A Streaming Multi-Threaded Model

Eylon Caspi
University of California, Berkeley
eylon@cs.berkeley.edu

André DeHon
California Institute of Technology
andre@acm.org

John Wawrzynek
University of California, Berkeley
johnw@cs.berkeley.edu

ABSTRACT

We present SCORE (*Stream Computations Organized for Reconfigurable Execution*), 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.

1. OVERVIEW

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. A key benefit of the ISA model is that those semantics decouple software from hardware development. A piece of software, written and compiled once, is guaranteed to run on any ISA-compatible device. This guarantee allows hardware to evolve over time, growing larger and faster with each process generation. The existing software base is preserved, and its performance automatically improves with each hardware generation. The ISA abstraction has been instrumental in protecting our investment in software and allowing it to ride Moore's law to better performance. Two shining examples are the IBM 360 and Intel x86 architectures, which have survived commercially for decades. The latter, in its 23 years of existence, has seen clock speeds increase nearly 400x and transistor counts grow nearly 10,000x.[1]

Increasingly, however, ISA uniprocessors are running out of headroom for performance improvement, due primarily to the increasing costs of extracting and exploiting instruction level parallelism (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.

The remainder of this paper is organized as follows. Section 2 ("Related Work") cites prior work in data-flow, streaming, and architecture that has inspired SCORE. Section 3 ("The SCORE Model") discusses the basic primitives and properties of the SCORE model. Section 4 ("SCORE for Reconfigurable Architectures") presents a binding of SCORE for reconfigurable logic, including a hardware model, programming model, scheduler, and simulation results for a media processing application. Section 5 discusses ideas for a binding of SCORE for processor architectures.

[1] Comparing the Intel 8086 at 5MHz (29,000 transistors, introduced June 1978) to the 0.18μ 2GHz P4 (221 million transistors, introduced August 2001). [14]

To appear in the Third Workshop on Media and Stream Processors, in conjunction with MICRO-34, Austin, Texas, Dec. 2001.

2. RELATED WORK

SCORE draws heavily on the prior work of numerous parallel models and architectures. This section highlights only a few of those works. We refer the reader to [7] for a more complete treatment.

Hoare's Communicating Sequential Processes (CSP) [13] was one of the first strong models for parallel computation. SCORE shares with CSP the general notion of computation as a collection of communicating, independent, processes with local control. The SCORE model is more stylized, making it amenable to virtualization and use of a strong hardware-software interface. SCORE emphasizes a design that preserves deterministic behavior regardless of target hardware size and scheduling.

Data-flow processing from Dennis [10] and Arvind [3] introduced parallel models and architectures with more flexible scheduling. Later work in Culler's Threaded Abstract Machine (TAM) [8] and Active Messages (AM) [11] was an important attempt to capture the essence of a parallel programming model at the software-hardware boundary and to make communication lightweight. SCORE's use of persistent, streaming data-flow is significant in overcoming many of the overheads that still made TAM expensive to implement.

Mosaic [21], J- and M-machines [9] [12] were early multicomputers pioneering the tight integration of communication into the processor ISA. Nonetheless, their use of dynamic messages still required a few tens of cycles to send each message [18]. Streaming in SCORE enables pipelined communication, reducing the time required to send or receive a message to as little as one machine cycle, with proper hardware support.

A wide range of shared-memory machines, including DASH [19], Alewife [2], and FLASH [16] showed that the memory abstraction was useful for some communication. However, they observed that relying solely on shared-memory cache-coherence for communication was not sufficient to broadly deliver high performance. Shared-memory programming remains much more difficult than single-threaded programming, owing primarily to the fact that determinism by synchronization is entirely the burden of the programmer.

Streaming data-flow has been used heavily in Digital-Signal Processing (DSP). Lee's Synchronous Dataflow (SDF) [4] is a heavy influence for SCORE. Lee's SDF is restricted to static rates and static flow graphs, suitable for systems modeled completely at compile-time. Later work, such as Buck's Boolean-Controlled Dataflow [6], supports some dynamic programming constructs, but still with a compile-time focus. SCORE expands on these models to handle more dynamic characteristics such as dynamic flow rates and graph evolution, as well as variable hardware resource availability.

