Scaling Task Graphs for Network Processors

Martin Labrecque[1] * J. Gregory Steffan[1] †

[1] *Department of Electrical and Computer Engineering,*
University of Toronto,
10 King's College Road, Toronto, Ontario, M5S 3G4, Canada

Abstract.
Modern network processors (NPs) are highly multithreaded chip multiprocessors (CMPs), supporting a wide variety of mechanisms for on-chip storage and inter-task communication. Real network processor applications are hard to program and must be tailored to fit the resources of the underlying NP, motivating an automated approach to mapping multithreaded applications to NPs. In this paper we propose and evaluate compiler-based automated task and data management techniques to scale the throughput of network processing task graphs onto NPs. We evaluate these techniques using a NP simulation infrastructure based on realistic NP applications, and present an approach to discovering performance bottlenecks. Finally we demonstrate how our techniques enhance throughput-scaling for NPs.

Keywords: Network Processors, Compiler techniques, Feedback-directed optimization, Multithreading, Multiprocessors, Concurrency, Simulation, Methodology

1 Introduction

The deployment of *network processors* (NPs) has become increasingly important as networking applications continue to push more processing into the network. NP applications are typically composed of many threads that share memory, synchronize and communicate frequently. Even today, for most NP architectures, writing code means managing several concurrent software threads in hand-coded assembly to ensure the most efficient code possible, and to fully exploit the wide variety of instructions commonly available for synchronization and communication. These programming challenges are limiting the widespread adoption of NPs.

Instead of writing complex parallel applications, we would rather the programmer be able to express the application as a **graph of tasks** [3, 5] through a high-level language. Ideally, the compiler would automatically insert all synchronization, signalling, and manage memory, allowing the high-level application to scale up to available NP resources. In this paper we introduce a parametric compilation and simulation framework that allows us to explore such compiler transformations for scaling task graphs for NPs.

1.1 Related Work

The notion of compiler support for automatically transforming network processing applications is of growing interest in the parallel systems community. The Shangri-la project [3] shares our approach of an integrated, profile-driven compilation; however, it suggests a custom programming language and uses an older NP (the IXP2400) that has less on-chip parallelism and memory bandwidth available. Shangri-la uses a task-based programming model that, with its packet processing functions and communication channels, resembles ours. Li *et al.* [10] explore automatic multithreading by compiler-inserted synchronization. However, they do not address how shared variable are identified and the case where two shared variables depend on each other's value. Dai *et al.* [5] present an elaborate form of task partitioning but since they evaluate it in isolation of multithreading, they are concerned with making balanced partitions and not with improved data locality and schedulability. Mudigonda *et al.* [11] performed some evaluation of wide memory accesses but did not use compiler feedback to group memory accesses and selectively use memory bandwidth, as well, it was not evaluated in conjunction with parallel PEs.

1.2 Contributions

This paper makes the following contributions: (i) it describes, combines and evaluates task and memory transformations and compilation techniques to automatically scale the throughput of an application to the resources of the underlying NP; (ii) it presents the NPIRE infrastructure, an integrated environment for network processor compiler and architectural research; (iii) it presents an integrated evaluation of realistic parallel NP applications and multithreaded NPs.

2 Compiler Techniques for NPs

The goal of automated compilation for an NP is to increase parallelism and hence throughput by scaling a high-level specification of an application to exploit all available processing and memory resources through increased parallelism. In this section, we first describe the programming model that we use, and discuss the different types of memory that it implies. Next, we explore compilation methods for increasing parallelism through task transformations and for tolerating memory latency.

2.1 Programming Model

Rather than defining a new programming model, we use the Click Modular Router [8] which has also been used in a number of other studies of NP design [4, 14]. Click provides a large library of predefined network processing tasks called *elements* which can be connected

*martinl@eecg.utoronto.ca
†steffan@eecg.utoronto.ca

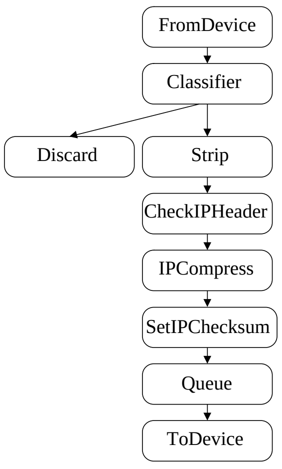
```
c0 :: Classifier(...);

FromDevice(...)
      -> c0;

c0[0]
      -> Discard;

c0[1]
      -> Strip(...)
      -> CheckIPHeader
      -> IPCompress(...)
      -> SetIPChecksum
      -> Queue(...)
      -> ToDevice(...);
```

(a) Configuration script



(b) Resulting task graph

Figure 1: A simple IP compression application described in Click. A `Classifier` filters IP packets based on header fields. Non-IP packets are sent to a `Discard` element, while the rest are by the sequence of tasks starting by `Strip`.

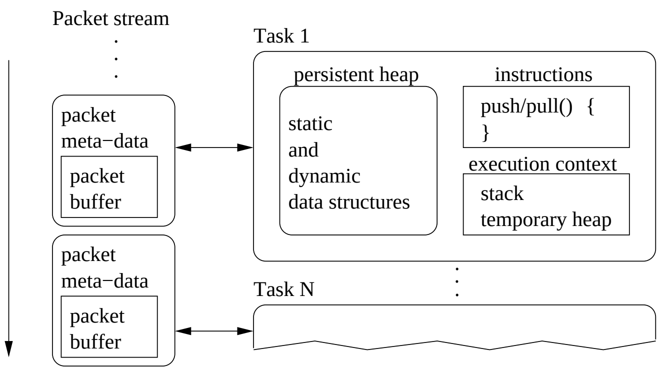


Figure 2: Separation of different types of memory.

in task graphs to form a wide variety of NP applications. Figure 1 shows an example Click configuration (which compresses valid IP packets) and the resulting task graph.

Click is coded in a high-level language (`C++`), which is not a traditional choice for NPs due to its overheads. However, the modularity and the absence of global variables make Click a good candidate for parallelization and source-level optimization. In our evaluation, we attempt to reduce the overheads of `C++` by inlining most method calls, and by optimizing the transitions between elements, as our compiler understands the Click configuration script.

