A SPMD Programming Model for PRET Architectures

Peter Donovan

1 Acknowledgement

This work is inspired by technical problems that arose while working with Prof. Shoeberl and with members of Prof. Lee's research group. However, any foolishness presented here (in retrospect, there certainly is some) is that of the author.

2 Abstract

A methodology for programming multicore precision-timed (PRET) machines is described and evaluated on small examples. Discussion is included on the suitability of the hardware architecture for multicore applications, and changes to the hardware are implemented and briefly evaluated. A collection of software primitives for writing tightly coordinated parallel programs are implemented and tested.

3 Introduction

The notion of a precision-timed (PRET) architecture, as imagined by Edwards and Lee[1], is in some ways not far removed from the implementation of low-cost microcontrollers today, or of processors that were used four or five decades ago [3]. PRET architectures do not have multilevel caches, branch predictors, or dynamically scheduled out-of-order pipelines, and as a result their timing behavior is relatively easy to reason about. However, they are distingished in that they document these restrictions as hard guarantees and in that are intended to deliver relatively performance despite these restrictions. For example, some PRET architectures may be used in configurations with large numbers of cores and many hardware threads per core.

This project is an attempt to benchmark a small program written to take full advantage of the timing predictability of the PRET design called FlexPRET [4].

3.1 Contributions

The RVG template language has not been described in any publication or used in any other coursework. Significant parallelism-related extensions to its standard library were made for the purpose of this assignment, including parallel for loops, splittable iterators, critical sections, all-to-all communication between hardware threads ("harts"), and scatter-gather communication between cores.

Modifications to an existing hardware simulator are proposed to allow more convenient expression of parallelism.

4 Design Overview

In the process of writing programs in which exact execution times are of interest, there is a range of human involvement that may be required. At one extreme, a human programmer may implement an explicit specification of the number of cycles required to execute a given control flow block. This approach is difficult to scale because manually counting cycles is error-prone and because any change to one block may result in a cascade of changes in the guarantees of blocks that use it. At another extreme, the programmer may write code using a subset of C, compile it using an optimizing compiler, and use a separate tool to analyze the C code and compute its worst-case execution time (WCET). This latter approach requires technology that is hardware-specific and difficult to implement well; therefore, although WCET analysis tools exist and are effective for many programs that are of practical importance, they may not exist for a given hardware platform and may not give tight execution time bounds. Practical solutions fall between these extremes. The PATMOS project, for example, does not use a general-purpose optimizing compiler but instead uses a WCET-aware compiler that places limitations on what the programmer can write and that produces machine code that is optimized to have WCETs that are small and calculable.

Another intermediate approach is to allow the programmer to use the first approach, with assembly, but with the aid of a metaprogram that automates the simplest cycle-counting tasks, such as computing the execution time of sequential code that is free of control flow and that executes on simple hardware. The programmer can then write unchecked assertions about execution times in terms of arithmetic expressions and then amortize the cost of manually verifying the assertions by writing them once in a reusable control-flow construct. Because the same metaprogram produces both the program and its execution time, this forces the program to take a form that makes the execution time analysis sound and tractable.

This is the approach pursued here and is visualized in Figure 1. The core is a minimal templating language based on the lambda calculus. It has no explicit support for recursion and is implemented in approximately 1000 lines of OCaml. `std.ml` provides useful functions that are difficult to implement without recursion, arithmetic on regular OCaml numbers rather than Church numerals, and other builtins; `assemblyParse.ml` implements analysis and cycle counting of assembly code based on constants defined in the environment; the RVG file `manyhart-table.rvg` defines these constants based on the thread cycles consumed by instructions on FlexPRET when the number of harts exceeds 3; `ctrl.rvg` defines control-flow primitives, such as branches, loops, and iterators, which must be implemented in a particular way in order to make execution times easy to predict; `platform.rvg` defines platform-specific utilities, such as the function for printing a number in simulation; and `parallel.rvg` implements primitives that are required in parallel programs, such as critical sections, parallel for loops, and the fork-join pattern.