Also inspirational to SCORE are heterogeneous systems that use streaming data-flow to tie together arbitrary processors (conventional, special-purpose, and/or reconfigurable). Examples include MIT's Cheops [5], MagicEight [23], and Berkeley's Pleiades [1]. These systems provide interesting performance point with a mix of processing elements. Our work on SCORE for hybrid reconfigurable hardware (Section 4) builds on these models by defining a common programming model for all processing elements (microprocessor and reconfigurable in our case) and by virtualizing the number and type of elements.

3. 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 [15] and more specifically, Dataflow process networks [17]. 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.[2] 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 may not be able to take full 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 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.

4. SCORE FOR RECONFIGURABLE ARCHITECTURES

Our first target for SCORE is as a model for scalable reconfigurable systems. Reconfigurable

[2]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.

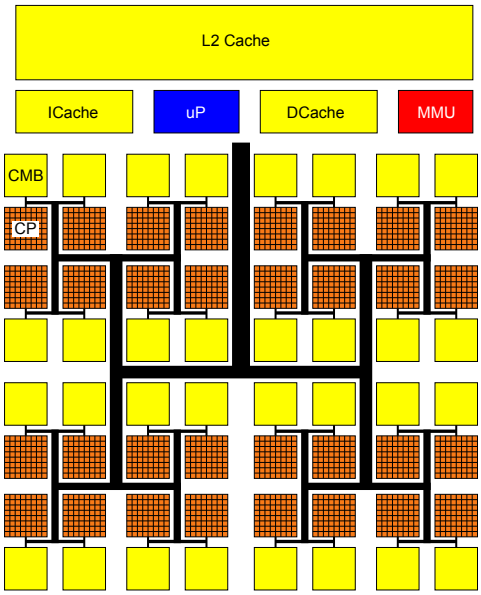

Figure 1: Hypothetical, single-chip SCORE system

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 re-compilation, 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 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.

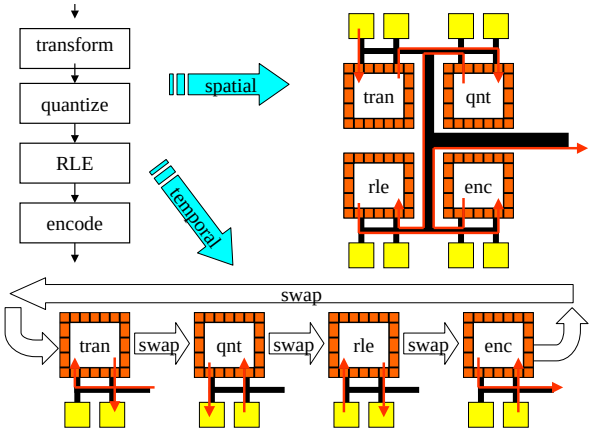

Figure 2: Hypothetical executions of JPEG Encode on a large and small device

Microarchitecture. 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 boolean 4-input lookup tables, or "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 [22] that is circuit-switched, fat-tree structured, and pipelined for high-frequency operation.

Execution. Figure 2 illustrates possible executions of a JPEG encode application on two instances of the microarchitecture described above. The application is described as a data-flow graph of stream-connected page threads (page configurations). A spatial execution may be used if the target hardware is large enough to simultaneously execute all page threads. In this case, each thread is loaded into a compute page, and each stream is mapped to on-chip interconnect. Primary inputs and outputs may be mapped to buffers in on-chip CMBs, which may be periodically flushed/refilled by the microprocessor. On the other hand, a time-multiplexed execution may be used if

```
select (input  boolean    s,
        input  signed[16] t,
        input  signed[16] f,
        output signed[16] o)
{
  state S(s):  if (s) goto T; else goto F;
  state T(t):  o=t;   goto S;
  state F(f):  o=f;   goto S;
}
```

Figure 3: TDF code for a *select* operator which selectively passes input `t` or `f` to output `o` based on the select input `s`

the target hardware is small. In this case, threads are clustered into groups, each of which is loaded in turn onto available compute pages. Any stream that outputs to a non-loaded thread is routed to a CMB for buffering; similarly, inputs from a non-loaded thread are routed from a CMB buffer. A cluster of pages may run until it exhausts its input buffers and/or fills its output buffers, at which point the next cluster of pages is loaded. The actual page loading order is determined by an automatic scheduler on the microprocessor, described below.

