

Worst-Case Execution Time Estimation for Hardware-assisted Multithreaded Processors

Patrick Crowley & Jean-Loup Baer
Department of Computer Science & Engineering
University of Washington
Seattle, WA 98195-2350
{pcrowley, baer}@cs.washington.edu

Abstract

This paper introduces a method for bounding the worst-case performance of programs running on multi-threaded processors, such as the embedded cores found within network processors (NPs). Worst-case bounds can be useful in determining whether a given software implementation will provide stable (e.g., line rate) performance under all traffic conditions. Our method extends an approach from the real-time systems literature, namely the implicit path enumeration technique [7], and casts the problem of bounding worst-case performance as an integer linear programming problem. To evaluate the method, we model the Intel IXP 1200 microengines and compare worst-case estimates to a selection of results gathered via simulation of that NP architecture.

1 Introduction

Network processing systems are real-time systems: if packet processing times exceed inter-arrival times, system instability will result (e.g., due to input buffer overflows). Consequently, network processing systems must process packets at network line rates. When a packet processing function is implemented in hardware, the packet processing rate, or throughput, is typically fixed and evident in the circuit timings. However, when such a function is implemented in software on a programmable processor, such as a network processor, the packet processing rate becomes variable, e.g., because of differing packet lengths.

To provide some estimate of throughput in the presence of this variability, some model, or trace, of expected traffic is used to measure packet processing rates. Examples of such traffic include minimum size packets or traces gathered in networks where deployment of the system is anticipated. This provides the designer with an expected level of performance, referred to here as an expected packet processing rate.

In this paper, we propose a complementary approach which bounds packet processing rates in a traffic-independent fashion. Our approach determines the peak performance attainable under worst-case circumstances, thus setting a bound on the processing rates for which we can insure that the system will deliver packets at the advertised rate. There are two additional reasons to follow such an approach: 1) traffic can vary widely and expected performance might not be characteristic, 2) the relationship between program performance and traffic characteristics is poorly understood and it is thus, in general, difficult to know how changes in traffic will effect performance. We see great value in being able to accurately describe both the expected and worst-case performance of software-based network processing systems.

Our interest is in determining the worst-case packet processing rate for a network processing system built around a programmable network processor (NP). This goal involves certain challenges: 1) programs can have complex control flow, 2) the packet handling process is generally structured as a software pipeline,

and 3) NPs typically employ multiple multithreaded processors to implement functionality. The method described in this paper deals with these challenges as follows.

The basic method used here to deal with the complexity of programs employs an implicit path enumeration technique (IPET) [7] to find a bound on the worst-case execution time (WCET) of a program. IPET reduces the problem of finding the WCET to that of finding the most expensive path through a network flow graph, a problem which is readily solved with integer linear programming methods.

If the packet handling process is structured as a pipeline of programs, and if parallel resources (such as multiple processors) can be used to exploit pipeline parallelism between packets, then we can determine the worst-case throughput of the pipeline (e.g., the worst-case packet processing rate) by finding the pipeline stage with the greatest WCET. This slowest pipe stage determines how quickly packets may safely enter the pipeline; the inverse of this WCET is the worst-case packet processing rate.

Modern NPs feature multiple embedded processors and thus facilitate this parallelization of pipeline stages. Furthermore, many NPs, notably Intel's IXP and IBM's PowerNP, include hardware-assisted multithreading in each embedded processor. This multithreading provides low- or zero-cost context switches so that as one thread begins to stall waiting on some long latency operation, such as a DRAM read, another thread can be swapped in to use the otherwise idle processor resources. Multithreading increases resource utilization and, therefore, can increase system throughput with the potential danger of increasing the latency, i.e., worst-case processing time, of an instance of a packet processing function. The method proposed in this paper takes this flavor of multithreading into account when determining the worst-case packet processing rate.

The general theme of this paper stems from the realization that it is important to describe the worst-case performance for software-based implementations of network processing systems. To this effect, we introduce and evaluate a method for bounding worst-case performance on a programmable, multithreaded network processor. The evaluation in this paper targets the Intel IXP1200 NP; worst-case processing rates for a number of programs are compared to some corresponding simulated processing rates.

The remainder of the paper is structured as follows. Section 2 first provides greater background on the structure and implementation of functionality on multithreaded NPs and then sketches the method used to bound system performance in such systems. Section 3 introduces the base method of determining a program's worst-case throughput by finding its WCET. Section 4 introduces the multithreading extension to the base method by considering the case of two threads; section 5 discusses the case of four threads. Limitations and future work are discussed in section 6. The paper concludes with a summary and discussion of contributions in section 7.

2 Background & Motivation

Both the sequential (e.g., pipelined) nature of packet handling and the parallel, multithreaded organization of modern NPs are dominant factors in determining a system's worst-case packet processing rate. In this section, we will motivate both factors and then discuss the flavor of multithreading considered in this paper.

2.1 Handling Packets with Software Pipelines on Multithreaded Chip-multiprocessors

The steps involved in handling a packet are inherently sequential. When a packet arrives, it must first be moved from the external transmission medium into memory. Once in memory, packet processing begins and typically proceeds according to the layers in a protocol stack. When a TCP/IP packet is received over Ethernet, for instance, the Ethernet headers are processed first, followed by the IP Headers. Such processing can proceed up the networking stack, including application specific processing in the data payload (e.g., such as HTTP URL parsing). Once all processing has completed, including classification, modification, next hop lookups and any other application specific processing, the packet must be transmitted out of memory and onto the appropriate output link. These steps in the packet handling process can be described as stages

in a pipeline. This pipelined nature can be exploited for increased performance: if stages can execute in parallel (say on different embedded processors) and if the amount of computation in each stage is balanced, then throughput can be increased by exploiting pipeline parallelism.