Arbitrary functions can be provided for dynamic validation in many of the places where type annotations would be used in a statically typed language. They can make assertions about execution time in cycles, check that values have certain attributes, require that an expression denotes a valid register, and perform any other programmatically-expressible validations that are needed in order to detect mistakes early.

5 Hardware

5.1 Unresolved hardware limitations

In the process of completing this project, hardware limitations were encountered in the ISA and in the interconnect. These limitations define what it means to achieve machine peak bandwidth and machine peak compute.

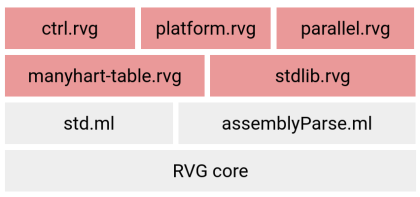

Figure 1: The design of the code generation system used to produce the unrolled and parallelized assembly code.

An in-order pipeline with time-division multiplexing can achieve high utilizations of the hardware resources required to implement the minimal RV32I instruction set. However, it cannot achieve high utilization of more specialized hardware, such as a floating point unit or even an integer multiplication and division unit, simply because even in a numerically intensive task such as a BLAS3 operation, not all instructions can be multiplications or divisions. This is particularly true in the case of a RISC-V or MIPS-like architecture in which the only instructions that can interact with the memory system are load and store instructions. The consequence of this limitation is that solutions for achieving high performance per unit cost when performing specialized arithmetic operations are not available.

Fine-grained multithreading can also increase the number of loads and stores that are required in order to achieve a given task in a given period of time in comparison to a single-threaded architecture that devotes hardware resources to forwarding instead of multithreading. This is because when only one or two tasks are ready to execute, the fastest software implementation is the one that makes the threads cooperate. This may result the same data being loaded from memory individually into each of the eight register files, whereas the single-threaded hardware would only need to load the data into one register file.

Another limitation is that the TDM routing system cannot scale to large numbers of cores. On a chip with 64 cores, the TDM schedule must be at least 64 cycles long, and in the design currently available for FlexPRET it would be 87 cycles long. This means that a given virtual communication channel between two cores becomes ready to move a word exactly once out of every 87 cycles. Although this is not as severe a limitation as it may seem given that a given thread already can only make progress once out of every eight clock cycles, it is still considered too long a period to allow low-latency communication between any pair of cores.[1] The current proposal is to extend the interconnect to be hierarchical. Communication strategies such as Dragonfly that use randomness to achieve high bandwidth with high probability at low hardware cost are, of course, not an option in the types of applications targeted by FlexPRET, so an older, less scalable design such as a fat tree is more likely to be used. A fat tree, however, cannot deliver high bandwidth using the existing hardware-software interface which requires one core to write each individual word to the correct memory-mapped address for a given virtual communication channel, because the speed of the communicating processor then places a limit on the fatness of any edge of the tree.

5.2 Hardware Modifications

The original implementation of the CPU interface to the NoC was a finite state machine; in order to send a word on the network, the CPU needed to write an address indicating the destination of the word, and then send the word. Additionally, the FSM used to get trapped in a waiting state whenever a read was attempted when no data was available. When eight harts are cooperating on a send operation, they all submit data with different destinations as a large number of interleaved transactions. This type of usage is required in order to approach the peak bandwidth that is achievable in a scatter or broadcast operation because a given core has a fixed amount of bandwidth to all other cores at all times, regardless of whether the bandwidth is being used. In addition to causing invalid behavior in the absence of a critical section implementation, the FSM design results in poor performance even without concurrent accesses because two instructions are required to specify one operation. For these reasons, I removed the finite state machine and changed the hardware to use the address of the load or store operation to determine the destination core; this allows the data and the destination core to be specified in the execution of a single instruction, which allows the use of a stateless interface.

