The XMT Big Little Computer System: A Revolutionary Approach to Ubiquitous Parallel Computing

U. Vishkin
Students: E. Berkovich, S. Dascal, J. Nuzman
New: Prof. M. Franklin. D. Naishlos.

Alternative title

Explicit Multi-Threading (XMT): A Theorist's Proposal for a Big Little Computer System

A fascinating question:

Will a 2cm $\times$ 2cm Piece Of Silicon Provide The Fastest Computer?

Suggested order for 1st time readers of these slides: 1,2,3,10-12,3(again),9,4,13,5,22-27.
For 2nd time readers: 1-27.

Objective

Technical goal Use on-chip resources to reduce single task completion time for "general-purpose" computing.
Opportunity Next decade: "Billion transistor chip era"; $> 10^4$ increase from 1980s.
Question **"What to do with all this hardware?"**

Current uniprocessing: Limited parallelism.
Current multiprocessing: Coarse-grained SW&HW parallelism due to high overhead SW&HW mechanisms; hence programmability problems. Large inputs.
On-chip: Diminishing returns for adding device; bandwidth no problem (Dally-Lacy'99). **Need: new context & low overhead (e.g., HW) mechanisms.**
PRAM-on-chip breakthrough? 2cm$\times$2cm PRAM??

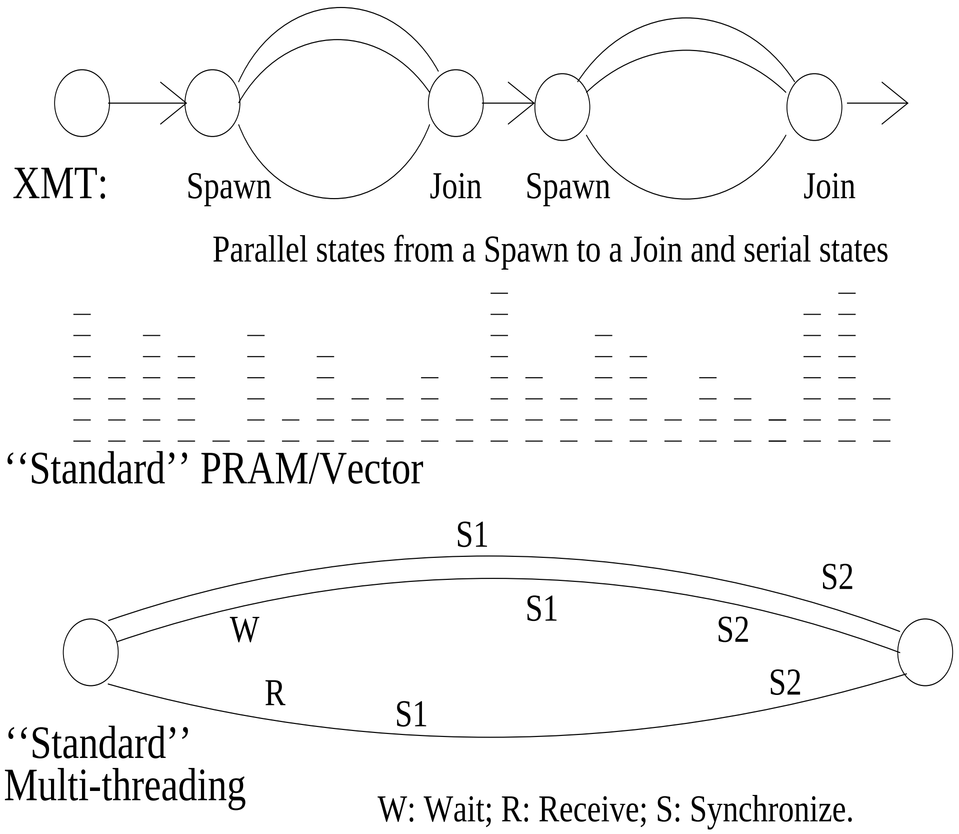
Background: PRAM (parallel random-access model) algorithmics: *knowledge-base 2nd only to serial algorithms.*
General idea: **Look to PRAM alg & MPP ideas (Ultra,NESL,etc); seek SW&HW mechanisms for more instruction-level parallelism (ILP).**
For code compatibility (without more ILP): "standard serial code" = single PRAM processor.

Spectrum of Explicit Multi-Threading (XMT) Framework

Algorithms $--$ > architecture $--$ > implementation.

