Task Partitioning for Multi-core Network Processors

Robert Ennals[1], Richard Sharp[1], and Alan Mycroft[2]

[1] Intel Research Cambridge,
15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK
[2] Computer Laboratory, Cambridge University,
15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK
{robert.ennals, richard.sharp}@intel.com
am@cl.cam.ac.uk

Abstract. Network processors (NPs) typically contain multiple concurrent processing cores. State-of-the-art programming techniques for NPs are invariably low-level, requiring programmers to partition code into concurrent tasks early in the design process. This results in programs that are hard to maintain and hard to port to alternative architectures. This paper presents a new approach in which a high-level program is separated from its partitioning into concurrent tasks. Designers write their programs in a high-level, domain-specific, architecturally-neutral language, but also provide a separate Architecture Mapping Script (AMS). An AMS specifies semantics-preserving transformations that are applied to the program to re-arrange it into a set of tasks appropriate for execution on a particular target architecture. We (i) describe three such transformations: pipeline introduction, pipeline elimination and queue multiplexing; and (ii) specify when each can be safely applied.

As a case study we describe an IP packet-forwarder and present an AMS script that partitions it into a form capable of running at 3Gb/s on an Intel IXP2400 Network Processor.

1 Introduction

This paper addresses *an instance* of a perennial general problem in the compilation of concurrent systems to parallel hardware architectures:

> *Given a program which expresses problem-oriented concurrency, and hardware which has multiple processing elements, how can we efficiently map one to the other?*

The instance we attack concerns the domain of packet processing applications such as Internet routers, firewalls and similar network devices. The parallel hardware architectures we target are Network Processors (NPs) [1, 6, 10, 23]: specialised programmable chips designed for high-speed packet processing.

NPs typically contain multiple processor cores, allowing multiple network packets to be processed concurrently. To make a program run fast on such an

R. Bodik (Ed.): CC 2005, LNCS 3443, pp. 76–90, 2005.

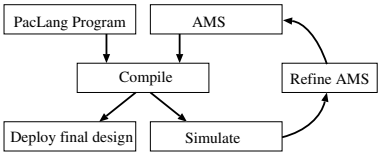

Fig. 1. Design flow using our compiler

architecture it is necessary to partition it into a number of separate concurrent tasks, such that the number of tasks matches the number of cores on the target architecture. Furthermore, tasks arranged in a pipeline configuration should be balanced, with similar latencies.

State-of-the-art programming techniques for NPs are invariably low level, requiring the programmer to explicitly code a separate process for each core and explicitly pass state between processes. In this way, the programmer is forced to combine high-level application functionality with low-level architectural details in a way that makes them difficult to separate. This results in programs that are hard to maintain and strongly tied to a particular revision of a particular architecture.

This paper describes a new approach, in which the high-level application functionality is completely separated from the architectural details of any specific NP. Our compiler takes two files as input: a high-level packet processing program and an *Architecture Mapping Script* (AMS). The AMS specifies (*i*) how the high-level program should be transformed into a new set of concurrent tasks suitable for execution on a particular NP architecture; and (*ii*) how these tasks should be mapped to the NP's processing cores[1]. The compiler checks that the transformations specified in the AMS are semantics-preserving.

We use the domain-specific language PacLang [4] to express the high-level behaviour of packet processing applications in an *architecturally-neutral* way (i.e. without encoding assumptions about any particular target architecture). A PacLang program consists of multiple concurrent tasks that communicate via shared queues. Such parallelism, and the controlled non-determinism it introduces, is essential if one is to conveniently express packet processing algorithms. For example, in an IP forwarder, non-critical, computationally expensive packets can be processed on different tasks to critical packets, allowing critical packets to overtake the non-critical ones.

Figure 1 illustrates the design flow that we intend programmers to follow when using our compiler. After writing a PacLang specification and an initial AMS for a particular architecture, the compiler is invoked and the resulting code simulated using architecture-specific tools. Based on the profiling results derived from simulation, the AMS is iteratively refined to explore different partitionings and timing behaviours.

[1] Although there is scope for generating an Architecture Mapping Script automatically for particular architectures, that is not the topic of this paper.

The contributions of this paper are: (*i*) a methodology for programming multi-core Network Processors that separates architectural-details from high-level application specification; (*ii*) a set of semantics-preserving program transformations that re-arrange a concurrent program into a different set of concurrent units (i.e. tasks and queues); and (*iii*) a whole-program analysis that determines when it is safe to pipeline a specified task in the wider context of a whole concurrent program.

