

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

```
1 void loop_task_func( void* a, int start,
2                      int end, int step=1 )
3 {
4   args_t* args = static_cast<args_t*>(a);
5   int* dest = args->dest;
6   int* src0 = args->src0;
7   int* src1 = args->src1;
8   for ( int i = start; i < end; i += step )
9     dest[i] = src0[i] + src1[i];
10 }
```

(b) Loop-Task Function Generated by Macro

Figure 4: XPC 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 XPC runtime, since our cross-compiler does not yet support C++11 lambdas.

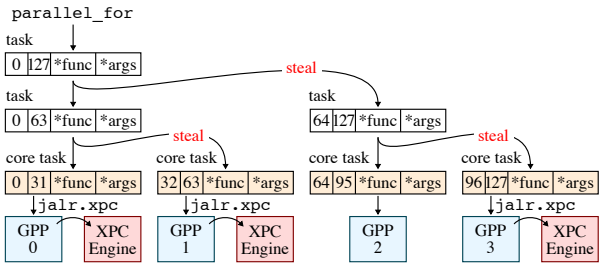


Figure 5: Example XPC Runtime Task Partitioning – XPC runtime partitions tasks into *core tasks* which are distributed across cores to exploit loop-task parallelism. Core tasks are executed using the `jalr.xpc` instruction. If an XPC engine is not available, a `jalr.xpc` instruction acts as a standard indirect function call. If an XPC engine is available, the `jalr.xpc` instruction enables an XPC engine to further exploit loop-task parallelism within a core.

the full suite with particular difficulty for the more irregular kernels. The LOC were the highest for the GPGPU implementations.

We condense the insights from this study into four key observations. First, our (admittedly qualitative) productivity analysis suggests that most SW frameworks are less productive than we might hope, except for TBB (without manual vectorization) which involved straight-forward modifications of the scalar implementation. Second, it can be difficult to easily port applications across different HW architectures, even when using the same SW framework. Third, exploiting loop-task parallelism across cores and within a core does not always yield a multiplicative effect in performance. Fourth, it is very difficult to achieve high performance on *irregular* loop-task parallel applications proportional to HW resources. **In this project, we argue that SW alone is not enough and instead careful SW/HW co-design is required to elegantly address these weaknesses in a unified manner.** We take a SW/HW co-design approach to enable us to: (1) raise the level of abstraction in the HW to better match proven, productive task-based programming models; (2) execute the same binary used for CMPs on accelerators improving portability; (3) reduce overheads and achieve a true multiplicative effect in performance; and (4) improve the performance of *both* regular *and* irregular applications.

2.3. XPC Software

In order to maintain the productivity of TBB that we observed in Section 2.2, we use the `parallel_for` primitive to express loop-tasks that can be exploited both across *and* within cores (see Figure 4(a)). Loop-tasks are functors applied across a range of loop iterations. More specifically, loop-tasks are expressed as a four-tuple of a function pointer, an argument pointer, and the start/end indices of the range. Figure 4(b) illustrates the loop-task function generated in this example, which is applied to the range $\langle 0, size \rangle$. The `step` argument is called the *range step value* and is hidden from the application-level programmer but provides flexibility in the microarchitecture (see Section 2.4).

Although we could port an existing full-featured library such as TBB to the XPC platform, we chose instead to develop our own task-based work-stealing runtime inspired by TBB. This enabled us to focus on just those features supported by the current version of the XPC platform and to ensure we had detailed

insight into all aspects of the software stack. The XPC runtime is responsible for efficiently distributing tasks across cores, and it employs child-stealing, Chase-Lev task queues, and occupancy-based victim selection. Figure 5 illustrates how a work-stealing runtime recursively partitions loop-tasks into subtasks to facilitate load balancing. Tasks are partitioned until the range is less than a configurable *core task size* at which point the subtask is called a *core task*. The basic 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 less core tasks with larger ranges (worse load balancing, lower overhead).[1]

One of the key differences in an XPC runtime is how the runtime actually executes core tasks. A traditional runtime simply uses an indirect function call (i.e., `jalr`) on the core task's function pointer with the given range and argument pointer, while the XPC runtime uses a new `jalr.xpc` instruction. A `jalr.xpc` is still an indirect function call with the same semantics as a `jalr`, except that a `jalr.xpc` can only be used to call a loop-task function pointer with the special signature in Figure 4(b). A `jalr.xpc` acts as a hint to the HW that it can potentially use the XPC engine to further partition the core task into *micro-tasks* (μtasks) and that these μtasks can be executed concurrently in any order by the XPC engine using *micro-threads* (μthreads). If an XPC engine is not available, the `jalr.xpc` hint can be treated as a standard `jalr`. The runtime can execute core tasks the same way regardless of XPC engine availability; a single implementation of an application can be used on any architecture that implements `jalr.xpc`, greatly improving portability.

Another key difference in an XPC runtime is how the runtime partitions tasks into subtasks. A naive partitioning can result in core tasks sizes that are not an even multiple of the number of μthreads within an XPC engine, causing poor intra-core resource utilization. If t is the total number of μthreads within an XPC engine, then an XPC runtime will ensure that in each partitioning step at least one subrange is evenly divisible by t. This can increase μthread utilization and performance when using an XPC engine, and causes negligible overhead when an XPC engine is not available.

2.4. XPC Hardware

An XPC engine partitions a core task into μtasks which are then mapped to μthreads. There is a large design space for organizing these μthreads in space and/or in time. Throughout this section, we will use terminology from traditional vector processors, where μthreads may be organized spatially across *lanes* and/or temporally with *chimes*.

Figure 6(a) illustrates one approach for an 8-μthread XPC engine that *tightly couples* μthread execution in both space and time. At a high level, the task-management unit (TMU) receives information about a core task from the general-purpose processor (GPP), divides this core task into eight μtasks, and (compactly) sends the μtasks to the fetch/dispatch unit. The HW is responsible for setting the argument registers for each μthread appropriately. Figure 6(b) illustrates how the core task that is mapped to GPP 0 in Figure 5 might execute on this tightly coupled XPC engine. All eight μthreads must execute in lock-step in space across the four lanes and also in lock-step in time across the two chimes. To expose potential memory structure across μtasks, μthreads on neighboring lanes execute consecutive μtasks. To enable this, the HW sets the *range step value* mentioned in Section 2.3 so that μthreads execute iterations at a stride of eight. Tightly coupled execution enables the XPC engine to exploit arithmetic, control, and memory structure across μtasks. Unfortunately, tightly coupled execution means that if one μthread stalls due to a RAW dependency or cache miss, then all μthreads must stall (e.g., marked with $\times$ in Figure 6(b)). Overall, tightly coupled XPC engines perform best on regular loop-task parallelism, but can perform poorly on irregular loop-task parallelism.

Figure 6(d) illustrates a different approach for an 8-μthread XPC engine that *loosely couples* μthread execution in both space and time. Figure 6(e) illustrates how the same core task from the previous example might execute on this loosely coupled XPC engine. 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). Additional stalls may occur due to conflicts at shared

[1] Sensitivity studies indicate that $k = 4$ is a reasonable design point, although obviously this is input dependent. Setting k adaptively is an interesting direction for future work.