- XMT: design point for fine-grained parallelism
- New elements are added only where needed

Snapshot: XMT High-level and assembly languages



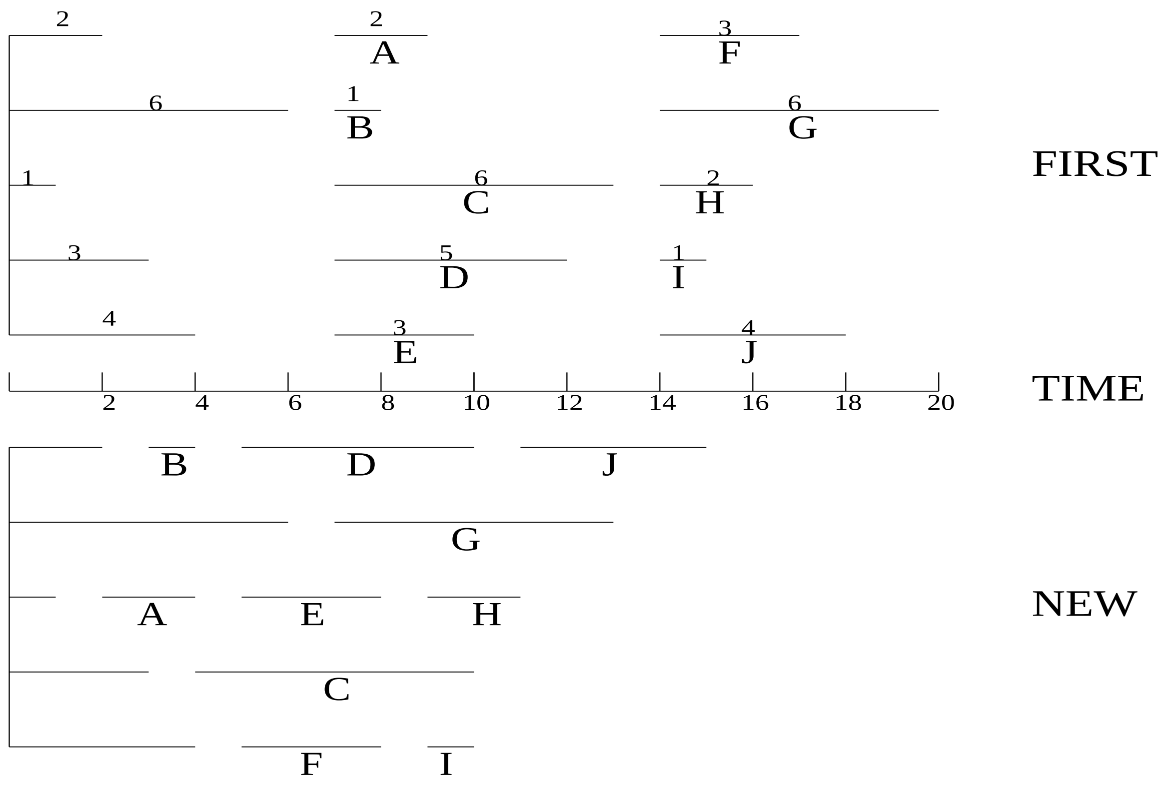
Cartoon Spawn creates threads; a thread progresses at its own speed and expires at its Join. Synchronization: only at the Joins. So, virtual threads avoid busy-waits by expiring. New: Independence of order semantics.

The implementation picture

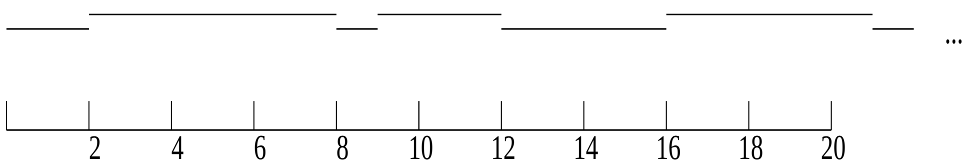
Illustrated: 5 thread control unit (5 prog. counters) executing 15 virtual threads (of one Spawn-Join).
First: Spawn more virtual threads once all current TCUs terminate.
New: **Quickly reuse freed TCUs** for remaining virtual threads.

Principle: be guided by "NBW-FSMs" ideal

(for no-busy-waits finite state machines). For progress: share work among FSMs. No delayed FSM can force other FSMs to busy-wait.



Notes: NBW-FSMs "doesn't make sense" for hardware For progress, the FSMs need to interact and exchange intermediate results: **don't** know to approach the NBW-FSMs ideal directly in hardware. Above (CS-style) idea: use cartoon (virtual design).
(Impl. may leave context of expiring threads.)
Fundamental differences with respect to Cray-vector and VLIW For performance, they couple activities whose coupling is not necessitated by program semantics; but, delay in one delays more. Such "tightly-coupled orchestration" (see paper) may not fit technological constraints: the inability to predict response time to memory requests.
Processor for Tera MTA Suppressed: (i) threads may be shorter (ii) multiprocessing.