2.2 Memory Types

The programming model that we assume allows the compiler to easily distinguish between different types of memory, identify shared memory, and map variables and data structures to the various types of storage available

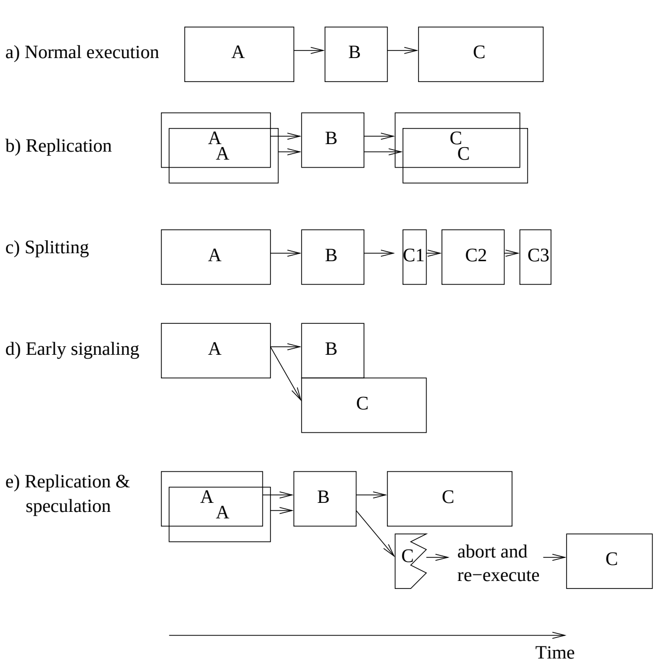


Figure 3: Task transformations to increase parallelism; `A`, `B`, and `C` are each distinct tasks.

in the NP architecture. Figure 2 illustrates the four different types of memory that we support. First, there are the instructions that comprise a task, which are typically read-only. Second is the *execution context*, the data which is private to a task such as its execution stack, registers, and any temporary heap storage. Third is the *persistent heap data* which is maintained across instances of a distinct task. The data structures that exist after initialization is called *static* while the data structures created as packets are processed are called *dynamic*. Fourth is the packet data, including the actual *packet buffer* (header and payload) as well as any *meta-data* attached to the packet by tasks. One benefit of this model of memory is that the only way for two distinct tasks to communicate is through either packet meta-data or potentially through a modified packet payload. To extract memory types, the runtime profiling system (see Section 3) tracks the creation of elements and other `C++` objects at initialization time. Later, all memory accesses are tagged and classified by our profiling tool: because the memory types are separate (no cross-referencing between the types), the memory accesses can be classified without ambiguity.

2.3 An Overview of Task Transformations

Figure 3 illustrates four task transformations for increasing parallelism in the task graph of a network processing application. Figure 3(a) represents the normal, sequential execution of three distinct tasks `A`, `B`, and `C`. To increase parallelism, and to allow a minimal task graph to scale up to a larger number of PEs and hardware contexts, we employ *task replication*: in Figure 3(b), tasks `A` and `C` are each replicated. In Figure 3(c), we employ *task splitting* which breaks a large task (eg., `C`) into smaller tasks allowing these smaller task-splits to be scheduled on multiple PEs, thus improving load balance. In Fig-

Table 1: Potential dependences through memory.

Dependence Type	**Dependence Location**
Between distinct tasks	packet meta-data packet payload
Between task replicas	persistent heap
Within a task and between task-splits	stack temporary heap persistent heap packet meta-data packet payload
With an early-signalled task	(none)
When speculating between task replicas	persistent heap

ure 3(d), we illustrate *early signalling*: when two independent tasks (eg., B and C) are both guaranteed to execute, the execution of those two tasks can be aggressively overlapped. Finally, Figure 3(e) demonstrates how *speculation* can be used to aggressively execute potentially dependent tasks in parallel (eg., task C and its replica). Speculation can be used to avoid synchronization in the case of an unlikely dependence with a replica, but ensures correctness by aborting and re-executing a task which violates such a dependence. In the remainder of this section, we describe these task transformations in more detail.

2.4 Task Replication

Giving the illusion of programming a machine where everything happens in sequential order greatly simplifies the programmer's work by reducing the need to perform dependence management. For a task which is not sensitive to packet ordering,[1] we can increase parallelism by replicating that task. A task and its replica(s) (i) can either occupy two hardware contexts on the same PE or occupy multiple PEs, (ii) can potentially share an instruction store, and (iii) can share the persistent heap. Thus there will potentially be dependences between the replicas through the persistent heap: for example, if a task increments a persistent counter for every packet. Hence, dependences between task replicas can be considered to be *unordered*: the order of execution of the task replicas does not matter as long as they execute atomically with respect to the shared persistent heap. Such atomic execution is provided by the insertion of synchronization, i.e., a lock and unlock pair which create a critical section around the memory accesses that result in unordered dependences. For dependent shared memory accesses in persistent heap identified by alias analysis augmented with our memory typing system, we automatically insert synchronization in a fashion similar to Li *et al.* [10] plus take care of grouping dependent shared variables in the same synchronized section. The synchronization placement is such that:

1. the task acquires a lock before the first read or write to a given shared location;
2. the task releases the lock after the last read or write to that location;
3. for any critical section that partially overlaps with another, both critical sections are combined into one.

While more aggressive or more fine-grained synchronization strategies are possible, this method has the benefit of avoiding deadlock situations, since only one lock is held at a time. A more advanced approach would attempt to decrease the size of the resulting critical section through instruction scheduling [16] or implementing thread folding [6]. In summary, replication can be used to increase the throughput of a bottleneck task, but can be limited by intra-task dependences.