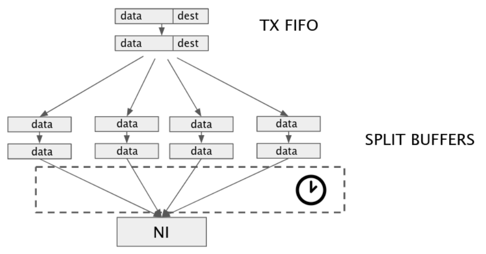

Figure 2: One possible design for the interface between the CPU and the network-on-chip.

In the original implementation, all pending words were submitted to a shared queue (shown at the top of Figure 2) before being submitted to the network port. Like the stateful CPU interface, the shared queue causes complex timing interactions between harts via head-of-line blocking. This is because the element at the end of the queue always waits until the NoC's time-division-multiplexing (TDM) schedule gives it a time slot; therefore, all other elements on the queue may wait for it for a substantial amount of time, allowing their time slot to pass by. The shared queue served no purpose other than to provide additional buffering between the

[1]One "quick fix" would be to create multiple parallel channels between the chips to alleviate this. The network-on-chip is cheap enough for this to be imaginable, but this does not solve the underlying problem that true all-to-all communication has poor asymptotic scaling with respect to core count, either in terms of virtual channels or in terms of physical channels.

CPU and the network, and I removed it because a predictably-scheduled hard real-time hart can always time its NoC accesses such that the even a buffer of depth two does not overflow. Indeed, to conserve hardware resources, the non-shared data buffers on either end of the network are of depth two.

6 Evaluation

6.1 Micro-benchmark 1: Scatter

The following code is the entire benchmark. Implementations for `using-noc`, `single-hart`, `delay`, `iterate-at-rate`, `noc-read`, `sim-print`, `memregion`, `cyclify`, `rng-seeded`, and `pariterate-at-rate` are omitted for brevity.[2]

```
[mu []
[def (do-receive)
[using-noc a3 [mu []
{
    [single-hart t1
    {
        [delay a4 60]
        [iterate-at-rate t2 of [.. t4 0 to 5] doing 2 per 42 cycles
        {
            [noc-read t0 0]
            [sim-print t0 t3]
        }]
    }]
}]]]
[def (do-send)
{
    [def (others) [memregion t0 t1 noc-base-address words 2 to 16 by 1]]
    [def (scattering) [cyclify a0 a1 others times 5]]
    [def (zipped) [zip
    t4 of [rng-seeded t2 t3 bits 5 length 16 seeded 7]
    and t6 of scattering]]
    [[nonzero-team pariterate-at-rate] {rand,addr}
        of zipped doing 4 per 42 cycles
    {
        sw [fst rand,addr] [snd rand,addr]
    }]
}]
[[file {Reset_Handler} {
Reset_Handler:
    [id init]
    [[single-else get-coreid] a3
    {
        [[single-else get-hartid] t5 {}
        [id do-send]]
    }
    {
        [id do-receive]
    }]
    [id die]
```

[2]All code can be found at `https://github.com/petervdonovan/rvg/tree/laziness-hack`, `https://github.com/petervdonovan/rvg-stdlib`, or `https://github.com/petervdonovan/cs267-final-project`

```
}]]
]
```

The problem setup is as follows: Core 0 has access to some stream R of data, but R does not fit in Core 0's local memory. For example, perhaps R is the result of some computation, or perhaps R is being read from a UART. Core 0 must distribute R between the other cores without storing R in its local scratchpad or in the buffers in the network interface of the network on chip. In fact, to conserve hardware resources, the buffers in the network interface only have enough room for two words per destination core.

One implementation would read one word from the stream, perform a handshaking procedure with the receiving core, send the word, and repeat. The handshaking procedure would ensure that the buffer does not overflow and that no core attempts to read from an empty buffer. This implementation, however, has high overhead.