EXPLICIT MULTI-THREADING (XMT) FRAMEWORK

COMPONENTS	SOME SPECIFICS	PRINCIPLE
PRAM-Style Algorithms	Existing Arbitrary CRCW PRAM Modified/new (APRAM-like)	Virtual No-Busy-Wait Finite-State-Machines (NBW-FSMs)
SPMD Programming	Spawn-Join Independence of Order Sematics (IOS) Synchronization only at Joins	
SPMD Assembly Language	3 multioperand primitives added Explicit inter-thread parallelism Standard intra-thread parallelism	
Decentralized Implementation	Spawn, Join, Prefix-Sum in ‘‘constant’’ time Enhanced thread spawning to benefit from NBW-FSMs	Approach NBW-FSMs

XMT EXTRAs

- Latency hiding (standard).
- Extension to multi-processing (standard).
- Register file organization (some new).
- Hardware supported nesting of Spawns.

WHY NOT XMT?

May encourage algorithms with higher data movement.
Proposed answers:
- "On-chip" promises high bandwidth and low latency communication.
- 64MB DRAM on chip memory needs 0.5B transistors (IRAM).

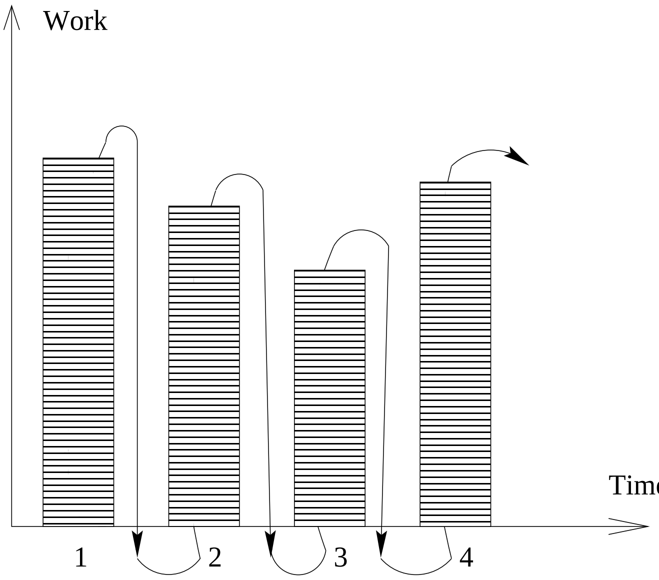
WHY XMT?

- The PRAM knowledge base.
- Scheduling Lemma.
- Empirical results: speed-ups with respect to standard serial (e.g., superscalar, VLIW).
- Competitive already for small inputs (more than MPPs).
- Simpler hardware: parallelism mandated by code.
- Decentralized impl.: modular, hierarchy, pipelining.

Traditionally, PRAM algorithms

(i) operate in lock-step fashion, and
(ii) satisfy serial semantics.

Namely, an algorithm is described in steps, each comprises parallel operations. *Lock-step* means: all operations of a step terminate; then operations of the next step begin. Total order among the operations of a step extends to total order (*"serial semantics"*) among all operations. (The k operations of a step are given ordinal numbers from 0 to $k-1$; if the hardware can execute only $p < k$ operations at a time, operations 0 through $p-1$ are performed first, then operations p through $2p-1$, and so on.)



Lock-step and serial semantics

Important goal

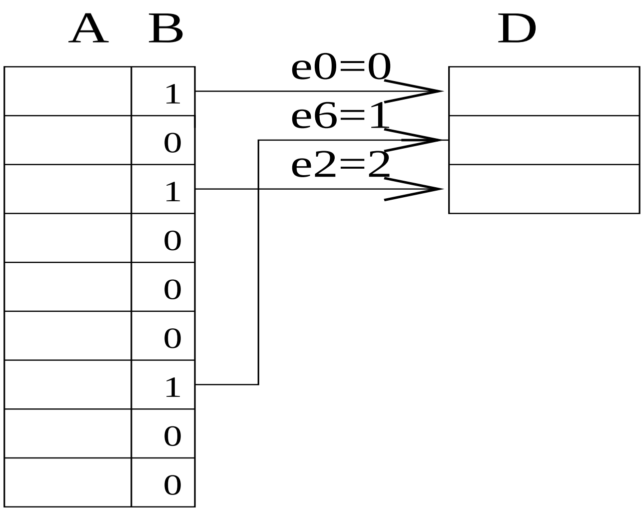
See that PRAM algorithms are translatable into assembly code where
(1) PRAM parallelism maps into a thread structure;
(2) assembly language threads are not-too-short (to increase locality of reference); and
(3) the threads satisfy IOS.

For this, observe:
(i) An instruction in a high-level language (or "granularity level of PRAM algorithmic thinking") translates into several assembly language instructions. The programming of PRAM algorithms (or the algorithms themselves) may also contribute to thread length.
(ii) Arbitrary CRCW PRAM generally **sufficient** for PRAM algorithms: one among concurrent-writes into the same memory succeeds; so, one thread could write into the memory location while the others bypass to their next instruction.