2.5 Task Splitting

To improve load balance, we can break a large task into smaller tasks through *task splitting*, allowing the new task-splits to be scheduled on multiple PEs. To split a task requires an analysis of all dependences between the task-splits. When we attempt to split a task, we must preserve any of the original dependences within the task that now cross task-split boundaries. Dependences across task-splits can exist in any of the types of memory used by a task (as summarized in Table 1), and are therefore difficult to manage. These dependences are *ordered* since the results of the producer task-split must be forwarded to the consumer task-split as input. Furthermore, if a task replica is split, any locks held across a split point must migrate from the producer task-split to the consumer task-split.

Our compiler performs task splitting by iteratively splitting candidate tasks, measuring the resulting performance through feedback, and repeating while performance improves. Upon each splitting iteration we break a task (or a task-split) in two task-splits of roughly the same dynamic execution duration. We chose the candidate tasks for splitting to be the top three bottleneck tasks. This criteria is estimated by the time that packets are queued awaiting processing by distinct tasks. In our test cases, we found that there was generally no benefit to performing more than five iterations of splitting, for two reasons: (i) each split incurs an overhead for communicating the live data set between the resulting splits that leads to diminishing returns; and (ii) we currently do not create tasks splits that span across loops boundaries for performance reasons.

A more advanced form of splitting would facilitate *task pipelining*—allowing a split task to operate on multiple packets at once. While automatically pipelining in the presence of potential dependences within a task is potentially beneficial, it is complex and hence our infrastructure does not support it yet.

2.6 Signalling a Task Early

Signaling a task early allows us to exploit inter-task parallelism by executing independent tasks at the same time instead of serially. Any task that *post-dominates* an

[1]To support packet ordering while not being overly conservative in the common case we assume a mechanism for the application developer to specify that ordering be preserved at a given point in the task graph.

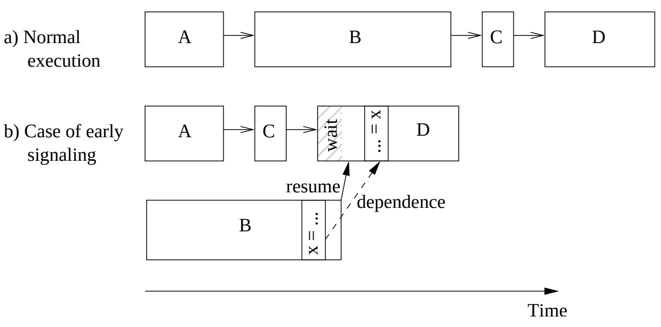


Figure 4: Example of early signaling requiring extra inter-task synchronization. Task B can start as early as task A, however task D must wait for a resume signal from task B because of an ordered dependence between tasks B and D.

earlier point in the task graph is a candidate for early signalling. However, the compiler must ensure there are no dependences between the candidate task and any intermediate tasks, and ensure that the candidate task does not deallocate memory (which may still be in use) nor perform packet output.

In Figure 3(d) we illustrated a simple case of early signalling when two independent tasks (eg., B and C) are aggressively overlapped by signalling C to execute early, i.e., after A completes. However, if we consider a slightly larger task graph the situation becomes more complex, as shown in Figure 4. Assume that the compiler has decided that B is independent of A. B can therefore be signalled early (i.e., at the same time as A). If the compiler has also decided that C is independent of B, then C may also be signalled early (ie., when A completes). The problem is as follows: since when C completes it signals D, D may inadvertently execute in parallel with B; if there is an ordered dependence between B and D, then the dependence might be violated leading to an incorrect execution. Rather than making D wait to begin execution until B completes, we take the more fine-grained approach of having D execute up to the dependent operation, *then* wait until B completes. This synchronization is performed using two additional types of signals: i) D *waits_for* B before performing the dependent operation, and ii) B *resumes* D upon completion. Note that this scheme for early signalling cannot be applied in some situations, such as when the candidate task appears in multiple locations in the task graph.

2.7 Speculation

When a task is replicated, the task and its replica may have dependences through the persistent heap. We initially solved this problem by creating synchronized sections around the accesses to shared data. In the case where actual sharing of data is rare, such synchronization can be a wasteful overhead—instead we would rather execute the critical section optimistically through support for *speculation* [7, 16] or *transactional execution* [13]. Such support involves two key mechanisms: (i) the ability to detect violated data dependences between speculative critical sections; and (ii) the ability to checkpoint a critical section and roll-back execution in the event of such a violation. When a task and its replica have conflicting critical sections, we must decide which should succeed and which must re-execute. Since these critical sections are unordered we must impose an order, such as the order in which the tasks entered the critical section. In addition to eliminating potentially unnecessary synchronization, speculation can also simplify the compiler since dependences do not have to be completely understood.

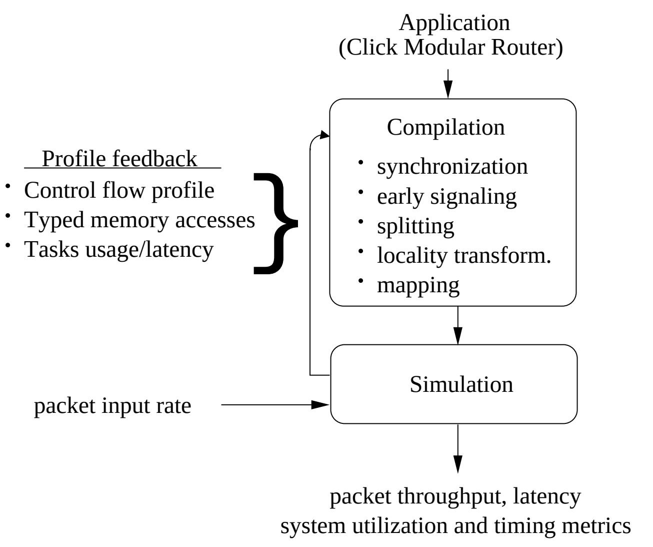


Figure 5: NPIRE: an infrastructure for integrated NP simulation.

2.8 Locality Transformation

In a typical mapping of a network application to a NP, the packet data is mapped to a memory with a large latency but high throughput. To help tolerate this latency and to reduce request traffic, NPs typically support “wide” memory operations where a single request results in the transfer of multiple memory locations—essentially, this results in the implementation of a software-managed cache. Compiler support for managing memory must be aware of this ability and automatically target wide memory operations when accessing a large data structure or when accessing several consecutive memory words.