Since modern NPs feature multiple processors, pipeline parallelism can be exploited. Such an arrangement suggests that a packet will be processed by several processors (one for each pipe stage). This is no requirement, however. Another approach would be to simply implement the full packet handling process on each processor and let them proceed in parallel without any pipelining. This certainly has scalability benefits over the pipelining approach, since the parallelism available with a pipeline is limited to the number of pipe stages. However, it also has greater costs. One practical consideration is the increased amount of control store needed by each processor; modern NPs typically have small control stores available at each embedded processor that cannot hold the programs necessary to handle all packet processing functions. Thus, NPs tend to favor pipelined or pipelined/parallel hybrid implementations over fully parallel ones.

Regardless of whether a given processor implements the full pipeline or only one stage, we can view the software it executes as a program that processes packets. The rate at which this program processes packets will influence the worst-case processing rate of the whole system; it will either determine it completely if the program implements the whole packet processing pipeline, or it will contribute to it as a single stage.

Modern NPs are multithreaded to help hide the long latencies required to access memory and other external resources. Thus, throughput can be improved by running multiple copies of the packet processing program in different threads on the same processor. Again, this applies regardless of how the pipeline is mapped onto processors.

In this paper, we will consider replication of identical threads, i.e., replication of pipeline stages, but the method is applicable to the other arrangement as well.

2.2 Non-preemptive, Hardware-assisted Multithreading

The flavor of multithreading we consider in this paper can be described as non-preemptive and hardware-assisted. The term non-preemptive refers to how threads are scheduled: it is up to each thread to yield control to the scheduler. This is in contrast to a preemptive scheduler that can interrupt the execution of one thread to switch control to another. The term hardware-assisted refers to both the hardware scheduler and the resources provided to each thread. When multithreading is hardware-assisted, each thread is allowed to save its architectural state (control and data registers, PC, etc.) in private hardware. This is opposed to a thread having to save its state to memory prior to a context switch.

This paper presumes a threading model identical to that found in the Intel IXP1200. Each processor has support for up to 4 threads of execution, and context switches require a single cycle (which can be hidden via delayed execution as is common with branch instructions). On the IXP1200, all long latency operations, such as memory operations, accept an optional parameter requesting a context switch. Since the arbiter is non-preemptive, the programmer, or compiler, must include these optional parameters often to keep any one thread from monopolizing the processor.

3 Processing Throughput of a Single Thread of Execution

Our approach for determining the worst-case packet processing rate of a program begins with determining the program's worst-case execution time (WCET). In this paper, we use the implicit path enumeration technique (IPET) from [5, 6, 7]; this is one, and in our view the most promising, of a number of recent results in the area of determining WCET on modern processors [3, 8, 9, 10, 2]. For a program running on a single threaded processor, the worst-case processing rate is the inverse of the program's WCET.

3.1 WCET Analysis via Implicit Path Enumeration

The WCET of a program depends on two factors: the path of instructions to be executed, and the microarchitecture of the processor. In IPET, the possible, or feasible, execution paths through a program

are described implicitly with a set of linear, network flow constraints extracted from the program’s control flow graph; these constraints describe how often each basic block in the program can be executed.

To determine the execution cost, in cycles, of each basic block, the microarchitecture of the processor must be considered. The work in [7] describes how to account for pipelined execution units as well as instruction and data caches. Since the NPs we consider in this paper have no caches, we do not consider cache-related effects; we return to this topic when discussing future work in Section 6. When pipeline stalls might occur or when delayed branch operations (present in the NP engines that we model) are used, we assume the worst cases; these assumptions introduce some pessimism into the WCET estimate.

Once the execution costs for all basic blocks are known, they can be combined with the flow constraints to form an integer linear program. To find the WCET, the objective function, which sums the product of basic block frequencies and costs, is maximized subject to the constraints via integer linear programming.

In order for this optimization problem to be decidable, we must place certain restrictions on how programs are written. This set of sufficient restrictions have been described by [4, 12]. The restrictions are: 1) no recursion, 2) no dynamic data structures, and 3) no unbounded loops. These are not overly restrictive conditions, as we are concerned with applications for NPs which, for precisely the same reasons, do not generally have these restricted features.

We now show how to construct the linear program from a program’s control flow graph. We illustrate this construction with a small program; For a more complete discussion, see [7].

3.2 Constructing the Linear Program

Suppose we have N basic blocks, B_i , in our CFG. Let the variable x_i represent the execution frequency of block B_i and let c_i represent its execution time, or cost, in processor cycles. The program’s total execution time can be expressed as:

$$\text{Total Execution Time} = \sum_i^N c_i x_i. \tag{1}$$

The purpose of the linear program is to find the set of feasible x_i values that maximizes the objective function (Equation 1). The x_i variables are constrained by the feasible flow in the control flow graph. Figure 1 depicts a program fragment and its associated CFG. This code checks an array of integers, of size `datasize`, for any negative numbers; if a negative number is found, the function immediately returns a 0, otherwise it examines all entries and returns 1. Each basic block has an execution frequency variable as well as some number of in and out edges, denoted d_i , which describe the flow between basic blocks.

There are two types of constraints in the graph in Figure 1: structural and functional. The *structural* constraints require that the flow into a node equals the flow out. In general, we have

$$x_i = \sum d_l = \sum d_m, \tag{2}$$

where the d_l are the incoming edges and the d_m are the outgoing edges.