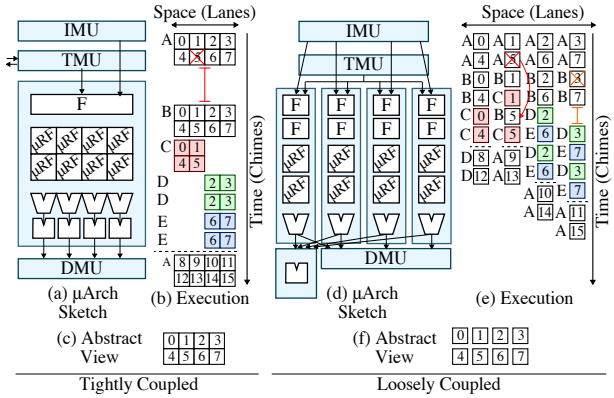


Figure 6: Tightly vs. Loosely Coupled XPC Engines – Two 8-μthread XPC engines each with 4 lanes, 2 chimes: (a–c) μthreads are tightly coupled in space and time; (d–f) μthreads are loosely coupled in space and time. Instruction sequence is denoted by letters (e.g., "μthreads 0, 1, 4, and 5 execute instructions A, B, and C") and divergent control flows are colored differently. IMU = instr mgmt unit; TMU = task mgmt unit; DMU = data mgmt unit; F = fetch/dispatch unit; μRF = μthread regfile.

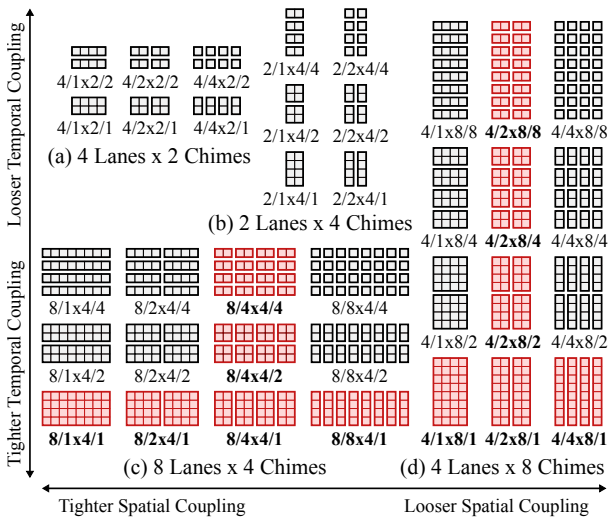


Figure 7: Task-Coupling Taxonomy – All possible spatial and temporal task-coupling configurations for: (a) 4 lanes, 2 chimes; (b) 2 lanes; 4 chimes; (c) 8 lanes, 4 chimes; (d) 4 lanes, 8 chimes. For given subfigure, most-coupled configuration is bottom left and least-coupled configuration is top right. Configurations likely to be study in next year of the project are highlighted.

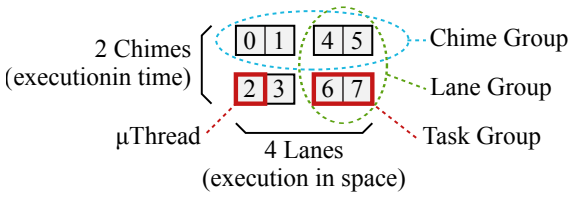
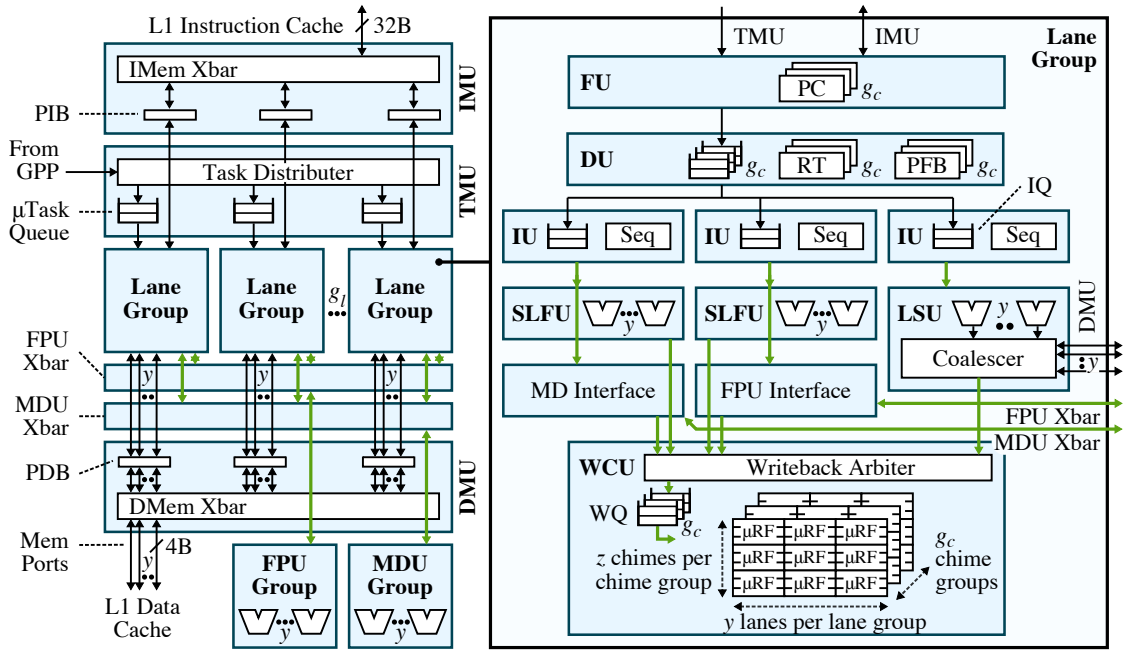


Figure 8: Terminology for Task-Coupling Taxonomy – Example of 8-μthread XPC engine with 4 lanes (l) and 2 chimes (c). 8 μthreads are divided into 4 task groups (g_t) which execute in lock-step in both space and time. The four lanes are partioned into two lane groups (g_l), and the two chimes are partitioned into two chime groups (g_c), representing the *XPC-4/2x2/2* configuration.

resources (e.g., marked by × on B3 in this example). Loosely coupled execution enables the XPC engine to better tolerate irregular control flow and memory latencies since each μthread can independently fetch, decode, dispatch, issue, and execute instructions. Loosely coupled execution also simplifies dynamically sharing expensive execution resources across the μthreads. Unfortunately, loosely coupled execution is not able to exploit structure across μtasks, potentially reducing area and energy efficiency.

The microarchitectures in Figure 6 are at two ends of a task-coupling spectrum. To simplify our discussion, Figures 6(c,f) and 8 illustrate abstract diagrams of how μthreads are coupled within an XPC engine. In Figure 8, the μthreads are divided into four *task groups*, which execute in lock-step in both space and time. This example has two *lane groups* and two *chime groups*, which can be executed spatially or temporally in a loosely coupled manner. For the remainder of this work, we abbreviate different XPC configurations with the following scheme: *num_lanes/num_lane_groups x num_chimes/num_chime_groups*. For example, Figure 8 represents the *XPC-4/2x2/2* configuration (4 lanes organized into 2 lane groups, 2 chimes organized into 2 chime groups).

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 7. Figure 7(a) shows the six configurations for the 8-μthread XPC engine we have been discussing with four lanes and two chimes. Figure 7(b) presents an alternative 8-μthread XPC engine with two lanes and four chimes Although we have performed extensive simulations of 8–64-μthread XPC engines, the rest of this work will focus on the 12 configurations for 32-μthread XPC engines highlighted in Figures 7(c) and (d). In general,



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

Figure 9: XPC Engine Template – IMU = instr mgmt unit; TMU = task mgmt unit; DMU = data mgmt unit; PIB = pending instr buffer; FPU = floating-point unit; MDU = integer mult/div unit; PDB = pending data buffer; FU = fetch unit; DU = dispatch unit; IU = issue unit; Seq = chime sequencer; SLFU = short-latency integer functional unit; LSU = load-store unit; WCU = writeback/ commit unit; PC = program counter; RT = rename table; PFB = pending fragment buffer; 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). Thick green arrows indicate channels that can transfer y worth of data in a single cycle.

more tightly coupled configurations in the lower left of the taxonomy perform better on regular loop-task parallelism, while more loosely coupled configurations in the upper right of the taxonomy better tolerate irregular loop-task parallelism.