3 The NPIRE Framework

In this section, we present NPIRE (Network Processor Infrastructure for Research and Evaluation), the basic structure of which is shown in Figure 5. The goal of NPIRE is to allow the investigation of high-level compiler transformations and NP architectures by mapping a benchmark application (composed of a graph of tasks) to a parameterizable NP architecture simulator. NPIRE has two major components: (i) a compiler, built on LLVM [9]; and (ii) a trace-based architectural simulator, which is integrated into the benchmark application by the compiler. The simulator provides feedback that allows the compiler to iteratively mold the application to the underlying NP. The remainder of this section describes NPIRE's compiler and simulator in more detail.

3.1 Compilation

The compiler divides Click's modular *elements* into tasks, and performs the transformations described in Section 2. However, the compiler must also perform the important tasks of memory mapping, task mapping, and task scheduling.

Memory Mapping The role of the compiler is to map each of the application-level memory types described in Section 2.2 to the different forms of physical memory available in the underlying NP: i.e., memory that is (i) local to the processor; (ii) shared at the processor level (e.g., for next-neighbor communication); (iii) shared chip-wide (e.g., a scratchpad); or (iv) shared in external (off-chip) memory. The mapping implemented in NPIRE of each application-level type of memory to physical memory is summarized later in Table 2. Our approach here in the mapping was to map dynamically sized application buffers to SRAM, the fastest external storage. Packet data is mapped to DRAM because of the potentially large amount of data to store. Stack is allocated in registers so that it is private to a context. Static persistent heap goes in local PE storage so that it can be shared by several contexts while providing a fast access to a few contexts.

Task Mapping While task mapping and scheduling are not the focus of this work, they are crucial steps that must be performed well to allow us to evaluate compiler task transformations. Automatically mapping a task graph to PEs is indeed a very challenging problem which gives rise to a strong tension between locality and parallelism. Locality is optimized by mapping related tasks to few PEs so that storage and communication are minimized, while parallelism is improved by mapping tasks to many PEs and exploiting more resources. In NPIRE, the mapping process is iterative, based on profile feedback from simulation, and proceeds in the following steps.

1. An initial measurement is made where each task runs on its own PE, assuming an infinite number of PEs.
2. Using a greedy algorithm, we then re-assign tasks to the parameterized number of PEs while minimizing the expected utilization of each PE.
3. Next we replicate the task with the longest queue time (according to profile feedback). Replicas can optionally be assigned to the same PE, or to different PEs. We repeat this step until the NP is well utilized.
4. Once a base mapping is decided, we attempt to improve it through simulated annealing using a faster, coarse-grain simulator to provide fast feedback.

Task Scheduling Once tasks have been mapped to PEs, there still remains some flexibility in the scheduling of tasks to the hardware contexts of each PE. We capitalize on the fact that only heap data is persistent

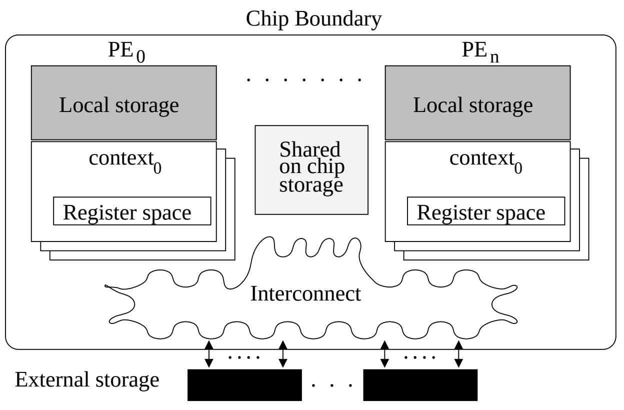


Figure 6: Generalized architecture of simulated NPs.

across task instances in a programming model such as Click's—this allows us greater flexibility in task scheduling, since a given task need not be bound to a certain hardware context within a PE. Instead, an instance of a task (identified by a task ID and a pointer to the packet that it will work on) is queued before each PE. A task is executed when it reaches the head of the queue and a hardware context is free to execute it. To improve load-balance, task replicas may be assigned to a small group of PEs. For such cases, we model a shared queue such that a replica may execute on the next PE in the group with an available hardware context. This model assumes that the code for each task or task replica has been loaded into each target PE.

3.2 Simulation

The simulation component of NPIRE is deeply parameterized, allowing us to vary the number of *processing elements* (PEs), the number of hardware contexts per PE, the interconnection between the PEs, and the implementation of the various forms of memory. Modern NP architectures are typically organized as a highly-multithreaded CMP executing either (i) independent threads having a global view of all shared on-chip resources; or (ii) a pipeline of processing stages where each processor is specialized to perform a certain task. A third alternative is a hybrid model (such as the Motorola C-5e [2] and the Intel IXP processors [6]) where the processing of a packet can flow through a general interconnection of processing elements that share memory. For generality, we focus on these hybrid architectures where the programmer is still free to adopt a pipelined or run-to-completion programming model.

The high-level architecture of the NPs that we can simulate with NPIRE is shown in Figure 6. The goal of NPIRE is to focus on thread-level parallelism and inter-task communication, similar to the Crowley and Baer [4] and the Intel IXP architecture tool [6]— hence NPIRE does not yet model the micro-architecture of PEs in detail. Instead, we assume a single-cycle-per-instruction PE model where instruction counts are obtained using a conversion factor from the intermediate representation to RISC instructions [1]. This abstraction technique, also

Table 2: Storage devices to which each memory type is mapped for our benchmark applications.

Device	**Application** / **Type**	Router	NAT
External SRAM	Packet Meta-Data	42B	36B
External DRAM	Packet Buffer	23B	45B
Local storage	Persistent Static Heap	5B	22B
Registers	Stack	0B	96B
External SRAM	Dynamic Heap	44B	49B

used by Vin *et al.* [15], gives us a reasonable approximation of the instruction counts and allows for a relative comparison of the impact of the proposed compiler transformations.