Another implementation would read many words from the stream, store all of them in a buffer, and then invoke a carefully optimized software routine that sends all of the words at a precise rate. The receiving core would then call a similar routine that reads the words at the same rate. This routine can be made reasonably efficient, but it results in an unnecessary copy, and the process of implementing these special timing-critical routines in plain RISC-V assembly is not straightforward because it requires that the routines access the buffer at a very precise rate. Furthermore, to achieve high performance, such routines must be implemented specifically for each of several communication patterns – point-to-point, scatter, broadcast, etc. – rather than implementing many patterns via reduction to one or two optimized kernels. This is because, since the NoC uses a fixed TDM schedule, sending data to every other core at once takes approximately the same amount of time as sending data to just one core. Therefore, if there are P cores, then "scatter" cannot be implemented by reducing it to a sequence of "send" and "receive" operations without incurring $O(P)$ additional latency.

The implementation pursued in this project aims to make it as simple as possible to write routines of this type, and it further endeavors to copy R directly into the network-on-chip in real time.

6.1.1 Design Choices

The core problem is to concurrently do reads and writes to the network at a fixed rate.

The initial design was to simply adapt the timing-predictable `for` loop implementation to suit this purpose. This is simple: The time to perform each loop iteration is calculable, and the time to branch to the beginning of the loop is calculable, so it is possible to pad the loop body with no-ops to give it a fixed period (in cycles). However, this simple solution does not work.

In order to have a predictable execution time, the loop must have static bounds: A general C-style `for` loop that allows initialization, condition, and advancement cannot work without additional work to ensure that its number of iterations is known at code generation time. Therefore, the initial design was to only allow loops that start at a fixed value, end at a fixed value, and have a fixed stride, as these are by far the most prevalent type of loop in the textual representation of C programs. However, this is not the most prevalent type of loop in the output of an optimizing compiler; for example, when iterating over an array, efficient compiler-emitted code simply increments a pointer. It does not increment an index and repeatedly add it to the pointer in each iteration. Therefore, without the aid of an optimizing compiler, the obvious way to implement a fixed-frequency `for` loop results in poor performance.

The straightforward `for` loop also does not work because it cannot take advantage of immediates, which appear in machine code to statically set offsets from addresses that are dynamically available in registers. It is possible to circumvent this by adding to an address, using it in a load/store with offset zero, and then subtract from the address; however, this incurs a cost of at least two additional instructions. Because FlexPRET has an in-order pipeline, no measurement is required to ascertain that the cost of such an operation, when performed in an inner loop, is significant.

A third problem is that the most straightforward `for` loop is not suitable for streams such as the one in the problem statement, where integers are provided on the fly and do not fit in the scratchpad of a single core.

The solution to these problems is to implement an iterator abstraction. The iterator interface mandates that iterators provide their (finite) length at compile time, as well as a function that generates unrolled assembly for them and a means of checking at run-time when they are empty. Importantly, all iterators must provide a `split` function, which divides the elements of the iterator into equally-sized parts[3] at code generation time. These parts do not correspond to different segments of machine code and so do not increase code size, but instead are encoded by their dynamic accesses to the current rank ID.

This abstraction allows a single implementation of the `iterate` control-flow construct to be used to accurately compute the execution times of many types of iterables, including integer streams that are not stored in memory, (possibly repeating) sequences of immediate values, and arrays implemented using incrementing pointers, without special cases for the different types.

6.1.2 Performance relative to machine peak

Because the TDM schedule is 19 cycles long, it can transmit a word every 19 clock cycles. Because integers in this example are source from a random number generator, they can be produced at a rate of at most 2 words out of 26 *thread* cycles. Because a thread makes progress in only one out of every 8 clock cycles, this corresponds to $26 \times 8/2 = 104$ clock cycles per word, which corresponds to a utilization of only 18 percent.

Furthermore, the processors receiving data receive one word every 104 cycles, which gives them very little computational work, and core 1 is not utilized at all because the thread that would send data to core 1 would also send data to core 0 unless another special case were written into the program, which could reduce performance.