Figure 9 illustrates the XPC engine microarchitectural template we will use for design-space exploration. Our proposed template can be configured at design time with any number of µthreads, lanes, lane groups, chimes, and chime groups.

The *task management unit* (TMU) is the interface between the GPP and XPC engine. The TMU is responsible for dividing the core task sent by the GPP into µtasks, then dynamically scheduling these µtasks across lane groups by injecting them into per-lane-group µtask queues. Upon receiving a new core task, the TMU initializes a pending µtask counter with the number of generated µtasks. Lane groups assert a completion bit when they finish executing a µtask. The TMU aggregates the completion bits and decrements the pending µtask counter accordingly. The TMU acknowledges the completion of a core task once the counter is zero by sending a completion message to the GPP. Currently, the GPP stalls until it receives this completion message.

A *lane group* manages a set of µthreads organized in task groups. Each µthread executes one of the µtasks assigned to the lane group in the corresponding µtask queue. Lane 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 earlier. Note that load balancing occurs naturally as lane groups that finish µtasks faster will obtain more µtasks 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 varying the number of frontends per lane group that can sustain separate instruction streams.

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 lock-step 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, and supports coalescing across µthreads within the same chime. 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 that prioritizes fragments in current loop iterations. We denote task groups as being spatially diverged when µthreads across lanes in the same chime have diverged, or temporally diverged when µthreads across chimes in the same lane have diverged. Lane groups also support density-time execution, which allows the sequencer to skip scheduling chimes that have no active µthreads.

Expensive resources are shared across lane groups to improve area efficiency and to exploit the elastic pipeline within lane groups. These shared resources include the instruction memory port, which is managed by the *instruction management unit* (IMU), the long-latency functional units (LLFUs) like the floating-point unit (FPU) and the integer multiply-divide unit (MDU), which are organized into an FPU group and an MDU group, and the data memory ports, which are managed by the *data management unit* (DMU). The number of FPUs, MDUs, and data memory ports is equal to the number of lanes per lane group, which means that the number of shared resources decreases as the number of lane groups increases (assuming a constant number of total lanes in the XPC engine). While we recognize that resource sharing is orthogonal to spatial task coupling, the irregular application kernels that benefit from increasing the number of lane groups usually have low LLFU intensity. This motivates significant sharing of resources across lane groups to improve area efficiency.

At the top level, IMUs consist of 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. DMUs consist of pending data buffers (PDBs) that can store a task-group-worth of 4B words to facilitate access/execute decoupling, and a crossbar with round-robin arbitration. Data bandwidth across ports can be combined for wide coalesced accesses.

2.5. Experimental Methodology

In this section, we describe the details of our vertically integrated research methodology spanning applications, runtime, architecture, and VLSI. In our first year report, we described a new XPC taxonomy that helps classify both spatial and temporal coupling. Figure 8 illustrates the naming convention we will use, and Figure 7 illustrates all possible spatial and temporal task-coupling configurations for XPC engines with both eight and 32 µthreads. The 12 XPC engine configurations used in this study are highlighted.

			DynInst (M)			Avg Size		Intensity				MC	MC	XPC-8/1x4/1				XPC-8/4x4/1				XPC-8/4x4/4			
Name	**Suite**	**Input**	**S**	**P**	**T%**	**Task**	**Iter**	**slfu**	**llfu**	**mem**	**IO**	**-IO**	**-O3**	**I**	**A**	**S%**	**M**	**I**	**A**	**S%**	**M**	**I**	**A**	**S%**	**M**
nbody	pbbs	3DinCube_1000	92	93	99%	1000	31K	18%	43%	33%	239.7	3.6	3.6	0.04	30	9.5	0.4	0.14	31	21.2	0.2	0.32	25	14.8	1.0
bilateral	cust	256×256 image	26	27	99%	66K	409	25%	51%	16%	61.9	4.3	3.8	0.03	31	8.5	2.1	0.14	31	20.4	1.2	0.26	31	11.6	1.2
mriq	cust	100-space, 256 pts	11	11	99%	256	23K	53%	20%	21%	13.9	4.0	4.1	0.04	32	6.0	0.0	0.14	32	15.4	0.0	0.27	32	6.0	0.0
sgemm	cust	256×256 fp mtrx	75	76	99%	576	131K	47%	19%	21%	113.3	3.8	3.9	0.03	32	2.5	3.4	0.13	32	12.8	0.9	0.26	32	5.6	0.8
rgb2cmyk	cust	1380×1080 image	43	43	99%	1380	31K	47%	0%	39%	57.1	3.3	2.5	0.04	29	0.5	9.7	0.15	30	0.8	4.4	0.27	31	0.0	7.1
dct8x8m	cust	782 8x8 blocks	55	55	99%	50K	1096	4%	64%	30%	81.4	4.0	3.9	0.03	31	0.0	24.6	0.13	32	0.0	18.4	0.25	32	0.0	18.2
knn	pbbs	2DinCube_10K	35	43	33%	9867	716	17%	32%	37%	57.3	1.5	1.3	0.21	5	17.5	10.7	0.41	11	32.0	10.6	0.57	15	17.6	10.6
bfs-nd	pbbs	randLocG_J5_150K	23	55	81%	36K	99	56%	0%	26%	88.6	1.5	1.1	0.07	17	0.0	23.5	0.22	21	0.0	16.4	0.36	24	0.0	16.1
radix-2	pbbs	exptSeq_500K_int	57	69	81%	46	92K	59%	0%	33%	102.3	1.1	1.1	0.04	30	0.0	49.5	0.14	31	0.0	48.9	0.27	30	0.0	48.7
radix-1	pbbs	randomSeq_1M_int	93	104	94%	229	74K	57%	0%	33%	175.1	3.4	3.6	0.05	26	1.0	35.9	0.18	28	0.8	34.5	0.30	29	0.8	34.7
rdups	pbbs	trigrSeq_300K_int	36	56	99%	508K	23	56%	0%	21%	87.1	3.0	2.0	0.12	11	0.0	30.5	0.28	18	0.0	22.5	0.41	22	0.0	22.5
sarray	pbbs	trigrString_200K	68	75	86%	76K	50	56%	0%	29%	203.9	2.5	1.7	0.10	11	2.5	60.5	0.22	20	1.6	41.1	0.35	24	0.8	42.2
strsearch	cust	210 strs, 210 docs	20	20	99%	210	49K	57%	0%	19%	30.0	3.3	3.7	0.20	6	1.5	0.5	0.51	10	0.8	0.6	0.49	18	0.0	0.6
bfs-d	pbbs	randLocG_J5_150K	23	35	95%	50K	75	56%	0%	26%	88.6	2.4	1.5	0.11	10	0.5	13.7	0.29	16	0.8	10.0	0.41	21	0.0	9.8
dict	pbbs	exptSeq_1M_int	39	51	99%	451K	25	66%	0%	19%	82.1	3.4	1.7	0.06	20	0.0	23.4	0.17	27	0.0	13.1	0.29	29	0.0	13.0
mis	pbbs	randLocG_J5_400K	14	32	99%	400K	27	52%	0%	25%	79.4	3.4	1.6	0.25	5	0.0	20.1	0.49	10	0.0	13.9	0.61	15	0.0	14.2
maxmatch	pbbs	randLocG_E5_400K	23	49	94%	1.7M	19	58%	0%	19%	154.3	3.4	1.3	0.05	25	0.0	16.3	0.17	28	0.0	12.2	0.30	29	0.0	12.3