The following equations describe the structural constraints that can be automatically extracted from the CFG shown in Figure 1B.

$$d_1 = 1 \tag{3}$$

$$x_1 = d_1 + d_6 + d_8 + d_9 = d_2 + d_3 \tag{4}$$

$$x_2 = d_2 = d_4 + d_5 \tag{5}$$

$$x_3 = d_4 = d_6 \tag{6}$$

$$x_4 = d_5 = d_7 + d_8 \tag{7}$$

$$x_5 = d_7 = d_9 \tag{8}$$

$$x_6 = d_3 = d_{10} + d_{11} \tag{9}$$

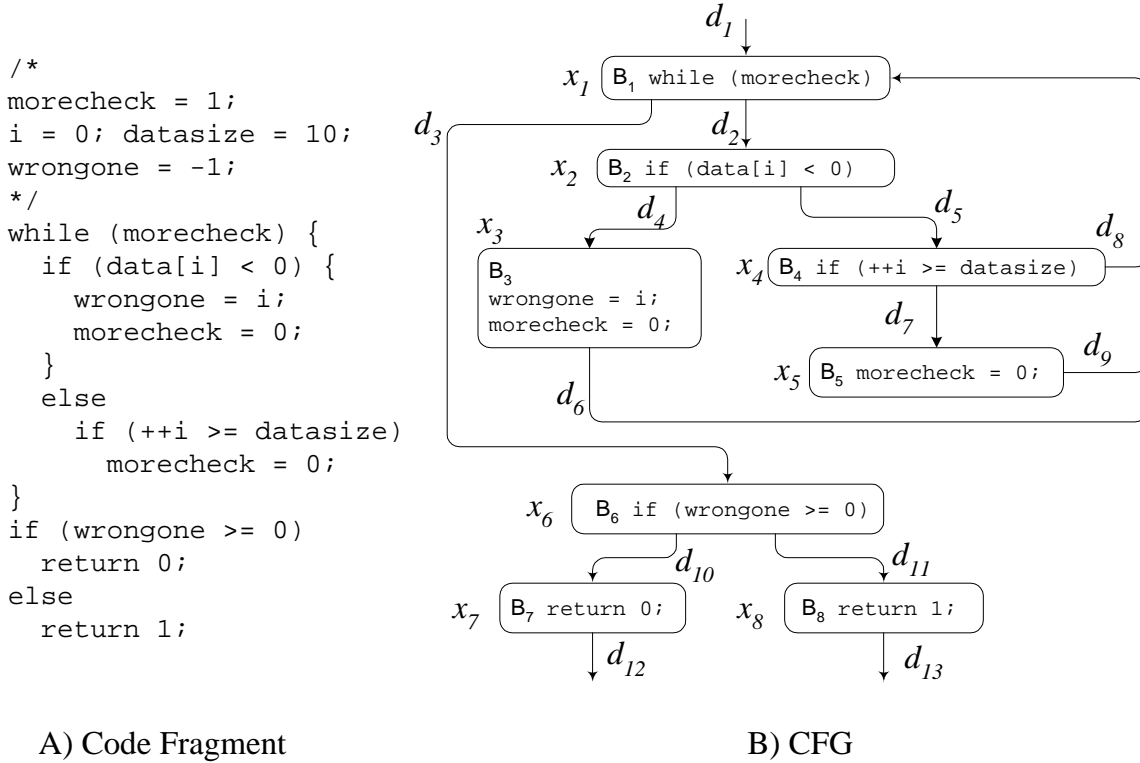


Figure 1. A sample code fragment and corresponding control flow graph.

$$x_7 = d_{10} = d_{11} \quad (10)$$

$$x_8 = d_{11} = d_{13} \quad (11)$$

The *functional* constraints constrain flow through the graph according to the program semantics. Loop bounds are the principal example and are indeed the only required functional constraints. In the code fragment, we see that the while loop will be taken between 1 and 10 times. The following equation expresses this constraint.

$$10d_1 \geq x_1 \geq 1d_1 \quad (12)$$

Other information may also be present in the program's semantics to obtain a tighter WCET estimate. For instance, the else block, B_3 can be executed at most once each time we enter the code fragment; we can add the following constraint to express this condition.

$$x_3 \leq 1d_1 \quad (13)$$

Once the objective function, structural and functional constraints have been constructed, the program may be solved with an LP solver. Many such solvers exist; in this paper, we use the publicly available program `lp_solve` (ftp://ftp.es.ele.tue.nl/pub/lp_solve). As these optimization problems are network flow problems, solutions are found very quickly; problems with hundreds of basic blocks are solved in a few seconds on a modern PC.

There are other complicating factors, not described here, such as the proper handling of procedure calls. For a more complete treatment of the IPET method, see [7].

3.3 Benchmarks and Single-Threaded Results

In this section, we: describe our simulation and analysis methodology, present WCET estimates for a selection of programs and compare those estimates to simulated execution times gathered on the Intel IXP1200 cycle-accurate simulator. These programs, along with descriptions and code size when compiled for the IXP1200 microengines, are listed in Table 1.

Program	Description	IXP μ -words	Basic Blocks
reverse_array	Reverses an array of N elements	60/54	14/13
chk_data	Checks array for negative values [11]	70/61	16/17
sort	Bubblesort of worst-case input (-1,-2,...)	103/70	23/23

Table 1. Sample programs used in this paper. Sizes and basic block counts are reported for unoptimized and size-optimized compilation, respectively

These programs are all written in C; we use the Intel IXA microengine C compiler, version 2.01a, to generate assembly code. The same assembly code that gets assembled and executed in the simulator is used as input to our WCET estimation tool. This tool constructs the control flow graph from the assembly code, determines the cost of each basic block, formulates the linear program (user must provide loop bounds and other functional constraints), and calls `lp_solve` to find the WCET estimate.

The microengine C compiler supports a number of optimization levels. We present results for programs compiled without optimization and for programs optimized for size. There is also a speed optimization, but some of our programs were too large to apply this optimization correctly, hence, we do not report those numbers here.

In the simulations, each program has been given input as close to algorithmic worst-case input whenever possible. For instance, the program `chk_data` has been given an input with no negative numbers, so the entire array gets traversed.

