A Streaming Multi-Threaded Model

Extended Abstract

Eylon Caspi, André DeHon, John Wawrzynek

September 30, 2001

Summary. *We present SCORE, a multi-threaded model that relies on streams to expose thread parallelism and to enable efficient scheduling, low-overhead communication, and scalability. We present work to-date on SCORE for scalable reconfigurable logic, as well as implementation ideas for SCORE for processor architectures. We demonstrate that streams can be exposed as a clean architectural feature that supports forward compatibility to larger, more parallel hardware.*

For the past several decades, the predominant architectural abstraction for programmable computation systems has been the instruction set architecture (ISA). An ISA defines an instruction set and semantics for executing it. The longevity of the ISA model owes largely to the fact that those semantics decouple software from hardware development, so that hardware can improve without sacrificing the existing software of an architectural family (*e.g.* the now 23-year old Intel x86 family; the IBM 360).

Increasingly, however, ISA uniprocessors are running out of headroom for performance improvement, due primarily to the increasing costs of extracting and exploiting ILP. Today's state-of-the art processors expend much of their area and power in hardware features to enhance ILP and tolerate latency, including caches, branch prediction units, and instruction reorder buffers. Recently, new architectures have emerged to exploit other forms of parallelism, including explicit instruction parallelism (VLIW), data-level parallelism (vector, MMX), and thread-level parallelism (chip multiprocessors, multi-threaded processors). These architectures typically sacrifice some of the desirable properties of the single-threaded ISA model, be it ease of programming, compiler analyzability (*e.g.* obscured inter-thread communication patterns), or forward compatibility to larger hardware (*e.g.* VLIW). In this paper we present SCORE, a scalable, multi-threaded computation model and associated architectural abstraction that maintain some of the best properties of the ISA model. The model is highly parallel, efficient, and supports forward compatibility of executable programs to larger hardware.

At the heart of SCORE is the premise that streams (inter-thread communication channels with FIFO discipline) should be a fundamental abstraction that is exposed both in the programming model and in the architectural model. We rely on streams because: (1) streams expose inter-thread communication dependencies (data-flow), allowing efficient scheduling; (2) streams admit data batching (data blocking) to amortize the cost of context swaps and inter-thread communication—in particular, the per-message cost of communication can be made negligible by setting-up and reusing a stream, and this reuse can be fully automated as part of thread scheduling; (3) streams can be exposed as a clean architectural feature, resembling a memory interface, that admits hardware acceleration, protection, and forward compatibility.

SCORE draws heavily on the prior work of numerous parallel compute models and system implementations. Due to space constraints in this paper, we refer the reader to [1] for a more complete treatment of related work and implementation details.

In the remainder of this paper we discuss: (a) the basic primitives and properties of the SCORE model; (b) a binding of SCORE for reconfigurable logic, including a hardware model, programming model, scheduler, and simulation results for a media processing application; and (c) ideas for a binding of SCORE for processor architectures.

The SCORE Model.

A SCORE program is a collection of threads that communicate via streams. A *thread* here has the usual meaning of a process with local state, but there is no global state shared among threads (such as shared memory). The only mechanism for inter-thread communication and synchronization is the *stream*, an inter-thread communication channel with FIFO order (first-in-first-out), blocking reads, non-blocking writes, and conceptually unbounded capacity. In this respect, SCORE closely resembles Kahn process networks [2] or, as we shall see later, dataflow process networks [3]. A *processor* is the basic hardware unit that executes threads, one at a time, in a time-multiplexed manner when there are more threads than processors.

Since threads interact with each other only through streams, it is possible to execute them in any order allowed by the stream data-flow, with deterministic results.[1] The data-flow graph induced by stream connections exposes inter-thread dependencies, which reflect actual inter-thread parallelism. Those dependencies can be used to construct schedules that are more efficient for a particular hardware target (*i.e.* minimize idle cycles) and which can be pre-computed. In contrast, other threading models tend to obscure inter-thread dependencies (*e.g.* pointer aliasing in shared memory threads) and restrict scheduling using explicit synchronization (*e.g.* semaphores, barriers). Such restricted schedules cannot take advantage of larger hardware and thus undermine forward compatibility.

Threads can be time-sliced to batch-process a large sequence of inputs. Batching is useful for amortizing the run-time costs of context swapping and of setting up stream connections and buffers. This batching mechanism is similar in spirit to traditional data blocking for better cache behavior, but in our case the blocking factor is determined in scheduling and can be tailored to the available hardware (namely to

[1]The blocking read and FIFO order of streams guarantee determinism under any schedule. Non-determinism can be added to the model by allowing *non-blocking* stream reads, in which case thread execution becomes sensitive to scheduling and communication timing.

available buffer sizes) as late as at run-time. This late binding of the blocking factor contributes to forward compatibility and performance scaling on larger hardware. In contrast, data blocking in non-streaming models is fixed at compile time and cannot improve performance on larger hardware.

