Explicit Multi-Threading (XMT) Bridging Models for Instruction Parallelism *(extended summary and working document)

Uzi Vishkin, Shlomit Dascal, Efraim Berkovich and Joseph Nuzman

July 7, 1998

Abstract

The paper envisions an extension to a standard instruction set which efficiently implements PRAM algorithms using explicit multi-threaded instruction-level parallelism (ILP); that is, Explicit Multi-Threading (XMT), a fine-grained computational paradigm covering the spectrum from algorithms through architecture to implementation is introduced; new elements are added where needed.

The more detailed presentation is by way of a bridging model. Among other things, a bridging model provides a design space for algorithm designers and programmers, as well as a design space for computer architects. It is convenient to describe our wider vision regarding "parallel-computing-on-a-chip" as a two-stage development and therefore two bridging models are presented: Spawn-based multi-threading (Spawn-MT) and Elastic multi-threading (EMT).

The case for Spawn-MT (or, alternatively, EMT) as a bridging model relies on the following evidence. (1) Spawn-MT comprises an "instruction set level", whose hardware implementability is justified. (2) Parallel and serial algorithms and programs are presented at both the "algorithm level" of Spawn-MT and its instruction set level. (3) Careful comparison of the new kind of code with standard serial code, for several ("irregular") test applications, illustrates a large potential advantage for Spawn-MT (and EMT); this advantage holds even for rather small inputs. One of the applications (searching DAGs) is general enough to be which is reflected in every computation but does not explicitly appear in standard parallel benchmarks. (4) Overall hardware realization is likely to be simpler than for standard approaches as more parallelism will be mandated by code, alleviating the need for the hardware to find parallelism in the face of control dependences and ambiguity of data references. (5) A certain ("independence of order") parallel algorithms semantics, the bridging model and support by several hardware implementation gadgets, put together, facilitate independent progress of concurrent threads; each thread progresses at its own pace from initiation to termination without the need for suspending concurrent threads. A scheduling lemma attests to the effectiveness of this overall design.

*Supported by NSF grant 9416890.

Contents

1. PREVIEW AND INTRODUCTION

A current trend in processor design is to increasing ILP - the number of instructions whose executions overlap in time. This trend is presenting ever greater challenges to compiler writers and others in their effort to find increasing amounts of parallelism in sequential code. This raises the following question: can programmers and algorithm designers more easily find the parallelism to effectively employ the ILP resource? This parallelism is fine grained, by contrast with the relatively coarse grained parallelism found on present parallel machines. Thus we are examining models of computation that (a) require only a small number of additional instructions that can be reasonably implemented, and (b) support fine grained parallelism. As it happens, there has been extensive study of fine-grained parallel algorithms, namely in the PRAM model.

The opportunity It is widely believed that we will have a Billion transistor chip within the next decade. "Parallel computing on a chip" provides a unique opportunity for faster single task computing, since it offers the following. (i) *High bandwidth communication.* This has also been possible for the more standard multi-processing approaches. (ii) *Low latency communication.* This promise contrasts with "massively parallel processors", where latency and send+receive overheads are measured in micro-seconds; see e.g. [HP96], p. 602; note that use of new multi-operand functional units for implementing communication functions in the current paper could reduce latency further. That is, low latency communication would mean a new reality for parallel computing; this calls for reexamination of premises, rationales and practice. (iii) *Better performance already for small input size.* That is, items (i) and (ii) above allow a parallel algorithm to become competitive with its serial counterpart for a much smaller input size than for other parallel system paradigms. Typically, the parallelism provided by a parallel algorithm increases as the input size increases. When implemented on a parallel computer system part of this algorithm parallelism is used for hiding system deficiencies (such as latency). When latency is a smaller problem, more algorithm parallelism can be applied directly to time speed-ups. So, parallel algorithms can become competitive already for *much smaller inputs.* For some algorithms, smaller inputs mean applicability to practical input sizes, as opposed to astronomical sizes that do not occur often in practice. (iv) *Explicit parallelism for simpler hardware.* Explicit ILP generally means "static" use of ILP, mandated by instruction code, which allows for simpler hardware than for dynamically (i.e., by hardware) extracted ILP. Stating simpler hardware as the main motivation, industry has demonstrated its interest in explicit instruction-level parallelism (ILP), by way of heavily investing in it.

This paper presents a preliminary exploration as to the extent to which relying on parallel algorithms could yield *greater hardware simplicity and performance gains.* The paper focuses on the interplay between parallel algorithms and instruction-level parallel architectures, trying to interject an algorithm-driven point of view. A number of issues need more study and a variety of expertise.

Organization of the paper For most readers, it will be best to first read a much more concise description of the XMT computational paradigm, as given in the "extended abstract" version of this paper [VDBN98]; it can be downloaded from a web site listed in Appendix A. For reading the current version it may be useful to first read the perspective in Section 10. Section 1.1 begins with an overview of some features of the new XMT computational paradigm, and is followed by an explanation of the bridging model concept as used in the current paper in Section 1.2. This is followed by an overview of the main contributions in Section 1.3 and the Spawn-MT model in Section 2. A detailed example for how a program and its instruction code may look in Section 3 is followed by a more elaborate description of the instruction set in section 4 and a very brief reference to another mode in section 5; note that a convenient quick reference guide is included in Appendix B. The EMT model is briefly described in Section 6. After presenting an execution model, and a scheduling lemma in Section 7, the empirical results are reviewed in Section 8. Section 8 also references the appendices which provide more information; fine details of the experiments can be downloaded from the web site in Appendix A. Relying on [P+97], Section 9 notes that a billion transistor chip would allow 64MB of on-chip DRAM memory, which can greatly enhance our envisioned approach; the section sheds some more light on our overall approach, discusses some possible extensions, and includes references to work on improving performance of registers in Section 9.1. This is followed by a comparison to the literature in Section 11. We conclude with revisiting the new ideas presented, and relating our direction to a fundamental parallel computing problem and to industry's input in Section 12.

1.1. The Explicit Multi-Threading (XMT) Computational Paradigm - Salient Features

The next paragraph begins with a "wishful thought" computer organization design ideal, and an explanation how the ideal is approached by a (virtual) design of programming and assembly languages. Following a preview of the XMT paradigm, PRAM algorithms and their translation into assembly code are discussed. Review of the assembly language, the high-level language, and the implementation. This section ends with mentioning execution models.

For **general motivation**, we first discuss two principled approaches for coping with complex designs. We then explain the interesting way in which one of them ended up guiding the view of the computer organization part in the XMT computational paradigm.
(i) *Tightly-coupled orchestration* of all components of a system at all levels as a design ideal (for better performance). One instance of the ideal, which is metaphorically referred to as *"the Swiss train system"*, is where a carefully designed overall schedule provides punctual orchestration of all components; for example, this instance appears to have guided the ingenious design of the Cray vector computers. This approach could work perfectly if no component ever fails to meet its schedule. However, it could incur drastic efficiency degradation even if only one component at any level is delayed. VLIW architectures appear to have also been guided by the tightly-coupled orchestration ideal. Several (otherwise unrelated) instructions are batched into a long instruction. This can be advantageous if all instructions execute promptly. However, delayed completion of one instruction could delay more instructions than necessitated by the original program, adversely affecting potential performance advantages. The tightly-coupled orchestration ideal is intuitively very appealing. However, technological constraints have necessitated a departure from that ideal in currently successful processors. An example for such a constraint is the inability to predict the response time to memory requests. One reason for such unpredictability is the memory hierarchy. Our search for an applicable ideal led to the following alternative.
(ii) The *no-busy-waits finite state machines (NBW FSMs)* as a design ideal. Progress is achieved by sharing the work among FSMs; however, a delayed FSM cannot suspend progress of other FSMs.

The latter means that no FSM can force other FSMs to waste cycles, by having to do busy-waiting. The difficulty is of course that FSMs working towards a common goal often need to interact with and rely on intermediate results of other FSMs. For this reason, we do **not** know how to approach the no-busy-waits FSMs ideal directly in hardware.

Instead, **the no-busy-waits FSMs ideal guided the design of our high-level programming language and assembly language**. Note that FSMs that will be reflected in these languages are **virtual**. We extended standard instruction sets (for assembly languages) to enable transition back and forth from serial state to parallel state and allow in the parallel state any number of (virtual) threads. A Spawn instruction marks a transition from a serial state to parallel state and a Join instruction marks a transition in the opposite direction. See Figure 1. The semantics of the assembly language could allow each thread to progress at its own speed from its initiating Spawn instruction to its terminating Join instruction, without having to ever wait for other threads. This semantics is henceforth called the *"independence of order semantics (IOS)"*, since the order of progress among threads does not matter (i.e., asynchrony among threads). Synchronization occurs only at the Join instruction; all threads must terminate at the Join instruction before the execution can proceed. That is, in line with the no-busy-waits FSMs ideal, a virtual thread avoids busy-waits by terminating itself ("commiting suicide") just before the thread could have run into a busy-wait situation. So the thread structures for each of these languages is actually an **assemblage of virtual FSMs** incorporating the no-busy-waits FSMs ideal.

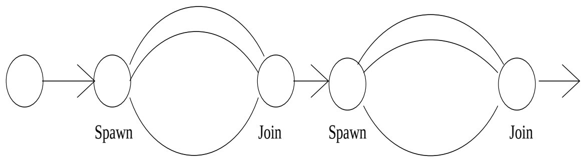

Figure 1: Parallel states from a Spawn to a Join and serial states

The **XMT computational paradigm** covers the full spectrum from algorithms through programming languages and instruction set architecture (i.e., assembly language) to implementation. It suggests to start from a PRAM-style algorithm, proceed to its programming, translate the program into assembly code, and finally reach the implementation. It is appropriate to mention here that the mapping of the assembly code into the implementation seeks to preserve as much as possible the advantages of the no-busy-waits FSMs ideal (although, as explained above, we believe that it is impossible to fully follow this ideal all the way to the implementation). A scheduling lemma attests to the efficiency of this mapping.

Although it may not be immediately clear to a novice, **PRAM algorithms** have traditionally operated in lock-step fashion and satisfied serial semantics. That is, an algorithm is described in steps. In each step some number of operations can be performed in parallel. *Lock-step* means that all operations of a given step must end before the operations of the next step can begin. There is a total order among the operations of a given step. That is, in a step which includes k operations, the operations are typically given ordinal numbers from 0 to $k-1$ (and 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). A total order (or *serial semantics*) among all operations follows by simply extending the in-step order to an order on all operations.

An important goal was to see that **PRAM algorithms are translatable into assembly code** where, (1) PRAM parallelism maps into a thread structure; (2) the (assembly language) threads are not-too-short (to increase locality of reference); and (3) the threads satisfy IOS (i.e., where each thread can progress at its own speed from beginning to end without "busy-waiting" for other threads). For this we observe that: (i) An instruction in a high-level language (which is more similar than assembly language to the "granularity level of PRAM algorithmic thinking") typically translates into several assembly language instructions. Also, the programming of PRAM algorithms (and sometimes the algorithms themselves) may also contribute to the length of threads. (ii) In the arbitrary CRCW PRAM, concurrent-writes into the same memory location result in having an arbitrary one among these writes succeed. So concurrent-writes coming from several threads could be implemented by having one thread write into the memory location and the others bypass the write and simply proceed to their next instruction. Note that the arbitrary CRCW PRAM is generally sufficient for presenting PRAM algorithms.

We are ready for a first visit to the **assembly language**, which is a single-program multiple-data (SPMD) language. A typical Spawn instruction initiates n threads labeled by 0 through $n-1$. The notation \$ is used as a place holder for thread ID. Each thread can have local registers, and there are global registers accessible by all threads. For example, the MIPS-like instruction $lw\ R1\$,\ 0(R3)$ loads a word from the memory address in global register $R3$ into register $R1\$$, which is local to thread \$. Obviously, this last instruction can be decoded (for execution) into up to n specific instructions, one for each thread.

The assembly language includes *multi-operand* instructions. These represent a new kind of instruction demonstrated by the **prefix-sum** (PS) instruction below. The PS instruction is defined through grouping of "individual prefix-sum" instructions.

$PS\ Ri\ Rj$, an *individual prefix-sum* with respect to registers Ri and Rj, means the following inseparable ("atomic") outcome:

(i) add the value of register Ri to the value of register Rj and store the result in Ri, and

(ii) store the original value of Ri in Rj.

In and of itself, this instruction has an effect similar to a simple "add" instruction. However, the difference between the PS instruction and an "add" instruction is that several PS instructions may be cascaded into one *multiple-PS* instruction. For example, 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* $R2, R3, ..., R(k+1)$ into $R2, R3, ..., R(k+1)$ and $R1$, respectively, which means the following:

$R2\ =\ R1;\ R3\ =\ R1\ +\ R2;\ ...;$

$R(k+1)\ =\ R1\ +\ R2\ +\ R3 + ... +\ Rk;$

$R1\ =\ R1\ +\ R2\ +\ R3 + ... +\ Rk\ +\ R(k+1).$

In other words, the sequence of prefix-sum instructions becomes a multi-operand instruction.

The more interesting way (for this presentation) in which a multiple-PS instruction occurs in the assembly code is where each of the individual prefix-sum instructions (to be grouped into a multiple-PS instruction) comes from a different thread. Due to the assumed asynchrony among threads (i.e., IOS), any order of grouping individual prefix-sum instructions having the same base into a multiple-PS is acceptable. The paper [Vi97] explains why if the number of operands does not exceed k, for some parameter k which depends on the hardware, and each of the elements is a small integer (say, between 0 and 3) multiple-PS can be implemented (in hardware) very efficiently (in "unit time"), even where the base is any integer.

The multiple-PS instruction leads to the following implementation of (arbitrary) concurrent writes. For each global address into which concurrent-writes may occur, a dedicated "gatekeeper" address is initialized to 0. Prior to each write an individual PS instruction increments the gatekeeper by 1. Exactly one of the attempting threads gets the value 0 from the gatekeeper and performs the write while any other thread proceeds directly to its next instruction. No busy-waits occur. (This function of implementing concurrent writes can actually be done using a multioperand instruction which is simpler than multiple-PS.)

The **high-level language** for presenting PRAM algorithms is an SPMD extension of standard C which includes Spawn and multioperand instructions such as prefix-sum. Join instructions are not (explicitly) needed; following standard scoping in C, the scope of a Spawn instruction is enclosed between a pair of brackets; the closing bracket ending the scope of a Spawn instruction automatically implies a (hidden) Join. The Join comprises a parallel sum operation for monitoring the number of terminating threads.

Highlights of a possible implementation of the instruction set is depicted in Figure 2. There are t (physical) thread control units (TCUs), each having its own program counter (PC), and a few local registers. There are also shared (global) registers. In the serial state, only the first TCU is active. A Spawn instruction causes a transition into the parallel state, and the actual spawning is done through a (special) Spawn control unit (SCU). The SCU first broadcasts the Spawn on a bus which connects to the TCUs. If the Spawn specifies t or fewer virtual threads, they are all initiated into the TCUs. If more than t virtual threads are specified: (a) The first t of these virtual threads are initiating into the TCUs. (b) The SCU will be responsible to issue the remaining threads, as explained later.