We have implemented a PacLang compiler targeting Intel IXP2400 Network Processors [10]. To demonstrate that the techniques described in this paper are applicable to realistic networking applications, we present a case-study showing how an architecturally-neutral PacLang IP packet-forwarder can be transformed into a form suitable for implementation on an Intel IXP2400 chip. We present performance figures, showing that executing the compiled code on a 3-port Gigabit Ethernet IXP2400-based system [20] achieves a forwarding rate of 3 Gb/s (full line-rate).

For expository purposes, this paper initially presents our partitioning transformations (Section 3) and safety analysis (Section 4) in the domain of *Core PacLang*—a simplified version of PacLang. Later, the transformations and safety analysis are then extended from Core PacLang to the full PacLang language (Section 5). We finish by presenting a case-study (Section 6), related work (Section 7), and our conclusions (Section 8).

2 Core PacLang Language

We start by considering Core PacLang: a simplified version of the PacLang language. Unlike PacLang proper, Core PacLang is untyped, supports only value types (no references or pointers) and has no user-defined functions.

$e \leftarrow c \mid x \mid op\ (e_1, \ldots, e_k)$	constant, local variable, primitive op
$s \leftarrow$ `if` $(e)\ s_1$ `else` s_2	conditional
$\mid$ `while` $(e)\ s$	while loop
$\mid\ s$`;` s	sequence
$\mid$ `skip`	do nothing
$\mid\ x$ `=` e	imperative assignment
$\mid\ q$`.enq` $(e_1, \ldots, e_k)$	enqueue
$\mid\ (x_1, \ldots, x_k)$ `=` q`.deq()`	dequeue
$d \leftarrow$ `task` $t\ \{s\}$	task t with body s
$\mid$ `queue` q`;`	global queue q
$p \leftarrow d_1 \ldots d_n$	Core PacLang program

Fig. 2. Abstract Syntax of Core PacLang

The abstract syntax of Core PacLang is presented in Figure 2. A program consists of a set of declarations, each of which is either a *task*, t, or a *queue*, q. Tasks are the unit of concurrent execution: during program execution, all task bodies run concurrently. Tasks are statically declared; there is no dynamic thread creation in PacLang. (Note that tasks' names have no significance other than providing a convenient way of referencing them in AMSes.)

Each task has a body, s, which it executes repeatedly. One can imagine that task bodies are surrounded by an invisible "`while (true)`" loop. When a task body restarts all its local variables are uninitialised. This ensures that there are no loop-carried-dependencies between subsequent iterations of a task body.

Queues provide inter-thread communication. In Core PacLang we assume that queue-read operations block when the queue is empty and queue-write operations never block (i.e. queues are unbounded). We write `q.enq(`$v_1, \ldots, v_n$`)` to atomically enqueue values $v_1, \ldots, v_n$ in the queue declared with name `q`. Similarly, `q.deq()` returns (multiple) values dequeued from `q`. Built-in queues, `receive` and `transmit` represent the external network interface. `receive.deq()` reads a value from the network[2]; `transmit.enq(`v`)` schedules value v for transmission.

In untyped Core PacLang, variables do not have to be declared explicitly and are scoped by their enclosing task.[3] To simplify the presentation of subsequent transformations we assume that all declared names (i.e. local variable names, queue names and task names) are globally distinct.

3 Semantics-Preserving Transformations for Partitioning

In this section we present three semantics-preserving transformations that allow programs to be repartitioned into different numbers of concurrent tasks. The first transformation, *PipeIntro* (Section 3.1), divides a single task into two separate, concurrent tasks connected in a pipeline configuration. The second transformation, *PipeElim* (Section 3.2), allows two pipeline stages connected via a queue to be fused into a single task. The third transformation, *QueueMux* (Section 3.3), allows multiple queues to be multiplexed onto a single queue. We first present the transformations in the domain of Core PacLang; Section 5 shows that they naturally extend to deal with the full PacLang language.

Although we do not prove our transformations formally in this paper, it is necessary nonetheless to define precisely what we mean by semantics-preserving. In previous work we presented a small-step transition semantics for PacLang [4]. The semantics is non-deterministic, making no guarantees about the interleaving of concurrent tasks' operations and making no guarantees about progress or fairness. With reference to this semantics, we say that a transformation is semantics-preserving iff the set of *possible behaviours* of the transformed program is a *subset* of the possible behaviours of the source program, where the

[2] Full PacLang supports a structured *packet* datatype to represent such packets.
[3] In contrast, full PacLang supports C-like variable declaration and lexical scoping.

possible behaviours of a PacLang program are the set of possible traces of values on external queues (`receive` and `transmit`). In other words, transformations can increase determinism, narrowing the set of possible behaviours, but any behaviour exhibited by the transformed program must have also been a possible behaviour of the source program.