The results can be seen in Table 2. The WCET estimate is generally within a factor of 2 of the simulated execution time. Smaller programs, with fewer loop iterations, are quite close; longer running programs with more complex control flow yield more pessimistic estimates. For the case of `sort`, it is difficult to closely model the algorithmic worst-case with only two nested loop bounds; thus, the sort estimate is overly pessimistic. The algorithmic worst case for an initially sorted array in reverse order has $\frac{N^2}{2}$ comparisons while the two nested loop model results in a pessimistic n^2 comparisons. Hence, the factor of 2 increase in the estimate. Note that the constraints could be tailored for a better fit; in general, greater care in the construction of functional constraints can increase the tightness of the bound.

Program	Unoptimized		Optimized for Size	
	Simulated WCET	Estimated WCET	Simulated WCET	Estimated WCET
reverse_array	718	733	659	699
chk_data	795	807	530	568
sort	211339	423125	134794	274002

Table 2. Simulated and Estimated WCETs for unoptimized and size-optimized programs.

4 Processing Throughput of Two Threads

The method described in the previous section applies to a single-threaded program. As noted previously, NPs typically provide hardware support in each embedded processor for multiple threads of execution. In this section, we introduce our method for finding the worst-case packet processing rate of a program running two threads on a multithreaded processor.

Our method extends the IPET method for estimating WCET to accommodate multiple threads. The principle goal of our extension is to support control flow and latency hiding between threads; these notions are not present in the original IPET method. To handle inter-thread control flow, we introduce yield nodes and edges; to handle latency hiding, we keep an accounting of the portion of a thread's memory latencies that is hidden via execution in another thread.

When considering the performance of a multithreaded program, two natural metrics arise: 1) the worst-case execution time of any one thread, and 2) the worst-case throughput of the system. In this section, after describing yield nodes and their construction and how concurrency is modeled, we show how the linear program may be formulated to model either of these metrics.

4.1 Yield Nodes & Edges

Basic blocks consist of sequences of instructions with single entry and exit points. When memory operations invoke a context switch, they, in essence, fragment their enclosing basic blocks; the context switch is an exit point. Figure 2 depicts a control flow graph before and after fragmentation. This fragmentation step introduces *yield nodes* into the CFG which are basic blocks containing only the yielding operation. Furthermore, each yield node has some number of outgoing *yield edges* which represent the transfer of control from this thread to another; control returns to this thread, and specifically to the yielding operation's subsequent basic block, by returning along a yield edge from another thread. In the single threaded case there was no other thread to yield to, so control proceeded directly to the subsequent node; in the multithreaded case, this flow gets routed through one or more other threads.

A yield edge begins at a yield node within one thread and, generally, ends at a basic block that follows a yield node in some other thread. A yield edge connecting two nodes within different threads has a precise meaning; namely, that once the source thread executes the yield operation, it is possible that the target node will be the next basic block to execute. To construct all yield edges for a given yield node, then, we must determine which nodes might next execute. This involves two questions, which we consider in order: which threads might next be scheduled, and which nodes within those threads might next be executed.

Yield Edge Placement and Thread Scheduling Policies

Yield edges are placed between threads according to the scheduling policy employed by the thread scheduler. That is, a yield edge is placed between threads A and B , if and only if the thread scheduling policy would allow thread B to be scheduled immediately after thread A yields control. In this paper, we only consider a strict round-robin scheduling policy; here, strict round-robin means that the order in which threads get scheduled follows a fixed round-robin pattern. We discuss other possibilities such as the non-strict round-robin policy of the IXP1200 as future work in Section 6.

Note that our assumption of strict round-robin is a good first approximation. In particular, if all threads yield only when accessing the same external resource, then the completion of those external resources will typically be serialized and, thus, so will the thread scheduling order. In other words, if we restrict our benchmark programs to use only a single external resource, such as one of either DRAM or SRAM, then the microengine scheduler will produce something similar to a strict round-robin scheduling pattern. However, it will not necessarily be a precisely strict round robin pattern. In any case, we assume that any difference due to a non-strict schedule will only improve performance. Thus, our worst-case bound will still be valid.

A strict round-robin scheduling policy means that each thread will only yield to one other. Thus, yield edges need only be constructed between each thread and the one that follows it in the strict round-robin

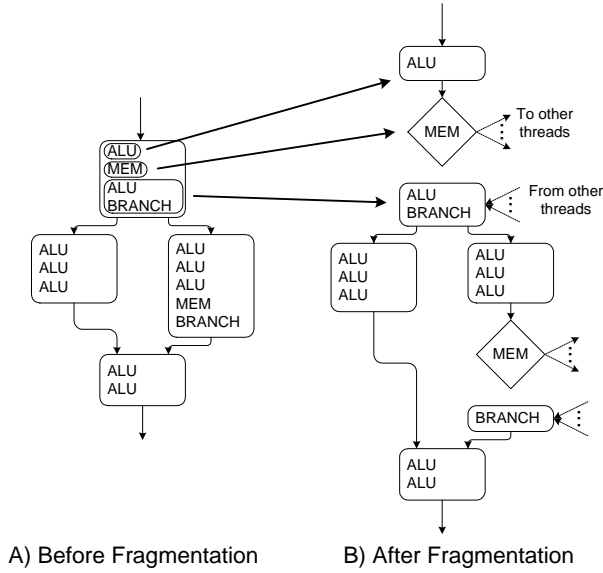


Figure 2. CFG Fragmentation due to yield operations.

schedule. For example, if we have N threads, and the round-robin schedule is $1, 2, \dots, N$, yield edges must be constructed between thread i and thread $(i + 1) \bmod N$.