Table 2: Application Kernels – Suite = benchmark suite {PBBS or custom}; Input = input dataset; DynInsts = dynamic instruction count in millions (S for serial impl, P for parallel impl); T% = percent of total dyn. insts in tasks; Avg Task Size = average number of dyn. insts per task; Avg Iter Size = average number of dyn. insts per iteration; { slfu, llfu, mem } Intensity = percent of total dyn. insts that are {short-latency arithmetic, long-latency arithmetic, memory operation}; IO = number of cycles (in millions) of optimized single-threaded implementation on in-order core; MC-IO = speedup of multi-threaded implementation on four in-order cores over single in-order core; MC-O3 = speedup of multi-threaded implementation on four out-of-order cores over single out-of-order core; I = ratio of total inst fetches to total dyn. insts; A = average active µthreads in XPC engine per dyn. inst; S% = percent of execution time stalled due to non-memory microarchitectural latencies; M = misses in L1 D$ per thousand dyn. insts. Coupling of XPC engines: *XPC-8/1x4/1* (tight spatial x tight temporal), *XPC-8/4x4/1* (moderate spatial x tight temporal), and *XPC-8/4x4/4* (moderate spatial x loose temporal).

2.5.1. 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. Most application kernels were ported from the problem-based benchmark suite (Shun et al., SPAA'12) and the remaining were developed in-house. Table 2 illustrates their diverse application-level characteristics with respect to task-level and instruction-level characteristics. A brief description of each kernel is included below.

nbody computes the net 3D force vector on each particle when subject to forces arising from other particles. The Barnes-Hut (BH) approximation is used with a traditional divide-and-conquer approach. Recursive spawn-and-sync is used to build an oct-tree, which organizes particles and is used to approximate the center of mass of particle groups. *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. *mriq* is an image reconstruction algorithm for MRI scanning; we parallelize across the output magnetic field gradient vector. *sgemm* performs a single-precision matrix multiplication for square matrices using a standard blocking algorithm; we parallelize across blocks. *rgb2cmyk* performs color space conversion on an image and is parallelized across the rows. *dct8x8m* calculates the 8x8 discrete cosine transform on an image using the LLM algorithm; we parallelize across 8x8 blocks. *knn* finds the nearest neighbor to each point in a 2D array. A quad-tree is built to speed up neighbor lookups, and work is then parallelized across the nodes to find the nearest neighbors. The quad-tree is built with recursive spawn-and-sync. *bfs-nd* uses a non-deterministic algorithm to do a bread-first search which finds the shortest path from a given source node to all target nodes. *radix* executes a stable sort of fixed-length unsigned integer keys in ascending order. We include two datasets for *radix*, since it exhibits strong data-dependent variability. *rdups* uses a parallel loop to insert elements into an internally deterministic hash table. *sarray* is a parallel variant of the Karkkainen and Sanders algorithm that generates the suffix array of a sequence of strings. *strsearch* implements the Knuth-Morris-Pratt algorithm with a deterministic finite automata to search a collection of byte streams for a set of substrings; we parallelize across different streams. The search

	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
XPC	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 of various work-stealing runtimes using 12 threads on a Linux server with two Intel Xeon E5-2620 v3 processors. Cilk+ = uses Cilk+'s `cilk_for`; TBB = uses TBB's `parallel_for`; XPC = uses x86 port of the XPC runtime. All apps are compiled with Intel C++ Compiler 15.0.3.

	Area	**XPC**	**Area**	**XPC**	**Area**	**XPC**	**Area**	**XPC**	**Area**
IO	0.61	*4/1x8/1*	1.34	*4/2x8/2*	1.23	*8/1x4/1*	1.74	*8/8x4/1*	1.27
O3	0.76	*4/2x8/1*	1.23	*4/2x8/4*	1.24	*8/2x4/1*	1.46	*8/4x4/2*	1.33
		4/4x8/1	1.17	*4/2x8/8*	1.24	*8/4x4/1*	1.32	*8/4x4/4*	1.34

Table 4: XPC Engine Area Estimates – Area is shown in mm^2 for single-core configurations and includes the L1 I$ and D$. Area is estimated using the component-based model described in Section 2.5.4.

Technology	TSMC 40nm LP, 500MHz nominal frequency
ALU	4/10-cycle Int Mul/Div, 6/6-cycle FP Mul/Div, 4/4-cycle FP Add/Sub
IO	1-way, 5-stage in-order, 32 Phys Regs
O3	4-way, out-of-order, 128 Phys Regs, 32 Entries IQ and LSQ, 96 Entries 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 latency, 12.8GB/s bandwidth SimpleMemory model

Table 5: Cycle-Level System Configuration

is parallelized by having all threads search for the same substrings in different streams. The deterministic finite automata used to model substring-matching state machines are also generated in parallel. *bfs-d* is similar to *bfs-nd* except it uses a deterministic two-phase reserve-commit algorithm. *dict* measures performance of batch operations on a dictionary data structure. Work is parallelized across batch inserts, deletes, and searches of sequences of values and accommodates repeated keys. *mis* finds the maximal independent set of an undirected graph, parallelizing across nodes and resolving conflicts using atomic compare-and-swap operations. *maxmatch* finds a maximal matching on an undirected graph, parallelizing work across edges. Threads claim endpoints using compare-and-swap operations.

2.5.2. Validating the XPC Runtime

To show that our basic XPC runtime is competitive with other popular task-based work-stealing runtimes, Table 3 compares the performance of the x86 port of the XPC runtime (using a standard indirect function call instead of `jalr.xpc`), Intel Cilk+, and Intel TBB with the same setup as in the first year report. The results verify that the XPC runtime has comparable performance to Intel TBB and is slightly faster in some cases because it is lighter weight (e.g., no support for C++ exceptions or task cancellation).

2.5.3. Cycle-Level Methodology

For cycle-level performance modeling, we utilize a co-simulation framework with gem5 and PyMTL, a Python-based hardware modeling framework. Table 5 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; we show both baselines as reference points in our evaluation. Multi-core configurations have four cores. The cycle-level models for XPC engines were implemented in PyMTL. Each core and its XPC engine 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, stalling, and memory system sensitivity for three specific XPC engine configurations.

2.5.4. Area/Energy Modeling

Area/energy are estimated using component/event-based modeling based on VLSI results from Verilog RTL implementations of microarchitectures somewhat comparable to XPC engines previously developed

by the PI's research group. We synthesized and placed-and-routed these designs using Synopsys Design-Compiler and IC Compiler with a TSMC 40 nm LP standard-cell library characterized at 1 V. We used the results to identify the dominant contributors to area/energy and inform our component- and event-based models. We modeled SRAMs with CACTI, since we did not have a memory compiler.

We used FG-SIMT from our work presented in ISCA'13 to model the area of a lane-group, DMU, and D$ crossbar network; and we used XLOOPS from our work in MICRO'14 to model the area of the IMU and TMU. The dominant contributors were the L1 caches (33%), regfiles (26%), LLFUs (20%), SLFUs (10%), and assorted queues (7%). We modeled the D$ crossbar network area by scaling roughly quadratically with the number of ports. Because we did not have RTL for the rename table, reorder buffer, and arbitration logic, we modeled these using flop-based 1r1w regfiles, integer ALUs, and muxes. Table 4 shows the area estimates for different XPC configurations. Since we did not have *O3* RTL, we determined a reasonable scaling factor for *O3* vs. *IO* ($\approx 3\times$) based on rough McPAT area estimates.