"Artificial" Example

The array compaction problem

Input: Array $A = A(0), \ldots, A(n-1)$ of elements; 2nd array $B = B(0), \ldots, B(n-1)$ of bit values (0 or 1).
For each i, $0 \leq i \leq n-1$ for which $B(i) = 1$, copy $A(i)$ into a different entry of a 3rd array
$D = D(0), \ldots, D(s-1))$; s number of ones in $B(i)$, over $0 \leq i \leq n-1$. Mapping into D need not be order preserving.



The array compaction problem.
Array B: bit values for the compaction.
For the program below: e0, e1 and e6
are the e values for threads 0, 2 and 6; x is 3.

XMT-C: High-level language

Single-program multiple-data (SPMD) extension of standard C. Includes Spawn and multioperand instructions (e.g., PS).

Algorithm level (high-level program)

$int\ x\ =\ 0;$
$Spawn(0, n)$ /* Spawn n threads; $\$$ ranges 0 $\rightarrow n-1$ */
$\{\ int\ e\ =\ 1;$
  $if\ (B[\$] == 1)$
    $\{\ PS(x, e);$
      $D[e]\ =\ A[\$]\ \}$
  $\}$
$n = x;$

Notes: (i) PS is explained next (think F&A). See results for e0,e2, e6 and x. (ii) Using C-style scoping, Join instructions are implicit.

XMT Assembly Language

Standard assembly language, plus 3 new instructions: Spawn, Join, and multi-operand (e.g., PS).

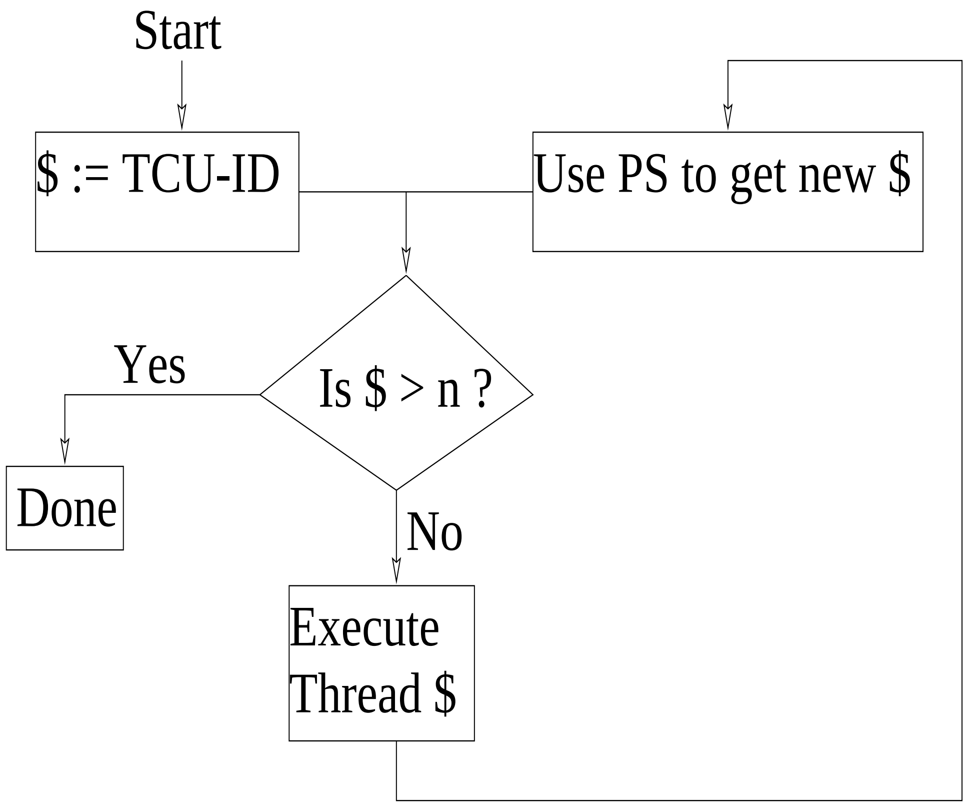
The PS multi-operand instruction

New kind of instruction: *Prefix-sum* (PS).
Individual PS, $PS\ Ri\ Rj$, has an inseparable ("atomic") outcome: (i) Store $Ri + Rj$ in Ri, and (ii) store original value of Ri in Rj.
Several successive PS instructions define a multiple-PS instruction. E.g., the sequence of k instructions: $PS\ R1\ R2$; $PS\ R1\ R3$; ...; $PS\ R1\ R(k+1)$
performs the prefix-sum of the *base* $R1$ and the *elements* $R2$, $R3$, ..., $R(k+1)$ to get: $R2 = R1$; $R3 = R1 + R2$; ...; $R(k+1) = R1 + ... + Rk$; $R1 = R1 + ... + R(k+1)$.
Several ind. PS's can be combined into one multi-operand instruction.
Executed by a **new multi-operand PS functional unit**.
Example for multiple-PS: same-base ind. PS from a different thread; due to IOS, any order is acceptable.