SCORE for Reconfigurable Architectures.

Our first target for SCORE is as a model for scalable reconfigurable systems. Reconfigurable logic (*e.g.* field programmable gate arrays—FPGAs) is a promising performance platform because it combines massive, fine-grained, spatial parallelism with programmability. Programmability, and in particular dynamic reconfiguration, aids performance because: (1) it can improve hardware utilization by giving otherwise idle logic a meaningful task, and (2) it allows specializing a computation around its data, later than compile time. Although reconfigurable systems (FPGAs, CPLDs) have been available commercially for some time, they have no ISA-like abstraction to decouple hardware from software. Resource types and capacities (*e.g.* LUT count) are exposed to the programmer, forcing him/her to bind algorithmic decisions and reconfiguration times to particular devices, thereby undermining the possibility of forward compatibility to larger hardware.

SCORE hides the size of hardware from the programmer and the executable using a notion of hardware paging. Programs and hardware are sliced into fixed-size *compute pages* that, in analogy to virtual memory pages, are automatically swapped into available hardware at run-time by operating system support. A paged computation can run on any number of compute pages and will, without recompilation, see performance improvement on more pages. Inter-page communication uses a streaming discipline, which is a natural abstraction of synchronous wire communication, but which is independent of wire timing (a page stalls in the absence of stream input), and which allows data batching. Batching is critical to amortizing the typically high

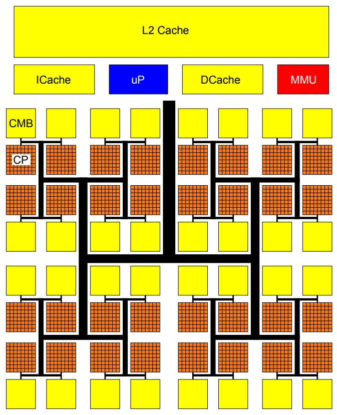

Figure 1: Hypothetical, single-chip SCORE system

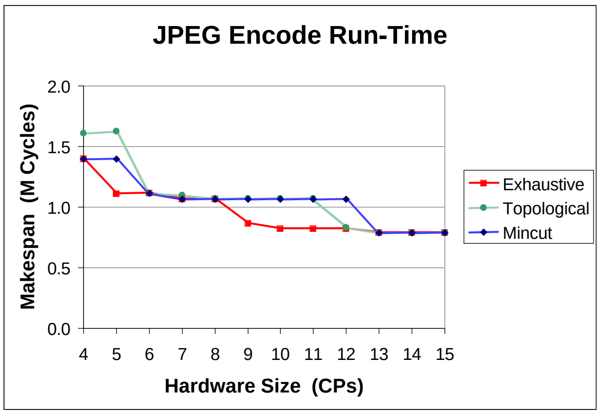

Figure 2: Run times for JPEG Encode

cost of reconfiguration. With respect to the abstract SCORE model, a compute page serves the role of a processor, whereas a page configuration serves the role of a thread.

The basic SCORE hardware model is shown in Figure 1. A *compute page* (CP) is a fixed-size slice of reconfigurable fabric (typically logic and registers) with a stream network interface. The fabric kind, size, and number of I/O channels (streams) are architecturally defined and are identical for all pages (we presently target a page of 512 4-LUTs). The number of pages can vary among architecturally compatible devices. A page's network interface includes stream flow control (data presence and back-pressure to stall the page) and a buffer of several words per stream (for reducing stalls and draining in-flight communication). A *configurable memory block* (CMB) is a memory block with a stream network interface and a sequential address generator. The CMB memory capacity is architecturally defined (we presently target a CMB of 2Mbit). A CMB holds stream buffers, user data, and page configurations, under OS-controlled memory management. A conventional microprocessor is used for page scheduling and OS support. The common streaming protocol linking these components allows a page thread to be completely oblivious to whether its streams are connected to another page, to a buffer in a CMB, or even to the microprocessor. The actual network transport is less important, though it should ideally have high bandwidth. We presently assume a scalable interconnect modeled after the Berkeley HSRA [4] that is circuit-switched, fat-tree structured, and pipelined for high-frequency operation.