We developed a suite of 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 total power estimates and breakdowns. We then built an event/component-based modeling tool that can parse event traces from cycle-level simulations to estimate energy. Events in the XPC engine and O3 with no corresponding RTL were estimated using carefully tuned McPAT component-level models.

2.6. Evaluation

In this section, we begin by exploring the design space for a single 32-µthread XPC engine by first examining the impact of spatial task coupling on performance, area, and energy efficiency, then doing the same for temporal task coupling. Next, we use the most promising XPC engines to evaluate the multiplicative effect of exploiting loop-task parallelism across cores using the XPC runtime and within a core using these XPC engines. We conclude with a qualitative evaluation of the productivity and portability of the XPC platform.

2.6.1. XPC Engine: Spatial Task-Coupling

Figure 10 shows the single-core performance of different XPC engines each with one chime group but varying spatial task coupling. **Tighter spatial task coupling improves the performance on regular kernels with high LLFU instruction density.** Examples include *nbody*, *bilateral*, *mriq*, *sgemm*, and *dct8x8m*. This is due to the higher LLFU bandwidth available with more lanes per lane group. In *bilateral*, the percent of time when stalled waiting on LLFUs decreases from 33% on *XPC-8/2x4/1* to 17% on *XPC-8/1x4/1*. Similar trends between *XPC-8/4x4/1* and *XPC-8/1x4/1* are also shown in Table 2, column S%. One tradeoff with bigger task groups is that more µthreads stall on a cache miss since task groups execute in lock-step; this behavior is seen in *sgemm* and *dct8x8m*. In *sgemm*, the percent of time when stalled waiting on D$ increases from 16% on *XPC-8/4x4/1* to 35% on *XPC-8/2x4/1*. Conversely, **looser spatial task coupling improves the performance on irregular kernels.** Examples include *rdups*, *strsearch*, *bfs-d*, *mis*, and *maxmatch*. By increasing the number of lane groups, we allow more (smaller) task groups to be executed concurrently. This effect is most beneficial when µthreads across lane groups exhibit high control divergence. In *strsearch*, the average number of active µthreads per cycle across all lanes increases from 6 on *XPC-8/1x4/1* to 14 on *XPC-8/8x4/1* (see also Table 2, column A). This does not address temporal control divergence since density-time already eliminates inactive chimes. In general, 8-lane configurations will outperform 4-lane configurations due to the increased SLFU and LSU bandwidth. On average, the XPC engines achieve 4–5$\times$ speedup vs. *IO* and 2–2.5$\times$ speedup vs. *O3*.

Looser spatial task coupling improves area-normalized performance as sharing resources reduces absolute area while not severely impacting average performance. Shared resources include the LLFUs and D$ memory ports, the latter of which impacts how the cache is banked and ported, as well as the crossbar network used to access cache banks. For the largest XPC engine in this study, *XPC-8/1x4/1*, the area of the LLFUs and the D$ crossbar network is 30% of the total area, which decreases as we share more resources. However, this is partially offset by an increase in area from per-lane-group PIBs and frontend

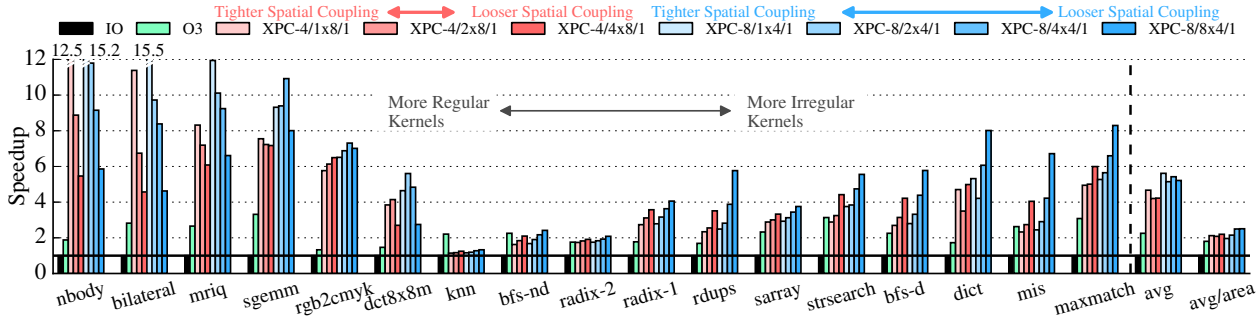


Figure 10: Performance of Single-Core XPC Engine with Variable Spatial Task-Coupling – All XPC engines have 32 µthreads and all results are normalized against the performance of the baseline scalar in-order core for each kernel.

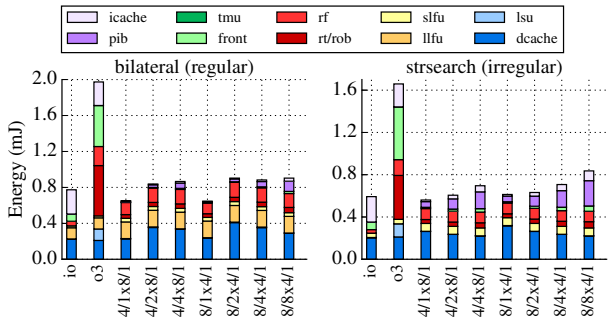


Figure 11: Energy Breakdown (Spatial Task-Coupling) – Results are shown for one kernel with regular loop-task parallelism (e.g., *bilateral*) and one kernel with irregular loop-task parallelism (e.g., *strsearch*).

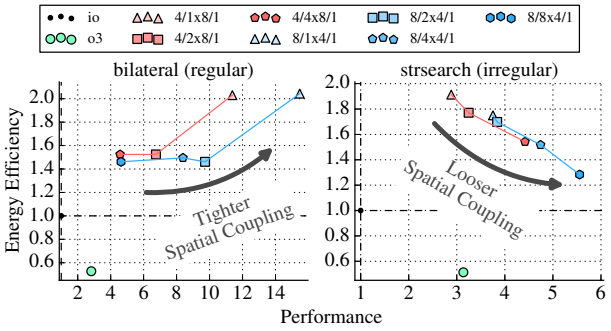


Figure 12: Energy Efficiency vs. Performance (Spatial Task-Coupling) – Each point represents the energy efficiency and performance of a different configuration in the XPC task-coupling taxonomy normalized to the baseline scalar in-order core.

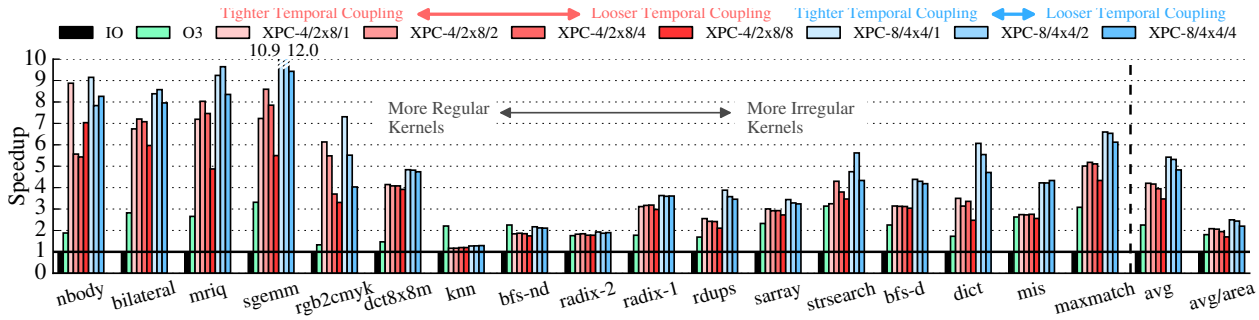


Figure 13: Performance of Single-Core XPC Engine with Variable Temporal Task-Coupling – All XPC engines have 32 µthreads. All results are normalized against the performance of the baseline scalar in-order core for each kernel.