3.1 PipeIntro Transformation

The PipeIntro transformation facilitates pipelining, allowing a task t to be transformed into two separate, concurrent tasks, t_1 and t_2—see Figure 3. Here, and throughout the rest of this paper, we let $\mathcal{A}$ and $\mathcal{B}$ range over statements. Queue Q is used to transfer the variables required by $\mathcal{B}$ (i.e. the live variables in task t at the program point between $\mathcal{A}$ and $\mathcal{B}$) from t_1 to t_2. Recall that statements may themselves include sequences of other statements. This, and the fact that we make the "`;`" operator associative, allows the PipeIntro transformation presented in Figure 3 to split task t between any two statements that are not nested within a `while` loop or a conditional.

In order to preserve the semantics of a Core PacLang program, the PipeIntro transformation can only be applied under certain conditions. In Section 4 we present the technical details of a static analysis that determines when it is safe to apply PipeIntro. We spend the remainder of this section highlighting the need for a safety analysis, by giving examples of *unsafe* applications of PipeIntro. First consider:

```
queue q1;
task t { x=q1.deq(); y=q1.deq(); transmit.enq(x,y); }
```

Task `t` continually reads pairs of values from `q1` and writes them to `transmit` in the order they were read. If we were allowed to apply the PipeIntro transformation arbitrarily we might choose to split between the two queue read operations, yielding:

```
queue q1; queue Q;
task t1 { x=q1.deq(); Q.enq(x); }
task t2 { x=Q.deq(); y=q1.deq(); transmit.enq(x,y); }
```

$$\texttt{task } t\ \{\mathcal{A};\mathcal{B}\} \quad \longrightarrow \quad \begin{array}{l}\texttt{queue } Q;\\ \texttt{task } t_1\ \{\mathcal{A};\ Q\texttt{.enq}(x_1,\ldots,x_k)\}\\ \texttt{task } t_2\ \{(x_1,\ldots,x_k)\ \texttt{=}\ Q\texttt{.deq()};\ \mathcal{B}\}\end{array}$$

where Q, t_1 and t_2 are fresh names and $x_1, \ldots, x_k$ are the live variables of task t at the program point between statements $\mathcal{A}$ and $\mathcal{B}$. (Recall the x's in t_1 are different from the x's in t_2 because they are locally scoped.)

Fig. 3. The PipeIntro Transformation

In the transformed program, the values on `transmit` might not appear in the same order that they were read from `q1`. For example, task `t1` may consume the first 5 elements from the `q1` before task `t2` has had a chance to read `q1` at all.

The unsafe application of PipeIntro given above may lead the reader to think that a suitable safety condition may be that, in the source program, the queues accessed (read or written) by the statements before the split point should be disjoint from the queues accessed by the statements after the split point. However, this condition is not sufficient in general. Consider the following program:

```
queue q;
task t { q.enq(1); transmit.enq(2); }
task connect_q_to_transmit { transmit.enq(q.deq()); }
```

Task `t` task writes a "1" to `q`, then writes a "2" to the `transmit` queue and then loops. Task `connect_q_to_transmit` reads elements from `q` and writes them to the `transmit` queue. If we now apply PipeIntro to `t`, splitting between the two queue write operations, we get:

```
queue q; queue Q;
task t1 { q.enq(1); Q.enq(); }
task t2 { ignore = Q.deq(); transmit.enq(2); }
task connect_q_to_transmit { transmit.enq(q.deq()); }
```

These two programs are not semantically equivalent (even though, in the source program, the statements on either side of the split point access disjoint queues)—e.g. in the transformed program the trace $\langle 1, 1, 1 \rangle$ may appear on the `transmit` queue; this is not a valid trace of the source program[4].

Informally the problem is that `t1` affects `connect_q_to_transmit` which shares a queue with `t2`. In Section 4 we present a static analysis that determines when PipeIntro can be safely applied.