Yield Edge Placement and Target Nodes

Once we've established the yield edge destination thread, we must find the target nodes within that thread. As previously mentioned, yield edges begin at yield nodes and end within another thread, usually at nodes following yield nodes. Each thread's entry node also receives yield edges. This determines the major source of our target nodes: we must add a yield edge from the source yield node to each node in the target thread that follows a yield block as well as the target thread's entry node. An example for the case of two threads is shown in Figure 3.

Although not pictured in Figure 3, a thread's exit node is also treated as a yield node. This is in order to permit threads to restart if the structural and functional constraints allow it to do so. This models the completion of the thread (and the satisfaction of its structural and functional constraints) and allows more flow to enter at the start node. We return to this subject in Section 4.3.

Linking Yield Edges to Basic Block Frequencies

The fragmentation due to yield node edges has an important effect upon the structural constraints of the graph. Specifically, there are no d edges linking yield nodes to their subsequent basic blocks. To repair intra-thread constraints, in addition to the expected flow conservation constraints linking node execution frequency to flow on yield edges, denoted y_k , we require that the sum of yield flow out of a yield node equals the sum of yield flow into the subsequent basic block. In general, we augment the remaining structural constraints (from Equation 2) with

$$x_i = \sum y_k \quad (14)$$

$$x_j = \sum y_l \quad (15)$$

where: x_i is a yield node, the y_k s are outgoing yield edges from x_i , x_j is a node following a yield node, and the y_l s are incoming yield edges ending at x_j .

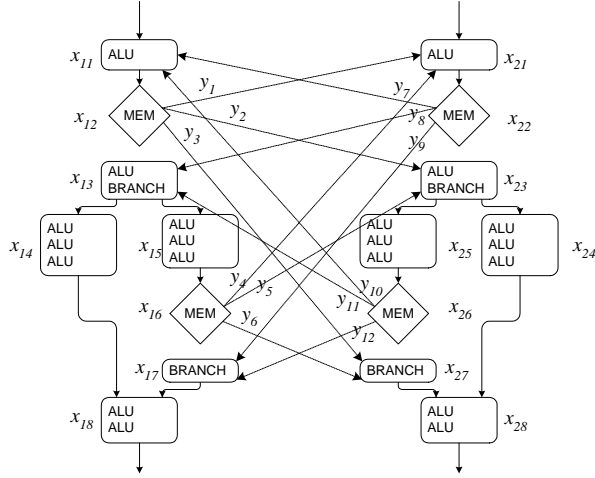


Figure 3. CFG Fragmentation due to yield operations. Exit nodes and their corresponding yield edges are not pictured.

For the example in Figure 3, we have the following yield flow constraints.

$$x_{11} = y_7 + y_{10} \quad (16)$$

$$x_{12} = y_1 + y_2 + y_3 \quad (17)$$

$$x_{13} = y_8 + y_{11} \quad (18)$$

$$y_1 + y_2 + y_3 = y_8 + y_{11} \quad (19)$$

$$x_{16} = y_4 + y_5 + y_6 \quad (20)$$

$$x_{17} = y_9 + y_{12} \quad (21)$$

$$y_4 + y_5 + y_6 = y_9 + y_{12} \quad (22)$$

$$x_{21} = y_1 + y_7 \quad (23)$$

$$x_{22} = y_7 + y_8 + y_9 \quad (24)$$

$$x_{23} = y_2 + y_5 \quad (25)$$

$$y_7 + y_8 + y_9 = y_2 + y_5 \quad (26)$$

$$x_{26} = y_{10} + y_{11} + y_{12} \quad (27)$$

$$x_{27} = y_3 + y_6 \quad (28)$$

$$y_{10} + y_{11} + y_{12} = y_3 + y_6 \quad (29)$$

In the single-threaded case, we began 'execution' by setting the entry node's incoming d edge to 1 in the set of structural constraints. Likewise, for multiple threads, we will set the first thread's incoming d edge to one, but we must arrange that the first node of each of the other threads be reachable via yield edges.

4.2 Modeling Concurrency

The purpose of fine-grained multithreading is to increase resource utilization by overlapping one thread's memory accesses, or other long latency external operations, with another thread's execution. In this method, yield edges describe the possible flow of control between threads; specifically, each yield edge connects a yield operation with basic blocks (non-yielding operations, i.e., computational blocks) that

might execute concurrently.

To model concurrency for our target system, we begin with the observation that total run time has two components: execution time and stall time. Since no computation occurs in parallel, execution time is simply the sum the execution times of all computation blocks in all threads; this can be computed by adding up the effective execution time of those blocks along the worst-case path. Stall time results when a long latency yield operation cannot be completely overlapped with execution in another thread. This is our key to modeling concurrency: we will account for the portion of each yield operation that, in the worst-case, forces a thread to stall. To this end, we will assign values to yield edges, and add those yield edges to the objective function, in order to calculate each yield operation’s contribution to stall time, and hence total run time. Only rarely will a yield operation’s full latency contribute to stall time. For example, if a yield operation with a 30 cycle latency only contributes 10 cycles of stall time along a given path, then the yield edge value would be set to -20 so that when the yield latency and yield edge value are summed in the objective function the true contribution to execution time, namely 10 stall cycles, will be represented.