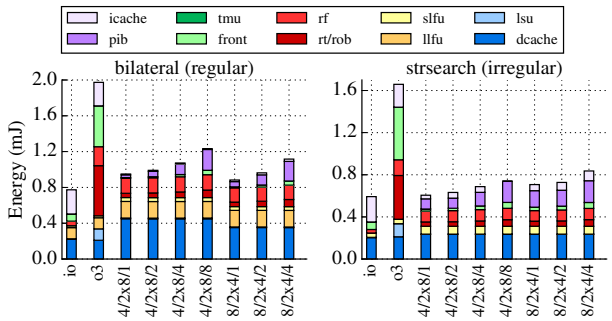


Figure 14: Energy Breakdown (Temporal Task-Coupling) – Results are shown for one kernel with regular loop-task parallelism (e.g., *bilateral*) and one kernel with irregular loop-task parallelism (e.g., *strsearch*).

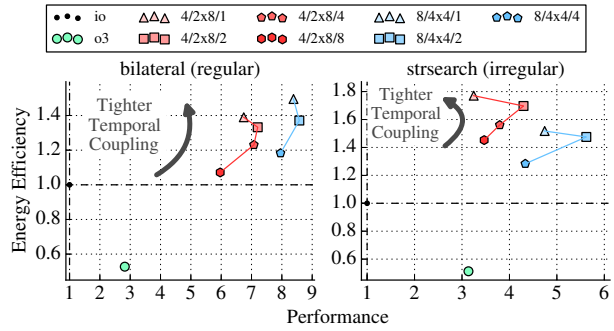


Figure 15: Energy Efficiency vs. Performance (Temporal Task-Coupling) – Each point represents the energy efficiency and performance of a different configuration in the XPC task-coupling taxonomy normalized to the baseline scalar in-order core.

units (i.e., FU, DU, IU). On average, XPC engines achieve improvements in performance/area of 2× vs. *IO* and 2.5× vs. *O3*.

Figure 11 shows the absolute energy breakdowns of XPC engines with varying spatial task coupling for one regular kernel, *bilateral*, and one irregular kernel, *strsearch*. The results show that **tighter spatial task coupling improves energy efficiency by amortizing the front-end energy across more µthreads.** This amortization is limited to *active* µthreads, thus the energy reduction is more noticeable on regular kernels. Regular kernels also benefit from memory coalescing with larger task groups, but this is offset by the increased energy in a higher-ported crossbar network. Increasing the number of lane groups increases energy due to redundant accesses for the same instruction; this is especially true for regular kernels. Figure 12 shows the energy efficiency vs. performance of XPC engines with varying spatial task coupling for *bilateral* and *strsearch*. XPC engines achieve average improvements in energy efficiency of 1.5× vs. *IO* and 3.1× vs. *O3*.

Although the average performance in Figure 10 is similar across a given lane configuration, the key is that the extremes are specialized for either regular *or* irregular kernels. The results suggest that **moderate spatial task coupling is ideal for achieving a reasonable compromise across both regular *and* irregular kernels in terms of performance, area, and energy.** As such, we select *XPC-4/2x8/1* and *XPC-8/4x4/1* as the most promising configurations.

2.6.2. XPC Engine: Temporal Task Coupling

Figure 13 shows the single-core performance of the XPC engines selected above with varying temporal task coupling. **Moderate temporal task coupling can sometimes improve performance on both regular and irregular kernels.** Examples include *bilateral*, *mriq*, *sgemm*, *strsearch*, and *maxmatch*. One reason for this is that increasing the number of chime groups creates smaller task groups, which as discussed before, means less µthreads will stall on a cache miss. Another reason is that increasing the number of chime groups allows more (smaller) task groups to better hide microarchitectural latencies. This can be seen in Table 2, column S% between *XPC-8/4x4/1* and *XPC-8/4x4/4*. However, **increasing the number of chime groups can reduce performance by significantly increasing I$ conflicts and decreasing IU utilization.** As the number of total task groups increases, the amount of frontend amortization becomes limited and the number of redundant instruction fetches inflated. Futhermore, fewer chimes per chime group also keeps the functional units busy for less cycles, which can create a fetch/dispatch bottleneck that outweighs these benefits (compare column I of *XPC-8/4x4/1* and *XPC-8/4x4/4* in Table 2).

Figure 14 shows the absolute energy breakdowns of XPC engines with varying temporal task coupling for *bilateral* and *strsearch*. The results show that **looser temporal task coupling increases the energy due to redundant work in the frontend, while only marginally improving performance**. Area-normalized performance also suffers as more frontend logic is added with each new chime group. Figure 15 shows the energy efficiency vs. performance of XPC engines with varying temporal task coupling for *bilateral* and *strsearch*.

These results seem to suggest that tighter temporal task coupling might be the most promising approach. However, these kernels mostly fit within the L2 cache. Additional experiments show that for larger kernels with longer memory latencies, more moderate temporal task coupling yields optimal performance. We have also experimented with XPC engines with up to 128 µthreads and observed that for greater numbers of µthreads, looser temporal task coupling has larger relative performance impact. Finally, examining loose spatial task coupling with no LLFU sharing (i.e., each lane has a dedicated LLFU) revealed that while performance on regular applications increased substantially, the energy efficiency was further decreased. We conclude that **moderate temporal task coupling with moderate spatial task coupling can achieve high performance on both regular and irregular kernels with relatively high area and energy efficiency.** We identify *XPC-4/2x8/2* and *XPC-8/4x4/2* as the most promising configurations.

2.6.3. 3P's of XPC Platform

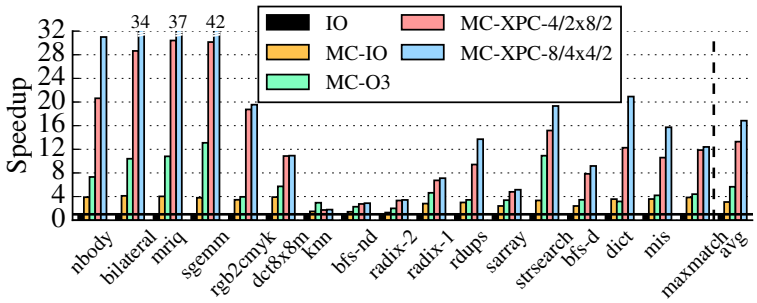


Figure 16: Performance of XPC Platform on Multi-Core System – Results are shown for baseline cores and XPC engines on a 4-core system. All results are normalized against the performance of a single in-order core for each kernel.

Figure 16 shows the performance of a quad-core system with either an *XPC-4/2x8/2* or *XPC-8/4x4/2* engine per core. The results confirm that the **XPC platform is able to achieve multiplicative effects from exploiting loop-task parallelism both across cores and within a core.** Referring back to Figure 13, the average speedup of the most promising XPC engines is 4.2–5.3× over *IO*, and using the XPC runtime on *MC-IO* yields an average speedup of 3.1× (see Table 2). We see that both *MC-XPC-4/2x8/2* and *MC-XPC-8/4x4/2* are able to achieve ideal multiplicative speedups of 13–16×. Compared to *MC-IO*, the XPC platform has an average performance per area of 1.9× and energy efficiency of 1.2×. Even compared to a more aggressive *MC-O3*, the XPC platform improves raw performance by 3.0×, performance per area by 2.5×, and energy efficiency by 2.5×.