Modern NPs typically provide very low-latency switching between hardware contexts within a PE. In combination with non-blocking loads modeled in NPIRE, we can tolerate the significant latency of memory accesses. While in Intel's IXP NPs, a context switch is triggered explicitly by the programmer, our simulator triggers a context switch whenever a long-latency memory access occurs. Because we have replicated tasks containing critical sections, we must address potential deadlock scenarios through support for pre-emption, although we do not discuss this further here.

In NPIRE, to provide support for speculation (when speculation is used) the compiler first inserts code to checkpoint the context of a task at the entry of a critical section, and to restore this context in the event of a dependence violation and rollback. Next, in the simulator we model hardware support for detecting data dependence violations similar to that proposed previously [7, 16]—our focus is not to propose specific hardware support for speculation in NPs but instead to evaluate the potential performance benefits of optimistic critical sections.

4 Experimental Setup

We evaluate two benchmark applications: a RFC1812-compliant router (Router) and a network address translation application (NAT), both of which are adapted from those created by Kohler *et. al.* [8], and have been used previously to benchmark NPs. Both applications perform IP header processing—i.e., the packet payload is irrelevant. Also, each application has two input and two output interface tasks. In Router, packets are each processed by an average of 16 tasks (with a standard deviation of 1.12); 17% of the dynamic instructions that process one packet are in synchronized sections (due to the buffering of packets before being sent out after processing). In NAT, packets are each processed by an average of 12 tasks (with a standard deviation of 3.37), 32% of dynamic instructions that process one packet are in critical sections because of again output buffering and because of the packet rewriting tasks that create and access per-flow records used to translate packet header fields. Table 2 shows the devices to which each application-level memory

Table 3: NP Architecture Simulated. The total latency to access a form of storage is equal to the sum of all parts. For example, to access external SRAM takes $4 + 51 + 5 + 81 = 141$ cycles, of which 132 are pipelined.

Storage Type		**Unpipelined (cycles)**	**Pipelined (cycles)**
External DRAM	access	12	R 226 / W 0
	bus	4	59
External SRAM	access	5	81
	bus	4	51
On-chip shared SRAM	access	3	R 21 / W 8
	bus	3	37
Remote PE registers	access	1	12
	bus	1	1
Local store		4	11
Registers		4	0
Next-neighbor PE registers		4	4

Other Parameters	**Value**
processing element frequency	1 GHz
hardware contexts per PE	8
queue size for bus and memory controllers	40
pending loads allowed per context	3
rollback on failed speculation	40 cycles
context switch latency	0 cycles

type is mapped, as well as the average amount of data accessed per packet for each memory type. Finally, we measure our benchmark applications using modified packet traces from the Teragrid-I 10GigE NLANR trace [12].

One goal of this work is to evaluate the ability of compiler task transformations to scale a NP application up to larger numbers of PEs—hence we model NPs with a representative memory organization and varying numbers of PEs. Table 3 summarizes the architectural parameters of our modeled processor, in particular the latencies to access the various storage types available. While these parameters are inspired by the IXP2800 NP [6] (which has 16 PEs), our infrastructure is far more flexible than available IXP simulators, allowing us to target a wide variety of architectures. We model several types of storage and bus interconnects, each of which have both pipelined and unpipelined components (as described in Table 3). The simulated NP has seven external memory channels: three for DRAM and four for SRAM. There are four DRAM buses which can communicate with all three DRAM channels through a time-multiplexed connection. The NP is divided in half such that half of the PEs each have two dedicated DRAM buses; of the two buses, one is for reads and one is for writes. SRAM is accessed through an additional four on-chip buses that have the same organization as the DRAM buses. Furthermore, through another bus each PE has access to a shared on-chip SRAM as well as shared registers on remote PEs. Finally, each PE has direct access to local storage, its own registers, and certain registers of its next-neighbor PEs.

To evaluate and compare different task transformation and mapping techniques and their ability to effectively scale an application to many PEs, we need a method for finding the maximum sustainable packet input rate. We derive this rate by finding, through a bisection search, the smallest effective packet inter-arrival time where the packet inter-departure time is equivalent (i.e., the NP can keep up). When we refer to this rate as to the resulting “throughput” of an experiment in the next section.

5 The Impact of Task Transformations

In this section, we use NPIRE to evaluate the impact on packet throughput for the two benchmark applications (ROUTER and NAT) of the compiler task transformations described in Section 2, and their ability to allow the applications to scale to NPs with larger numbers of PEs. For each application we also identify the performance bottleneck, and determine whether it is a limitation in the application or instead a saturated resource in the underlying NP. Note that since both applications each have two input and two output interface tasks, four is the minimum number of PEs allowing us to bind each interface task to a PE. Hence in all of our result graphs, throughput speedup is computed relative to the maximum sustainable packet throughput on four PEs with no task transformation.

Replication In this section, we evaluate four different replication scenarios for the *replication* task transformation presented in Section 2.4. The simplest scenario (*no_transformation*) has no replication and simply extends the mapping to the number of available PEs. Next, we investigate the case where a task and its replicas are limited to execute on a certain PE (*rep_on_PE*), but may use any available hardware context on that PE. A more flexible form of replication allows a task and its replicas to execute on any available context within a small subset of PEs; we call this scheme “subset replication” (*subset_rep*). Finally, we examine the case where any task or replica can execute on any available context of any PE, in a sense modeling a global task queue (*rep+global_task_queue*).

In Figure 7(a), we first observe that with no transformation, ROUTER does not scale at all as we increase the number of PEs from four. Next, when the replication transformation is performed but the replicas are limited to a single PE (*rep_on_PE*), we see that throughput is improved by a nearly constant amount (by about 5x), but that this does not facilitate scaling either. We observe that subset replication is ineffective until the number of PEs is greater than the average number of active tasks in the application (i.e., greater than 16 PEs, as described in Section 4). For this transformation with larger numbers of PEs (such as 28 and 32 PEs), we see that the mapping is imperfect, leading to inconsistent throughput scaling—this underlines the importance and difficulty of mapping. Finally, we see that ROUTER with a global task queue scales asymptotically up to 30 PEs. As the number of PEs increases, PE utilization steadily decreases due to increasing contention on the external SRAM and DRAM memory and the corresponding buses. In this case, the maximum throughput obtained on 30 PEs is 27.2 times the throughput of the application with no transformation—hence the combination of task replication with more flexible task scheduling can be quite powerful.