Decentralized implementation in a nutshell

Given is a Spawn (of n threads). How does the hardware know which virtual thread to execute next?

The program of a thread control unit (TCU)



For now only: TCU similar to processor in an MPP.
Details follow.

Example cont'd: instruction set (assembly lang.) level

SPMD language. *Spawn* initiates n virtual threads 0 through $n-1$. Each thread can have local registers. Global registers accessible by all threads.
MIPS-like instructions: $lw\ R1\$,\ 0(R3)$ loads from memory address in global register $R3$ into register $R1\$$, which is local to thread $\$$. Instruction can be decoded (for execution) into n specific instructions, one for each thread.

	Instruction	
	...	
	li	R1,0
	move	R2, Rn
	Spawn	R3,0,R2,3
	li	R1$,1
	lwa	R2$,B_OFF(R0), 4[R0$]
	bne	R2$,R1$,L
	PS	R1,R1$
	lwa	R3$, A_OFF(R0), 4[R0$]
	swa	R3$, D_OFF(R0), 4[R1$]
$L:$	Join	R3, R2
	move	Rn, R1
	...	

Notes

Load-immediate (li) initializes $R1$ to 0. n is assumed to be in Rn; move copies Rn into $R2$. Spawn spawns $R2$ virtual threads indexed 0 to $R2$-1; use 3 local registers per thread; the Join matching the Spawn counts terminating threads into $R3$. $R0\$$ always includes the thread index $\$$; it will be a read-only local register. As in MIPS: $R0 = 0$. B_OFF is the base address (i.e., offset) for array B. A "Scaled"-like Load–word-array (lwa) instruction for directly accessing array addresses; multiplier of 4 since MIPS addressing is done in bytes (assume, each array entry is 4 bytes). If $R2\$$ equals 1, a (parallel) PS (see next) increments counter $R1$; $R1\$$ provides address into which to copy $A(\$)$; $A(\$)$ copied into compacted array D using lwa and store-word-array (swa) instructions. Each thread reaching Join increments $R3$ by one using a parallel-sum integer instruction (SUMI), as part of Join; once $R3$ equals $R2$, Join switches to serial state. Size of compacted array copied into Rn.

Version of example in paper: more believable that threads running time is "unpredictable".

More notes

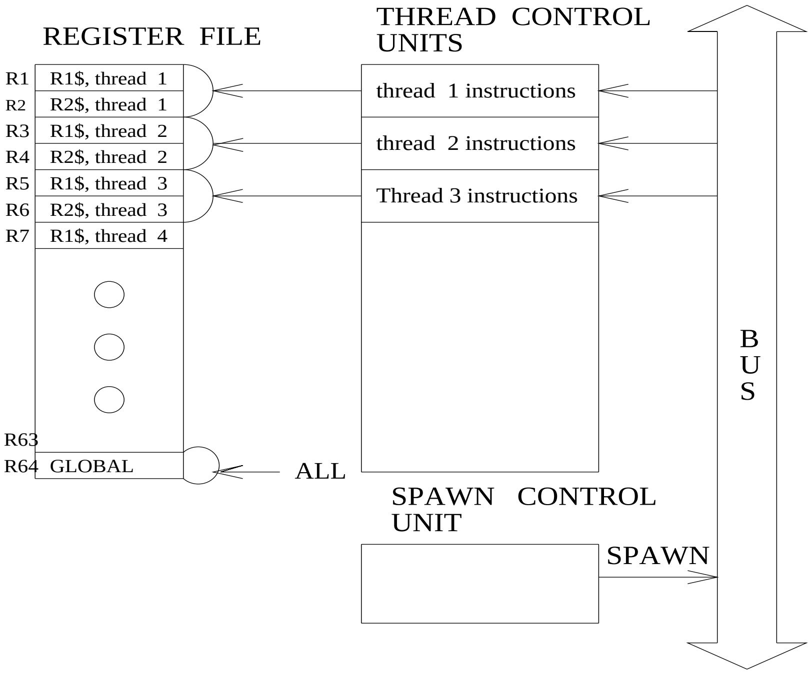
Observation (V-SPAA97) If $\leq k$ operands, for some hardware dependent k, and every element is a small integer (say, between 0 and 3), multiple-PS can be implemented (in hardware) very efficiently ("unit time"); the base can be any integer.

Implementing arbitrary concurrent-writes

using multiple-PS. Suppose: concurrent-writes into some global address may occur. Have dedicated "gatekeeper" address. Initialize to 0. Prior to each write use individual-PS to increment gatekeeper by 1. One attempting thread gets 0 from gatekeeper and performs the write. Other threads proceed directly to the next instruction. No busy-waits occur. (Actually, a simpler multioperand instruction is used.)