Compilation. We have designed a language (TDF, the Task Description Format) and associated compiler (tdfc) for specifying page threads and inter-page data-flow. A thread in TDF is a *streaming finite state machine* (SFSM), essentially an extended FSM with streaming I/O, that describes the cycle-by-cycle behavior of a compute page. TDF 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. The firing action is straight-line code; looping is accomplished by re-entering the state and re-evaluating its input signature. Figure 3 shows an example TDF thread to implement a *select* operation. TDF also includes syntax for composing persistent data-flow graphs of stream-connected page threads and supports a hierarchy of sub-graphs.

A TDF thread can be compiled into one of two forms. A thread can be compiled into compute page logic, which in the present implementation is evaluated by a device simulator. Alternatively, a thread can be compiled "for the microprocessor" as a POSIX thread with a stream API. The API allows stream communication between threads on the microprocessor and threads on compute pages. It also allows threads on the microprocessor to spawn and connect new threads.

At present, TDF threads must be explicitly written to match the hardware constraints (area, I/O, timing) of a compute page. That is, an SFSM must fit in a compute page and must execute each state action within a device cycle. In general, such device details must not be exposed to the programmer, since they tie the program to the device and thus undermine forward compatibility. We are presently working on automatic synthesis and partitioning of SFSMs, to transform arbitrarily large and complex SFSMs into groups of stream-connected, page-size SFSMs.[3] The primary challenge for page partitioning is that inter-page communication delay (stream latency) is not known at compile time. Communication delay depends on device size, page placement, and page scheduling. Communication delay may be arbitrarily large between threads that are not simultaneously executing in hardware. Our basic partitioning approach is as follows. First, we attempt to hoist code out of the SFSMs and into pipelines, so as to shrink the size and delay of the cyclic state machine cores. Resulting SFSMs that are still larger than a page are decomposed by clustering states into pages so as to minimize the frequency of inter-page state transitions. State transition frequencies can be profiled beforehand in an execution of pure microprocessor threads. SFSM area and timing is estimated using a 4-LUT area/time model of the data path components.

Page Scheduling. 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. The scheduler is responsible for reconfiguring the hardware (CPs, CMBs, interconnect), including data transfer between primary memory and the CMBs. The general policy of every scheduler is to execute communicating pages together so as to reduce communication latency (especially under inter-page feedback), to reduce the number of buffered streams, and to reduce reconfiguration frequency.

Our initial scheduler was completely dynamic, making all decisions at time-slice intervals. The advantage of a dynamic scheduler versus a static one is that it can use dynamic information about stream buffer fullness and page stalls to construct efficient, reactive schedules. We used a list-based scheduling policy to select for execution those pages with the most buffered input available, the intent being that those pages would run the longest before requiring reconfiguration. In practice, however, performance suffered from high scheduling overhead and from occasional stalling when the page loading order violated the page graph's dependence order.

To reduce run-time overhead and improve analysis, we subsequently designed three static schedulers that compute a single, repeated page loading order for each page graph.[4] The "topological" scheduler chooses a loading order to

[3]The need to resize a thread to match a page is an artifact of the reconfigurable hardware target. It may be unnecessary or optional for other hardware targets. For example, partitioning an SFSM into compute pages is somewhat analogous to restructuring microprocessor code for better locality in virtual memory pages.

[4]SCORE allows dynamic spawning of page threads and page graphs. A static schedule must be generated separately for each graph. In our actual implementation, we compute optimized, graph-specific schedules off-line using profile information collected by a previous, unoptimized execution.