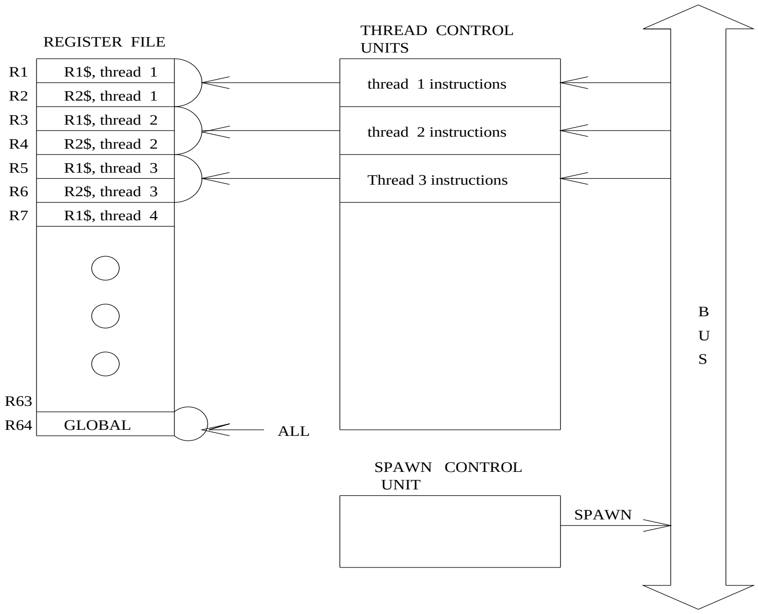

Figure 2: Datapaths, control units, registers, and a Spawn instruction

Instruction supply: each instruction in the instruction code is duplicated (possibly using the bus) for each thread in which it is executed.

A *thread terminates* at a Join instruction. The Join implementation comprises a parallel summation instruction which accumulates the number of threads reaching the Join instruction and is executed using a dedicated *parallel sum functional unit* at the SCU. Once all threads of the previous Spawn terminate, the first TCU is activated in serial state.

The case where several TCUs contribute instructions issued at the same cycle is called *inter-thread parallelism.* The case where a TCU contributes more than one issued instruction to a cycle is called *intra-thread parallelism*; the latter is similar to ILP in current superscalar uniprocessors. (In practice, TCUs could be divided into groups. TCUs of a group can share functional units, similar to [TEL95]). We will assume that a Spawn (or Prefix-sum) instruction can be executed in unit time if the number of operands does not exceed some parameter, which *depends on the hardware*. To understand the robustness of these assumptions consider that the primary objective of our empirical performance studies has been performance comparison with the serial algorithm, rather than observing absolute performance results.

Hiding latencies: for simplicity, suppose that the hardware can issue p instructions per cycle; that is any combination of p instructions is possible and each TCU can contribute between 0 and p instructions to those p (for a total of p, or less); suppose further that an instruction can take between 1 and d time units, for some integer d. Different execution times can reflect arithmetic and floating point operations, or cache misses (for data supply). A design in which $t = pd$ enables issuing p instructions per cycle, based on inter-thread parallelism alone. That is, if a Spawn at the algorithm level provides at least t virtual threads (and $t = pd$), we can guarantee that it will be possible to issue p instructions per cycle as long as at least t virtual threads have not yet terminated. Note that *register file organization* is discussed in separate companion papers.

New mechanism for enhancing thread spawning using prefix-sum Spawning new threads once all current TCUs terminate is relatively straightforward. However, when some TCUs terminate, but others continue, a "gross violation" of the no-busy-waits FSMs ideal occurs. We outline a hardware based solution for quickly reusing the freed TCUs for processing new virtual threads (as long as not all virtual threads have been spawned). The solution makes yet another use of hardware implemented parallel prefix-sum (and again relies on the IOS semantics for the code between a Spawn instruction and its matching Join instruction). Suppose that the execution of several threads terminated, so that each of the TCUs that ran them can take a new thread, and the SCU is waiting to spawn new threads. To allocate new thread indices to these TCUs: (1) The hardware performs a prefix-sum among those available TCUs to get for each of them its serial number relative to the other available TCUs. (2) Broadcast a machine Spawn instruction (called, Spawn-Recur) on the bus, and this will result in each available TCU acquiring a thread index. This allocation of new threads to just freed TCUs will repeat itself following each cycle in which some TCU terminated until all virtual threads of an original Spawn instruction have been initiated.

"Simple_count" Our empirical performance measurements (see Section 7) have been done for a detailed execution model reflecting intra-thread parallelism, cache misses, prefetching and other practical issues. A more common alternative approach, called Simple_count, could have reflected only inter-thread parallelism, thereby providing an easier first estimate. For serial assembly code, the time (or work) count is simply the total number of instructions executed. For each Spawn-Join pair in the parallel code (see Figure 1), its time is the number of operations in the longest thread and the work is the total number of operations in all threads. The total time (resp. work) is the sum of time (resp. work) over all serial and parallel states. For maintaining focus on implementation Simple_count was not included.

1.2. Bridging models

As articulated by [Va90], the "von Neumann machine" has effectively originated the concept of *bridging model*, with respect to serial computing: (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.

Valiant's BSP bridging model aims to do the same for multi-processing as illustrated in the first (leftmost) structure of Figure 4. A closer look at serial computing, however, suggests that unlike [Va90] there is actually a dual bridging model. (i) *Algorithm level* (or *"design space"*): For design, analysis and study of algorithms, the random-access machine (RAM) model of computation coupled with several standard high-level programming languages, such as C, could be used. (ii) *Instruction set level* (or *implementation space*, or *"architecture"*): One of several standard instruction sets (e.g., X86) could be used. Hardware designers could limit their concern to implementing the instruction set, with relatively few (although significant) updates to the instruction set. Efficient emulations of the algorithm level by the instruction set level have contributed to the robustness of the the dual model. See the second structure in Figure 4. In this paper, we aim to provide an analogous dual bridging model for explicit multi-threading.

On the new bridging model: (i) The algorithmic level is essentially the PRAM model of computation. (We actually use a simple-minded language-based model, as explained later.) (ii) The instruction-set level is a multi-threaded extension to standard instruction set. The third (or fourth) structure in Figure 4 depicts the new bridging model; differences between the third and fourth structures are discussed later.

Actually, describing a parallel computer system by way of *two design spaces*, one for the algorithm designer and the other for the computer architect, was proposed before for a so-called Parallel-Design Distributed-Implementation (PDDI) general-purpose computer in [Vi84a]. The algorithm design space here is generally similar to [Vi84a]; however, while the "implementation space" here is based on ILP, there the implementation space was based on a theoretical notion of distributed computing which is more adequate for multi-processing.

Why bridging models? The focus on bridging models also enables presenting the wide scope of our ideas, and at the same time delimit our discussion and reserve judgment on many issues. Some of those issues need much more study and a variety of expertise. Similar to [Va90], the bridging model is intended neither as a hardware model nor a programming model, but something in between. For instance: (1) We chose additions to a standard instruction set which are sufficient for programs we looked at. We hope that the instruction set chosen provides a good approximation for the performance one could expect from our general direction. We leave it to future research to decide to what extent our instruction set need to be expanded, and then reevaluate our performance estimates. (2) The hard numbers we derive in our experiments are meant as a first approximation rather than final numbers. For this reason we often took a cautious approach for comparing our approach with the serial one: whenever we had to relax performance estimates we did that in a way which made our approach look worse than it is. So, our comparative performance figures are more meaningful than the absolute ones.

1.3. Contributions

A new computational paradigm, Explicit Multi-Threading (XMT), which applies to the full spectrum from algorithms through architecture to implementation, is presented; new elements were added where needed. (Figuratively: "although we also plant a few new trees the main message is in the forest".) We note a few things about the bridging models and the XMT paradigm:
(i) The *hardware implementability of the instruction set level is explored*; this includes *a new way for spawning threads, an approach to quickly reusing hardware vacated by terminating threads for execution of more threads (in the "asynchronous mode"), and a new way for implementing inter-instruction (and inter-thread) communication using multi-operand functional units.*

(ii) The instruction level presents a new *explicitly parallel paradigm for issuing instructions*.
(iii) *Simplicity of hardware realization is likely to be competitive* with standard computational paradigms as more parallelism is mandated by code, alleviating the need for the hardware to find parallelism in the face of control dependences and ambiguity of data references.
(iv) Careful *comparison of the new kind of code with standard serial code*, for several ("irregular") test applications, illustrates a *large potential increase in ILP*.
(v) Deviating from the standard PRAM approach, XMT incorporates *reduced synchrony*.
(vi) Both alternatives of not allowing nesting of thread spawning instructions as well as allowing such nesting are considered.
The next paragraph elaborates on item (iv), and the next section on items (v) and (vi).

On the empirical results and their significance Our empirical results were obtained using a *conservative approach* where *a (parallel) lower bound on the time of serial code* (i.e, a serial algorithm, and standard serial instruction code with ILP extracted out of it) *was compared with an upper bound on the time of parallel code* (i.e., a parallel algorithm using our new parallel instruction code - which is an extension of the standard serial instruction code). We also do *not penalize the serial approach for its more complex hardware* for extracting ILP. On the other hand our comparisons assume the same clock rate for different degrees of parallelism, while one would expect some degradation with more parallelism.
Based on our limited empirical results, our **most daring aspiration** is as follows. We hope that *a hardware investment of between* 2 *to* 3 *orders of magnitude (*10^2 *to* 10^3*) in ILP will generally return a speed-up, relative to the serial paradigm, which is an order of magnitude less than the investment* (i.e., in the range 10 to 100).
It is also instructive to *compare our explicit multi-threading (XMT) approach with multi-processing.* We do it here by way of example. The following table compares results for the Paragon reported in Figure 3 of [Si97b] with our XMT result for the same (list ranking) problem.

Multi-processing			**XMT**		
Input size	P	Speed-up	Input size	ILP	Speed-up
$5 * 10^5$	100	< 3	10^2	60	5
10^8	100	25	$5 * 10^3$	100	11.93
			$5 * 10^4$	200	28.53
			$5 * 10^4$	500	63.3

Although the XMT results are not for a real machine, we think they demonstrate that the low latency of ILP (relative to multi-processing) does not require trading much parallelism for hiding latency, but rather translate parallelism directly into performance gains; this enables *competitive results for relatively small inputs.* For some algorithms, smaller inputs means greater applicability to inputs which occur in practice.

1.4. Preview of the Bridging Models

The new bridging model comprises two modes for parallelism. (i) An *asynchronous mode.* This is the main mode, and it has two notable features. The first is reducing synchrony requirements (avoiding expensive busy-waits), thereby enabling our Scheduling Lemma; the Lemma guarantees time upper bounds for running a parallel algorithm on the bridging model; *these bounds are closer to the total amount of actual work required over the amount of available parallelism*, than appears to be possible without reduced synchrony. The second feature is the departure from the lock-step structure of most PRAM-related models (notable exceptions include [G89], [CZ89], or bulk-synchronous parallelism, BSP [Va90]) which was motivated by the following insight: although

PRAM algorithms are synchronous, their semantics allows some slack in the synchrony requirements of their (more refined) instruction code. This slack translates into reduced synchrony, as per the first feature. (ii) A *synchronous mode*. The synchronous mode is useful where rigid synchronization is available (for good performance in the synchronous mode, memory response time should be predictable).
The idea is that each piece of code will be mapped by the compiler into either a serial or a parallel *state*, and within the parallel state into one of the modes.

It will actually be convenient to describe our wider vision regarding "parallel-computing-on-a-chip" using *two* bridging models for a *two-stage* development: (1) *Spawn-based multi-threading (Spawn-MT)*: This is based on (asynchronous, or synchronous) nesting-free Spawn-Join instructions. This is the main model for the presentation in this paper. (2) *Elastic multi-threading (EMT)*: This more involved model concept enables nesting of Spawn-Join instructions (similar to NESL) and relies on the ability of the hardware to suspend, and later reactivate, threads. The third structure in Figure 4 illustrates Spawn-MT and the fourth illustrates EMT.
The Spawn-MT model can cope with irregular, or unpredictable (at compile time), flow of program execution. Such flow may incur due to dynamically varying amounts of parallelism, as an impact of hierarchy and distribution of memory, or due to control and data instructions which use indirect addressing; or due to the nested Spawns (which are only allowed in EMT). Threads can be very short or very light, initiation of new threads can be controlled by the system (using "Spawn-Recur"), and (in EMT only) once initiated they can be suspended to cope with unpredictable delays, and then resumed.

2. THE SPAWN-MT MODEL - OVERVIEW

The Spawn-based multi-threading model of parallel computation 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. For brevity, our description highlights only the most interesting triple of mode-state-level: the asynchronous mode, parallel state and instruction set level. Other possible triples of mode-state-level are simpler. Where this is not obvious, we explain the ways in which they differ from this triple.

Rather than using machine-based models (such as the PRAM, or the serial RAM), it will be convenient to rely for both levels on language-based models where language constructs define the allowed operations. (Such a classification of models and evaluation of performance based on language constructs appeared in [Bl96].) The reader should not look for anything deep (or new) in those language based models; they are just a convenience and are certainly not intended as full scale parallel languages. A key feature for an algorithmic model is its usefulness. Our algorithmic model is PRAM-related - the wealth of the PRAM knowledge base provides unique evidence that it is useful. A key feature of the instruction-set model is its similarity to the algorithmic level model. Its implementability is another important feature; we outline mapping into the hardware for some of the novel constructs in the instruction set model.

2.1. The Asynchronous Mode

We begin with an informal introduction to the asynchronous mode. All terms are defined later. Section 3 provides a comprehensive example for the asynchronous mode; it should be useful to occasionally look up the example while reading the current one. The main source of parallelism both at the algorithmic and at the instruction set levels will be "virtual" processors (called virtual threads, or just threads where their being virtual is clear from the context), each having its local

variables and unit time access to a shared memory (shared variables). Parallel threads are created by a Spawn instruction and terminated by a Join Instruction. In the asynchronous mode, we assume the following **Independence of Order Semantics (IOS)** for the (parallel) threads of the same Spawn instruction. *There is a total independence of order among the work of different threads; that is, each thread can progress at its own speed until its termination.*
Since this is a fundamental point, we rush to note the following. Given an application, it will (continue to) be the programmer's responsibility to make sure that the programs he/she submits are correct; that is, any possible execution of the program (subject to the language semantics) gives a correct solution for the application.

2.1.1. The algorithmic model level In the serial state, only one virtual thread is active. In the parallel state, threads can have concurrent-read or arbitrary concurrent-write access to the shared memory, or they can access it using a prefix-sum instruction. The program under execution switches between a serial and a parallel state. A transition from the serial to the parallel state occurs when the thread executes a Spawn instruction.
A Spawn instruction activates p threads indexed by integers (called IDs) between 1 and p. The value of p may be different for different Spawn commands. After being activated by a Spawn instruction each of the virtual threads will execute its copy of a shared program; using its ID, each thread will interpret the program differently. An important assumption is that each thread executes at its own speed. A virtual thread terminates its execution as a result of hitting a Join instruction. After all the threads (which have been activated by a Spawn instruction) terminate, the execution switches back to the serial state.
We mention several additional characteristics of the algorithm level mode.
(1) *Implementation of concurrent-writes without busy waits.* An important use of the Spawn and Join instructions is for relaxing the tight lock step synchrony at the instruction set level. Concurrent-writes could present a problem. However, we will implement concurrent-writes so that each thread can progress at its speed, free of the need to synchronize with other threads. The idea is that the instruction set implements arbitrary concurrent-writes as follows: (i) exactly one of the threads that attempts to write into a shared variable obtains exclusive access through a "lock substitute" (called later, gatekeeper) for the shared variable. (ii) Have the successful thread write into the shared variable, and the failed threads proceed directly to their next instruction.
(2) *Implementation of synchronization without busy waits.* A thread terminates upon hitting a Join, so that no busy-waits are required. This will become clearer with the specification of Join in the instruction set.
(3) For performance estimates at the algorithmic-level model, we assume that each instruction takes unit time.

Finally, the advantage of using Spawn, Join and lock-substitutes (for concurrent-writes) should be clear at the instruction set level. However, it is less clear at the algorithmic level: given a PRAM algorithm, the assumption that each thread progresses at its own speed may require more Spawn-Join pairs at the algorithmic level (since Join is the main synchronization instruction). This discussion gives us an opportunity to relate to other works. Designing (possibly different) parallel algorithms which are "less synchronous" was considered in the context of the Phase PRAM of [G89], APRAM [CZ89], or bulk-synchronous parallelism (BSP) [Va90]. However, our emphasis in the asynchronous mode is different. We remain with the same parallel algorithm; its proper implementation in the instruction-set level (which is more fine-grained than the algorithm level) could need less tight synchronization than lock-step; this could lead to better performance in our model.

