Compiler Support to Improve Work Stealing Scalability

Miloš Puzović, David Greaves
Computer Laboratory, University of Cambridge

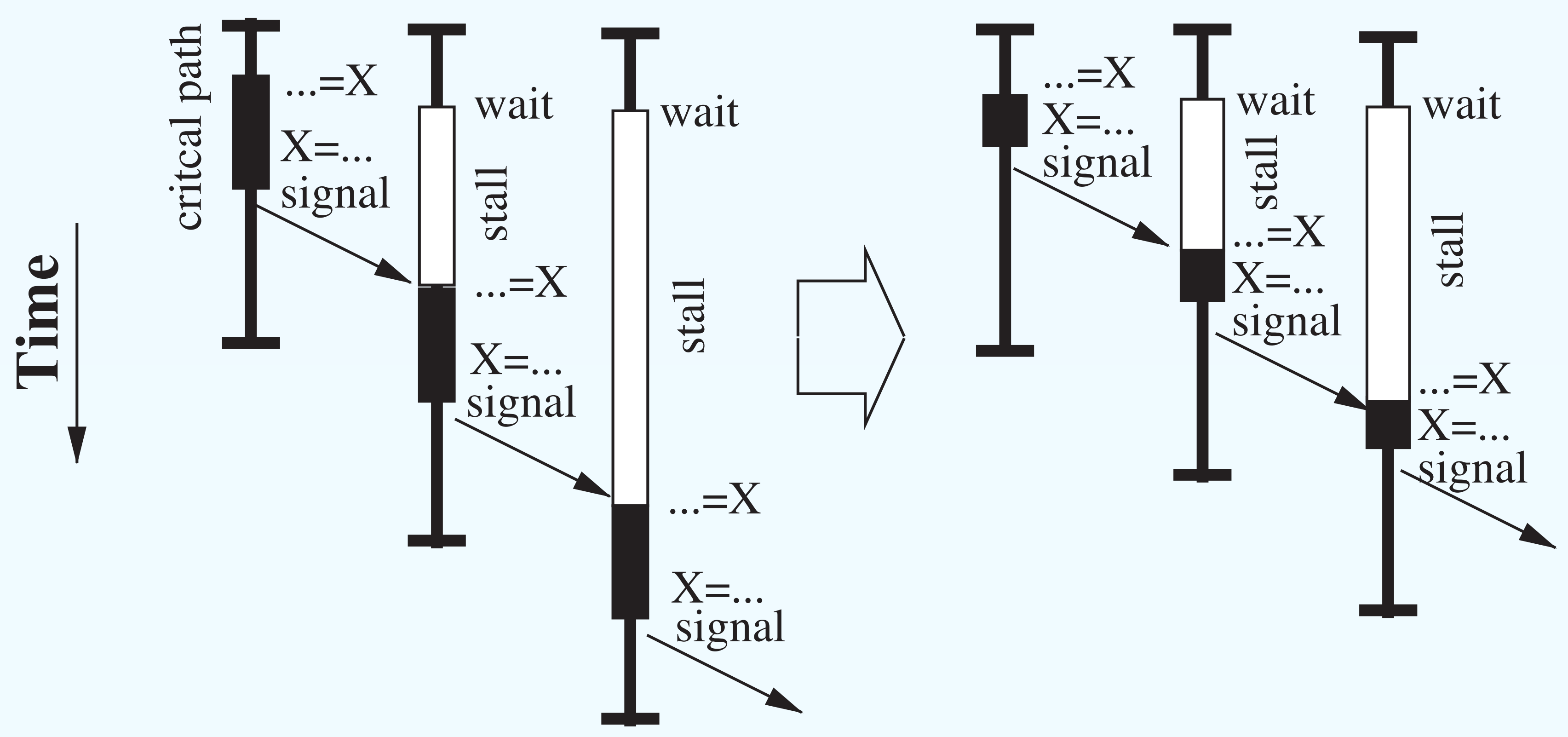
Motivation

The processor scheduling problem is to **map** a set of **precedence constrained tasks** T_i where $i = 1, \ldots, n$ onto a set of **processors** P_k where $k = 1, \ldots, p$ such that a specified objective function, for example, schedule length, mean flow time or processor idleness is *minimized*.

	assignment	**ordering**	**timing**
fully dynamic	run-time	run-time	run-time
static allocation	compile-time	run-time	run-time
self-timed	compile-time	compile-time	run-time
fully static	compile-time	compile-time	compile-time

Beyond assigning computation nodes to processor we also need to allocate communication resources for interprocessor data transfers. **We shouldn't separate these two concerns.**

Critical forwarding path [2]



A compiler can schedule instructions to reduce the *critical forwarding path*, therefore increasing parallel overlap and improving performance by placing **wait** instructions as late as possible and **signal** instructions as early as possible in the code.