We have designed a language (TDF) for specifying page threads and inter-page data-flow, as well as a compiler (tdfc) to compile threads into page simulation code or into POSIX threads for microprocessor execution. Microprocessor threads interact with simulated hardware using a stream API and can dynamically spawn and connect page threads. A TDF page thread is defined as a *streaming finite state machine* (SFSM), essentially an extended FSM with streaming I/O. The execution semantics specify that on entry into a state the SFSM issues blocking reads to those streams specified in the state's *input signature*. When input is available on those streams, the SFSM *fires*, executing a state-specific action described in a C-like syntax. At present, SFSMs must be explicitly written to match page hardware constraints (area, I/Os, timing), but we are working on an automatic partitioner to decompose arbitrarily large SFSMs into groups of stream-connected, page-size SFSMs.[2] We have written a number of media processing applications in TDF, including wavelet, JPEG, and MPEG coding.

We have developed and tested several page schedulers. Each such scheduler is a privileged microprocessor task responsible for time-multiplexing a collection of page threads on available physical hardware. We use a time-sliced model where, in each time slice, the scheduler chooses a set of page threads to execute in hardware and manages stream buffers to communicate with suspended pages. Schedules are quasi-static, with a page loading order determined once from the stream data-flow topology and applied repeatedly. Schedules strive to run communicating pages together to reduce

[2]Thread sizing (to match a page) is an artifact of the reconfigurable hardware target. It may be unnecessary or optional with other hardware targets and should, in general, not be exposed to the programmer. Partitioning SFSMs into reconfigurable pages is in some ways analogous to restructuring microprocessor code to improve VM page locality.

communication latency (especially under inter-page feedback) and stream buffering.

Figure 2 shows performance results for running JPEG encode (an application with 15 page threads) on simulated hardware of varying page counts, using three quasi-static schedulers. First, these results demonstrate that application performance scales predictably across hardware sizes—more hardware means shorter run-times. Second, these results demonstrate that time-multiplexing is efficient in the sense that the application can run on substantially fewer compute pages than there are page threads, with negligible performance loss. The application, as implemented, has limited parallelism that requires some threads to be idle some of the time—a time-multiplexed schedule can avoid loading such threads into idle hardware. Third, these results show that simple, heuristic schedulers based on stream data-flow topology (*topological*: topological order, *min-cut*: minimize buffered streams) perform almost as well an exhaustive-search to minimize idle cycles (*exhaustive*).

SCORE for Processor Architectures.

The SCORE architecture developed above can be easily extended with additional processor types. We have already defined three processor types: CP, CMB, and microprocessor. An extended architecture might use specialized function units (ALUs, FFT units, *etc.*) or additional microprocessors. Each processor must have a stream network interface with ability to stall the processor. The network fabric may vary.

Streams can be added to a microprocessor as a clean architectural feature, resembling a memory interface. The instruction set is extended with load/store-like operations: stream_read(*register*,*index*) and stream_write(*register*,*index*), where *index* enumerates the space of streams (like a memory address), and *register* denotes a value register. Stream reads must be able to stall the processor (like memory wait states), while stream writes can resume immediately (like memory stores to a write buffer). These features can be handled by stream access units, resembling memory load/store units, even with out-of-order issue. These units must transparently route stream accesses either to a network interface or to memory buffers. Stream state ("live" or "buffered") and buffer locations can be stored in a TLB-like stream table, backed by a larger table in memory (like a VM page table). Access protection can be added using a process ID in the stream table. Finally, stalled stream accesses should time out after a while, allowing a scheduler to swap out blocked threads. Many of these features could be emulated on a conventional microprocessor using memory mapping or a co-processor interface, plus an external controller to route stream accesses to a network or to buffer memory.

The scheduling policies for reconfigurable hardware are easily adapted to multiprocessor and multi-threaded processor architectures, provided I/O operations are as fast as memory operations. A chip multi-processor would be scheduled like a SCORE architecture with only microprocessors. A multi-threaded processor would be scheduled similarly, treating each hardware thread as a separate SCORE processor. Communication between threads loaded in hardware is "live" through registers, whereas communication to threads swapped-out to memory is buffered in memory. In either case, the scheduler prefers to co-schedule communicating threads.

References

[1] E. Caspi, M. Chu, R. Huang, J. Yeh, Y. Markovskiy, J. Wawrzynek, and A. DeHon. Stream computations organized for reconfigurable execution (SCORE): Introduction and tutorial. `http://brass.cs.berkeley.edu/documents/score_tutorial.pdf`, August 2000.

[2] G. Kahn. Semantics of a simple language for parallel programming. *Info. Proc.*, pages 471–475, August 1974.

[3] E. A. Lee and T. M. Parks. Dataflow process networks. *Proc. IEEE*, 83(5):773–801, May 1995.

[4] W. Tsu, K. Macy, A. Joshi, R. Huang, N. Walker, T. Tung, O. Rowhani, V. George, J. Wawrzynek, and A. DeHon. Hsra: High-speed, hierarchical synchronous reconfigurable array. In *Proc. International Symposium on Field Programmable Gate Arrays (FPGA '99)*, pages 125–134, February 1999.