2.1.2. The instruction-set level The instruction set is based on a standard (serial) instruction set (we chose the MIPS version used in [PH94]) with a few additions.

Highlights of the instruction-set (i) Transition from the serial to the parallel state is done by the Spawn instruction and a thread terminates when it hits a Join instruction, for a transition back into the serial state. These Spawn and Join instructions are meant to be as similar as possible to the algorithmic level for enabling a direct translation: (an algorithmic-level) Spawn will translate to (an instruction set) Spawn and a Join to a Join, keeping the same (variable) number of threads.
(ii) The program for each (executing) thread is derived from a single common program using the thread ID ("single program multiple data - SPMD"). The thread ID is denoted by $.
(iii) A thread can have (virtual) registers (denoted by $) as local thread variables; global variables include (virtual) registers and memory.
(iv) Prefix-sum instructions (and Mark instructions, see later) will facilitate the implementation of concurrent-writes similar to the implementation of concurrent-writes without busy waits as described for the algorithmic model level above.
(v) A thread may perform: (a) Global reads: read the contents of a shared variable into a local variable. A special array addressing mode is introduced to enable faster parallel access to addresses. (b) Global writes: write a local variable into a shared one. As noted above, use of global writes will be limited. (c) Local operations: any standard RAM operation. (d) Prefix-sum (or Mark): as defined later, an individual prefix-sum in a thread may form a multiple prefix-sum operation if other threads simultaneously attempt individual prefix-sums with respect to the same base; a sequence of individual prefix-sums relative to the same base within a thread may also form a multiple prefix-sum.
(vi) The semantics of our model allows each of the executing threads to progress independently at its own pace.

Non-standard features in the instruction set level are: (i) The threads in a Spawn remain virtual (and their number is independent of the hardware). (ii) All addresses (including registers) remain virtual. Next, we briefly overview mechanisms (some are new) which could facilitate these features.

Possible hardware/software implementation of the instruction set For concreteness we outline below possible implementations; typically, an implementation outlined represents one of several possibilities; unless otherwise stated, its importance is more in supporting implementability of the instruction set level (and the bridging model) than in the specific details. (i) There are t (physical) thread control units (TCUs), each having its program counter (PC). The TCU may have its own small private memory in the form of a set of local registers, or as a (dynamically allocated) part of a larger (e.g., global) register file.
(ii) The TCUs communicate through a shared memory. The shared memory encompasses shared register variables, and memory variables (that can physically reside in caches and lower levels of the memory hierarchy).
(iii) In the serial state only the first TCU is active. A transition into the parallel state occurs once the first TCU issues a Spawn command; a Spawn command specifies a number of (virtual) threads to be initiated. This is done through a (special) Spawn control unit (SCU), and is depicted in Section 4 by Figure 2. The execution begins by having the SCU broadcast the Spawn on a bus which connects to the TCUs. If the Spawn specifies t or fewer virtual threads, they are all initiated into the TCUs. If more than t virtual threads are specified the following happens. (a) The first t of these virtual threads are initiating into the TCUs. (b) The SCU monitors the (indices of the) virtual threads which are yet to be issued. The next paragraph outlines a new and interesting

mechanism which enables the SCU to initiate more virtual threads into TCUs quickly once they become available. Also, note in Figure 2 the register file, which includes one global register and two local registers per thread.
(iv) Each instruction in the instruction code is duplicated for each thread in which it is executed. Special instructions and hardware mechanisms will facilitate this run-time duplication into the TCU which executes the thread. We will assume that a Spawn, or Prefix-sum instruction can be executed in unit time if the number of operands does not exceed some parameter, which *depends on the hardware*.
(v) A thread terminates when it reaches a Join instruction; the Join instruction comprises a summation instruction; the summation instruction accumulates the number of threads reaching the Join. Once all threads of the previous Spawn terminated, the first TCU is activated in a serial state.
(vi) For simplicity, we will assume in this paper that the hardware will be able to issue p instructions per cycle; it is assumed that any combination of p instructions is possible and each TCU can contribute between 0 and p instructions to those p (for a total of p, or less). We see later that it could make sense to have t larger than p. The case where several TCUs contribute instructions issued at the same cycle is called inter-thread parallelism. The case where a TCU contributes more than one issued instruction to a cycle is called intra-thread parallelism; it is similar to ILP in, say, current superscalar serial processors. (In practice, TCUs can be divided into groups. TCUs of a group can share functional units, similar to [TEL95]).
(vii) An instruction can take between 1 and d time units, for some integer d. (Different execution times can reflect arithmetic and floating point operations, or cache misses, where execution time cannot be predicted.) A design in which $t = pd$ enables issuing p instructions per cycle, based on inter-thread parallelism alone. That is, if a Spawn at the algorithm level provides at least t virtual threads (and $t = pd$), we can guarantee that it will be possible to issue p instructions per cycle as long as at least t virtual threads have not yet terminated.

New mechanism for enhancing thread spawning using prefix-sum Spawning new threads once all current TCUs terminate is relatively straightforward. However, when some TCUs terminate, but others continue, a troubling decrease in efficiency can occur, and the challenge is to quickly reuse the freed TCUs for processing new threads. We outline a hardware based solution for doing that, which uses hardware implemented parallel prefix-sum. Our solution relies on the IOS semantics for the code between a Spawn instruction and its matching Join instruction.
Suppose that the execution of several threads terminated, so that each of the TCUs that ran them can take a new thread, and the SCU is waiting to spawn new threads. To allocate new thread indices to these TCUs, we suggest the following idea:
1. The hardware performs a prefix-sum among those available TCUs to get for each of them its serial number relative to the other available TCUs. (The hardware should, of course, be designed to permit that.)
2. Broadcast a machine Spawn command (called in the next section Spawn-Recur) on the bus, and this will result in each available TCU acquiring a thread index.

The performance model is described in greater detail in a later section. For the serial and parallel instruction set, we assume implementations which aggressively try to maximize intra-thread parallelism. For the parallel instruction set this is, of course, in addition to inter-thread parallelism.

Nesting For Spawn-MT, the asynchronous mode does not permit nesting of spawns. Our model is meant to serve as a bridging model between programs and the hardware. Therefore, it is important to understand that while nested spawns would be allowed for programming, they should be flattened by the compiler. The compiler technology of flattening nested parallelism has been developed in quite a few papers (see, e.g., [Bl90] and the references in `http://www.cs.unc.edu/~prins/nested_par.html`). In our test results for the Spawn-MT model, we used directly nesting-free Spawn-Joins. It will be instructive to compare them with test results for the EMT model where nesting is handled by the hardware.

2.2. The Synchronous Mode and the Possibility of an Alternative

Several multi-prefix primitives are the same as in the asynchronous case. The key differences between the asynchronous and the synchronous modes are as follows: (i) The Spawn operation in the synchronous mode (denoted *Spawnsync* instead of *Spawn*) is performed in lock-step, where all operations of the first time-step precede any operation in the second time-step. (ii) Multi-prefix operations in the synchronous mode are *stable*. That is, the multi-prefix operation is guaranteed to reflect the order of the individual prefix operations comprising it. See also the list of differences with respect to the asynchronous mode in Appendix B.

The way we see the relationship between our Spawn lock-step semantics and vectoring is as follows: (i) If the memory system is designed to deliver vector data at exactly the correct moment, there is no need for implementing a special lock-step mechanism on the TCU side. However, such memory systems are expensive. (ii) The Spawn lock-step mechanism can implement vectoring.

Possible alternative The only use that we found so far for the synchronous mode was for the following *segmented ID problem.*
Input. An array $A = A(1), ..., A(n)$ of bits. Assume (for convenience) that $A(1) = 1$. For every i, $1 \leq i \leq n$, denote by j_i the largest index $\leq i$ for which $A(j_i) = 1$. The segmented ID problem is to compute into some array B the value $B(i) = i + 1 - j_i$ for every $1 \leq i \leq n$.
The Spawn-MT code for searching directed acyclic graphs, (described in Section 8), was derived from its respective code for the EMT mode, by flattening nestings of Spawns that occured in the EMT code. The segmented ID problem occurred in that context.
It is straightforward to solve the segmented ID problem in software within the asynchronous mode, although at some cost. We also see an efficient solution in hardware.
A possible alternative to the synchronous mode in the parallel state could be based on a hardware primitive that can handle in parallel a segmented ID problem whose input length does not exceed some number s. Such a primitive could even be activated from a serial state.

2.3. Related Work and Extensions

Synchronous and asynchronous modes appear in [KS95]; however, they use it for high-level programming languages (rather than for an instruction set), and do not emulate virtual threads; their asynchronous mode appears not to be designed for describing general PRAM algorithms. Multi-prefix primitives were proposed in the NYU-Ultra computer, in [Vi83], as "Scans" in [Bl90], and for Wolfgang Paul's SB-PRAM machine [Ke96].
The instruction set concepts presented are a parallel *extension* of standard serial instruction set architectures. In the current paper, we present them as an extension of the MIPS architecture, but similar concepts could provide extensions to the X86 architecture, as well.

3. EXAMPLE FOR THE ASYNCHRONOUS MODE

We use the following somewhat artificial problem to illustrate a "depth-cut" of the bridging model from algorithm through instruction code to hardware execution.

The array compaction problem *Input.* An array $A = A(0), \ldots, A(n-1)$ of elements; a second array $B = B(0), \ldots, B(n-1)$, whose elements are integers between 0 and $n-1$; and a third array $C = C(0), \ldots, C(n-1)$ of bits (each valued zero or one).
The *compaction problem* is defined as follows. For each i, $0 \leq i \leq n-1$ for which $C(B(i)) = 1$, copy $A(i)$ into a different entry of a fourth array $D = D(0), \ldots, D(s-1))$; where s is the (a priori unknown) numbers of ones in $C(B(i))$, over $0 \leq i \leq n-1$. The mapping into D need not be order preserving. That is, if $A(i)$ is mapped to $D(k)$ and $A(j)$ is mapped to $D(k+1)$ then either $i < j$, or $j < i$. See also Figure 3.
The use of array B is what caused us to refer to the problem as artificial. The reason for using array B is that its indirect addressing makes it more convincing that the time to retrieve the value of $C(B(i))$ can be different for different values of i. This, in turn, could demonstrate the effectiveness of our asynchronous mode, since each thread progresses at its own pace, and once virtual threads terminate, more virtual threads (if such exist) can be immediately be spawned. So, the overall delay due to longer retrieval times for some values of $C(B(i))$ is minimized. This important advantage allowed us to rely on our scheduling lemma at the end of this section.

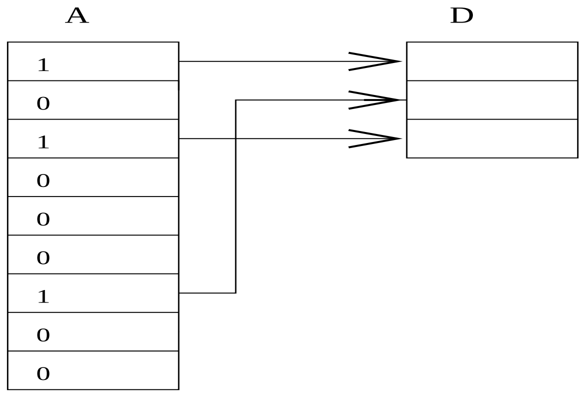

Figure 3: The array compaction problem. The bit values which guide the compaction are shown inside the entries of the array to be compacted (array A).

Algorithm level A high-level program follows.

....
$int\ x;$
$x = 0;$
$Spawn(0, n)$
$\{\ int\ e;$
  $e = 1;$
  $if\ (C[B[\$]] == 1)$
    $\{\ PS(x, e);$
      $D[e] = A[\$]\ \}$

}
$n = x$;
...

A verbal description follows. x is initialized to 0; it is the base variable for the prefix-sum computation. The Spawn instruction spawns threads 0 through $n-1$. Although not explicitly stated, a Join instruction is implied by the bracket "}" ending the scope of the Spawn instruction; the Join instruction comprises a parallel sum computation which increments an "invisible" global variable (let us call the variable y); this invisible y is initialized to 0 by the Spawn instruction. Thread \$ initializes e, its (possible) contribution to the prefix-sum computation, to 1; in case the condition for copying $A(\$)$ into the compact array D is met, the thread performs a prefix-sum with respect to the base x to find the location in D and then copies $A(\$)$ into that location of D. The thread terminates at the (invisible) Join after incrementing y. Once y reaches n, a transition into serial state occurs and n gets the size of the compacted array.
Finally, note that the program remains correct subject to IOS semantics.

Instruction set level For simplicity, we provide next an informal instruction code. For definitions see [PH94].

	Instruction		*Cycle no.*	*Comments*
	...			
	li	R1,0	1	
	move	R2, Rn	1	
	Spawn	R3,0,R2,4	2	
	li	R1\$,1	3	
	lwa	R2\$,B_OFF(R0), 4[R0\$]	$1-5$	*prefetch*
	lw	R3\$,C_OFF(R2\$)	$6-10$	
	bne	R3\$,R1\$,L	11	
	PS	R1,R1\$	12	
	lwa	R4\$, A_OFF(R0), 4[R0\$]	$8-12$	*prefetch*
	swa	R4\$, D_OFF(R0), 4[R1\$]	13	
$L:$	Join	R3, R2	14	
	move	Rn, R1	15	
	...			

The load-immediate (li) instruction initializes $R1$ to 0. The value of n is assumed to be in Rn; the move instruction copies Rn into $R2$. The Spawn instruction is to spawn $R2$ virtual threads indexed 0 to $R2$-1; use 4 local registers per thread; the Join instruction matching the Spawn instruction will count 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. We use our new Load–word-array (lwa) instruction for directly accessing array addresses; the multiplier of 4 is since MIPS addressing is done in bytes (and we assume that each array entry is 4 bytes). If $R3\$$ equals 1, a (parallel) prefix sum is performed incrementing the counter $R1$; $R1\$$ will provide the address into which to copy $A(\$)$; $A(\$)$ is then copied into the 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), which is part of Join; once $R3$ becomes equal to $R2$, Join switches back to serial state. The size of the compacted array is copied into Rn.

Scheduling instructions into cycles along the critical path of the dependence graph follows our general rules, as elaborated in Section 7. Whenever cache misses are possible we use a penalty of 5 (we say later a few words about this arbitrary number); when possible we do these memory loads by a prefetch; however, since we avoid speculations as much possible, we commit to a load only after its preceding control dependence has been resolved.

The most important thing to note in understanding how the execution proceeds is the mechanism for enhancing thread spawning using prefix-sum. The TCUs of all virtual threads that terminate in some cycle use this mechanism to acquire (in parallel) new virtual threads until all virtual threads have been initiated.

To get some hard numbers, assume that the input implies that $\frac{1}{4}$ of the original elements are in the compacted array. Using the framework of the Scheduling Lemma of Section 7, the work requirements of the parallel algorithm for input of size n is $W(n) = 4 + 5n + 3(1/4)n = 5\frac{3}{4}n + 4$, and the time is $T(n) = 15$. Assuming parallelism of p, the scheduling Lemma implies a time upper bound of $\frac{5\frac{3}{4}n + 4}{p} + 15$.

Comparison with serial code was done in Appendix G. The serial instruction code was optimized using loop-unrolling. For a certain loop unrolling (of 4), we got a time critical path of $T = 3.75n + 10(n \ mod\ 4) + 6$. The work is $W \leq 4.25n + 18$. Our comparison is between the parallel upper bound of $\frac{5\frac{3}{4}n + 4}{p} + 15$ and the serial lower bound of $T = 3.75n + 10(n \ mod\ 4) + 6$. For $p = 2$, and n which is big enough, the parallel upper bound is already lower than the serial upper bound. For $n = 100$ and $p = 50$ the parallel algorithm is nearly 15 times faster than the serial one. For $n = 10000$ and $p = 500$ the parallel algorithm is over 280 times faster.
Finally, we note that the optimized serial instruction code is considerably more involved and hard to follow than the parallel instruction code. This may suggest that it will be easier to write the parallel compiler.