Surprise Optimized serial code: more involved and harder to follow. Easier to write parallel compiler?!

Highlights of a possible implementation



Datapath, control units, registers and a Spawn command

t (physical) thread control units (TCUs). Each has: own program counter (PC); local registers. Shared (global) registers. In serial state, only the first TCU is active. Spawn causes transition into parallel state; actual spawning done by Spawn control unit (SCU). The SCU first broadcasts the Spawn on a bus which connects to the TCUs. If the Spawn specifies $\leq t$ virtual threads, they are all initiated into the TCUs. If $> t$ virtual threads are specified: (a) The first t virtual threads are initiating into the TCUs. (b) The SCU will be responsible to issue the remaining threads.

Instruction supply: using the bus, each instruction in the instruction code is duplicated at each TCU executing a thread.

A *thread terminates* at a Join command. The Join implementation monitors the number of threads reaching the Join command using a dedicated *parallel sum functional unit* at the SCU. Once all threads terminate, the first TCU is activated in serial state.

• Inter-thread parallelism. The case where several TCUs contribute instructions issued at the same cycle.
• Intra-thread parallelism. The case where a TCU contributes more than one issued instruction to a cycle (as in current superscalar uniprocessors). In practice, TCUs could be divided into groups. TCUs of a group can share functional units (as in U. Washington's SMT project, for OS-driven enhancement of processor throughput).

Assume: Spawn (or PS) takes unit time for a hardware dependent number of operands. The assumptions are robust in the following sense: the primary objective of our empirical performance studies is comparison with the serial algorithm; and not observe absolute performance results.

Hiding latencies: for simplicity, suppose the hardware can issue (any) p instructions per cycle coming from $\leq$ p

TCUs; suppose an instruction takes between 1 and d time units (due to various arithmetic and floating point operations, or cache misses for data supply).
if $t = pd$ we can issue p instructions per cycle, based on inter-thread parallelism alone; i.e., having $\geq t = pd$ threads for a Spawn at the algorithm level guarantees issuing p instructions per cycle, as long as at least pd virtual threads have not yet terminated.

Register file organization for aggressive ILP is doable.

Thread spawning

"Standard" thinking: spawn new threads once all current TCUs terminate. When some TCUs terminate, but others continue, a "gross violation" of the NBW-FSMs ideal occurs.

New mechanism for enhancing thread spawning

using prefix-sum: quickly reuse freed TCUs for remaining virtual threads. Uses hardware implemented parallel prefix-sum (and relies on the IOS semantics). Suppose the execution of several threads terminated; each of the TCUs that ran them can take a new thread; the SCU is waiting to spawn new threads.
To allocate new thread indices to these TCUs: (1) The hardware performs PS among these TCUs to get for each its serial number relative to these TCUs.
(2) Broadcast a machine Spawn instruction (called, Spawn-Recur) on the bus; result: each available TCU acquires a thread index. Until all virtual threads of a Spawn were initiated: following each cycle in which some TCU terminated, allocation of new threads to just freed TCUs.

Bridging Models

Similar to the "von Neumann machine", Valiant-90 and Vishkin-84:
(i) The diverse software industry can aim to write programs that can be executed efficiently on the bridging model, without explicit consideration of the hardware.
(ii) Hardware designers could have shared the common goal of realizing the von Neumann machine without having to be concerned with the software that it is going to execute.

The Spawn Multi-Threading (*Spawn-MT*) bridging model operates in two states: parallel and serial. The parallel state has an asynchronous mode and a synchronous mode. Each mode is described at two levels:

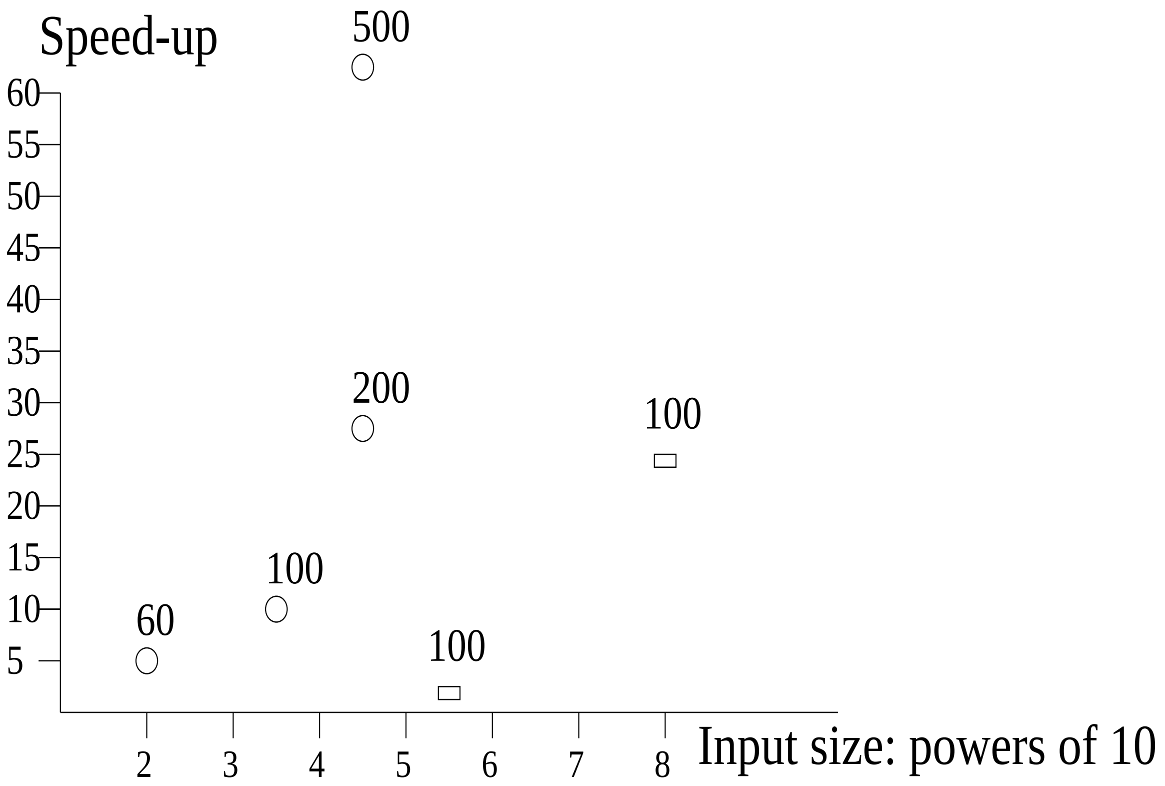
algorithmic model level and instruction-set level.

Above description highlights one mode-state-level triple: the *asynchronous mode, parallel state* and *instruction set level*.

Spawn-MT doesn't allow nesting of Spawn commands. Elastic Multi-Threading (*EMT*) allows nesting; nesting will be handled by the hardware.

Empirical Results

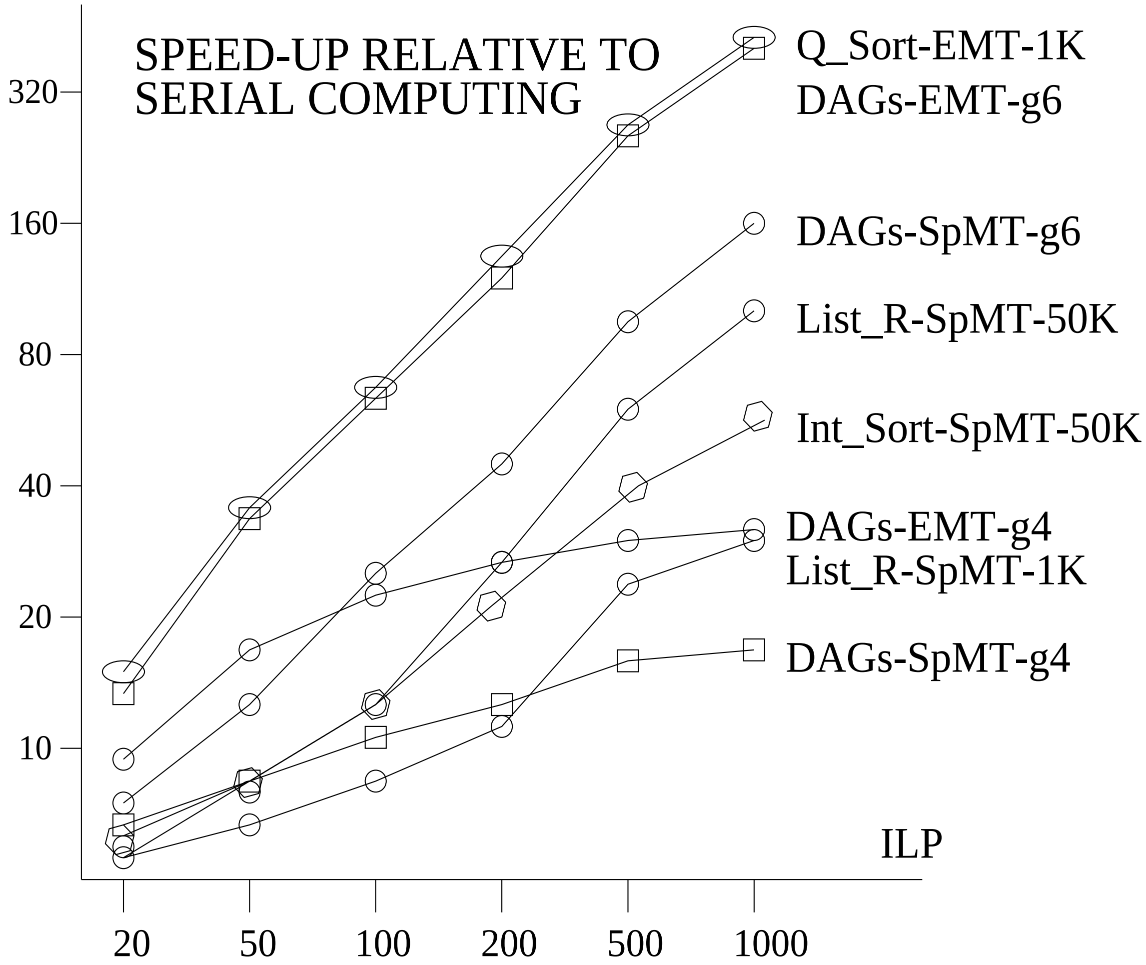
• **MPP versus XMT**. List ranking results for the Paragon reported in Sibeyn-97 versus our XMT results.