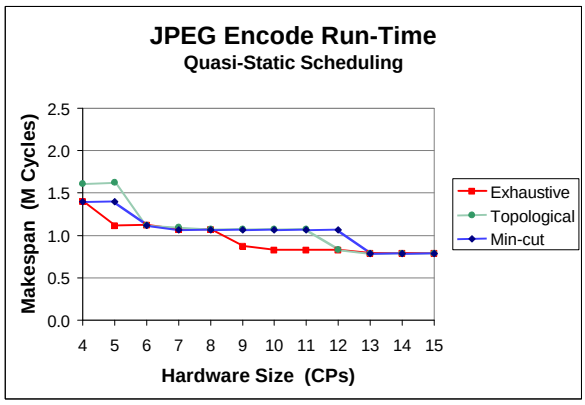

Figure 4: Run times for JPEG Encode using quasi-static scheduling

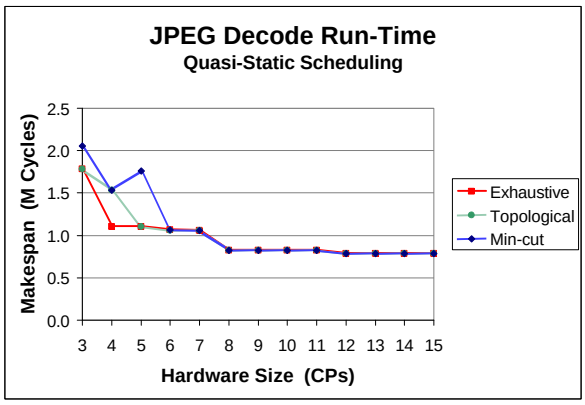

Figure 6: Run times for JPEG Decode using quasi-static scheduling

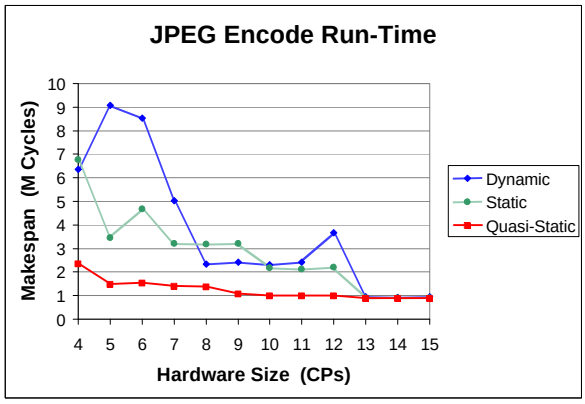

Figure 5: Run times for JPEG Encode using different schedulers ("exhaustive" static and quasi-static[5])

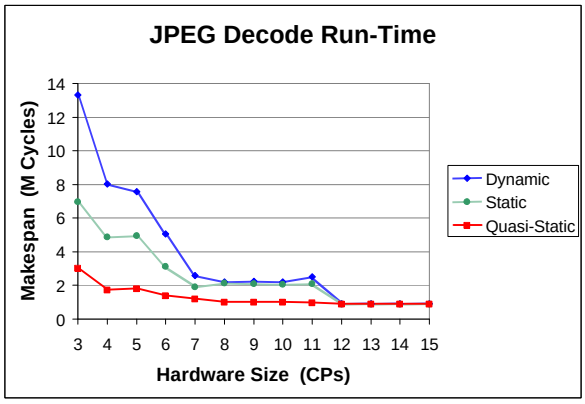

Figure 7: Run times for JPEG Decode using different schedulers ("exhaustive" static and quasi-static[5])

satisfy page precedence constraints, by topologically sorting the page graph. The "min-cut" scheduler chooses a loading order to minimize the number of stream buffers required in any time-slice, by min-cutting the page graph. The "exhaustive" scheduler chooses a loading order to minimize stalled cycles in loaded pages, by exhaustively searching the space of orderings, using profiled I/O rates to estimate input availability. Analysis (discussed below) shows that the heuristic schedulers ("topological," "min-cut") perform surprisingly well, yielding almost the same application run-times as the exhaustive search.

Interestingly, the best performance was demonstrated neither by the dynamic nor the static schedulers, but rather by a class of hybrid, quasi-static schedulers. The quasi-static schedulers extend the static schedulers with the dynamic ability to detect when all compute pages have blocked on stream access (due to empty/full stream buffers), and to immediately advance to the next time-slice. The same three policies apply: "topological," "min-cut," and "exhaustive."

[5]The astute reader will notice a small disparity between Figures 4 and 5 (also between 6 and 7) in the performance of the "exhaustive" quasi-static scheduler. The disparity is due to different accounting of overheads and is negligible for all but the smallest device sizes.