4. DETAILS OF THE ASYNCHRONOUS MODE

Instructions for spawning and joining threads, and for prefix-sum (which has several uses), are introduced. We will assume that each of these instructions can be executed in unit time if the number of operands does not exceed some parameter (which depends on the hardware). Description of the Spawn instruction is followed by the prefix-sum instruction, a simpler variant called Mark, Join, and a new array addressing mode.

4.1. Spawn

The Spawn instruction can instruct the spawning of any number of threads. Its execution enables to regulate the number of threads which are initiated, deferring the initiation of others. The instruction can specify the number of registers, local to each thread, and the number of global registers. A data-path for the execution of Spawn is illustrated in Figure 2.
Broadcast of a Spawn instruction from the Spawn control unit on the bus causes threads to be spawned. Recall that the hardware implementation we envision: (i) may also generate a Spawn-Recur command, thereby deferring spawning of the remaining threads in the original Spawn instruction, and (ii) may enhance spawning of deferred threads, taking advantage of the asynchronous mode, and the independence of order semantics (IOS).

Specifics of the Spawn instruction For concreteness, suppose that the instruction $Spawn(1, n; R1; 2)$ is to be executed. Suppose that there are 2 local registers per thread. Execution of the Spawn instruction means the following:
1. For some integer k, threads $1, 2..k$ are initiated. Namely the TCU i will execute next the first instruction of thread i, respectively, for $1 \leq i \leq k$.

2. Logical registers are mapped into physical registers.
3. If $k < n$, the following instruction will imply the execution of the remaining threads:
$Spawn - Recur(k+1, n)$.
The instruction is kept for later execution by the Spawn control unit. Later, it will spawn more threads, as well as possibly yet another $Spawn - Recur$ command.
4. The Spawn instruction implies that the command $R1 = 0$ is executed.
Determining the value k in conjunction with the Spawn-Recur command enables to regulate the number of threads which are initiated, deferring the initiation of others. The Join instruction which matches the above Spawn instruction is given later.

4.2. Prefix-sum (PS)

The *prefix-sum* instruction is defined through grouping of individual prefix-sum instructions.
$PS\ Ri\ Rj$, an *individual prefix-sum* with respect to registers Ri and Rj, means the following inseparable ("atomic") outcome:
(i) add the value of register Ri to the value of register Rj and store the result in Ri, and
(ii) store the original value of Ri in Rj.
In and of itself, this instruction has an effect similar to a simple "add" instruction. However, the difference between the PS instruction and an "add" instruction is that several PS instructions may be cascaded into a *multiple-PS* instruction. For example, 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* $R2, R3, ..., R(k+1)$ into $R2, R3, ..., R(k+1)$ and $R1$, respectively, which means the following:
$R2 = R1$
$R3 = R1 + R2$
$R4 = R1 + R2 + R3$
...
$R(k+1) = R1 + R2 + R3 + ... + Rk$
$R1 = R1 + R2 + R3 + ... + Rk + R(k+1)$
In other words, the sequence of prefix-sum instructions becomes a multi-operand instruction.
Important assumption If the number of operands does not exceed k, for some parameter k which depends on the hardware, and each of the elements is a small integer (say, between 0 and 3) the instruction takes unit time. Note that the base can be any integer.

How does a multiple-PS instruction occur in the code? We distinguish two possibilities:
Possibility (i). Each of the individual prefix-sum instructions (to be grouped into a multiple-PS instruction) comes from a different thread. Due to the asynchrony among threads, any order of concurrent single prefix sums instructions having the same base is acceptable. For example, suppose that the instructions $PS\ R1\ R2$, $PS\ R1\ R3$, $PS\ R1\ R7$ are provided by threads $1, 2$ and 6, respectively, and no other concurrent instructions with respect to base $R1$ are provided. Then, the asynchrony would allow any of the six permutations of $PS\ R1\ R2$, $PS\ R1\ R3$, and $PS\ R1\ R7$.
Possibility (ii). All the individual prefix-sum instructions of a multiple-PS instruction come from a single thread. Only in-order grouping of the individual PS instructions is allowed (similar to [Vi97]).

Comments: 1. Prefix-sum is very similar to the Fetch-and-Add instruction; see [AG94] who discuss its potential use in multi-processor systems. Using Fetch-and-Add as a multiprefix priority-concurrent-write primitive in a parallel programmer's model for a multi-processor parallel machine, was considered in [Vi83]. See also: (i) a discussion of the Scan primitive and a demonstration of its usefulness in multi-processors in [Bl90], (ii) the multiprefix as an operator and instruction in [Ke96] in the context of the SB-PRAM multiprocessor, and (iii) an example for extracting PS from sequential parallel loops by compiler in [FG94]. In other words, the usefulness of multi-prefixes in compact and elegant parallel programming is well-established and the current paper relies on that. In contrast to these references, however, the current paper considers the value of (multiple) prefix-sum to uniprocessors.
2. It is envisioned that multi-operand prefix-sum functional units will be available (possibly implemented by a unique multiple fan-in circuit). Generally, one would expect each (single) prefix-sum to need an adder, (see, e.g. [AG94]). However, an efficient hardware implementation for "small-integer" prefix-sum, where all the operands except the base must be small integers (e.g., between 0 and 3), which uses much fewer adders, is given in [Vi97].
3. Availability of prefix-sum functional units would enable to use it for synchronization and load balancing by the system itself, in addition to the instruction code.
4. In addition to PS instructions which will refer to registers only, one algorithm we implemented assumes that a PS instruction whose base address is a memory location is allowed. The format for such an instruction is PSM 13(R1) R2$ (this means that the base is MEMORY[13 + R1]).
It could be performed by first caching the base address and second creating a copy in a register, which is hidden from the assembly language programmer. The main implementation issue will be to make sure that at a given time all accesses referencing the base address are to this register. This will enable faster access by future PS instructions: each such instruction can become a register-only instruction, coupled with a write-back policy to the cached copy. This PS instruction could implement access of a PRAM algorithm to a shared memory location.
5. The algorithm requiring PS to memory and an additional algorithm we implemented both require the ability to perform concurrently prefix sums with respect to several bases, while the other algorithms require this ability with respect to one, or two, bases.
We mark clearly those algorithms which use the hardware capabilities under items 4 and 5 above in the context of their implementation, later in the paper.
6. It may be easier to implement in hardware an unordered prefix-sum, where the hardware has the freedom to execute several concurrent individual prefix sums (with the same base) as if they were given in some order, but not necessarily in the order in which the threads produced them. Such an implementation would often suffice for the asynchronous mode.

4.3. A Mark Primitive for Synchronization-free Exclusive-write Implementation of Concurrent-writes

A useful operation for parallel algorithms is the so-called concurrent write operation. In a PRAM algorithm, several threads may attempt to write simultaneously into the same memory location. The Arbitrary concurrent-write convention would result in having one of the attempting threads succeed, without prior knowledge which one.

We could implement concurrent writes into a register R1 using (even unordered) prefix sum, as follows. We will need an auxiliary register R2 (called "gatekeeper"), which is initialized to 0, and a local register R1$, which is initialized to 1 for each thread attempting a concurrent-write into R1. Now, each participating thread executes the instruction PS R2, R1$; (only) the thread for which R1$ = 0 goes ahead with the write into R1 and proceeds with executing the remainder of its program; all other threads proceed directly to the remainder of their program. Note that: (i) one thread obtained the exclusive right to write into $R1$; (ii) all threads were able to proceed without any synchronization. We summarize this by saying that, using prefix sum, concurrent-write was implemented by way of synchronization free exclusive write. Concurrent-writes to memory can be similarly implemented using prefix-sum to memory.

However, this is an overkill. There is a cheaper way to implement concurrent-writes using the following Mark instruction (which is similar to standard atomic exchange (see, e.g., [HP96]). Using the registers, as the ones above, and initializing the gatekeeper $R2$ to 0, we define:
Mark R2, R1$
according to the following rule. If R2=0, then the Mark instruction implies: R2=1 and R1$ = 0 atomically. If R2 $\neq$ 0 , then R1$ = 1. Now, for each participating thread, following an instruction Mark R2, R1$ only the thread for which R1$ = 0 will go ahead with the write into R1.

A similar instruction, MarkM 11(R2) R1$ is used to obtain locks to a memory location, and for implementing concurrent-writes into a memory location.

Note that a concurrent-write not through a lock results in an unspecified order.

Together, the Mark instruction and gatekeeper facilitate representing a concurrent-write as a sequence of "*commutative read-modify-write*" instructions - an attractive compiler technology attribute, see [SUIF].

An idea for simplifying hardware. Consider now deriving intra-thread parallelism (in either the parallel or the serial state); that is, where serial (in-order) semantics applies. Superscalar processors use expensive hardware methods for facilitating run-time data disambiguation. We refer the reader to [Vi97] for understanding how to use the Mark (or PS) primitive for possibly reducing these costs. Suppose that an algorithm mandates parallelism and for some reason the choice is that parallelism is to be expressed using intra-thread (i.e., serial) semantics. The basic new idea in [Vi97] is, first, to use the gatekeeper above for implementing concurrent-writes; second, all parallel accesses to memory addresses that cannot be disambiguated as disjoint at compile time will also be expressed (or encoded) in code using a mechanism similar to the gatekeeper above (as if concurrent-writes were involved, even if this was not the case); the original parallelism will then be readily found. It should be mentioned that the gatekeeper mechanism requires additional hardware for implementing the gatekeepers in lieu of the hardware saved. Note that the current comment is somewhat out of place in the current section as its topic is the asynchronous mode; this was done for the sake of brevity.

4.4. Join

The Join instruction is used to terminate threads and end a Spawn block. It incorporates a parallel sum instruction SUMI (see description of the SUMI instruction in Appendix B. Specifically, the instruction $Join(R1, n)$ will have the following effect.
(i) The instruction $SUMI(R1, 1)$ will be executed. That is, $R1$\$ participates in a parallel sum computation with respect to variable $R1$. (ii) Thread $ will terminate (note its local registers can now be reused). (iii) Once the value of $R1$ reaches n, we switch back to a single thread mode.
Note that no "busy-waits" are needed for the synchronization obtained; the transition into single thread mode is initiated by the event of $R1$ becoming n.
In the EMT model described later a Join will imply return to a "parent" thread.

4.5. Array Addressing Modes

In serial code, array access is often done by incrementing a previous address. In parallel code, however, simultaneous access to several addresses, where the distance between each successive pair of addresses is the same, is needed. The new addressing mode facilitates such access; it is actually quite similar to a standard addressing mode called "scaled" in [HP96], page 75.

We continue to use the indirect MIPS addressing mode. In addition, we will use a new array indirect addressing, which is presented next by way of example.
The instruction lwa R3, 24(R1), 4[R2]
is an extension of the standard lw (load word) instruction (the letter a stands for array), which means the following:
R3 = Memory[R1+ 4*R2 + 24].
R1 contains a base address (say, where the part of the array that needs to be accessed begins). The number 4 represents the distance among two successive addresses. R2 holds an index, which could range between 0 and some integer for each of the different threads (it could be a thread ID, or to have been derived from it). 24 is an (additive) offset, which, for instance, for an index value of 0 gives the exact address.

Another example follows. Let lwa R2$, 24(R1$), 8[R0$] be a load in array mode, where the registers local to threads are used. This instruction means, R2$ = Memory[R1$ + 8*R0$ + 24].
Mnemonics for load and store instructions. In addition to the load and store instructions available in MIPS, we use new instructions (lwa, swa, lba, laa, etc. - the "a" stands for "array") for incorporating the new array addressing mode described above.

5. DETAILS OF THE SYNCHRONOUS MODE

We already overviewed briefly the main differences between the asynchronous and synchronous modes in Section 2.2. As progress in lock-step is one of the main differences, it should be clear what constitutes a step. We just note that, unlike the asynchronous mode, prefix-sum across threads will have to be in order. The examples of parallel algorithms, and the information in Appendix B will help illustrate that.

6. HANDLING NESTING BY THE SYSTEM

In the elastic multi-threading (EMT) model nesting is handled by the hardware. Consider the asynchronous mode of the algorithmic level, or the instruction set level, with Independence of Order semantics. Then, nesting of Spawn instructions is allowed. Allowing such nesting is the only difference with respect to the Spawn-MT models, as demonstrated by our empirical results.

Migration of threads from the TCUs to lower levels of the memory hierarchy could be done for threads whose progress is temporarily undesirable. (Progress in a particular thread could be temporarily undesirable if the thread, and threads it may spawn, may drastically increase overall memory requirements.) At the hardware level, it will be possible to suspend threads by context switching them out of the TCUs into the cache (or even into a lower level in the memory hierarchy) and to reactivate threads by context switching them back. This will require a third kind of memory; in addition to the standard data and instruction memories, there will be an Enabled Instructions (EI) memory.

For Spawn-MT, we left open whether the registers local to a TCU reside in the TCU, or are in shared register files. For EMT, it could be advantageous to keep all registers global and local in a shared register file, and have each thread know where its local registers (which correspond to virtual registers marked by $) are; this will enable making local threads global in cases where a thread itself spawns threads, and local registers of the parent thread may be needed as global registers. It is beyond the scope of the current paper to describe this in greater detail.

Consider the issues addressed by the left-to-right depth-first policy for initiation and suspension of threads, due to [BL93]; recall that those issues include generating sufficient parallelism out of a parallel program while keeping the memory requirements under control; these issues are suppressed in the current paper. It will be interesting to see the extent to which recent work on scheduling of parallelism for multi-processors (as in [BGMN], for example) will be applicable to future studies.

7. HOW EXECUTION IS MEASURED

For a preliminary "proof-of-concept" of the model, we considered a few significant, and challenging, test problems, whose algorithms are so-called *irregular*. The more the flow of control of an algorithm is data-dependent, and the more it involves indirect and pointer-based addressing the more irregular it is considered. Traditional parallel computing has made remarkable performance headway concerning regular algorithms, but much less concerning irregular ones. Obtaining good performance for algorithms that involve extensive indirect addressing is sometimes not easy even in the serial domain.

Comparing parallel and standard instruction codes We wrote: (i) Spawn-MT and EMT instruction code for several problems, implementing (elementary) parallel algorithms, as well as (ii) standard instruction code for the same problems, implementing serial algorithms. Our *principal goal* was to compare for the test problems the performance of these two kinds (i.e., parallel and serial) of instruction code. Our results demonstrate that performance can be improved considerably if sufficient ILP is available in hardware.
Our results are reported in Section 8.