Clearly, performance is limited by the rate at which core 0 can produce random numbers. The random number generator is an efficient non-cryptographic quality xorshift RNG that produces two new numbers in 6 cycles each, and with 2 more cycles each to copy data from the random state to the output vector. 6 more cycles are an unnecessarily duplicated initialization sequence that appears between loop iterations, 2 more cycles are control flow, and 2 more decrement counters, giving a total of 26 cycles. Of these, the first 16 cycles mentioned are unavoidable and the remaining ten can be amortized by loop unrolling. For example, increasing the loop unrolling by a factor of 2 amortizes these ten cycles of overhead proportionally and allows 2 words to be sent for every 21 cycles.

7 Primitives Implemented and their Pertinence to Parallel Programming and Performance Engineering

Critical sections are implemented in software using a technique similar to the one described by Schoeberl and Puschner [2]. They have the precondition that all threads reach the critical section at the same time. Upon reaching the critical section, each thread reads its thread ID, and from its ID it computes the fixed amount of time that it must wait for its turn to access a resource. When it is done using the resource, it waits a predictable amount of time for all other threads to finish using the resource. No communication between threads is required in this simple protocol.

Reductions and all-gathers between threads on the same core are performed in the trivial way. On startup a shared buffer is allocated for the purpose of thread-to-thread communication, and the threads write to the shared buffer so that they all have access to all data. This is approximately as efficient as a tree reduction when the cost of combining results is small because there are at most eight threads per core, the logarithm base two of eight is not much smaller than eight, and the tree reduction would require more memory operations. Memory operations are as costly as all other instructions – they cost one cycle – regardless of how fast the CPU can

[3]Non-divisibility results in errors at code generation time.

access its nearest memory. Similarly, even when the number of cores is large, it is difficult to attain improved performance on InterPRET by using an efficient algorithm for reduction across multiple cores because time slots are statically allocated for all possible communication regardless of whether one attempts to optimize some communication away.

The implementation of the fork-and-join pattern is as follows. Each rank loads its ID and repeatedly either branches to its appropriate task (if its ID is zero) or decrements its ID. When it is finished with its task, it branches to a precisely chosen position in a sequence of no-ops such that all ranks exit the sequence of no-ops at the same time.[4]

Parallel-iterate (`pariterate`) constructs are implemented simply as splitting apart a compile-time iterator abstraction and then letting each rank at run time iterate over its own part in the standard (non-parallel) way. Parallel-for (`parfor`) constructs are similar.

7.1 The iterator abstraction

The iterator abstraction was motivated in section 4. It provides clear performance advantages in comparison to reducing a wide range of iteration types into loops over indices, and it enables parallelism by providing a `split` function. Iterators are a well-known and thoroughly understood concept, but the implementation of iterators for the purpose of this project was successful in a few ways which deserve some discussion.

The first is that they provide perhaps the thinnest possible abstraction over the difference between concepts that are available at compile time (such as matrix and array objects that provide code generation APIs) and concepts that exist only at runtime (such as the values physically stored in registers). This is possible because the iterator implementations and the code generation constructs that use them operate on untyped "output" objects that make very few guarantees, except in that they act as "handles" for controlling which values are accessible, and where. The result is that iterators that produce output objects of either of the two kinds can be zipped into one iterator, converted into a longer iterator that cyclically repeats the values in the original iterator many times, or passed through a general `map` function that converts from one type of an "output" object to another. For example, address objects that include both an immediate and the name of the register that will contain the address at runtime can be mapped to more abstract array objects that are themselves iterables.