LEGEND: 100 □ Means Multi-processing with 100 processors.

200 ○ Means XMT with ILP of 200

• Relative to serial computing.



Pleasant postscript: Work in progress with a UMD micro-architect confirms results using their simulation tools.

Note

• Execution model: reflects intra-thread parallelism, cache misses, prefetching.
• Compiler feasibility.
• A **conservative comparison approach**.
Avoid unfairly favoring the performance estimates of the ILP code over standard serial code. Concretely, (1) For standard serial code, we took its (parallel) T as the running time. Rationale: this is a **lower bound** on the (parallel) execution time achievable.
(2) For parallel code, we took its $W/p + T$ as the running time. Rationale: this is an **upper bound** on the execution time achievable.
• The XMT paradigm is broader (more comprehensive) than this talk could convey. Please look up the paper, etc.:
http://www.umiacs.umd.edu/~vishkin/XMT/

Summary of the XMT design point

Context: From work-time algorithm description **to** HW implementation. (Extend [SV-MaxFlow82].)
Low-overhead mechanisms:

SW1 SPMD XMT-C. IOS.
SW2 Concurrent spawn of many threads.
SW3 Multi-operand PS instruction.

ARCH1 SPMD XMT-assembly. IOS.
ARCH2 Distributed concurrent Spawn&Join of many threads. Quick reuse of freed TCUs.
ARCH3 Multi-operand PS FU (64-bit base, 64 1-bits).
ARCH4 Spawn: bus-broadcast of instruction code.

ALG1 PRAM algorithmics. Arbitrary CRCW → IOS.
ALG2 Longer threads: C → assembly, thread consolidation, other programming.

GLUE NBW FSM ideal as guiding principle.
OVERALL EFFECTIVENESS Scheduling lemma, empirical results. Also see slides 6&7.

Conclusion

- **M$?: PRAM alg implementable?**
Not a minute too early to try (even if theorist...).
New ground rules; rich MPP&serial idea base.

- Tangibles Given parallel BFS & DAG search, how much parallelism reaches implementation?
By XMT Scheduling Lemma: competitive!

- Using multiple program counters (Multiscalar)
SMT: same cycle execution of instructions from different programs.
None aspires/claims to support PRAM-like algorithms (no scalable mechanisms for impl. very fine-grained SPMD code).

- Need to exploit parallelism due to fundamental limits; not just technological predictions as when funding focused on parallel computing! Signal at half the speed of light advances 7.5cm per a 2GHz clock. Chips exceed 2cm on the side. Also, miniaturization reduces speed! SIA roadmap: soon, only 16% of a chip will be accessible in a clock cycle! Indeed: Intel/HP Explicitly Parallel Instruction Computing (EPIC) VLIW technology (Merced, 2000?).

• Related empirical work measuring ILP in elegant trace-based models. Analysis needed unrealistic assumptions (e.g., perfect memory disambiguation). Make gap between real figures and their reported upper bounds unclear, and so are resulting comparisons. No such problem with our conservative approach.

• Why care for speed-ups for smaller inputs?
Hypothetical numeric example S - base serial system. Compare A&B, two parallel systems. Suppose: For 95% of the running time on S, system A runs 100 times faster; for 5%: running time unchanged. System B runs well also on smaller inputs. For 99% of the running time on S, system B runs 100 times faster; for 1%: running time unchanged. How do systems A&B compare? Factor: $\doteqdot 3$!

• Quest for "paradigm shift" (T. Kuhn) in CS&E
Find a general-purpose way for minimizing computation completion time by way of parallelism.

• Last but not least. Inspired by (partial list): PRAM, BSP/LogP, Cilk&MILAN, Cray, DataFlow, Hennessy Patterson'96, HEP/Tera, HPF, IBM360, NESL, NYU-Ultra, RISC, SB-PRAM&FORK, SUIF, VLIW.