An execution model Our empirical results were derived as follows.
(i) Given the code, we tried to place each instruction as early as possible without violating data and control dependences.
(ii) *Speculation.* Generally, we were able to obtain our results without resorting to speculation; however, we allowed speculation for branches which are known to happen with high probability (e.g., branching on whether an element of a linked list is last in the list fails for all but one of the elements of the list).
(iii) *Cache misses.* We examined every memory access. If, due to spatial-locality or temporal-locality, the address could be found in the cache, we assumed unit-time memory access. If a case for spatial or temporal locality could not be made, we applied a penalty of $d = 5$ time units (to simplify our analysis).
This is unquestionably an arbitrary decision which oversimplifies an involved situation. The issue is how to account for memory response time without resorting to simulations; the problem is that to reach the level of detail that merits the use of simulations, we will have to specify the hardware in ways which exceed our more general bridging model and our basic intent to reserve judgement on hardware implementation issues which complement our new ideas. For problems, which are not exceedingly large, the memory could fit within the levels of the cache (which reached 16MB for all cache levels in 1995, with a maximum response time of 15 cycle, see [HP96]). So, the number 5 is not unreasonable for average response time. Also, the sensitivity of our speed-up results to replacing 5 by slightly bigger or smaller numbers was marginal.
(iv) *Prefetching.* To alleviate this cache-miss penalty, prefetching could begin prior to a branch or spawn instruction, that precedes the memory access; however, to minimize speculative execution,

their commit is deferred to after the resolution of the branch or spawn.
(v) *Spawn and Join instructions.* Our analysis uses the following simplifying assumption: a Spawn instruction contributes one to the operation count, and a Join instruction (in each thread) is counted as one operation. Our rationale follows. The principal goal has been comparing the performance of two kinds of instruction codes: serial and parallel. We were faced with a situation where the extensive hardware used currently in practice in order to extract ILP from serial code and control its execution is not accounted for in the instruction code. This gives an unfair advantage to the serial code since it is likely to require more complicated hardware than for implementing our explicitly parallel code; one reason is that *parallelism mandated by the code frees the hardware from verifying that no dependences occur.* This stresses our case. It explains why in terms of devoted hardware, standard serial code is unlikely to be implemented faster, or using less hardware, than our envisioned parallel code (for the same level of parallelism). Still, we take the extremely conservative approach of counting the Spawn and Join instructions, while essentially not counting anything on the serial code end. The only case where we applied to serial code the same costs as to the parallel one was were loop parallelism which can be determined at run-time is needed. Note that this still reflects a "discount" for the serial code: in practice only some fixed loop-unrolling is possible for serial code, while our Spawn mechanism has no problem handling this for parallel code.
(vi) *Unit time instructions.* All other instructions are assumed to take unit time.
(vii) *Scheduling Lemma, number of registers and code length.* The following Scheduling Lemma assumes that the number of registers is unlimited and that the length of the code can be increased with no penalty (as needed for loop-unrolling). (Note, however, that our empirical results consider limiting these assumptions, as well.)
Beyond understanding the Scheduling Lemma and its proof, it should be instructive for the reader to think about the overall design of the Spawn-MT model, and how it was meant to facilitate the Lemma.

Scheduling Lemma Assume that a Spawn-MT program is provided and that only the asynchronous mode is used. Suppose first that each instruction can be executed at the earliest possible time subject to control and data dependence; in particular, we assume the instructions of each serial (or parallel) state are finished before proceeding to the next state. Assume that for some set of inputs, the instructions for executing a program are divided into T unit-time steps, for some integer T, so that at each time i, $1 \leq i \leq T$, at most W_i instructions can be issued, where W_i is a non-negative integer; a thread can contribute 0 instructions to some W_i (if it issued instructions in previous virtual cycles which are waiting for a cache miss to be resolved, and it does not have other ready-to-issue instructions), it can contribute 1, or a larger integer (due to intra-thread parallelism). Assume that any instruction that does not involve a cache miss takes unit time, and any one that does takes $\leq d$ time units. Denote $W = (W_1 + W_2 + \ldots + W_T)$. Assume now that the hardware can issue (any) p instructions at every step; that is, if x instructions are ready for issue (i.e., ready subject to control/data dependences) in the TCUs, then we assume that the hardware can draw all the x instructions for issue in the next round if $x \leq p$, or p instructions otherwise. Assume further that the hardware allows $t = dp$ TCUs. Then, all these W instructions can be executed within $\leq W/p + T$ time units.

Proof The proof is based on a "charging scheme". Each time unit in the execution will either be presented as a charge of p towards W, or a charge of 1 towards T; and the lemma will follow. The interesting case is the parallel state. Assume that a Spawn instruction for s threads is given. Assume first that $s \geq t(= dp)$, and the hardware starts off t threads. As long as those t threads

have not finished, the hardware should have at each cycle at least p more instructions which are ready to issue. Since we have a mechanism for supplying new threads to TCUs which become available, this shows that as long as the number of threads that have not yet finished is $\geq dp$, it will be possible to issue p instructions per cycle. Suppose now that some threads in the original Spawn instruction finished, and the number of threads which have not yet finished is smaller than dp. Consider the first time cycle for which the hardware cannot find p instructions to be issued. Let i be the lowest numbered virtual time cycle for which there is an instruction under execution (among the instructions issued in the current cycle, or in a previous cycle, but are delayed due to a cache miss). We will "charge" this time cycle to virtual time i. Observe that the earliest virtual cycle for which an instruction has not been issued by the current cycle cannot be earlier than i; since if there existed one instruction from a virtual cycle prior to i which had not yet been issued, the earliest among such instruction should have been ready, and we would have used it now to fill up the issue capacity of p. Furthermore, by the current cycle all instructions of virtual cycle i have been issued. Consider now the second time cycle for which the hardware cannot find p instructions to be issued. We charge it to the earliest virtual cycle from which an instruction is still under execution; however, if that cycle was already charged, we charge the actual cycle to virtual cycle $i+1$. The above explanation implies that the earlier virtual cycle for which an instruction is issued in the current cycle cannot be earlier than $i+1$. Observe that by the current cycle all instructions of virtual cycle $i+1$ have been issued. Inductively, this shows that every cycle either issues p instructions, or is charged to a different virtual cycle. The Lemma follows.
Note that to obtain the performance guaranteed by the Lemma any greedy issue of instructions will do.

Timing Lower Bound Modeling Assume that unlimited parallelism is provided by the hardware, and the instructions for executing a program are divided into T unit time steps, for some integer T. Each instruction is put in the earliest possible time in which it can be executed. At each time i, $1 \leq i \leq T$, some non-negative number of instructions can be issued. The number of instructions can be 0 if all future instructions depend on cache misses. T will be the lower bound on the execution of the program.

A conservative comparison approach We made every effort to avoid unfairly favoring the performance estimates of the new code over standard serial code. Concretely, we followed the following guidelines.
(1) For standard serial code, we took its T as the running time, since 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, since this is an **upper bound** *on the execution time achievable.*
Clearly, this puts the code we favor at a disadvantage. We hope that this increases the confidence in our claims.

8. EMPIRICAL RESULTS

Preview of selected problems Our selection of problems and algorithms reflects basic elements of the PRAM theory of parallel algorithms. We anticipate that the efficiency of implementing this sample will correlate well with implementing other algorithms. Integer sorting appears in many parallel benchmarks. Quicksort is one of the most heavily used sorting routines. We explain later why searching directed acyclic graphs is not just a basic problem in computer applications: it cannot be avoided, as it implicitly lies at the heart of every computation. List ranking is a core routine in a large number of parallel algorithms on lists, trees and graphs.

The following table briefly summarizes the results of our experiments. All results provide speedups obtained by the best parallel algorithm relative to a (parallel) lower bound on the best serial algorithm. The experiments, as presented in [Be98], [Nu98a], [Nu98b], and [Da98], respectively, are described in sections 8.1, 8.2, 8.3, and 8.4, respectively. More results are given in appendices D, C, E, and F. A web site from which the documented code and analysis of [Be98], [Nu98a], [Nu98b], and [Da98] can be downloaded is given in Appendix A. Section 8.5 concludes this section with a short discussion of the validity of the empirical results.

			P					
Problem	Input size	Model	20	50	100	200	500	1000
DAGs	graph4	Spawn-MT	4.17	7.59	10.71	13.78	16.45	17.72
	graph6	Spawn-MT	5.79	14.02	26.69	43.97	100.46	159.69
DAGs	graph4	EMT	9.27	17.81	23.58	28.25	32.06	33.57
	graph6	EMT	14.54	35.53	68.42	127.51	265.44	417.09
Quicksort	10^3	EMT	15.30	37.37	71.95	133.90	277.02	430.32
Integer sort	$5 * 10^4$	Spawn-MT	3.61	7.71	13.21	21.84	40.15	60.88
List ranking	10^3	Spawn-MT	2.68	5.29	7.83	12.81	23.47	32.49
	$5 * 10^4$	Spawn-MT	3.09	7.62	14.89	28.53	63.30	106.60

Legend For searching a directed acyclic graph (DAG), graph4 represented graphs with 3200 edges, 400 vertices, and a longest path of 21; graph6 represented graphs with 100000 edges, 20000 vertices and a longest path of 21. The results quoted were in the middle of the range of results achieved for such graphs.

8.1. Search of directed acyclic graphs (DAGs)

Let the input be a directed acyclic graph (DAG) where the length of each edge can be any number. Find the longest path leading to each of the vertices in the graph. A parallel algorithm for searching a DAG will start by marking all vertices whose in-degree is 0, and then, in parallel rounds, will mark those vertices whose incoming edges all come from previously marked vertices. It is easy to see how to use such a DAG search for finding the longest path leading to every vertex of the DAG. The experiments with coding the DAG algorithms led to a few things which are of independent interest. The issue of flattening nested Spawns and a possible hardware primitive for facilitating such flattening are reviewed in Section 2.2. Another issue that came up is a comparison between two implementations of the standard "balanced binary tree algorithm"; that is, the PRAM algorithm for $\log n$-round summation of n numbers: the first round produces $n/2$ numbers through $n/2$ independent pairwise summations, the second produces $n/4$ numbers and so on. A first ("synchronous") implementation does that in a straightforward manner using a sequence of $\log n$ Spawn-Join pairs. A second ("asynchronous") implementation uses a single Spawn-Join pair. There are n threads, each starts at a leaf of the binary tree, and climbs (asynchronously with respect to other threads) up the tree towards the root. A thread terminates once it is the first to reach an ancestor node in the balanced binary tree; that is, the second thread to reach an internal node of the tree performs the required summation and continues to the parent of the node. The comparison of these implementations include: the number of threads, length of threads, as well as, time and work. The outcome is a bit involved and is rather sensitive to the exact (assumptions of the) execution model.

Why the DAG search problem has the flavor of a "universal" benchmark DAGs have been used for modeling precedence among tasks comprising a complex operation, as in the classic

PERT charts. They provide the minimal time for completing a complex operation as well as parallelism available for executing the tasks comprising the operation.
Perhaps more important to our paper is that this kind of modeling, and this kind of search algorithm, are commonly used for scheduling (e.g., loop-unrolled) parts of a computer program for executions. Compiler analysis uses DAGs extensively in connection with program dependence graphs. At a micro scale, DAGs lie at the heart of current ILP hardware enhancements (dynamic) methods, such as scoreboarding or Tomasulo's algorithm.
We elaborate on the relevance of this discussion for the rest of the paper. Given a general application, programmers will first have to provide programs with good parallelism (or good DAGs). Deriving parallelism from DAGs, whether done by the application programmer or by the system, is similar to our DAG search problem. Therefore, efficient execution of the DAG search problem is needed for a high-performance parallel computer system to be successful.
When possible, it will be best if static (i.e., compiler) methods will be used in order to explicitly instruct the hardware to execute parallel DAG search. Our parallel code can help in that. However, when this is not possible (i.e., in case where the DAG is not available at compile time), the actual DAG and its parallel search need to be "encoded" differently, and in a more ad-hoc manner.

8.2. Quicksort

The second algorithm considered is Quicksort. It was implemented only for the EMT model. The main differences between the parallel version and the serial one were that: (i) once a pivot was found, the elements smaller than it were gathered together using a PS instruction, and so were the elements larger than it, and (ii) the processing of these two groups could proceed concurrently.

8.3. Integer sorting

The bin-sort procedure sorts integers whose values are taken from a limited range. Using a standard method, radix-sort sorts integers whose values are taken from a larger range by applying bin-sort in iterations.

Suppose that n integers which range between 1 and $\sqrt{n}$ are given.
1. Bin-sort will partition them into groups. To optimize performance, the number of groups is some function of n and p. For the current high-level description it will be convenient to assume that the number of groups is $\sqrt{n}$. We will have $\sqrt{n}$ threads. Each thread works on one group. It ranks each element in its group with respect to other elements with the same value. This takes $O(\sqrt{n})$ steps.
2. For each value v between 1 and $\sqrt{n}$, a parallel prefix-sum routine is applied to the following array of values: the number of elements whose value is 1 in group 1, the number of elements whose value is 1 in group 2, till the number of elements whose value is 1 in group $\sqrt{n}$, the number of elements whose value is 2 in group 1, till the number of elements whose value is 2 in group $\sqrt{n}$, and the same for each value and group till the number of elements whose value is $\sqrt{n}$ in group $\sqrt{n}$.
3. For each element, its final rank is computed. Let the value of the element indexed i be v, and suppose that it is in group k. The rank of element i is the sum of two numbers: (i) The number of elements which precede value v of group k. (Observe that this number is provided by the prefix-sum computation of Step 2.) (ii) The number of elements in group k whose value is also v, and whose initial index was not larger than i (as computed in Step 1).

Comment concerning the experiments. For integer sorting the range of values for the inputs was 2^{64}. This range is not meaningful for absolute performance results; however, it is adequate for comparison with the serial algorithm, which is what matters to us in the current paper.

8.4. List Ranking

List ranking is one of the "hardest nuts" in the area of parallel algorithms. List ranking is also a well motivated problem since it arises in parallel algorithms for many combinatorial problems, such as on lists, trees and graphs. Empirical parallel algorithms for list ranking have been considered by many including [HR96] [Re94] [SGS], [Si97a] and [Si97b]. We are not aware of previous reports of results for this problem which are competitive with the counterpart serial algorithm for inputs not exceeding 1000. Another difference is that our envisioned implementation paradigm is more fine-grained than in previous work.

Suppose that the input is an array of elements, each having a pointer to its successor in the list (the successor can be anywhere in the array). The pointer is labeled with the distance it represents. The *list ranking problem* is to find for each element in the list its total distance from the end of the list. The serial algorithm for the problem consists of first finding the "head" of the list; this head is the only element of the list that is not the successor of any other element in the list. Second, the list is traversed to find the ranking of the "head", determining for each element by how much it is closer to the "tail" of the list than its head. Finally, the ranking of all elements in the list are derived.

The main parallel algorithm we used is randomized and works in iterations. Assume that an array with precomputed results of "randomly tossed coin" is provided. Each iteration begins by assigning the result of a such a coin tossing to each element of the list. Each element whose coin toss is 1 and whose successor's coin toss is 0 is "selected"; other elements are not selected. Note that: (i) on the average a quarter of the elements are selected, and (ii) it is impossible to select two successive elements. The main progress in the computation is achieved as follows: for each selected element, "pointer-jump over" successor elements, stopping once a selected element is reached. All selected elements (which have not been jumped over) are included in a smaller array. Compaction into the smaller array is similar to Section 3. The smaller array is the input for the next iteration. This will result in finding the rank of the head of the list. Playing "backwards" the iterations will extend the ranking to all elements of the list. We chose this algorithm (dubbed "no-cut") as a practical version of [Vi84]; our evaluations showed that this algorithm requires less work (but needs more time) than alternative versions or algorithms. A variant of the no-cut algorithm (dubbed "6-cut") achieves better time by doing more work; the only difference is that a selected element stops pointer-jumping after at most 6 elements. Another parallel algorithm we used for comparison with the serial algorithm is Wyllie's parallel algorithm (see [Ja92]); that algorithm consists of steps, each comprising parallel pointer jumping by all elements, till all elements are ranked. An appendix reports speedups by the three algorithms relative to the serial one.

8.5. On the Validity and Relevance of the Reported Empirical Results

Since a system in which programmers use instruction code is unacceptable, we confirmed, for each of our test instruction codes, that some high-level description (in the language C) could have been translated into the same instruction code. Observe that restricting ourselves to establishing feasibility of a translation, rather than the translation itself, enabled us to get significant results while circumventing the very demanding task of having compiler technology in place. For example, the reader can verify that the algorithm level code of Section 3 has a feasible translation into the instruction set level code given there.

The code written and the scheduling of instructions to parallel time-units represent our best efforts. It is possible that these can be modified to provide improved performance, although we will be surprised if significant improvement can be achieved.

Finally, many more applications from a variety of application domains will be needed for advancing our proof-of-concept. This is a topic for future work.