To calculate the stall time contribution for a yield operation, we make the conservative assumption that the only cycles of overlap are due to the minimum number of execution cycles that can take place at the yield edge destination. The equation for determining the yield edge value is as follows. Note that the `min` function assures that no amount greater than the source yield latency will be hidden in the case of large execution sequences (this avoids negative stall times).

$$\text{Yield Edge Value} = -\min(\text{Minimum Destination Execution Cycles}, \text{Source Yield Latency}) \quad (30)$$

As we show, this approach in determining a yield node’s stall time contribution is conservative in two ways. We illustrate them beginning with simple example in Figure 4. It depicts two threads each with one yield operation. The first source of pessimism is present in Figure 4. Since we assume that only the destination execution cycles overlap with the source yield operation, we imply that no overlap occurs between yield operations. Since the IXP1200 supports pipelined SRAM requests, this is a safe but pessimistic assumption. This assumption results in 10 cycles of pessimism in this example. Furthermore, the pessimism seems unwarranted in this simple example. Indeed it is, in this particular case, as we clearly see that the first 20 cycles of thread 2’s yield latency are overlapped with thread 1’s yield operation. We see this example clearly because each of the two yield nodes has only one incoming edge. In general, however, as shown in Figure 2, each yield node will have many incoming edges, and it will not be clear how to choose a safe amount of overlap between yield operations.

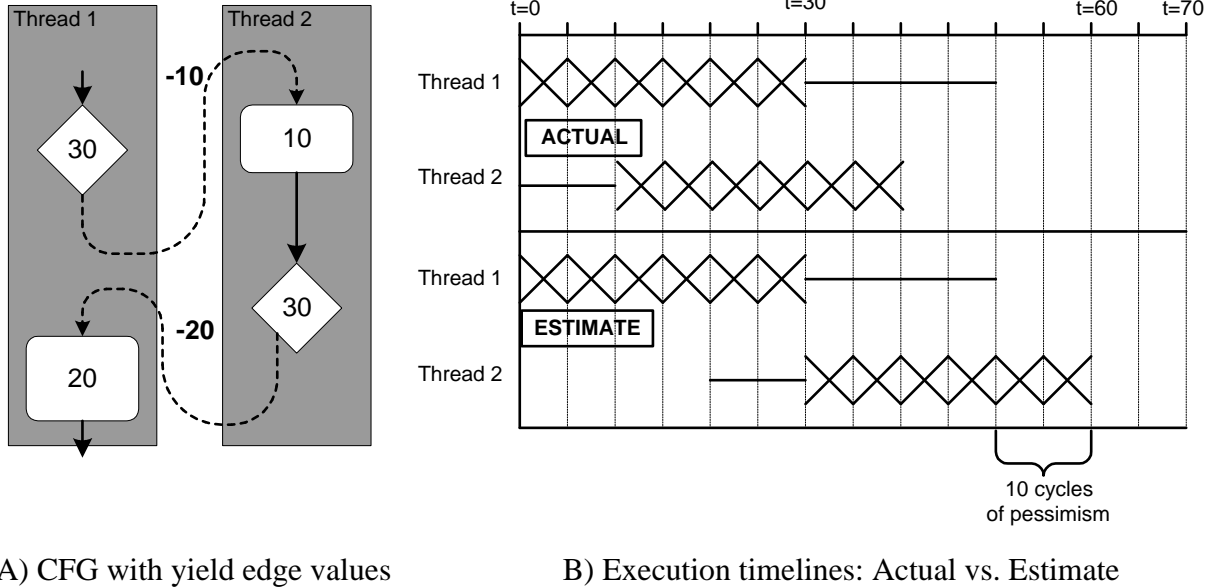
This approach for determining a yield node’s stall time contribution is conservative in a second way, as shown in Figure 5. There, we see the potential pessimism due to choosing the minimum number of execution cycles in Equation 30. In general, we must base our stall time contribution on the minimum number of execution cycles, rather than the maximum, because we only get to choose one value for each yield edge; this value must make a conservative contribution to execution time, taking into account all possible paths. In the case of Figure 5, the yield edge from thread 1 to thread 2 is followed by two possible paths, one with cost 10, the other with cost 20. Since, during the course of execution, flow may follow either path, we base our stall time on the conservative value of 10.

One might be tempted to choose to use the value of 20 since that is the costlier exit path from thread 2; after all, our method is meant to find the worst-case path. Note, however, that the structural and functional constraints might require that the shorter path be taken (e.g., at a loop exit). In this case, a value of 20 would over-estimate the amount of overlap and lead to an under-estimate of execution time.

These figure have depicted simple target execution paths; Figure 4 has only a single path, while Figure 5 has two. In general, however, the control flow could be arbitrarily complex, involving loops and branches. To find a safe value for each yield edge, we must first find the minimum cost exit path for each yield edge destination node. The procedure for determining this is a simple iterative breadth-first traversal in which cumulative cost is maintained for each path. At each step, we compare the cost of the newly extended path with the minimum cost completed path seen so far. Any paths with cost greater than the minimum

are discarded; any completed path with a lesser cost becomes the new minimum. The procedure completes when no paths remain with cost less than the current minimum.

When all yield edge credit values are found, they must be added to the objective function to accurately model the cost of flow through the CFG. The updated objective function is shown in Equation 31.



A) CFG with yield edge values

B) Execution timelines: Actual vs. Estimate

Figure 4. Modeling concurrency with yield edges. In the timeline, the X's denote memory operation and the horizontal lines denote execution.

$$\text{Total Execution Time} = \sum_i^N c_i x_i + \sum_j^M c_{y_j} y_j, \text{ where } c_{y_j} \text{ is the yield edge value for yield edge } y_j \quad (31)$$

4.3 Worst-case Latency vs. Worst-case Throughput

When a stage of our packet processing pipeline is implemented in software on a multithreaded processor, we will be interested in both the worst-case latency that a packet might experience and the worst-case throughput achievable by the pipeline stage. Note that these are indeed different metrics. The worst-case latency for a single packet might arise when the other threads in the system achieve very short latencies; so, despite one packet in the system experiencing worst-case latency, overall throughput might still be good. Likewise for worst-case throughput. An individual thread might not be experiencing its worst possible latency because doing so would shift the contention for resources in such a way as to *improve* the latency of some other thread by a greater amount. Our method distinguishes these two target metrics by constraining each thread's entry and exit frequencies in different ways.