3.2 PipeElim Transformation

The PipeElim transformation allows two tasks t_1 and t_2 connected by a single-reader, single-writer queue q to be fused into a single task t. In essence the code for t_2 is inlined into t_1 in place of its write to q—see Figure 4. Since the queue write operation can occur anywhere within a t_1 (e.g. nested inside conditionals or while loops) we express PipeElim in terms of a *context* [22], $\mathcal{C}$, defined below:

$$\begin{aligned}\mathcal{C} \leftarrow\ & [\cdot] \mid s;\mathcal{C} \mid \mathcal{C};s \mid \texttt{while (}e\texttt{) } \mathcal{C} \\ & \mid \texttt{if (}e\texttt{) then } \mathcal{C} \texttt{ else } s \mid \texttt{if (}e\texttt{) then } s \texttt{ else } \mathcal{C}\end{aligned}$$

In joining concurrent tasks, the PipeElim transformation essentially picks a static interleaving of operations from t_1 and t_2, encoding this schedule explicitly in the order of statements in t. For the sake of simplicity, the transformation shown in

[4] The source program ensures that: (the number of 1's on the transmit queue) $\leq$ (the number of 2's on the transmit queue) + 1.

```
queue q                                              task t {
task t1 {C[q.enq(e1, ..., ei)]}          ⟶            C[A; x1=e1; ...; xi=ei; B]
task t2 {A; (x1, ..., xi) = q.deq(); B}              }
```

where there are no other references to q in the rest of the program; we assume that task-local variables in t_1 and t_2 have been renamed so as to be disjoint.

Fig. 4. The PipeElim Transformation

Figure 4 just inlines the body of t_2 into t_1. Note, however, that PipeElim is merely an instance of a more general *transformation schema* which may interleave the statements from $\mathcal{A}$ and $\mathcal{B}$ with the statements of t_1's body in a variety of ways, exploring different static schedules.

Depending on the static schedule implicitly specified by an application of PipeElim, deadlock may be introduced. For example $\mathcal{B}$ might block waiting for a queue that t_1 would have written to immediately after writing to q. Although such deadlocks are consistent with our subset interpretation of semantics-preserving, they are clearly undesirable. In this paper we do not consider deadlock detection further; however, we are currently implementing a "deadlock and timing analyser"[5] that checks whether (transformed) PacLang programs meet user-specified timing constraints.

3.3 QueueMux Transformation

The QueueMux transformation is used in conjunction with PipeElim to fuse concurrent tasks that are *not* connected in a pipeline configuration (i.e. concurrent tasks that cannot be fused using PipeElim alone).

The effect of a QueueMux transformation on program structure is shown in Figure 5. We start with n queues $(q_1, \ldots, q_n)$ each read by a single reader task. After transformation, a task body that previously wrote a value, v, to q_i $(1 \leq i \leq n)$ now writes a *pair* of values (i, v) to a *Combined Queue*, Q. A *Demux* task dequeues these (i, v) pairs, testing the value of i to determine which of the original queues v should be forwarded to.

Once a QueueMux has been applied, PipeElim can be applied as many times as required to combine each of the reader tasks with the Demux task (see Figure 5). The case study in Section 6 demonstrates this technique in practice.

3.4 Architecture Mapping Scripts

For a particular NP architecture, A, and an architecturally-neutral PacLang program, $\mathcal{P}$, an Architecture Mapping Script (AMS) specifies both:

[5] After all, for real-time reactive systems, deadlocks are just a special case of failing to meet timing requirements!

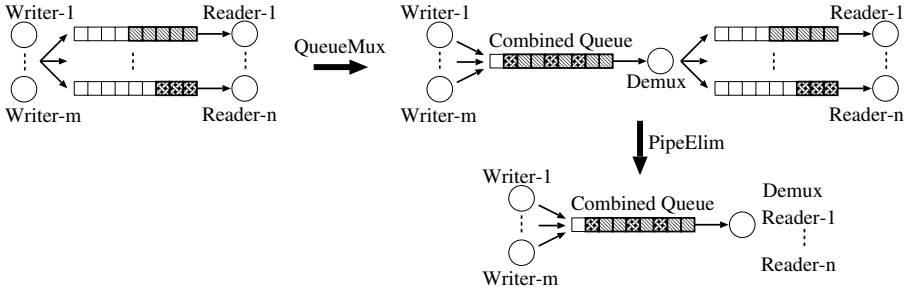

Fig. 5. Applying the QueueMux Transformation, followed by PipeElim

– how the PipeElim, PipeIntro and QueueMux transformations should be applied to $\mathcal{P}$ in order to refine it into a form suitable for execution on A; and
– how the tasks and queues after transformation are to be mapped onto the low-level resources of A.

The precise syntax of Architecture Mapping Scripts is straightforward. Although the technical details are omitted from this paper due to space constraints, the interested reader may download real examples of AMSes from the web [11].

4 Safety Analysis for PipeIntro Transformation

Here we present a static analysis which enables the PipeIntro transformation by conservatively determining whether the transformation is safe.