Another advantage is that the iterators naturally express loop unrolling. These `zip` and `cyclify` functions define iterators in terms of other iterators, allowing the unrolling to be delegated to the lowest-level iterator which is in the innermost loop and which therefore benefits the most from unrolling. In particular, when simultaneously broadcasting over multiple virtual channels of the network-on-chip, it makes sense to `cyclify` the `memregion` construct to cycle over the memory-mapped IO addresses of each of the destination cores; this effectively puts the `memregion` construct inside of a loop. A loop unrolling requirement manually specified by the user at the level of the enclosing `iterate` control-flow construct is passed among the hierarchical levels of iteration so that it affects the part that benefits from the unrolling.

References

[1] Stephen A Edwards and Edward A Lee. "The Case for the Precision Timed (PRET) Machine". In: ().

[2] Martin Schoeberl and Peter Puschner. "IS CHIP-MULTIPROCESSING THE END OF REAL-TIME SCHEDULING?" In: ().

[3] Martin Schoeberl et al. "Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach". In: *Predictability and Performance in Embedded Systems (PPES 2011)* Open Access Series in Informatics (2011).

[4]With fine-grained multithreading, there is a slight offset between the times when the instruction counters of each thread on the same core are incremented. The phrase "at the same time" is used here to mean "within the same sequence of eight cycles such that at any time during that period, a read to the thread cycles CSR would result in the same value."

[4] Michael Zimmer et al. "FlexPRET: A processor platform for mixed-criticality systems". In: *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2014, pp. 101–110. DOI: 10.1109/RTAS.2014.6925994.

A Other Passing Test Cases

To convey an idea of the parallel programs written during the course of this project, a few programs are selected here from among the passing test cases.

This program fills a matrix and then reads from it.

```
[mu []
[def (A) [matrix 4 by 4 at t0]]
[[file {Reset_Handler} {
    Reset_Handler:
    [id init]
    [single t5
    {
        [A matrix-allocate]
        [iterate {row} of [[A matrix-rows] a0 a1 a2]
        {
        [iterate {addr} of [map [row t1 t2] mapaddr]
        {
            li t3 42
            sw t3 [id addr]
        }]
        }]
        [iterate {row} of [[A matrix-rows] a0 a1 a2]
        {
        [iterate {addr} of [map [row t1 t2] mapaddr]
        {
            lw a3 [id addr]
            [sim-print a3 t4]
        }]
        }]
    }]
    rdcycle a4
    [single-core a2 [critical-hart t5 a1 {
        [get-hartid t6]
        [sim-print t6 a0]
        [sim-print a4 a0]
    }]]
    [id die]}]]
]
```

This program populates two arrays and computes their elementwise maximum:

```
[mu []
[def (N) 8]
[def (A) {t2}]
[def (B) {t3}]
[def (C) {t4}]
[[file {Reset_Handler} {
    Reset_Handler:
    [id init]
    li A 0
    li B 0
    li C 0
    [single-hart a0 {
        [allocate N A]
        [allocate N B]
        [allocate N C]
        [for t0 t1 0 to [* 4 N] by 4 {
        add a1 A t0
        addi a2 t0 [/ N -8]
        sw a2 0(a1)
        }]
        [for t0 t1 0 to [* 4 N] by 4 {
        add a1 B t0
        li a2 [/ N 2]
        sw a2 0(a1)
        }]
    }]
    [single-core a2 {
        [reduce-shmem +acc A 0 a0 a1]
        [reduce-shmem +acc B 0 a0 a1]
        [reduce-shmem +acc C 0 a0 a1]
    }]
    [single-core a0
        [parfor t0 t1 0 to [* 4 N] by 4 {
        add a1 A t0
        add a2 B t0
        add a3 C t0
        lw a1 0(a1)
        lw a2 0(a2)
        [max~ a4 a1 a2]
        sw a4 0(a3)
        }]]
    [single-core a0
        [single-hart a1
        [for t0 t1 0 to [* 4 N] by 4
        {
            add a3 C t0
            lw a2 0(a3)
            [sim-print-int 42 a4 a5]
            [sim-print a2 a3]
        }]]]
    [single-core a0 [single-hart a1 [sim-print-int 99 a4 a5]]]
    [id die]}]]
]
```