To target the worst-case latency of a single thread (e.g., packet), we constrain the entry and exit node frequencies for the first thread to be one, as shown in Equation 32. Intuitively, this constraint allows the maximum possible amount of time to be spent in the second thread between the time execution enters and exits the first thread.

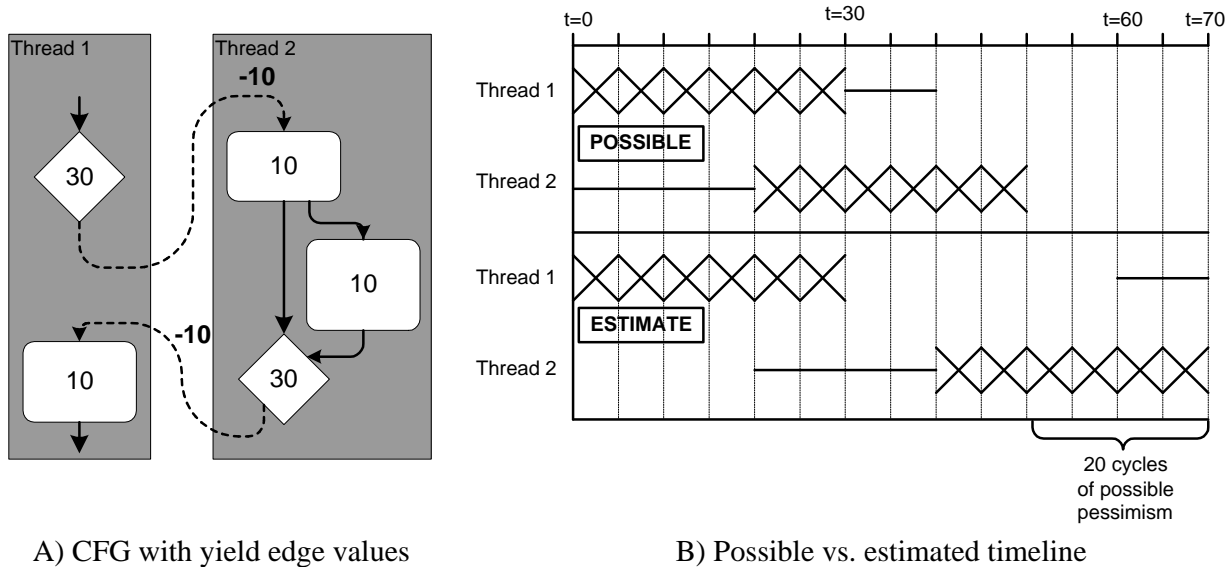


Figure 5. Determining yield edge debit value. In the timeline, the X's denote memory operation and the horizontal lines denote execution.

$$x_{\text{first thread entry}} = x_{\text{first thread exit}} = 1 \quad (32)$$

To target the worst-case throughput, we constrain 1) the entry nodes for all threads to be one or greater and 2) the exit nodes of all threads to be one. These constraints are shown in Equations 33 & 34. Intuitively, these constraints allow the maximum possible amount of time to be spent in all threads provided that each thread only completes once. A thread completion corresponds to a packet completion, so, in the case of two threads, this worst-case path through the combined CFG corresponds to the worst-case time needed to process two packets.

$$x_{\text{thread } i \text{ entry}} \geq 1 \quad (33)$$

$$x_{\text{thread } i \text{ exit}} = 1 \quad (34)$$

In our benchmarks, we have found no cases where targeting worst-case latency produces a significantly different result from target worst-case throughput. This is likely due to the memory-bound nature of our programs; with two threads, the compute resources are generally idle for half the total cycles. For this reason, along with the difficulties in comparing such an estimate to a measured result, we report only worst-case throughput results in this paper.

4.4 Benchmark Results with Two Threads

The worst-case execution time estimates for dual threaded versions of each of the programs, along with their simulation times, are shown in Table 3. With the exception of sort, all estimates are near the worst-case simulation time; the differences are, as expected, slightly larger than those seen in the single-threaded case because of the pessimistic assumptions explained previously. For sort, as was the case for a single thread, the estimate is greater than the simulated result by more than a factor of two. This is again due to the pessimistic mismatch between the loop constraints and the actual worst case achievable by bubblesort.

Program	Unoptimized				Optimized for Size			
	Simulated WCET	Estimated WCET	Basic Blocks	Yield Operations	Simulated WCET	Estimated WCET	Basic Blocks	Yield Operations
reverse_array	731	1162	23	7	675	1102	23	7
chk_data	822	1302	30	8	564	904	26	6
sort	210291	633786	33	7	133447	407700	31	6

Table 3. Simulated and Estimated WCETs for dual-threaded programs. The basic block and yield operation counts indicate the number of each per thread. These times indicate the number of cycles needed for both threads to complete.

It is also interesting to note that these dual threaded programs require approximately the same amount of time as their single-threaded counterparts (in the Simulated WCET column). This implies that the second thread, in each case, is taking advantage of idle resources without impacting the performance of the first thread. In these cases, multithreading has doubled throughput.

5 Processing Throughput of 4 Threads

The method for estimating the performance of dual-threaded programs described in Section 4 scales and applies directly to versions with 4 threads. The reason is that none of our assumptions change. Yield edges are drawn between neighboring threads according to the round-robin schedule (e.g., $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1$). Yield edge values, which account for the stall time contributions of their source yield nodes, are still based on estimates of the minimum number of execution cycles that will occur at the yield edge destination. None of this requires inspection of any threads but the ones joined via a yield edge. Had we taken a less conservative approach in Section 4, one that, say, tried to tighten the bound by accounting for the amount of overlap between subsequent yield operations, we would almost certainly require a change in strategy.