The PipeIntro transformation (as presented in Figure 3) allows a subsequent iteration of $\mathcal{A}$ to start before a previous iteration of $\mathcal{B}$ has finished. Therefore, the transformation is safe if an *observer* (who reads from transmit queues) is unable to infer that an execution step in an iteration of $\mathcal{A}$ *occurs before* an execution step in an iteration of $\mathcal{B}$. We model this observer by adding a task to the program that reads from all transmit queues. The analysis then determines whether this observer task might be able to infer that an execution step in an iteration of $\mathcal{A}$ occurs before an execution step in an iteration of $\mathcal{B}$.

We start by considering what information a task, t, might infer about the ordering of execution steps in other tasks. We note that a task can only infer ordering information about other tasks' execution steps by reading from a shared queue. (One cannot infer anything by performing a queue write, as writes return no information.)

We let u, v, w (in addition to t) range over tasks. We write $u \overset{t}{\rightsquigarrow} v$ to mean that task t may infer that an execution step of task u occurred before an execution step of task v by reading a queue. The relation '$\overset{t}{\rightsquigarrow}$' is defined as follows:

1. if t and u both read from q, then $t \overset{t}{\rightsquigarrow} u$ and $u \overset{t}{\rightsquigarrow} t$;
2. if t reads from q and u writes to q, then $u \overset{t}{\rightsquigarrow} t$;
3. if t reads from q and both u and v write to q, then $u \overset{t}{\rightsquigarrow} v$ and $v \overset{t}{\rightsquigarrow} u$.

We justify these three cases as follows:

1. If u and t both read from q then t may be able to infer the order of its reads w.r.t. u's reads—e.g. let q be a queue containing sequential integers starting from "1". If t's first read returns "2" then it knows that u must have read first.
2. If u writes to q and t reads from q then t may be able to determine that its read occurred after u's write—e.g. if t read the value written by u. However, it is not possible for t to infer that its read occurred *before* u's write. Nor is it possible for t to infer that any other task's read from q has occurred before any further task's write to q.
3. If u and v both write to q then t may be able to infer the order in which the writes occurred—e.g. t may perform two read operations and compare the values returned with those expected.

But we cannot just apply these rules and ask "can the observer infer that an execution step of $\mathcal{A}$ occurs before an execution step of $\mathcal{B}$". Firstly, consider the case where task u passes information (via a shared queue) to task v. The data transferred may reveal, to task v, the event orderings observed by task u. To simplify the analysis we conservatively assume that every task may get to know all orderings observed by all other tasks. Therefore, we define:

$$u \rightsquigarrow v \overset{\text{def}}{\iff} \exists t. u \overset{t}{\rightsquigarrow} v$$

i.e. $u \rightsquigarrow v$ holds iff *any task* may observe that an execution step in u occurs before an execution step in v. Secondly, we note that, if $u \rightsquigarrow v$ and $v \rightsquigarrow w$ then one *may* use this information to deduce that an execution step of u occurs before an execution step of w. It is thus necessary to consider $\rightsquigarrow^*$, the transitive closure of $\rightsquigarrow$.

The PipeIntro transform as presented in Figure 3 is safe if in the transformed program with queue Q removed, it is not the case that $t_1 \rightsquigarrow^* t_2$.

4.1 Algorithm for PipeIntro Safety Analysis

Taking the safety analysis presented above, and making the conservative assumption that all queues have readers, leads to the following simple algorithm for determining whether a PipeIntro transformation (as presented in Figure 3) can be applied:

1. Construct a graph, G, where nodes are tasks in the transformed program.
2. In the transformed program with queue Q removed (see Figure 3) consider each pair of tasks, u and v, that share a queue, q. Place a directed edge from u to v if:

(a) u and v both read from q; or
(b) u writes to q and v reads from q; or
(c) u and v both write to q.

3. If, there is no path in G from t_1 to t_2 then the PipeIntro transformation can be applied.

5 Dealing with the Full PacLang Language

The full PacLang language supports a number of constructs omitted from the core language of Section 2 including user-defined functions, references, arrays (of values or of queues) and global variables. Here we discuss how these additional features impact the transformations presented in Section 3.

User-defined functions can be dealt with straightforwardly: for the purposes of this paper we simply restrict functions to being non-recursive and then assume all user-defined function calls are inlined (although, in practice one need only inline a function call if the AMS requests that it be split across several tasks).

The introduction of generalised global variables is also largely straightforward[6]. The PipeElim and QueueMux transformations are unaffected by the introduction of global variables. However, the PipeIntro safety analysis needs to be extended accordingly (see Section 5.2).