9. EXTENSIONS

On the memory issue Spawn-MT (or EMT) appears to be combinable with promising directions for putting more memory on chip, such as the suggestion to put DRAMs on chip in [P+97]; they expect to be able to fit 64MB of memory on chip, using a one-transistor per bit (e.g., DRAM) technology, for around half of the Billion transistor budget. This, and other augmentations that concern intra-thread parallelism mentioned earlier, appear to be possible since they are essentially orthogonal to our focus on inter-thread parallelism in the current paper.

Reducing on-chip global communication Two elements in our presentation whose global communication demand is relatively high are *prefix sum* and *the bus used for spawning threads.* Fortunately, it turns out that both pipelining and hierarchical distributed implementations are possible for each of those elements. Such implementations will greatly improve their scalability. By way of example, suppose that we need to find the prefix sum of 640 single-bit numbers, and we can use at most 64-bit multi-operand functional units. A hierarchical implementation could partition the 640 bits into 10 groups and find the prefix sum for each group; since the sum of each group does not exceed 64 (which takes 6 bits), another 60-bit unit can perform prefix sums relative to the groups with the final prefix sums derived in one more step. We avoid elaborating further on the issue of distributing global communication in the current paper, but only note a few more things to further emphasize its importance: (i) At half the speed of light a signal advances 7.5cm per clock for a 2GHz clock - a reasonable clock rate for the next decade. A chip will exceed 2cm on the side, and since wires wrap around one way across-chip communication will soon exceed one cycle. (ii) The Semiconductor Industry Association [SIA-97] estimates that, within a decade, only 16% of the chip will be reachable in a clock cycle; due to miniaturization clock speed will improve; however, since miniaturization may slow signal speed decrease in wire length need not translate into sufficient decrease in wire crossing time.

The attribute "Elastic" in the acronym EMT is meant to convey flexibility inherent in a design. Parallel algorithms are often described as a sequence of parallel rounds, to be performed one after the other, where each round comprises a sequence of operations that can be executed concurrently. Slightly generalizing the PRAM model, such a description mode permits having a different (flexible, or "elastic") number of operations in different parallel rounds. This relates to the well known *work-time methodology* for describing parallel algorithms (and the VRAM model in [Bl90]). Since its introduction and original use in [SV82] and [Vi81], the work-time methodology has become a standard framework for describing parallel algorithms (see, e.g., [Ja92]). (Section 11 notes that much of the empirical work on studying the limits of ILP extractable from code, by Lam-Wilson and others, appear to fit a more abstract version of this model of parallelism.)
This level of elasticity has long been recognized as useful in the parallel algorithms domain, and is reflected in our main model: the Spawn-MT model.
The current paper also considers a richer notion of flexibility for the new Elastic Multi-Threaded model.

9.1. Implementing Shared Registers

The discussion below: (i) references a few available solutions for various situations which require high-bandwidth register files, and (ii) highlights solutions which imply that the total number of single ported register files need not exceed the number of operands required in a cycle. High bandwidth is required both for the global registers which are shared among threads, and for local registers which are shared among parallel instructions of the same thread.

Following [RGP], consider scratch-pad (or modular) register files. There, finding a viable schedule of operations depends on assigning each temporary value to a module in a way which will satisfy the following *general conflict-free rule*: values that are either written or read simultaneously must reside in distinct modules. In [RGP], assignment of temporary values to modules is performed by first forming *maximal compatibility classes (MCCs)*, where two values are *compatible* if they are neither both read nor both written during the same cycle. All members of an MCC may be assigned to the same module. A *cover* is a set of MCCs whose union includes all the values that have to be assigned. In case the number of MCCs in a cover does not exceed the number of modules, an adequate solution to the assignment problem follows simply by assigning all the values in one MCC to the same module.

A solution that requires a large number of registers was implemented by the context register matrix in the Cydra 5 [BYA, DT93], where each functional unit has a dedicated row for writes in the register matrix, and each row can be read in parallel by all functional units. This structure permits conflict-free register reads and writes for every functional unit. Two recent solutions rely on the following general idea. Rather than having each temporal value reside in a fixed module, allow moving values among the modules so that the conflict-free rule is satisfied. It should be noted that similar to the register matrix solution, both solutions also apply to the general case where the allocation of instructions to steps during execution is given. A static (i.e., compile-time) solution, was given in [MV] and a dynamic solution (i.e., that could be implemented at run-time) in [Vi97]. The new solutions provide high register read and write bandwidth with substantially less hardware; each register file bank has a single read port and a single write port; the solutions imply that the number of register file banks need not exceed the total number of operands required in a cycle.

10. PERSPECTIVE

Interplay of algorithms and architectures: compatibility As part of the reexamination and overhauling of current premises and practice, we need to characterize architectures and algorithms in a sufficiently compatible manner. For that we need to bridge across different terminologies and different research methodologies. We discuss some issues related to the compatibility sought.

Architectures Our quantitative computer architecture evaluation (i.e., the comparison of two architectures) relies on algorithms (and programs) designed and optimized for the systems compared. Quantitative evaluation of standard architectures through a benchmark suite of real programs has become standard for evaluating computer systems designed for the same programming interfaces as the ones for which these real programs were designed (see [HP96]). However, it was necessary to deviate from this methodology to avoid the risk of evaluating a next generation of computer systems by software optimized for the previous generation.

Algorithms (and programs) We need to produce meaningful hard numbers to match architectures and do so by a "quantitative" empirical approach. Traditionally, however, algorithms have been evaluated at a lower resolution level than hard numbers: asymptotic analysis. Asymptotic analysis has been a tremendous success story, as evidenced by the magnitude of the idea-wealthy theory that it produced; it also provides robustness across alternative computation models. The main practical

imperfection is that asymptotic analysis does not provide hard numbers; in particular, asymptotic analysis suppresses: (1) the input length for which the analysis applies, and (2) constant factors.
A quantitative approach Recently there is a growing awareness that not having a quantitative approach is not an option. Actually, many researchers have been working on this issue. It will be interesting to consider pros and cons of the quantitative approach that we picked.

Fine-grained all the way More specifically, we focus on the interplay between instruction-level parallel (ILP) architectures, on one hand, and parallel and serial algorithms, on the other hand. It so happened that many algorithms require fine-grained parallelism. An impetus for our study is that the parallelism in ILP architectures is (even more) fine-grained, resulting in fine-grained parallelism all the way from algorithms to architecture.
Comments. 1. Numerous works have related to matching parallel algorithms with multi-processor architectures; as multi-processor parallelism tend to be more coarse-grained than algorithm parallelism, these works have typically addressed issues which are different than the current paper.
2. Although not discussed in the current paper, we believe that our target multi-threaded processor architecture will fit naturally as the processor component in a variety of multi-processor organizations. For some tightly coupled multi-processor organizations, this could be an avenue for scalability for those applications that exhibit a great deal of parallelism. A related fit is being studied in the context of the M-Machine [F+95].

ILP and a limiting bottleneck ILP architectures are getting increasing attention. For instance, a considerable fraction of the standard text [HP96] is devoted to that, and each among the leading computer vendors came out in the last few years with processors supporting ILP (page 358 in [HP96] lists Intel P6, AMD K5, Sun UltraSPARC, DEC Alpha 21164, MIPS R10000, PowerPC 640/620 and HP 8000). This text implies that hardware capabilities will allow ILP of several hundreds by the beginning of the next decade. However, by the same text many studies established that the main bottleneck for making this capability useful is the (rather limited) ability to extract sufficient ILP from current general-purpose code. It is also noteworthy that in an October 1997 press release (almost a year after our project started), Intel/HP entitled the instruction-set technology for their joint Merced processor, due in 1999: explicitly parallel instruction computing (EPIC). Since having a large number of consecutive instructions free of data and control dependences would help to extract more ILP, it would be natural to look to parallel algorithms for help. The bridging models, presented in this paper, help make the connection with parallel algorithms (and programs).

Multiple-issue paradigms A multiple-issue processor can issue more than one instruction per cycle. There have been two main issue paradigms. The first concerns superscalar processors where the hardware can select instruction for issue subject to: (i) code dependence constraints, and (ii) hardware limits (e.g., how many instruction can be issued or executed concurrently). Further scheduling may be done by the compiler, or dynamically using techniques such as scoreboarding or Tomasulo's algorithm. The main problem with this approach is in its scalability: increased ILP requires a much higher increase in hardware costs. The second multiple-issue paradigm concerns VLIW processors. Here, in contrast to the first paradigm, the compiler creates a package of (a fixed number of) instructions that can be issued in the same cycle. This is a more cost-effective hardware approach but has other problems. For instance, if one instruction in a package is delayed, all instructions (including ones which are otherwise unrelated) are delayed. Our approach can be viewed as a compromise between these two approaches. It borrows from VLIW the notion of statically mandated parallelism, and from superscalar the ability to dynamically progress with as many of the ready tasks as possible.

11. COMPARISON TO KNOWN APPROACHES

Motivated by a need to measure the amount of ILP extractable from code, the works [AS], [LW], and [W93] provided a very interesting empirical methodology. Given the execution trace of a program, they place each instruction in the trace, in turn, in the earliest "parallel round" in which it could have been placed by a hypothetical compiler. Typical assumptions are that unlimited parallelism is possible in each round, and that memory disambiguation (also called memory aliasing) can always be resolved at compile time. As they note, these assumptions are often unrealistic. The understanding that an instruction is ready for execution in a certain parallel round comes often in an implicit form and these works assume that this understanding can be turned imperative. They did not specify how to do this in general, nor have they provided the tools to do so. While their approach did not account for increased parallelism due to program transformation, it provided numerous insightful empirical results, which upperbound the amounts of ILP extractable from code for many important problems, with or without branch prediction, with single- or multiple-flow of control, and subject to other useful alternatives. Note that, in contrast to these, all the parallelism we report actually exists, and the inter-thread parallelism that the Spawn-MT (or EMT) instruction code provides is already in an imperative form. This last contribution resembles the contribution of the PRAM model in providing imperative algorithms; see for instance the contributions of [SV81] and [BH85] in turning Valiant's [Va75] decision-tree algorithms imperative, and the contribution of [SV82] in proposing to use Brent's non-imperative theorem in the work-time methodology for presenting (imperative) PRAM algorithms where "all overheads are taken into account".

The paper [Vi97] presented the following theoretical insight. Using prefix-sum as the only addition to a standard instruction-set, a "completeness" theorem was given. The theorem established that any parallel algorithm can be written ("encoded") using standard instruction code (without the prefix-sum), and then, by compiler methods only, have its full parallelism be decoded for ILP execution. That paper did not give architectural tools for expressing ILP in an inter-thread manner and required lock-step progression on all parallel threads. Its empirical results had to rely on Lam-Wilson's platform, mentioned above. However, the thread structure facilitated by the Spawn-MT (or EMT) instruction set (especially in the asynchronous mode), permits progression within threads. This should improve locality, as reusing data which is already in the registers is more likely to happen. The EMT concept also facilitates suspension, and reactivation, of threads.

For comparison with our work, we note that data-flow architectures have not been shown to have a tight connection with PRAM algorithms; while they presented spawn-join mechanisms they have not offered spawn mechanisms into larger than 2 threads (or corresponding join mechanisms); also their multi-threaded paradigms are far less close to the no-busy-wait FSMs ideal than XMT: multiple threads performing a common job can exchange information in more ways among them, and they can synchronize, which means waiting for one another. For more on the data-flow concept, see [AG94]. The focus of the Fetch-and-Add mechanism has been as a synchronization primitive for multi-processors, and its implementation relied on "combine" elements, rather than a unique multiple fan-in circuit. The Multiscalar project at the University of Wisconsin [Fr93] aims at increasing ILP by splitting a standard serial program, which is given using standard serial instruction code, into multiple, possibly dependent, tasks in parallel. Note that while [LW] estimates that for serial code multiple flow of control can yield considerable performance gains, refined estimates in [Fr96] show much lower performance gains. These later findings explain why a Multiscalar-like organization is even in a greater need for parallel algorithms. Several principled differences with respect to our approach are outlined below. The Multiscalar approach is constrained by existing instruction code, and they anticipate that "most of the execution will be speculative in nature". The Simultaneous Multithreading (SMT) project (e.g., [TEL95]) reflects an interesting interplay between operating systems and architecture with a focus on multiprogramming. A possible analogy is that while they try to project their operating systems background, the current paper tries to project a parallel algorithms background. The discussion considers sharing of functional units among threads, but not registers; inter thread communication is done through the cache. Their view of concurrent memory access is guided by multi-processor approaches and is implemented in the memory system using coherence protocols. Indeed, according to [Le97], at a first approximation parallel computer programs (as discussed in [L+97]) in their ready-to-run form for SMT are not much different than for shared-memory multi-processors. In comparison, our approach allows much lighter threads: inter thread communication can be handled by the registers using multi-operand function units, and not only the memory system (i.e., *multi-operand operations qualifies inter thread communication as "computation", potentially reducing further communication latency*). It also appears that activation, overlap, and termination of threads is more flexible in the Spawn-MT model. And so is also suspension, and reactivation, in the EMT model. None of these architecture approaches supports our instruction set, or anything close to it. The advantage of doing that is the ability to efficiently support code originating from a PRAM algorithm, or just fine-grained threads in general. The thread structure of PRAM algorithms is very dynamic: new threads are generated frequently, and often threads are very short. It will be interesting to find out whether our ideas can be used in conjunction with architectures such as SMT, or Multiscalar.

A comparison with [Ke96] (in the context of the SB-PRAM multiprocessor) reveals that while allowing an asynchronous mode in their most recent version, they abandoned virtual processor emulation, limiting the number of processes to hardware resources. This is a notable difference which does not apply to a previous version (that did not include the asynchronous mode). Some other differences are due to the fact that they are aiming at a multiprocessor, while the primary interest in the current work is ILP on a uniprocessor. However, it is more interesting to note a connection. For instance there is similarity in that they have a asynchronous mode. *A contribution of our bridging model is that it brings the concepts of a high-level language* (high-level as Fork95, or NESL) *down to an instruction code language.* This would make parallel computing much more like serial computing where code in high-level languages, such as C, resembles instruction code much closer. In other words, we somewhat tried to resemble the selection of primitives in [Bl90], and Fork95 in our instruction set; other specific choices for the instruction set would have been possible.

The edited book [IGHS94] is an excellent general reference for known approaches to multithreading. The approaches described there emphasize the case where in a given cycle instruction from only one thread can be issued. Another typical focus is on threads which are not too light as for instance in the compiler-based approach in [CSSvEW]; as noted above for the dataflow paradigm this focus distances them from the NBW-FSMs ideal.

A very interesting "multi-threaded" architecture is presented in the Tera computer [Tera]. What significantly limits the similarity of our approach to theirs is again that they focus on supporting a plurality of threads by constantly switching among threads; but they do not issue instructions from more than on thread at the same cycle; this, in turn, would limit the relevance of our multi-operand and Spawn instructions for their architecture.. However, relative to SMT Tera's threads are much lighter, as are our threads. In order to guide our expectations with respect to the Spawn-MT (and EMT) model, we take note that Tera (which was designed in the early 1990s) is based on 128 threads per processor, while SMT (which was designed later) limited itself to 8, expecting a limited number of ("heavy-thread") hardware contexts.

12. CONCLUSION

Notable new ideas presented: 1. An asynchronous mode for explicit multi-threading for ILP; i.e., virtual order-independent threads (the IOS semantics) are to be executed using multiple program counters. For that, four new mechanisms and an insight are used. 2. Spawn-Join instructions for producing multiple virtual threads; possible Spawn implementation via bus duplication to multiple thread control units. 3. A multi-operand prefix-sum primitive which enables interaction among threads,blurring the distinction between communication and computation; most notable is an interaction with no busy-waits, which often can be implemented using the (simpler) Mark primitive. 4. Use of the prefix-sum primitive for quickly generating more virtual threads once previous ones terminated. 5. The enabled-instruction (EI) memory (for EMT, only). 6. An insight which motivated the design of the asynchronous mode is that although PRAM algorithms are synchronous, their semantics allows some slack in the synchrony requirements of their (more refined) instruction code. All these enabled efficient implementation as per the Scheduling Lemma.