The simulation results and estimates for 4-threaded versions of the programs are shown in Table 4.

Program	Unoptimized				Optimized for Size			
	Simulated WCET	Estimated WCET	Basic Blocks	Yield Operations	Simulated WCET	Estimated WCET	Basic Blocks	Yield Operations
reverse_array	1083	2326	23	7	896	2204	23	7
chk_data	1201	2612	30	8	985	1808	26	6
sort	282973	1267570	33	7	192789	817980	31	6

Table 4. Simulated and Estimated WCETs for 4-threaded versions of the unoptimized and size-optimized programs. The basic block and yield operation counts indicate the number of each per thread. These times indicate the number of cycles needed for all four threads to complete.

The estimates are roughly within a factor of two of the simulation results. The program `sort` is once again the exception; its WCET estimate is approximately four times the simulated result, for reasons similar to those seen in the previous section.

6 Limitations & Future Work

The objective of this line of research is to provide tight and safe bounds on the performance of multi-threaded network processing programs. The method described in this paper represents a first attempt at bounding performance for a limited class of such programs under restricted circumstances. Each of these

limitations represent directions for future research. We now discuss these limitations and then consider additional opportunities for extending our approach.

6.1 Limitations

Multiple Yield Resources and Other Scheduling Policies. The method presented in this paper requires that all threads yield on the same external resource. So, for example, it is not possible for threads to yield on accesses to both SRAM and DRAM. We plan to extend the model to accommodate multiple yield resources in the near future.

Additionally, our approach pessimistically requires that threads execute according to a fixed round robin schedule. This is a fundamental limitation to our approach since this choice determines how yield edges are constructed between threads. In the near future, we plan to restate the problem as a generalized linear program, in which some coefficients are described with systems of equations, in the hopes of alleviating this limitation.

Asynchronous events. Certain network processing tasks are activated by external, asynchronous events. For example, in response to a packet arrival, a packet receive routine might allocate a buffer, perform record keeping and then move the packet into memory. The present method does not consider any events that are not synchronized with, or requested by, a thread.

Thread Interdependencies. The present method has assumed that threads have no interdependencies. This precludes, for example, implementing mutex regions within the threads to limit access to shared state. Such protections are critical for some important functions, and, as such, this is a high-priority area for future research.

6.2 Additional Future Work

Caches and Other Architectural Features. The established method for modeling caches is one benefit of using IPET to bound worst-case performance. In future work, we plan to incorporate the caching techniques into our extension. Caches are generally lacking in the NP embedded cores for two reasons: 1) conventional wisdom holds there is insufficient locality available, and 2) caches introduce non-determinism and complicate worst-case analyses. Providing sufficient worst-case bounds for a system with caches would help address both of these issues.

Formal Verification. We would like to establish the formal correctness and validity of our bounds. This will become more important as the method grows in complexity.

Restatement as a Generalized Linear Program. Tighter worst-case bounds could be achieved if some flexibility could be retained in the description of cost coefficients in the objective function. Having to choose a single constant to serve as a coefficient for yield edges costs, for instance, is more restrictive than strictly necessary. We feel the problem can be recast as a generalized linear program [1] in which certain coefficients can be described with systems of equations rather than constants.

7 Conclusions

In this paper, we have introduced the problem of estimating worst-case performance for network processing programs running on multithreaded processors. We have described an initial estimation method suitable for a limited class of programs executing under a restricted set of conditions. The method, which is based on the IPET approach [7] for worst-case program execution time estimation, casts the worst-case estimation problem as an integer linear program. To evaluate the method, worst-case processing rates for a number of programs running on Intel IXP1200-like microengines were compared to some corresponding simulated processing rates for that network processor. Results were presented for programs with 1, 2 and 4 threads. In the multiple thread cases, the estimates were found to be between 40% and 4 times greater

than the results gathered via simulation. The program `sort` is an outlier due to a pessimistic bound on the number of worst-case nested loop iterations.

In future work, we plan to address the current limitations and restrictions in order to broaden the class of programs supported by the method. Additionally, we plan to explore compatible extensions to the model, some of which have been described in the literature such as cache analysis, in order to investigate whether certain non-deterministic architectural features can be shown to provide sufficiently good worst-case performance to warrant inclusion in network processors.

References

- [1] George B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.
- [2] Jakob Engblom and Andreas Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proc. of 21st IEEE Real-Time Systems Symposium (RTSS'00)*, 2000.
- [3] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, pages 37–46, 1997.
- [4] Eugene Kligerman and Alexander D. Stoyenko. Real-time euclid: A language for reliable real-time systems. *IEEE Trans. on Software Eng.*, 12(9):941–949, September 1986.
- [5] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Workshop on Languages, Compilers, & Tools for Real-Time Systems*, pages 88–98, 1995.
- [6] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *IEEE Real-Time Systems Symposium*, pages 298–307, 1995.
- [7] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1996.
- [8] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong-Sang Kim. An accurate worst case timing analysis for RISC processors. *Software Engineering*, 21(7):593–604, 1995.
- [9] Sung-Soo Lim, Jung Hee Han, Jihong Kim, and Sang Lyul Min. A worst case timing analysis technique for multiple-issue machines. In *Proc. of IEEE Real-Time Systems Symposium (RTSS)*, pages 334–345, 1998.
- [10] Greger Ottosson and Mikael Sjödin. Worst-case execution time analysis for modern hardware architectures. In *ACM SIGPLAN 1997 Workshop on Languages, Compilers, and Tools for Real-Time Systems (LCT-RTS'97)*, 1997.
- [11] Chang Yun Park. *Predicting Deterministic Execution Times of Real-Time Programs*. PhD thesis, University of Washington, August 1992.
- [12] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, 1(2):160–176, September 1989.