Analysis. We have written and tested several media processing applications for SCORE, including wavelet encode/decode, JPEG encode/decode, and MPEG encode. Figures 4–7 show performance results for running JPEG encode and decode (each having 15 page threads) on simulated hardware of varying page counts, using different schedulers. The device model scales the number of CMBs with the number of pages. The device simulation does not model page placement and routing. Rather, it models page capacity and a fixed network delay between pages. We refer the reader to [20] for more details on simulation, scheduling, and these application results.

Figures 4 and 6 demonstrate several interesting properties of SCORE and of the 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 highly efficient for these applications, in the sense that each application can run on substantially fewer compute pages than there are page threads, with negligible performance loss (this is analogous to efficiency in virtual memory, which allows some programs to run in smaller memory with negligible performance loss). Each 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," "min-cut") perform almost as well an exhaustive search to directly minimize idle cycles ("exhaustive").

Figures 5 and 7 compare application run times under dynamic, static, and quasi-static scheduling approaches. We find that static scheduling generally outperforms dynamic scheduling, owing in part to better analysis and in part to the lower run-time overhead of computing schedules off-line. We also find that quasi-static scheduling outperforms the pure static and dynamic approaches, typically by a factor of 2–4. This performance improvement owes entirely to the addition of a single dynamic feature, namely the ability to advance the static schedule when all compute pages are blocked.

The quasi-static schedulers presently attain hardware utilization (non-idle page-cycles) of up to 50%. This limit seems to be due to I/O rate mismatches among scheduled threads leading to input starvation and/or buffer overflows. Future work includes scheduling to run rate-matched threads together and rate-mismatched threads in different time slices. Compiler transformations may also be used to intentionally change thread I/O rates, *e.g.* using serial arithmetic.

5. 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.*), DSPs, or additional microprocessors. Each processor must have a stream network interface with ability to stall the processor. The network fabric is completely abstracted by the stream network interface.

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

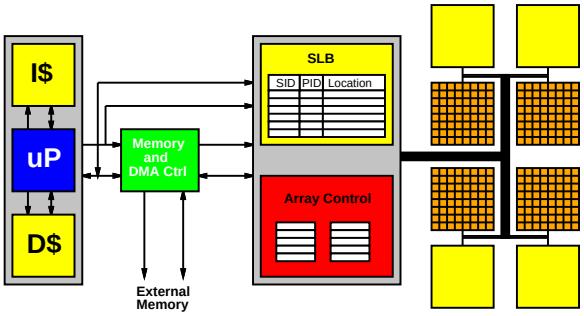

Figure 8: Hypothetical microprocessor with stream support, controlling a reconfigurable array

buffer locations can be stored in a stream table (analogous to a VM page table) and cached in a TLB-like stream look-aside buffer (SLB). The SLB enables single-cycle access to common streams and handles uncommon streams by trapping and being updated. 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. Figure 8 illustrates the hardware components for a microprocessor with stream support. Note that many of the necessary 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 multi-processor 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 context as a separate SCORE processor. Communication between threads loaded in hardware would be "live" through registers, whereas communication to threads swapped-out to memory would be buffered in memory. In either case, the scheduler prefers to co-schedule communicating threads.

6. SUMMARY

SCORE is a multi-threaded compute model built from the ground up with communication in mind to support software scalability and longevity on high capacity hardware. The model uses streams, *i.e.* FIFO channels with blocking read, as the only method of inter-thread communication and synchronization. Exposing the inter-thread dependencies allows highly flexible and efficient thread scheduling that can be tailored to available hardware, separately from thread compilation. Exposing streams as an architectural abstraction allows compiled programs to automatically benefit from additional compute and communication resources on future devices. We have demonstrated SCORE for reconfigurable hardware, which decouples a logic design from its target device size and enables binary compatibility and performance scaling on larger devices. Within that framework, we have developed several schedulers and have demonstrated heuristics that perform nearly as well as exhaustive search scheduling. We have

also proposed architectural support for SCORE streams on microprocessors. Finally, we note that SCORE extends naturally to large, heterogeneous systems, provided that every component has a compatible stream interface.

7. ACKNOWLEDGEMENTS

This research is part of the Berkeley Reconfigurable Architectures, Software, and Systems (BRASS) effort supported by the Defense Advanced Research Projects Agency (DARPA) contract DABT63-C-0048, by the California MICRO Program, and by ST Microelectronics.