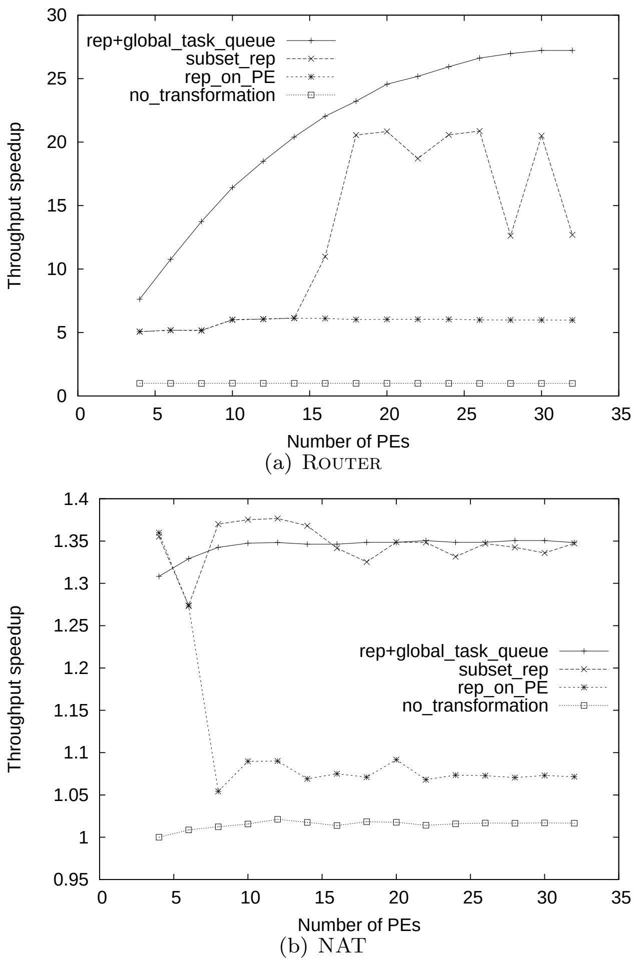
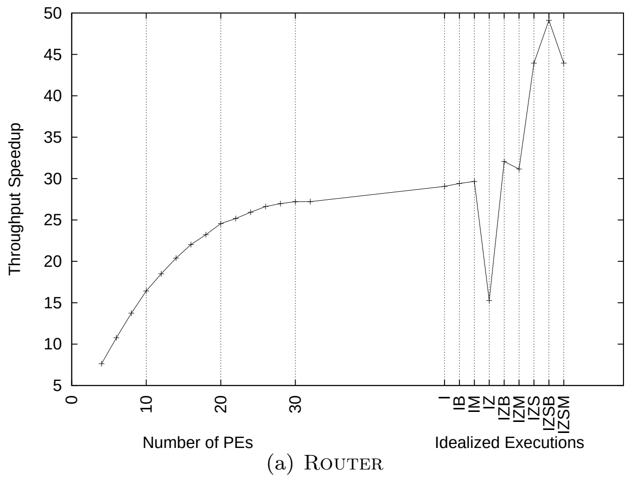
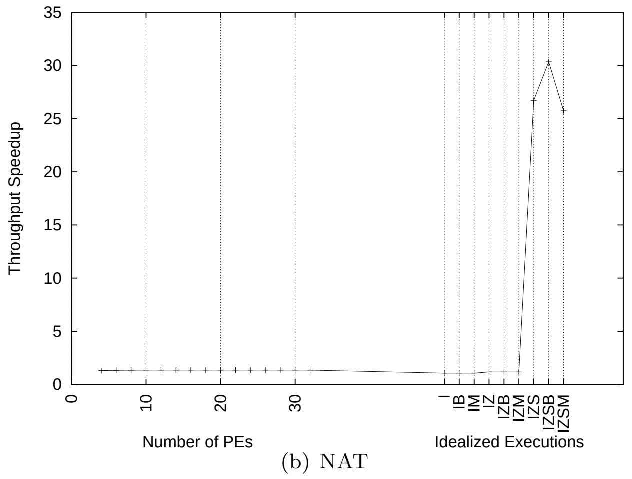


Figure 7: Impact of three variations of replication on throughput with varying numbers of PEs, relative to the application with no transformation running on 4 PEs. `subset_rep` means replication where a task and its replicas can execute on a small subset of PEs, while `rep_on_PE` means replication where replicas may only execute on a specific PE.

Because NPs are complex systems, with many tasks performing many communication and computation operations in parallel, identifying the performance bottleneck for a given configuration is difficult. One method for bottleneck identification involves idealizing potential bottleneck sources, as shown in Figure 8(a) where we focus on the experiment of the global task queue from Figure 7(a). In those “idealized executions”, we first observe that having infinite PEs alone (*I*) is ineffective, indicating that processing resources are not a bottleneck. Second, modeling zero-cycle instructions (*IZ*) actually makes things worse, because a large number of bus requests must retry, and the overall fairness for bus requests is reduced. Eliminating synchronization (**S**) provides the biggest win, motivating effort in more efficient or speculative critical sections. Finally, the further addition of perfect bus pipelining provides the best overall throughput (*IZSB*), confirming that bus contention is the next-most important bottleneck after synchronization for ROUTER.