Work Stealing

Work-stealing is a provably efficient, fully-dynamic scheduling algorithm whose expected time to execute a fully-strict, multi-threaded computation on P processors is:

$$E(T) = \frac{T_1}{P} + O(T_\infty) \qquad (1)$$

where T_1 is the minimum serial execution time of the multi-threaded computation and T_∞ is the minimum execution time with an infinite number of processors. [1]

When the number of available processors P is no more than the *average available parallelism* T_1/T_∞ the first term in equation (1) dominates the second term and therefore we obtain *linear parallel speedup*. If the number of available processors increases then $P = \omega(T_1/T_\infty)$, hence second term in equation (1) dominates. Therefore the work stealing algorithm stops scaling efficiently as we add further processors.

We investigate how to reduce computation depth (T_∞) by reducing the *critical forwarding path* between the first use and last definition of dependent values.

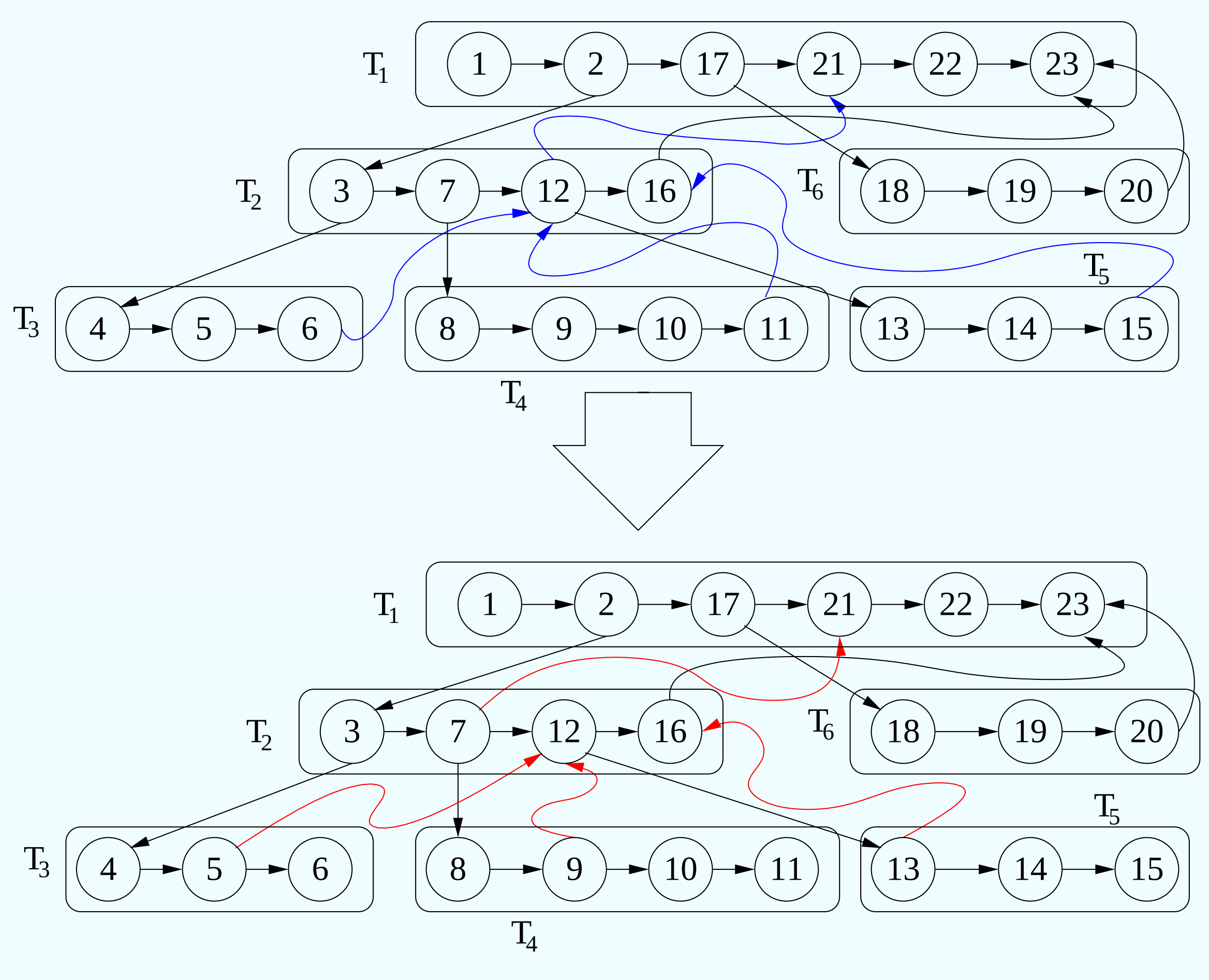
Accelerating value communication

Data-flow analysis conservatively assumes that all execution paths are taken. In practice, however only a small number of paths is frequently executed.

We use ideas from *deferred-data flow analysis* [3] to aggressively schedule instructions on the critical forwarding path by speculating on control and data dependencies. If the incorrect value has been passed we *squash* speculated thread and restart it with correct value. When speculation is correct we improve performance by increasing overlapped execution of instructions from multiple threads, while with incorrect speculation we still indirectly improve performance by increasing *instruction-level parallelism*.

Example

step	processor activity					
	p_1		p_2		p_3	
1	T_1 :	1				
2		2				
3	T_2 :	3	T_1 :	17		
4	T_3 :	4	T_6 :	18	T_2 :	7
5	T_3 :	5	T_6 :	19	T_4 :	8
6	T_3 :	6	T_6 :	20	T_4 :	9
7					T_4 :	10
8					T_4 :	11
9					T_2 :	12
10	T_1 :	21			T_5 :	13
11	T_1 :	22			T_5 :	14
12					T_5 :	15
13					T_5 :	16
14					T_5 :	23



step	processor activity					
	p_1		p_2		p_3	
1	T_1 :	1				
2		2				
3	T_2 :	3	T_1 :	17		
4	T_3 :	4	T_6 :	18	T_2 :	7
5	T_3 :	5	T_6 :	19	T_4 :	8