8. ADDITIONAL AUTHORS

Randy Huang, Yurym Markovskiy, Joseph Yeh

9. REFERENCES

[1] Arthur Abnous and Jan Rabaey. Ultra-low-power domain-specific multimedia processors. In *Proceedings of the IEEE VLSI Signal Processing Workshop (VSP'96)*, October 1996.

[2] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The mit alewife machine: Architecture and performance. In *In Proceedings of the 22nd International Symposium on Computer Architecture*, 1995.

[3] Arvind and R. A. Ianucci. Two fundamental issues in multiprocessing. In *Proceedings of DFVLR Conference on Parallel Processing in Science and Engineering*, pages 61–88, West Germany, June 1987.

[4] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. *Software Synthesis from Dataflow Graphs*, chapter Synchronous Dataflow. Kluwer Academic Publishers, 1996.

[5] Vincent Michael Bove, Jr. and John A. Watlington. Cheops: A reconfigurable data-flow system for video processing. *IEEE Transactions on Circuits and Systems for Video Technology*, 5(2):140–149, April 1995. <http://wad.www.media.mit.edu/people/wad/cheops_CSVT/cheops.html>.

[6] Joseph T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model*. PhD thesis, University of California, Berkeley, 1993. ERL Technical Report 93/69.

[7] Eylon Caspi, Michael Chu, Randy Huang, Nicholas Weaver, Joseph Yeh, John Wawrzynek, and André DeHon. Stream computations organized for reconfigurable execution (SCORE): Introduction and tutorial. <http://www.cs.berkeley.edu/projects/brass/documents/score_tutorial.html>, short version appears in FPL'2000 (LNCS 1896), 2000.

[8] David E. Culler, Seth C. Goldstein, Klaus E. Schauser, and Thorsten von Eicken. Tam — a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, June 1993.

[9] William J. Dally et al. The message-driven processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro*, pages 23–39, April 1992.

[10] Jack B. Dennis. Data flow supercomputers. *Computer*, 13:48–56, November 1980.

[11] Thorsten von Eicken et al. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, Queensland, Australia, May 1992.

[12] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Yevgeny Gurevich Andrew Chang, and Whay S. Lee. The M-machine multicomputer. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, Ann Arbor, MI, 1995.

[13] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[14] Intel microprocessor quick reference guide. http://www.intel.com/pressroom/kits/quickref.htm.

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

[16] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The stanford flash multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302–313, April 1994.

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

[18] Whay Sing Lee, William J. Dally, Stephen W. Keckler, Nicholas P. Carter, and Andrew Chang. Efficient protected message interface in the MIT M-machine. *IEEE Computer*, 31(11):69–75, November 1998.

[19] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, and John Hennessy. Overview and status of the stanford dash multiprocessor. In Norihisa Suzuki, editor, *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 102–108. Information Processing Society of Japan, April 1991.

[20] Yury Markovskiy, Eylon Caspi, Randy Huang, Joseph Yeh, Michael Chu, André DeHon, and John Wawrzynek. Analysis of quasi-static scheduling techniques in a virtualized reconfigurable machine. In *Proceedings of the Tenth International Symposium on Field-Programmable Gate Arrays (FPGA 2002)*, February 2002.

[21] Charles L. Seitz. Mosaic C: An experimental fine-grain multicomputer. In A. Bensoussan and J.-P. Verjus, editors, *Future Tendencies in Computer Science, Control and Applied Mathematics: Internantional Conference on the Occasion of the 25th Anniversary of INRIA*, pages 69–85. Sprinter-Verlag, December 1992.

[22] William Tsu, Kip Macy, Atul Joshi, Randy Huang, Norman Walker, Tony Tung, Omid Rowhani, Varghese George, John Wawrzynek, and André DeHon. HSRA: High-speed, hierarchical synchronous reconfigurable array. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*, pages 125–134, February 1999.

[23] John A. Watlington. MagicEight: An architecture for media processing and an implementation. Thesis proposal, MIT Media Laboratory, January 1999. `<http://wad.www.media.mit.edu/people/wad/tp/>`.