Hardware simplicity These new ideas should lead to hardware realization whose simplicity is competitive with standard approaches. Mandating parallelism, through the Spawn-Join mechanism, the PS (or Mark) mechanism for interaction among threads, and the IOS semantics, alleviates the need for the hardware to find parallelism in the face of control dependences and ambiguity of data references; section 4.3 mentions use of the Mark primitive for facilitating intra-thread parallelism, in lieu of run-time data disambiguation by hardware.

Overcoming a fundamental gap A fundamental problem [Fl96], or gap, in the area of parallel computing is the incompatibility of available parallel machines, which can be characterized as cellular-MIMD ("coarse-grained"), and many of the programming and algorithmic models, which allow "fine-grained" use of shared variables, as in the abstract PRAM algorithmic model. The latter models provide a huge knowledge base of algorithmic methods (see, e.g., [Ja92]), whose magnitude is considerably larger than offered by coarse-grained approaches to parallel algorithms and programming. The current paper is guided by a whole new strategy for alleviating this gap: develop fine-grained ILP computer architectures. The resulting machines will become much closer to desired programming models.

On-chip communication One of the big optimization challenges for hardware implementation of the XMT computational paradigm concerns the performance and cost of on-chip communication. While [M96] notes that "intrachip latency and bandwidth are decreasing and increasing, respectively, at much greater rates than the corresponding interchip figures", only explicit hardware specification would reveal the actual figures involved.

What industry does and says Finally, we suggest to listen to the industry as a technical, though interested, party, and try to understand its actions (investment). The Intel/HP reliance on explicit ILP (for simpler hardware) in their 1999 Merced processor (called "EPIC") is a very encouraging initiative in the general direction outlined in the current paper. However, even if EPIC expects to benefit from programmer provided parallelism, their business interest could still be to publicly present their new processor as relying only on existing code; only after the processor succeeds in the market place, their interest may change to acknowledging that parallelism provided by the application programmer can further improve performance. It is interesting to compare this understanding with two quotes in a technical position paper [SCDEFT97]. A lead sentence is:

"ILP's key advantage is that it exploits parallelism without requiring the programmer to rewrite existing applications". However, towards the end of the paper it is observed that "A variety of ILP-friendly enhancements to existing language and application development methodologies could improve the performance of future systems"; using compiler directives is one of the enhancements mentioned.
Acknowledgement. Helpful comments by Don Alpert, Richard Cole, Don Fussell, Bruce Jacob, Joseph JaJa, Dick Karp, Yossi Matias and Don Yeung are gratefully acknowledged.

References

[AG94] G.S. Almasi, and A. Gottlieb. *Highly Parallel Computing, Second Edition.* Benjamin/Cummings, 1994.

[Tera] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Int. Conf. on Supercomputing*, 1–6, 1990.

[AS] T. M. Austin and G. S. Sohi. Dynamic dependency analysis of ordinary programs. In *19th Annual International Symposium on Computer Architecture*, 1992.

[BYA] G. R. Beck, D. W. L. Yen and T. L. Anderson. "The Cydra 5 minisupercomputer: architecture and implementation". *The Journal of Supercomputing* 7, 143–180, 1993.

[Be98] E. Berkovich. Documented experiments with DAG search for the XMT computational paradigm. See Appendix A.

[Bl90] G.E. Blelloch. *Vector Models for Data-Parallel Computing.* MIT Press, 1990.

[Bl96] G.E. Blelloch. *Programming parallel algorithms. CACM*, 39(3), 1996, 85–97.

[BGMN] G.E. Blelloch, P.B. Gibbons, Y. Matias and G.J. Narlikar. Space-efficient scheduling of parallelism with synchronization variables. In Proc. 9th ACM-SPAA, 1997, 12–22.

[BL93] R.D. Blumofe and C.E. Leiserson. Space-efficient scheduling of multi-threaded computations. In *Proc. 25th ACM-STOC*, 362–371, 1993.

[BH85] A. Borodin and J.E. Hopcroft. Roting, merging and sorting on parallel models of computation. *J. Computer and Systems Sciences* 30 (1985), 130–145.

[CZ89] R. Cole and O. Zajicek. The APRAM: incorporating asynchrony into the PRAM model. Parallelism: In Proc. 1st ACM-SPAA, 1989, 169–178.

[CKPSSSSvE] D.E. Culler, R.M. Karp, D. Patterson, A. Sahay, E.E. Santos, K.E. Schauser, R. Subramonian and T. von Eicken. LogP: A practical model of parallel computation. *CACM*, 39(11), 1996, 78–85 .

[CSSvEW] D.E. Culler, A. Sah, K.E. Schauser, T. von Eicken and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler controlled threaded abstract machine. In *Proc. 4th ASPLOS*, 164–175, 1991.

[Da98] S. Dascal. Documented experiments with list ranking for the XMT computational paradigm. See Appendix A.

[DT93] J. C. Dehnert and R. A. Towle. "Compiling for the Cydra 5". *The Journal of Supercomputing* 7, 181–228, 1993.

[F+95] M. Fillo, S.W. Keckler, W.J. Dally, N.P. Carter, A. Chang, Y. Gurevich, and W.S. Lee. The M-Machine Multicomputer. In *Proc. 28th Int. Symp. on Microarchitecture*, Ann Arbor, MI, 1995. See also `http://www.ai.mit.edu/projects/cva/cva_m_machine.html`.

[FG94] A.L. Fisher and A.M. Ghuloum. Parallelizing complex scans and reductions. In *Proc. ACM-SIGPLAN PLDI*, 135–146, 1994.

[Fl96] M.J. Flynn. Parallel processors were the future... and may yet be. *Computer* 29,12, 151–152, December 1996.

[Fr93] M. Franklin. The Multiscalar Architecture. Doctoral dissertation, Department of Computer Science, University of Wisconsin, 1993.

[Fr96] M. Franklin. Private written communication.

[G89] P.B. Gibbons. A more practical PRAM algorithm. In Proc. 1st ACM-SPAA, 1989, 158–168.

[HP96] J.L. Hennessy and D.A. Patterson. *Computer Architecture A Qualitative Approach, Second Edition*. Morgan Kaufmann, San Mateo, California, 1996.

[HR96] T. Hsu and V. Ramachandran. Efficient massively parallel implementation of some combinatorial problems. *Theoretical Computer Science* 6(2), 1996, 297–322.

[IGHS94] R.A. Iannucci, G.R. Gao, R.H. Halstead and B. Smith (Editors). *Multithreaded Computer Architecture - A Summary of the State of the Art*. Kluwer, Boston, MA, 1994.

[Ja92] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, MA, 1992.

[KS95] C.W. Kessler, and H. Seidl. Integrating synchronous and asynchronous paradigms: the Fork95 parallel programming language. Technical report no. 95-05, Fachbereich 4 Informatik, Univ. Trier, D-54286 Trier, Germany, 1995.

[Ke96] C.W. Kessler. Quick reference guides: (i) Fork95, and (ii) SB-PRAM: Instruction set simulator system software. Universitat Trier, FB IV -Informatik, D-54286 Trier, Germany, May 1996.

[LW] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *19th Annual International Symposium on Computer Architecture*, pages 46–57, 1992.

[Le97] H.M. Levy. Personal communication, August 1997.

[L+97] J.L. Lo, S.E. Eggers, J.S. Emer, H.M. Levy, R.L. Stamm, D.M. Tullsen. Converting thread-level to parallelism via simultaneous multithreading. *ACM Tansactions on Computer Systems*, 1997, to appear..

[M96] T. Mudge. Strategic directions in computer architecture. *ACM Computing Surveys*, 28,4 (1996), 671–678.

[MV] S.M. Mueller and U. Vishkin. Conflict-free access to multiple single-ported register files. in *Proc. 11th IEEE-IPPS*, 1997.

[Nu98a] J. Nuzman. Documented experiments with Quicksort for the XMT computational paradigm. See Appendix A.

[Nu98b] J. Nuzman. Documented experiments with integer sort for the XMT computational paradigm. See Appendix A.

[P+97] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas and K. Yelick. Intelligent RAM (IRAM): Chips that remember and compute. In *1997 IEEE International Solid-State Circuits Conference*, San Francisco, CA, February 1997.

[PH94] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design The Hardware/Software interface*. Morgan Kaufmann, San Mateo, California, 1994.

[RGP] B.R. Rau, C.D. Glaeser and R.L. Picard, "Efficient code generation for horizontal architectures: compiler techniques and architectural support". *In Proc. ISCA*, 131–139, 1982.

[Re94] M. Reid-Miller. List ranking and list scan on the Cray C-90. In Proc. 6th ACM-SPAA, 1994, 104–113.

[SCDEFT97] M. Schlansker, T.M. Conte, J. Dehnert, K. Ebcioglu, J.Z. Fang, and C.L. Thompson. Compilers for instruction-level parallelism.. *Computer*, 30,12 (1997), 63–69.

[Sc98] M.L. Scott. *Programming Languages Pragmatics*. Morgan Kaufman, 1998, to appear..

[SIA-97] Semiconductor Industry Association. *The National Technology Roadmap for Semiconductors, 1997 Edition*. Semiconductor Industry Association, 1997.

[SV81] Y. Shiloach and U. Vishkin. Finding the maximum, merging and sorting in a parallel computation model. *J. of Algorithms*, 2 (1981), 88–102.

[Si97a] J.F. Sibeyn. Better trade-offs for parallel list ranking. In Proc. 9th ACM-SPAA, 1997, 221–230.

[Si97b] J.F. Sibeyn. From parallel to external list ranking. TR-MPI-I-97-1-021, Max-Planck-Institut Fur Informatik, Saarbrucken, Germany, 1997.

[SGS] J.F. Sibeyn, F. Guillaume and T. Seidel. Practical parallel list ranking. Preprint, 1997.

[SV82] Y. Shiloach and U. Vishkin. An O($n^2 log n$) parallel max-flow algorithm. *J. of Algorithms*, 3 (1982), 128–146.

[SUIF] R.P. Wilson, R.S. French, C.S. Wilson, S.P. Amarasinghe, J.M. Anderson, S.W.K. Tjiang, S-W. Liao, C-W. Tseng, M.W. Hall, M.S. Lam, and J.L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. http://suif.stanford.edu/suif/suif-overview/suif-overview.html.

[TEL95] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In Proc. 22nd ISCA, 1995.

[Va90] L.G. Valiant. A bridging model for parallel computation. *CACM*, 33(8), 1990.

[Va75] L.G. Valiant. Parallelism in comparison problems. *SIAM J. Computing* 4 (1975), 348–355.

[Vi81] U. Vishkin. Synchronous Parallel Computation. D.Sc. Dissertation (in Hebrew), Technion, Haifa, Israel,1981.

[Vi83] U. Vishkin. On choice of a parallel model of computation. TR-61, Dept. of Computer Science, Courant Institute, New York University, 1983.

[Vi84] U. Vishkin. Randomized speed-ups in parallel computation. In *Proc. 16th ACM-STOC*, 1984, 230–239.

[Vi84a] U. Vishkin. Parallel-Design Distributed-Implementation (PDDI) general purpose computer. *Theoretical Computer Science* 32 (1984), 157–172.

[Vi92] U. Vishkin. A case for the PRAM as a standard programmer's model. In *Proc. Workshop on Parallel Architectures and Their Efficient Use: State of the Art and Perspectives*, Lecture Notes in Computer Science, Volume 678, 1993, 11-19.

[Vi97] U. Vishkin. From Algorithm Parallelism to Instruction-Level Parallelism: An Encode-Decode Chain Using Prefix-Sum. In Proc. 9th ACM-SPAA, 1997, 260–271.

[VDBN98] U. Vishkin, S. Dascal, E. Berkovich and J. Nuzman. Explicit Multi-Threading (XMT) bridging models for instruction parallelism (extended abstract). In Proc. 10th ACM-SPAA, 1998, to appear.

[W93] D. W. Wall. Limits of instruction-level parallelism. RR 93/6, Digital Western Research Laboratory, Palo Alto, California, November 1993.

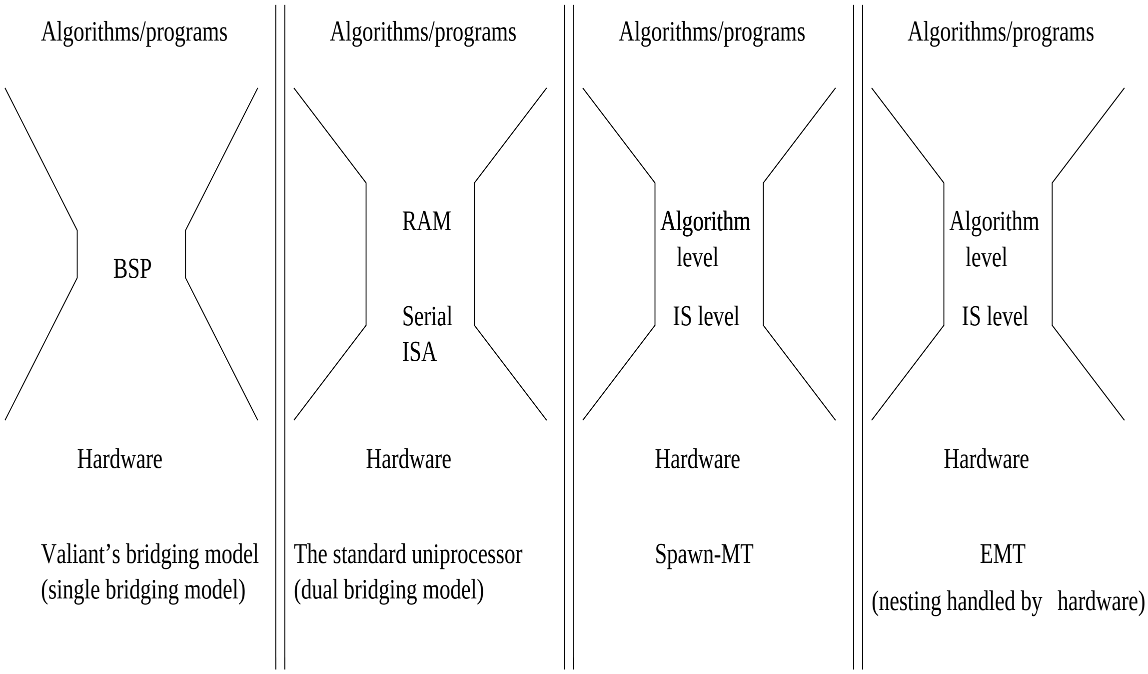

Figure 4: Valiant's BSP bridging model, a standard uniprocessor dual bridging model with the random-access machine (RAM) algorithm level and a standard instruction set level (e.g. X86), the Spawn multi-threading (Spawn-MT) and the Elastic multi-threading (EMT) dual bridging models

A. Supporting Documents

Several supporting documents are available through the following web site: `http://www.umiacs.umd.edu/~vishkin/XMT/`. They include a concise presentation of the XMT paradigm [VDBN98] and documentation along with the elaborate code and analysis for the following problems:
Searching DAGs. This is (around) 63 pages.
Integer sorting, 22 pages.
Quicksort, 15 pages.
List ranking, 76 pages.

B. Spawn-MT: Instruction Set Quick Reference Tables

Notation: $R1\$$ is local register number 1 of thread \$.