6	T_3 :	6	T_6 :	20	T_4 :	9
7	T_1 :	21	T_2 :	12	T_4 :	10
8	T_1 :	22	T_5 :	13	T_4 :	11
9	T_2 :	16	T_2 :	14		
10	T_1 :	23	T_2 :	15		

The graph above demonstrates an example when multi-threaded computation would benefit from reduced critical forwarding path. *Data-dependency edges* are represented by the blue and black curved edges and the table on the left is its 3-processor execution schedule. When our analysis is applied we replace blue curved edges with red curved edges and obtain the 3-processor execution schedule shown on the right. As it can be seen overall performance and processor utilisation is improved. Furthermore, there is now enough independent work to be able to use 4 processors at the same time.

Evaluation

Multi-threaded programming is usually done by library of threading primitives thus compilers are designed independently of threading issues. We propose to bridge this gap by extending LLVM Intermediate Representation (IR) with parallel primitives. Our compiler infrastructure will output LLVM bitcode and we will evaluate our analysis using PARSEC and SPECInt2000 benchmarks on a detailed machine model that is being developed as a part of the Communication-Centric Computer Design (C3D) project.

Future work

We are looking at different high-level programming models that can effectively map to many-core architectures: stream programming and join calculus. The important aspect from these models that we are interested in is amount of information we can obtain at compile-time to reduce amount of work we need to do at run-time. Also, we plan to consider what effect knowledge of underlying architecture has on scheduling and if it improves performance how to package it and pass to the compiler.

References

[1] R. D. Blumofe, C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing In *Journal of the ACM (JACM), v.46, n. 5, p. 720 - 748, September 1999*
[2] A. Zhai et al. Compiler and Hardware Support for Reducing the Synchronization of Speculative Threads In *ACM Transactions on Architecture and Code Optimization, May 2008*
[3] S. Sharma, A. Acharya, J. Saltz, Deferred Data-Flow Analysis. Technical Report, UMI Order Number: TRCS98-38., University of California at Santa Barbara