The impact of introducing references needs to be considered more carefully. If we permitted the *unrestricted* use of references then the PipeElim and QueueMux transformations would remain sound, but PipeIntro would not. In the following subsection we explain informally why PacLang's linear type system [4] is sufficient to ensure that PipeIntro (as already presented) remains sound in the PacLang domain, even when references are used.

5.1 References, Linearity and the PipeIntro Transform

The full PacLang language provides a *packet* datatype. Packets are dynamically allocated blocks of structured data that can be passed-by-reference. PacLang's linear type system restricts the ways in which these references can be manipulated, with the aim of enabling a number of optimisations, including PipeIntro.

Before considering PacLang's linear type system, let us first consider what would happen to the PipeIntro transformation if we permitted the *unrestricted* use of packet references. Figure 6 gives an example of how unrestricted references can lead to an unsound application of PipeIntro. For the sake of simplicity let us consider the case where `q` contains a single packet reference. In this case task `t` uses `q` to simulate a global packet variable: the first line of `t` reads a packet reference from `q` and then immediately writes it back again. Executing the program results in the packet's first word being repeatedly incremented and written to `q1`. As a result, a series of consecutive integers appears on `q1`.

[6] Recall that Core PacLang already supports global queues.

```
queue<packet*> q;
queue<int> q1;

task t {
   packet* p = q.deq();
   q.enq(p);
   p[0]++;
   q1.enq(p[0]);
}
```

⟶

```
queue<packet*> q, Q;
queue<int> q1;
task t1 {
   packet* p = q.deq();
   q.enq(p);
   p[0]++;
   Q.enq(p);
}
task t2 {
   packet* p = Q.deq();
   q1.enq(p[0]);
}
```

Fig. 6. This code, written in a PacLang-like language without a linear type system, shows that the unrestricted use of references can break the PipeIntro transformation

In the transformed program, task `t1` can loop round many times before `t2` has read anything from `Q` (the queue introduced by the PipeIntro transformation). As a result, by the time `t2` gets round to dereferencing its local copy of the packet pointer, the packet's first word may have been incremented several times. This allows traces to appear on `q1` that were not possible in the source program (e.g. $\langle 0, 5, 10 \rangle$). The problem is that `t1` and `t2` access shared state via their packet references. The PipeIntro safety analysis, in the form presented in Section 4, is not able to detect this sharing since it does not model aliasing. One solution would be to perform full alias analysis as a precursor to the PipeIntro safety analysis, using this approximate aliasing information to detect potential accesses to shared state. Fortunately this is unnecessary as PacLang's linear type system [4] prevents aliasing and so would disallow the source program in Figure 6. Thus, the PipeIntro safety analysis does not need to be modified at all.

PacLang's linear type system does not exist merely to make PipeIntro easier. It has a number of notable features that simplify the compilation of high-level programs for Network Processors, while naturally capturing the style in which many packet processing programs are already written [4].

5.2 Extending PipeIntro Safety Analysis to Full PacLang

Global Variables: The algorithm for determining safety of PipeIntro (Section 4.1) can be extended to deal with global variables by extending the graph, G, as follows. For each pair of tasks, u and v, that share a global variable, g, place directed edges from both u to v and v to u if: (i) both u and v write to g; or (ii) one of u and v writes to g and the other reads from g.

We note that global variables could be translated into operations on shared queues. However, if we do this then the safety analysis as presented in Section 4 would deduce that the order of two reads from the same global variable may be observed. Dealing with global variables directly leads to a more accurate analysis.

Bounded Queues: Consider adding language primitive *HowFull*(q) that returns the number of elements currently on queue q. We can extend the algorithm for determining safety of PipeIntro (Section 4.1) by extending the graph, G, as follows. For every task, u, that does *HowFull*(q) add edges (u, v) and (v, u) for any task, v, that reads or writes q.

The intuition is that if a task tests the fullness of a queue, q, then it may be able to determine the order of its *HowFull* operation w.r.t. reads and writes to q.

6 Case Study

We have written a simple IPv4 unicast packet forwarding engine that employs a longest-prefix-match route-lookup algorithm in 500 lines of architecturally neutral PacLang. In this section we illustrate how our tools allow the program to be transformed into a form capable of achieving 3Gb/s (line rate) on an Intel IXP2400 Network Processor.

The details of IP packet forwarding are not described here (for more technical information the interested reader is referred directly to the IETF standards [18] and our PacLang code [11]). The purpose of this case study is to show that our transformations can be applied to realistic, non-trivial programs.