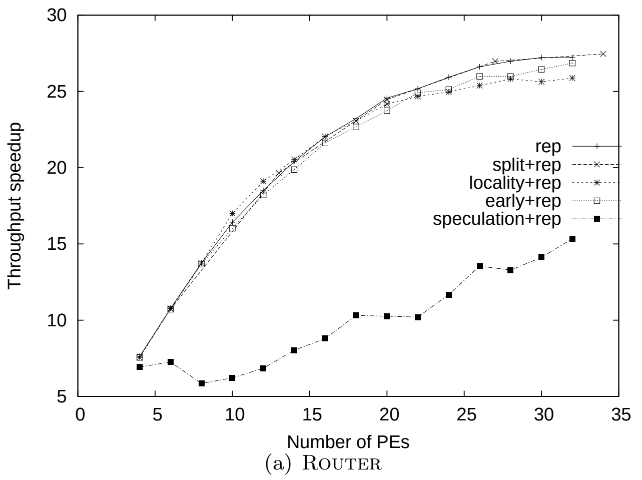


(a) Router

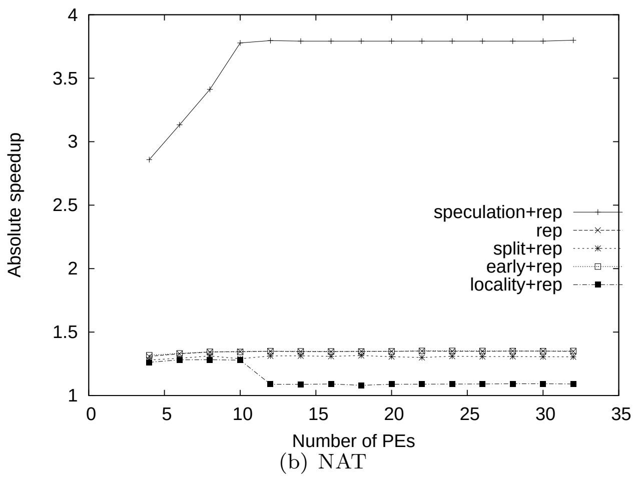


(b) NAT

Figure 8: Impact of replication with a global task queue on throughput for varying numbers of PEs, relative to the application with no transformation running on 4 PEs. Combinations of idealized executions are plotted to the right: infinite number of PEs (*I*); maximum bus pipelining (*B*); maximum memory pipelining (*M*, i.e. the unpipelined time for a request is 1 cycle); zero instructions (*Z*); and no synchronization (*S*, i.e. there are no critical sections and we ignore inter-replica dependences).



(a) Router



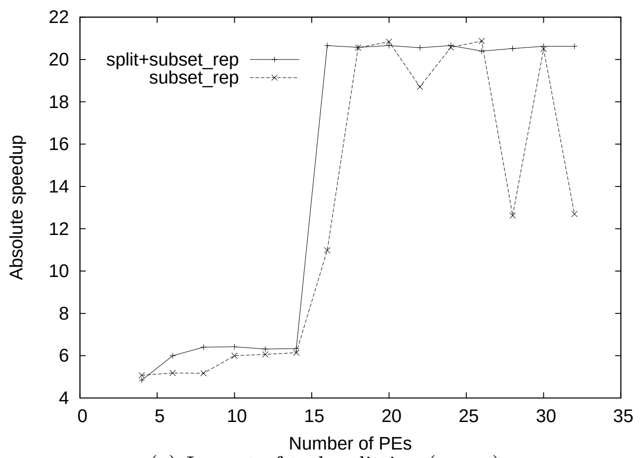
(b) NAT

Figure 9: Impact of the locality (`locality`), early signalling (`early`), and speculation (`speculation`) transformations on throughput for varying numbers of PEs, relative to the application with no transformation running on 4 PEs. Each experiment includes replication with a global task queue.

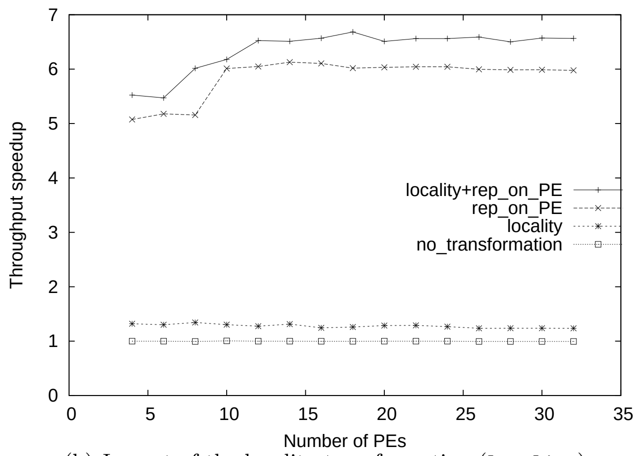
In Figure 7(b), the throughput of NAT with no transformation does not scale at all for increasing PEs. Replication limited to a PE provides a modest throughput speedup of 1.36 for 4 PEs, but this gain is negated as we increase the number of PEs: this is due to the reduction in locality for data in the persistent heap, underlining the tension between parallelism and locality. The greater flexibility of subset replication with a global task queue stabilize throughput, but the result is far from scaling nicely to increasing PEs—hence there is a severe bottleneck for NAT. Using Figure 8(b), we attempt to find this bottleneck. Similar to our result for Router, we find that synchronization (**S**) is again the key bottleneck.

Task Splitting As it can be seen on Figures 9(a) and 9(b), task splitting does not significantly impact the throughput of Router and NAT: the communication overhead between the task splits actually reduces the throughput of NAT by 3% (Figure 9(b)). Task splitting does not increase parallelism because the task-splits execute in sequence, and are mapped to the same PEs as the original unsplit task. The impact of task splitting is more evident when replication is limited, as shown in Figure 10(a) where we evaluate subset-replication with and without the task splitting transformation on Router. With subset replication, task splitting improves the throughput on a small number of PEs because the finer granularity of the task-splits improves the load balance. We also observe that splitting can ease mapping, since the throughput is stabilized when increasing the number of PEs from 16 (compared to no task splitting).