In terms of productivity, the kernels we used for the evaluation were ported from the corresponding TBB implementations with minimal modifications. Since the XPC platform uses loop-tasks as the common abstraction across SW and HW, porting the kernels was usually a simple matter of replacing TBB's `parallel_for` constructs with our own macros and linking in the proper libraries. Although further XPC-specific optimizations might improve performance, **the key point is that they are not necessary to extract high performance from the XPC platform for both regular and irregular kernels**. In terms of portability, a single implementation of the kernel can be **written and compiled once, then executed using the XPC platform on a system with any combination of GPPs and homogeneous or heterogeneous XPC engines**. The primary enabler for this is the `jalr.xpc` instruction that acts as a common SW/HW interface.

2.7. Other Research Related to XPC

This section briefly describes our work on extending XPC to handle asymmetric multicores, fork-join parallelism, and application-specific accelerators. We also describe simulator infrastructure research that was funded in part through this project, and was a key enabler for successfully demonstrating the potential of XPC.

- **Asymmetry-Aware Work-Stealing Runtimes –** A work-stealing scheduler is critical component of the XPC runtime, and XPC systems are fundamental heterogeneous since they can include a mix of general-purpose cores and cores augmented with various XPC engines. We also explored more generic asymmetry-aware work-stealing (AAWS) runtimes that target the static asymmetry (e.g., from core microarchitecture as in ARM big.LITTLE systems) and dynamic asymmetry (e.g., applying dynamic voltage/frequency scaling) in traditional multicore processors. AAWS runtimes use three key hardware/software techniques: work-pacing, work-sprinting, and work-mugging. Work-pacing and work-sprinting are novel techniques that combine a marginal-utility-based approach with integrated voltage regulators to improve performance and energy efficiency in high- and low-parallel regions. Work-mugging is a previously proposed technique that enables a waiting big core to preemptively migrate work from a busy little core. We proposed a simple implementation of work-mugging based on lightweight user-level interrupts. We used a vertically integrated research methodology spanning software, architecture, and VLSI to make the case that holistically combining static asymmetry, dynamic asymmetry, and work-stealing runtimes can improve both performance and energy efficiency in future multicore systems. This work was published in ISCA'16.

- **Smart Sharing Architectures –** The XPC architecture described earlier focuses on data-parallel computation, but we also explored support for fork-join parallelism. We proposed smart-sharing architectures (SSAs) which are a multicore-based solution where a general-purpose processor is augmented with conjoined lanes. Similar to the XPC lanes described above, SSA lanes have the same instruction set as the general-purpose processors. Unlike the XPC lanes describes above, SSA lanes actually execute the exact same work-stealing runtime as well. We conducted detailed studies using high-level instruction-set simulation to characterize the amount of instruction redundancy in these fork-join-centric parallel programs, and then we explored techniques to exploit this instruction redundancy to reduce cost, maximize efficiency, and improve performance. We proposed instruction coalescing based on a hybrid min-PC/round-robin reconvergence scheme, soft-barrier hints to synchronize task execution, and lock-step execution when sharing long-latency functional units. Our results showed the promise for extending XPC to even more diverse forms of parallelism. This work was a key part of Shreesha Srinath's Ph.D. thesis.

- **Application-Specific Accelerators for Amorphous Data Parallelism –** XPC uses hardware specialization to improve the performance and efficiency of general amorphous data-parallel applications, but we also pushed this idea to the extreme and explored hardware specialization for a single amorphous data-parallel application. In this work, we developed an architectural framework for building application-specific parallel accelerators meant for FPGAs. Our framework introduced a task-based computation model with explicit continuation passing to support dynamic parallelism in addition to static parallelism. In contrast, today's high-level design frameworks for accelerators focus on static data-level or thread-level parallelism that can be identified and scheduled at design time. To realize the new computation model, we developed an accelerator architecture that efficiently handles dynamic task generation and scheduling as well as load balancing through work stealing. The architecture is general enough to support many dynamic parallel constructs such as fork-join, data-dependent task spawning, and arbitrary nesting and recursion of tasks, as well as static parallel patterns. We also introduced a design methodology that includes an architectural template that allows easily creating parallel accelerators from high-level descriptions. The proposed framework was studied through an FPGA prototype as well as detailed simulations. Evaluation results showed that the framework can generate high-performance accelerators targeting FPGAs for a wide range of parallel algorithms and achieve an average of 4.0× speedup over an eight-core out-of-order processor (24.1× over a single core), while being 11.8× more energy efficient. This work was published in MICRO/18.

- **Multi-Core RISC-V gem5 Simulation –** Early in the project, XPC used a MIPS variant as its base instruction set architecture, but near the end of the project we transitioned to using RISC-V. The RISC-V ecosystem is becoming an increasingly popular option in both industry and academia. The ecosystem provides rich open-source software and hardware tool chains that enable computer architects to quickly leverage RISC-V in their research. While the RISC-V ecosystem includes functional-level, register-transfer- level, and FPGA simulation platforms, there is currently a lack of cycle-level simulation platforms for early design-space exploration. gem5 is a popular cycle-level simulation platform that provides reasonably extendable, fast, and accurate simulations. We led the work on extending gem5 to simulate multi-core RISC-V systems, and we demonstrated the ability for gem5 to execute programs using the upstream open-source RISC-V software tool chain and several popular task-based parallel programming frameworks. This work was published in CARRV'18.

- **Fast and Productive Cycle- and Register-Transfer-Level Modeling –** XPC required a mix of cycle- and register-transfer-level modeling. Early in the project, we decided to extend our work on PyMTL, a Python-based hardware modeling framework, for use in the XPC project. PyMTL is an example of a hardware generation and simulation (HGSF) framework which uses a single "host" language for parameterization, static elaboration, test bench generation, behavioral modeling, and simulation. Unfortunately, HGSFs often suffer from slow simulator performance which undermines their potential productivity benefits. We developed Mamba, an improved version of PyMTL that co-optimizes both

the framework and a general-purpose just-in-time compiler. We conducted a quantitative comparison of Mamba vs. traditional and emerging hardware development frameworks across both simple and complex designs. Our results suggest Mamba is able to match the performance of commercial Verilog simulators and is 10× faster than existing HGSFs while still maintaining the productivity of using a high-level language in hardware design. This work was published in DAC'19.

3. Management Summary

The proposed work included two research thrusts with four tasks per thrust. A brief summary of the completion of each task is discussed in more detail below. Figure 17 illustrates the updated project roadmap.

3.1. Research Thrust 1: XPC Architecture

- **Task 1.1. Design a new XPC instruction set –** We finished the new XPC instruction set in the first year of the project. This new ISA adds a new `jalr.xpc` instruction to a standard RISC instruction set. We did explore adding new hint instructions later in the project to enable better synchronization of task execution across parallel lanes.
- **Task 1.2. Evaluate XPC tiles with tightly coupled lanes using cycle- and/or register-transfer-level modeling –** In the second year of the project, we finished a detailed design-space exploration of our microarchitectural template which can be configured at design time to use tightly coupled lanes. We explored adding support for nested parallelism in the final year of the project.
- **Task 1.3. Evaluate XPC tiles with loosely coupled lanes using cycle- and/or register-transfer-level modeling –** In the second year of the project, we finished a detailed design-space exploration of our microarchitectural template which can be configured at design time to use loosely coupled lanes. We explored adding support for nested parallelism in the final year of the project.
- **Task 1.4. Evaluate XPC tiles with cooperative multicore using cycle- and/or register-transfer-level modeling –** In the final year of the project, we explored augmenting traditional in-order cores with hardware support for work distribution and for exploiting instruction similarity across these cores.