Figure 7(i) shows the initial structure of the PacLang IP forwarder. The program has five tasks, represented by white circles: Classify (`C`), IP options (`O`), ARP (`A`), IP Route Lookup (`I`) and ICMP Error (`E`). Queues are represented by filled circles. The receive (`r`) and transmit (`t`) queues are sources and sinks of network packets respectively.

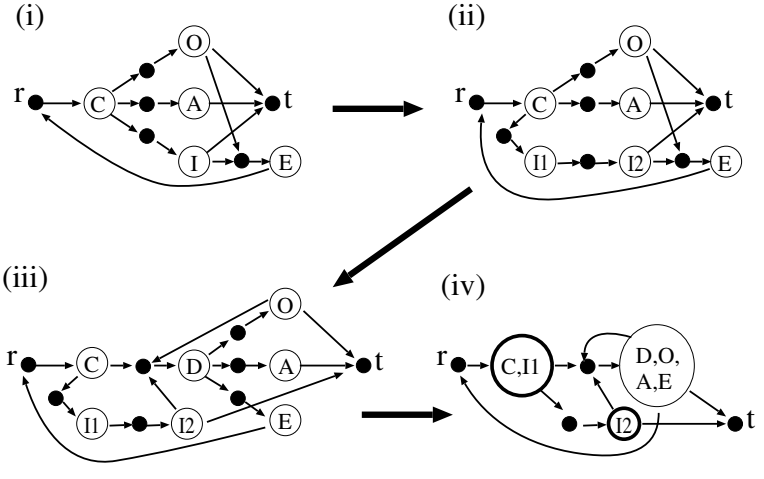

Fig. 7. Transforming the IPv4 unicast packet forwarder for IXP implementation. White circles represent tasks, filled circles represent queues

Our AMS for Intel IXP-series NPs applies the transformations shown graphically in Figure 7. First, PipeIntro splits `I` into `I1` and `I2`. Next, QueueMux is applied to the input queues of `O`, `A`, and `E`, creating a new Demux (`D`) task. Finally, PipeElim merges `D`, `O`, `A`, and `E` together, and merges `C` and `I1` together.

Our safety analysis deems that the PipeIntro transformation is applicable since, in the graph, G, constructed by the algorithm of Section 4.1, there is no edge from `I1` to any other task. Thus, there is no path from `I1` to `I2`—the two tasks created by the PipeIntro transformation.

The final structure of the transformed program (Figure 7(iv)) is well suited for IXP implementation. The tasks on the packet forwarder's critical path (the path taken by the vast majority of incoming packets) are highlighted with thick-lined circles. Timing analysis and simulation shows that, for our IXP2400, a 2-stage pipelined version of the critical path is sufficient to achieve 3Gb/s packet throughput (full line-rate on our 3-port Gigabit Ethernet board), for worst-case, min-size packets. If greater throughput was required (e.g. if we wanted line-rate for more than 3 ports) then we could apply PipeIntro again to increase the pipeline depth. Our AMS maps the two tasks on the critical path to separate micro-engines (small RISC processor cores on the IXP chip), the remaining task to the IXP's XScale processor core, and the queues to hardware scratch queues.

Both the source code, and the AMS that transforms it are available for download [11].

7 Related Work

Transformation-based approaches to program development have been around for a long time [3, 5] and applied to a variety of problems including circuit design [17] and hardware/software co-design [2]. The contribution of our research is to show that program-transformation is an appealing technique for bridging the gap between a high-level packet processing program and its low-level realisation on a multi-core network processor.

Software Pipelining [13, 8] is a transformation that superficially sounds similar to our *PipeIntro*, but is actually quite different. Software Pipelining reorders instructions in a loop so that instructions for future iterations may take place during the current iteration. This allows loads to be hoisted and allows better use to be made of multiple execution units on VLIW and superscalar architectures. Unlike our work, the aim is not to split a task over several processing elements, but to make better use of a single processor.

Our work has more in common with Hardware Pipelining [16, 19]: the division of a circuit specification into concurrent pipeline stages such that each stage is of roughly uniform size. However, unlike our work, the successive stages run in lock-step with no queueing between them—a model which is inappropriate for packet processing systems.

Previous work on automatic pipelining typically focuses on transforming a complete sequential program into a single pipeline. In contrast, our PipeIntro transformation and associated safety analysis extends this work, addressing the

more general problem of determining when it is safe to pipeline a *particular* concurrent task within the wider context of a whole concurrent program.

Task Assignment [15, 9] addresses the problem of assigning tasks to processors, taking into account the sizes of the tasks and the communication between them. While this work is similar to ours in that it explores the way in which a program can be mapped to several processors, there is no attempt to pipeline one task between several processors.