Early Signaling Figure 9(a) shows the impact of the early signalling transformation on Router, where throughput is reduced by 1%. This reduction is due to increased task switching since there is more inter-task syn-



(a) Impact of task splitting (`split`)



(b) Impact of the locality transformation (`locality`)

Figure 10: Impact of transformations on the throughput of ROUTER for varying numbers of PEs, relative to the application with no transformation running on 4 PEs.

chronization. However, we found that early signalling reduces packet processing latency by 8% on average across all simulated NPs (i.e., when varying the number of PEs), hence this transformation can still be compelling in some scenarios. In Figure 9(b), we see that for NAT early signaling increases the maximum throughput by only 0.1%. Furthermore, for NAT, there is no significant improvement in packet processing latency. A closer look revealed that few tasks were signalled early for NAT, indicating that for this application most tasks are inter-dependent.

Speculation Support for speculation allows the NP to optimistically execute critical sections, as described in Section 2.7 From Figure 9(a) we see that for ROUTER speculation has a significant negative impact. Surprisingly, the fraction of speculated critical sections for which speculation fails is fairly small: for example, speculative critical sections fail only 5% of the time for 8 PEs, and 6.5% for 32 PEs. This indicates that for ROUTER the overhead of mis-speculation is intolerable. In contrast, in Figure 9(b) we see that support for speculation results in a dramatic improvement in throughput for NAT, for two reasons. First, as we observed in Section 5, NAT is bottlenecked mainly on synchronization of critical sections. Second, we found that the fraction of speculated critical sections for which speculation fails is much smaller than for ROUTER: for example, in NAT speculative critical sections fail only 1.8% of the time for 12 PEs, and 2.2% of the time for 32 PEs. These results indicate that speculation must be used judiciously, but can offer compelling improvements in throughput for applications that are bottlenecked on synchronization.

Locality Transformation Combining accesses to external memory reduces the number of accesses at the cost of more bursty access behavior. As seen in Figure 9(a), the NP is able to accommodate the traffic burstiness of the locality transformation which results in a slight improvement in the throughput of ROUTER for between 10 and 20 PEs (for example, throughput is improved by 3% for 12 PEs). The average fraction of time a PE spends waiting for memory is reduced by nearly 50%, and overall DRAM utilization is reduced by 5%. However, utilization for both the SRAM read buses and the shared on-chip bus are increased by 10%, and the DRAM read bus utilization is increased by 3%. In Figure 10(b), the locality transformation on ROUTER leads to a decent improvement in throughput over in the application without replication and replication limited to a PE. For NAT, shown in Figure 9(b), the locality transformation degrades performance beyond 10 PEs: in this case, the benefit of combined accesses is negated by increased bus contention, which in turn leads to a significant increase in the amount of time spent in critical sections. In summary, the locality transformation is safer to use when task scheduling is more localized, and must be used carefully to avoid the situation where the resulting bursty traffic impedes the execution of critical sections.

6 Conclusions

We have demonstrated that a NP application described as a graph of tasks in a high-level language can be transformed automatically by a compiler to exploit NP resources, and to scale up to NPs with larger numbers of PEs. Of the transformations that we investigated, replication was the most effective at scaling an application to larger numbers of PEs. We also found that flexibility in scheduling tasks and their replicas is key to scaling throughput with the number of PEs. While early signalling did not significantly improve throughput, we observed that it can reduce packet processing latency. When the tasks and their replicas are limited to execution on a small number of PEs, we found that task splitting can be used to improve load balance. We demonstrated that support for speculation can result in a dramatic improvement in throughput when critical sections are executed optimistically, for applications where the synchronization of critical sections are a bottleneck and the frequency of mis-speculation is low. Finally, combining references to consecutive locations in external memory into a wide reference can improve throughput by reducing the number of references, but that the bursty behavior of this transformation can lead to a prohibitive increase in contention if the interconnect is already saturated.

References

[1] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. LLVA: A Low-level Virtual Instruction Set Architecture. In *MICRO-36*, San Diego, California, Dec 2003.

[2] C-Port. *C-5 Network Processor Architecture Guide*, C-5 NP D0 Release edition, 2002.

[3] M. K. Chen, X. F. Li, R. Lian, J. H. Lin, L. Liu, T. Liu, and R. Ju. Shangri-la: achieving high performance from compiled network applications while enabling ease of programming. In *PLDI '05*, pages 224–236, NY, USA, 2005. ACM Press.

[4] P. Crowley and J.-L. Baer. A modeling framework for network processor systems. *Network Processor Design*, 1, 2002.

[5] J. Dai, B. Huang, L. Li, and L. Harrison. Automatically partitioning packet processing applications for pipelined architectures. In *PLDI '05*, pages 237–248, NY, USA, 2005. ACM Press.

[6] S. Goglin, D. Hooper, A. Kumar, and R. Yavatkar. Advanced software framework, tools, and languages for the IXP family. *Intel Technology Journal*, 7(4), November 2003.

[7] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS 8*, October 1998.

[8] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.

[9] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO'04*, Palo Alto, California, March 2004.

[10] L. Li, B. Huang, J. Dai, and L. Harrison. Automatic multithreading and multiprocessing of C programs for IXP. In *PPoPP '05*, pages 132–141, NY, USA, 2005. ACM Press.

[11] J. Mudigonda, H. M. Vin, and R. Yavatkar. Overcoming the memory wall in packet processing: hammers or ladders? In *ANCS '05*, pages 1–10, New York, NY, USA, 2005. ACM Press.

[12] National Laboratory for Applied Network Research. Passive measurement and analysis. http://pma.nlanr.net/PMA/, Feb. 2004.

[13] R. Rajwar and J. Goodman. Transactional execution: Toward reliable, high-performance multithreading. *IEEE Micro*, 23(6):117–125, November/December 2003.

[14] G. Schelle and D. Grunwald. CUSP: a modular framework for high speed network applications on FPGAs. In *FPGA*, 2005.

[15] H. M. Vin, J. Mudigonda, J. Jason, E. J. Johnson, R. Ju, A. Kunze, and R. Lian. A programming environment for packet-processing systems: Design considerations. In *Workshop on Network Processors & Applications - NP3*, February 2004.

[16] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler optimization of scalar value communication between speculative threads. In *ASPLOS-10*, San Jose, October 2002.