3.2. Research Thrust 2: XPC Software

- **Task 2.1. Develop a new XPC programming framework –** We finished the XPC programming framework in the first year of the project. This framework enables programmers to express both loop-level, fork-join, and nested parallelism. We did make some small changes in the second and third years of the project to support more advanced C++11 features including lambdas.
- **Task 2.2. Port interesting amorphous data-parallel applications using the XPC programming framework –** We ported 16 C++ application kernels to use our XPC programming framework in the first year of the project. These applications primarily used amorphous loop-level parallelism. In the final year of the project, we added applications with fork-join and nested parallelism.
- **Task 2.3. Investigate dynamic binary translation to enable XPC applications to execute on legacy systems –** In the final year of the project, we implemented a simple dynamic binary translation scheme that enabled automatically rewriting our explicit parallel call instruction (`jal.xpc` to a traditional call instruction (`jal`). This enables XPC applications to execute on legacy applications.
- **Task 2.4. Explore an XPC runtime system that enables adaptively migrating XPC binaries between XPC tiles –** We finished development of a preliminary work-stealing runtime in the first year of the project. This runtime included support for child-stealing, occupancy-based victim selection, and Chase-Lev task queues. The runtime is able to take advantage of the new XPC instruction set. In the final year of the project, we revisited this task in the context of heterogeneous mix of tiles (e.g., tiles with general-purpose processors, tiles with loosely coupled lanes, and tiles with tightly coupled lanes).

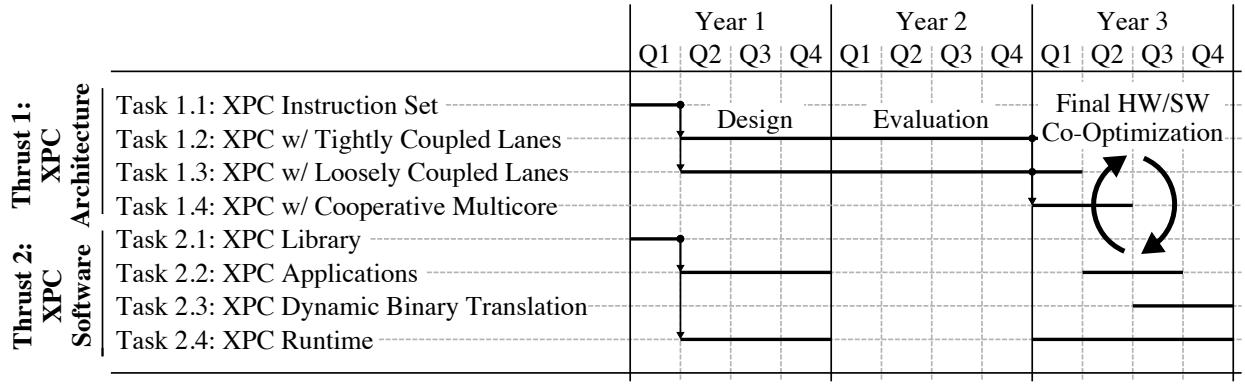


Figure 17: Project Roadmap – Roadmap was reorganized during the first year to better reflect the plan for work.

4. Financial Status Report

Figure 18 shows the budgeted and actual expenses for the entire project. It was not possible to recruit two students to start work right away in the fall of 2015. In the fall of 2016, I recruited a new student to work on the project. This new student had a fellowship, and thus I pushed some of the GRA funding into the final year. I had two students working on the project in the final year of the project.

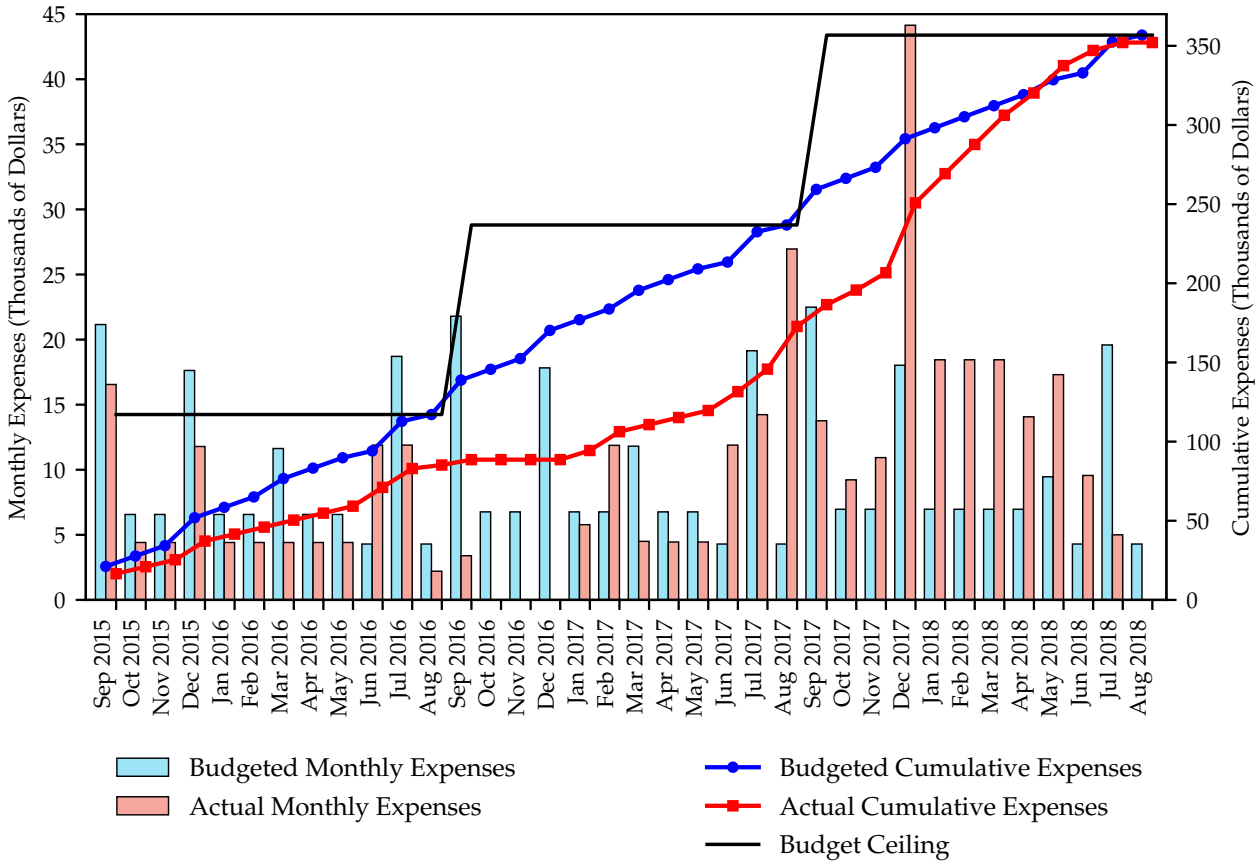


Figure 18: Project Finances – Left axis is for bars showing budgeted and actual monthly expenses, right axis is for lines showing budgeted and actual cumulative expenses.