A number of other languages for multi-core processors have been developed [7, 12, 21, 14], but these are all significantly lower level and do not allow the task structure of a program to be changed.

8 Conclusions and Future Work

We have (*i*) presented a transformation-based methodology for programming Network Processors that allows architectural details to be separated from high-level program specification; and (*ii*) validated this methodology by showing how it can be applied to a realistic packet processing application.

We have also presented a whole-program analysis that determines when it is safe to pipeline a PacLang task. This extends previous work on automatic pipelining by addressing the more general problem of determining when it is safe to pipeline a *particular* concurrent task within the wider context of a whole concurrent program.

We hope that the ideas presented in this paper can be applied to the automatic partitioning of high-level code across multi-core architectures more generally (i.e. not just Network Processors). Since industrial trends suggest that such architectures will become more prevalent (as silicon densities continue to increase) we believe that this is an important topic for future research.

Acknowledgements

This research was supported by (UK) EPSRC grant GR/S68941: "High-Level Languages for Network Processors".

References

1. Allen, J. R., Bass, B. M., Basso, C., Boivie, R. H., Calvignac, J. L., Davis, G. T., Frelechoux, L., Heddes, M., Herkesdorf, A., Kind, A., Logan, J. F., Peyravian, M., Sabhikhi, M. A. R. R. K., Siegel, M. S., and Waldvogel, M. PowerNP network processor: Hardware, software and applications. *IBM Journal of research and development 47*, 2–3 (3003), 177–194.
2. Barros, E., and Sampaio, A. Towards provably correct hardware/software partitioning using occam. In *Proceedings of the 3rd international workshop on Hardware/software co-design* (1994), IEEE Computer Society Press, pp. 210–217.
3. Burstall, R., and Darlington, J. A transformation system for developing recursive programs. In *JACM 24(1)* (1977).

4. Ennals, R., Sharp, R., and Mycroft, A. Linear types for packet processing. In *Proceedings of the European Symposium on Programming (ESOP) 2004* (2004).
5. Feather, M. A system for assisting program transformation. *ACM Transactions on Programming Languages and Systems 4*, 1 (January 1982), 1–20.
6. Freescale. *C-5 Network Processor Architecture Guide*, 2001.
7. George, L., and Blume, M. Taming the IXP network processor. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation* (2003), pp. 26–37.
8. Hwang, C.-T., Hsu, Y.-C., and Lin, Y.-L. Scheduling for functional pipelining and loop winding. In *Proceedings of the 28th conference on ACM/IEEE design automation* (1991), ACM Press, pp. 764–769.
9. Ikinci, M. Multilevel heuristics for task assignment in distributed systems. Master's thesis, Bilkent University, Turkey, 1998.
10. Intel Corporation. Intel IXP2400 Network Processor: Flexible, high-performance solution for access and edge applications. Available from: `http://www.intel.com/design/network/papers/ixp2400.htm`.
11. Intel Corporation. PacLang. `http://sourceforge.net/projects/paclang/`.
12. Intel Corporation. *Microengine C Language Support Reference Manual*, 2003.
13. Lam, M. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation* (1988), pp. 318–328.
14. Lam, M. Compiler optimizations for asynchronous systolic array programs. In *Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages* (1998).
15. Lo, V. Heuristic algorithms for task assignment in distributed systems. *IEEE Transactions on Computers* (1988), 1384–1397.
16. Marinescu, M.-C. V., and Rinard, M. High-level automatic pipelining for sequential circuits. In *Proceedings of the 14th international symposium on Systems Synthesis* (2001), ACM Press, pp. 215–220.
17. Mycroft, A., and Sharp, R. A statically allocated parallel functional language. In *Proceedings of the International Conference on Automata, Languages and Programming* (2000), vol. 1853 of *LNCS*, Springer-Verlag.
18. Network Working Group. RFC1812: Requirements for IP version 4 routers.
19. Papaefthymiou, M. C. On retiming synchronous circuitry and mixed integer optimization. Master's thesis, Massachusetts Institute of Technology, 1990.
20. Radisys. ENP-2611 network processor board. `http://www.radisys.com`.
21. Teja. Teja NP: The first software platform for multiprocessor system-on-chip architectures. `http://www.teja.com`.
22. Winskel, G. *The formal semantics of programming languages: an introduction.* Foundations of computing. MIT Press, 1993.
23. Yavatkar, R., and H. Vin (eds.). *IEEE Network Magazine. Special issue on Network Processors: Architecture, Tools, and Applications 17*, 4 (July 2003).