Asynchronous mode

Instruction	*Instruction Meaning*
$Spawn\ Rb, j, Rn, REGS$	Switch from serial to parallel state. Spawn Rn threads, indexed j, .. $j + Rn - 1$. Execute by spawning the first (say) k threads the hardware allows. The hardware implementation generates the rest using Spawn-Recur. Assign $REGS$ physical registers to local virtual registers. Typically initialized to 0, global register Rb is a base register for the SUMI instruction of the matching Join.
$Join\ Rb, Rn$	Switch from parallel to serial state. Using a SUMI instruction, each thread increments global register Rb. Once $Rb = Rn$, switch to serial state.
$PS\ Ri, Rj$	Atomically: (i) $Ri\ :=\ Ri\ + Rj$, and (ii) $Rj\ :=\ (original)Ri$. Ri is called: *base for PS.*
$PS\ Ri, Rj_f$ for $1 \leq f \leq g$	Multiple prefix-sum: If a sequence of g Prefix-sum instructions come from one thread: follows serial semantics. If the g instructions come from different threads: follow some (arbitrary) serial semantics. Execute by special hardware. Assumptions made: if each Rj_f between 0 and 3, then unit time execution if $g \leq k$ (base is any integer).
$SUMI\ Rb, Imm$	Given a sequence of such sum integer (SUMI) instructions with the same R_b, the sum of the immediate values (Imm) is produced in parallel. $0 \leq Imm \leq 3$, Imm is inserted at compile time, and the sequence takes unit time for $\leq k$ instructions.
PSM $15(Ri)Rj$	*Prefix sum to memory*, with base $Memory[15 + Ri]$. Multiple PSM: only inter-thread.
$Mark\ Ri, Rj$	If $Ri = 0$, then atomically: $Ri = 1$ and $Rj = 0$; else $Rj = 1$. Also: Multiple-Mark executed by special hardware.
$MarkM$ $15(Ri), Rj$	Mark to memory. Also, multiple MarkM.
$lwa\ R3, 24(R1), 4[R2]$	Array addressing mode. Demonstrated by an array load instruction: $R3\ =\ Memory[R1 + 4R2 + 24]$.

Differences in synchronous mode

1. $Spawnsync$ replaces $Spawn$.
2. Lock-step execution of the parallel state.
3. Multiple-PS across threads: in-order. Same for Mark.
4. No PSM, or MarkM.
5. Alternative possibility: not having a synchronous mode at all. Instead: incorporate in the serial state a hardware solution to the following segmented-ID problem. *Input.* Array $A = A(1), ..., A(n)$ of bits with $A(1) = 1$. For every i, $1 \leq i \leq n$, denote by j_i the largest index $\leq i$ for which $A(j_i) = 1$. Compute into array B, the value $B(i) = i + 1 - j_i$ for every $1 \leq i \leq n$. The (parallel) *segmented-ID* hardware primitive will be able to handle this computation for $n \leq k$, for some k. For $n > k$, the problem is solved in iterations.

Differences in Elastic-MT

1. In asynchronous mode: nesting of Spawns is allowed. A thread can be suspended and reactivated. Nesting will be supported by the hardware.
2. No nesting in the synchronous mode for EMT.

Representing intra-thread parallelism

Note method of [Vi97] for representing static intra-thread parallelism (in parallel or serial state) under serial semantics in lieu of hardware for data disambiguation in Section 4.3.

Potential future extensions

We reserved judgement concerning the extent and form in which several kinds of instructions will be represented, or implemented:

1. Predicated instructions (see, e.g., [HP96]).
2. Prefix-sums for general inputs.
3. Prefix-maxima.
4. Prefetch instructions.
5. A *segmented PS* instruction, similar to [Bl90], within the synchronous mode.

C. Quicksort Algorithm - Speedups

Speedup, comparing time upper-bound for the parallel algorithm with lower-bound for the serial algorithm.

N input size	**P**												
	10	20	30	40	50	60	80	100	200	500	1000	1500	2000
10^2	8.20	15.30	21.51	27.00	31.87	36.23	43.71	49.88	69.53	91.06	101.53	105.58	107.73
10^3	7.71	15.30	22.78	30.13	37.37	44.50	58.43	71.95	133.90	277.02	430.32	527.66	594.95
$5 * 10^3$	7.30	14.58	21.84	29.07	36.28	43.47	57.77	71.97	141.65	337.96	628.13	879.97	1100.61
10^4	7.15	14.29	21.42	28.54	35.65	42.74	56.90	71.00	140.85	343.71	661.09	955.07	1228.13
$5 * 10^4$	6.86	13.72	20.57	27.43	34.28	41.13	54.82	68.50	136.77	340.21	674.79	1003.89	1327.64
10^5	6.75	13.51	20.26	27.01	33.76	40.51	54.00	67.49	134.88	336.34	669.86	1000.61	1328.61
10^6	6.47	12.94	19.41	25.88	32.35	38.82	51.76	64.70	129.38	323.37	646.46	969.28	1291.82
10^7	6.26	12.52	18.77	25.03	31.29	37.55	50.07	62.58	125.16	312.90	625.78	938.63	1251.45
10^8	6.09	12.19	18.28	24.38	30.47	36.57	48.76	60.95	121.89	304.73	609.47	914.19	1218.92

D. DAG Longest Path Algorithms - Speedup

The results here are given as ranges of speedups. The lower speedups are for the worse performing graphs for the given parameters and the higher speedups are for the better performing graphs. However, all speedups are from comparing the upper-bound on the parallel algorithms with the lower-bound for the serial algorithm.

Speedups for the EMT parallel algorithm

Graph	P												
	10	20	30	40	50	60	80	100	200	500	1000	1500	2000
graph1	1.70 to 2.59	1.92 to 3.08	2.01 to 3.29	2.06 to 3.40	2.09 to 3.47	2.11 to 3.52	2.13 to 3.59	2.15 to 3.63	2.18 to 3.71	2.20 to 3.76	2.20 to 3.78	2.21 to 3.79	2.21 to 3.79
graph2	2.22 to 3.31	2.65 to 4.16	2.84 to 4.55	2.94 to 4.77	3.00 to 4.91	3.05 to 5.02	3.11 to 5.15	3.14 to 5.23	3.22 to 5.41	3.26 to 5.52	3.28 to 5.56	3.28 to 5.57	3.29 to 5.58
graph3	4.73 to 5.94	7.28 to 9.37	8.86 to 11.61	9.95 to 13.18	10.73 to 14.34	11.33 to 15.24	12.18 to 16.54	12.75 to 17.42	14.08 to 19.52	15.01 to 21.04	15.35 to 21.60	15.47 to 21.79	15.53 to 21.89
graph4	5.30 to 6.78	8.77 to 11.78	11.23 to 15.62	13.06 to 18.66	14.48 to 21.14	15.61 to 23.18	17.29 to 26.37	18.48 to 28.75	21.45 to 35.06	23.74 to 40.38	24.61 to 42.53	24.92 to 43.30	25.08 to 43.69
graph5	6.75 to 7.79	13.23 to 15.37	19.47 to 22.74	25.47 to 29.91	31.25 to 36.89	36.82 to 43.69	47.37 to 56.76	57.22 to 69.18	97.89 to 123.00	170.70 to 230.69	226.97 to 325.76	254.99 to 377.64	271.77 to 410.31
graph6	6.61 to 8.06	13.07 to 16.02	19.40 to 23.90	25.60 to 31.68	31.67 to 39.38	37.62 to 46.99	49.15 to 61.97	60.23 to 76.62	109.69 to 145.34	216.21 to 314.67	319.69 to 514.47	380.38 to 652.60	420.28 to 753.78

Speedups for the Spawn-MT parallel algorithm

Graph	P												
	10	20	30	40	50	60	80	100	200	500	1000	1500	2000
graph1	0.77 to 1.48	0.89 to 1.96	0.94 to 2.20	0.96 to 2.34	0.98 to 2.43	0.99 to 2.50	1.00 to 2.58	1.01 to 2.64	1.03 to 2.76	1.04 to 2.84	1.04 to 2.87	1.05 to 2.87	1.05 to 2.88
graph2	0.98 to 1.75	1.18 to 2.47	1.27 to 2.86	1.32 to 3.11	1.35 to 3.28	1.37 to 3.41	1.40 to 3.58	1.42 to 3.69	1.45 to 3.93	1.48 to 4.09	1.48 to 4.15	1.49 to 4.17	1.49 to 4.17
graph3	1.87 to 2.43	2.74 to 4.11	3.25 to 5.34	3.58 to 6.29	3.81 to 7.03	3.98 to 7.63	4.22 to 8.54	4.38 to 9.21	4.73 to 10.89	4.97 to 12.24	5.06 to 12.77	5.09 to 12.95	5.10 to 13.05
graph4	2.17 to 2.65	3.47 to 4.87	4.34 to 6.75	4.97 to 8.36	5.43 to 9.77	5.80 to 11.00	6.33 to 13.05	6.70 to 14.70	7.57 to 19.68	8.22 to 24.69	8.46 to 26.97	8.55 to 27.83	8.59 to 28.29
graph5	2.78 to 2.85	5.40 to 5.64	7.88 to 8.38	10.21 to 11.07	12.42 to 13.72	14.51 to 16.31	18.39 to 21.37	21.90 to 26.25	35.41 to 48.34	56.24 to 97.66	69.95 to 147.98	76.13 to 178.66	79.65 to 199.33
graph6	2.89 to 2.94	5.68 to 5.86	8.36 to 8.76	10.95 to 11.64	13.45 to 14.50	15.86 to 17.34	20.44 to 22.96	24.72 to 28.51	42.55 to 55.13	75.03 to 125.43	100.63 to 218.15	113.54 to 289.48	121.32 to 346.05

Descriptions of the input graphs

Parameter	Graph					
	graph1	graph2	graph3	graph4	graph5	graph6
num. edges	603	800	800	3200	25600	100000
num.. vertices	201	200	200	400	1600	20000
num. layers	61	51	11	21	11	21

E. Integer Sort Algorithm - Speedups

Speedup, comparing time upper-bound for the parallel algorithm with lower-bound for the serial algorithm.

N input size	P												
	10	20	30	40	50	60	80	100	200	500	1000	1500	2000
10^2	0.85	1.19	1.39	1.53	1.63	1.71	1.82	1.90	2.09	2.24	2.29	2.32	2.33
10^3	1.43	2.35	3.06	3.65	4.15	4.59	5.33	5.94	7.96	10.60	12.32	13.16	13.67
$5 * 10^3$	1.69	2.97	4.05	5.00	5.86	6.66	8.07	9.32	14.08	22.29	29.51	33.87	36.92
10^4	1.56	2.75	3.78	4.69	5.52	6.30	7.69	8.95	13.89	23.18	32.32	38.40	42.95
$5 * 10^4$	1.97	3.61	5.08	6.44	7.71	8.91	11.15	13.21	21.84	40.15	60.88	76.29	88.81
10^5	2.11	3.95	5.64	7.22	8.73	10.16	12.88	15.42	26.38	50.85	80.00	102.43	121.08
10^6	2.43	4.65	6.74	8.75	10.68	12.55	16.15	19.57	34.99	72.17	120.36	159.96	194.44
10^7	2.12	4.11	6.02	7.89	9.70	11.48	14.93	18.26	33.72	73.15	127.30	173.61	215.00
10^8	2.95	5.82	8.64	11.42	14.17	16.89	22.24	27.50	52.80	122.29	225.83	319.92	407.47

F. List Ranking Algorithms - Best Speedups

Best speedup, comparing time upper-bound for the parallel algorithms with lower-bound for the serial algorithm.

N input size	**P** 10	20	30	40	50	60	80	100	200	500	1000	1500	2000
10^2	1.44	2.47	3.31	3.98	4.54	5.00	5.73	6.28	7.78	9.08	9.62	9.81	9.91
	alg 2	alg 4	alg 4	alg 4	alg 4	alg 4	alg 4	alg 4	alg 4	alg 4	alg 4	alg 4	alg 4
10^3	1.47	2.68	3.69	4.55	5.29	5.93	6.99	7.83	12.81	23.47	32.49	37.26	40.22
	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 4	alg 4	alg 4	alg 4	alg 4
$5 * 10^3$	1.51	2.94	4.28	5.55	6.76	7.90	10.02	11.93	19.33	32.73	50.86	68.08	81.95
	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 3	alg 4	alg 4	alg 4
10^4	1.53	3.01	4.44	5.83	7.17	8.46	10.94	13.26	23.09	41.57	63.39	78.09	94.77
	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 3	alg 3	alg 4
$5 * 10^4$	1.55	3.09	4.61	6.12	7.62	9.10	12.02	14.89	28.53	63.30	106.60	138.09	165.09
	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 3
10^5	1.55	3.10	4.64	6.17	7.69	9.20	12.21	15.18	29.62	68.99	123.85	168.53	205.62
	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2
10^6	1.56	3.11	4.67	6.22	7.77	9.32	12.42	15.52	30.92	76.50	150.36	221.74	290.74
	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2
10^7	1.56	3.11	4.67	6.23	7.78	9.34	12.45	15.56	31.11	77.67	154.98	231.93	308.52
	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2	alg 2

Algorithms used:

- *alg 1* - serial algorithm;
- *alg 2* - work optimized coin tossing algorithm (no-cut);
- *alg 3* - time optimized coin tossing algorithm (cut-6);
- *alg 4* - Wyllie's (pointer jumping) algorithm.

The horizontal and vertical lines shown are boundary lines between different algorithms.

G. Serial Code for the Compaction Algorithm

For comparison with the parallel algorithm and code of Section 3, we wrote serial code, produced an assembly code, and then used a considerable enhancement by way of loop-unrolling. For a loop unrolling of 4, we got critical path time of $T = 3.75n + 10(n mod 4) + 6$. The work is $W \leq 4.25n + 18$. However, it is remarkable to note that the optimized serial instruction code is considerably more involved and harder to follow than the parallel instruction code.

Cooment. The choice of $k = 4$ for loop unrolling was arbitrary. One wants to balance parallelism with the size of code (an issue suppressed in the current paper).. Also, the computations involving k lead to preferring to use powers of 2 for k. A larger k could improve the parallelism but only marginally

Serial C-code:

```
...
int  i,j;

j = 0;
for (i = 0; i < n; i++)
{
  if ( C[B[i]] == 1)
  {
    D[j] = A[i];
    j++;
```

```
    }
  }
  n = j;
```

Assembly-code (non-optimized):

```
                                        Cycle # for       Cycle #
                                        initialization    for 2nd
                                        and 1st iteration iteration
        ...
        li      R7, 1                     1
        li      R1, 0                     1
        lw      R2, 0(Rn)                 1
        multi   R2, R2, ELEMENT_SIZE      2
        li      R3, 0                     1
        bge     R3, R2, L3                3
L1:     lw      R4, B_OFF(R3)             1-5                11-15
        lw      R5, C_OFF(R4)             6-10               16-20
        bne     R5, R7, L2                11                  21
        lw      R6, A_OFF(R3)             8-12               18-22
        sw      R6, D_OFF(R1)             13                  23
        addi    R1, ELEMENT_SIZE          14                  24
L2:     addi    R3, R3, ELEMENT_SIZE      13                  23
        blt     R3, R2, L1                14                  24
L3:     divi    R1, R1, ELEMENT_SIZE                                ...
        sw      R1, 0(Rn)
        ...
```

Let t be the time for finishing all iterations of the loop. That is t+1 is the first cycle in which L3 is reached. Then: $t = 10 * N + 4$.
Critical Path Time: $T = 10 * N + 6$.
Work: $W = 5 * N + 3 * 0.25 * N + 8 = 5.75 * N + 8$.

Optimized Assembly-code:
This was was obtained as follows. From C to SPARC assembly with the "-O2" optimization of the SPARC compiler. Then (in assembly code) it was translated to MIPS assembly code and further optimized manually. The direct MIPS optimizations tried were not as good. (Assume a loop-unrolling of k=4.)

```
                                                Cycle #
        ...
        li      R7, 1                              1
        li      R1, 0                              1
        lw      R2, 0(Rn)                          1
        li      R3, 0                              1
        and     R4, R2, 3                          2
        multi   R4, R4, 4                          3
        ble     R2, R0, END                        2
```

```
        #first (N mod k)   (original) loop iterations
        #the cycle scheduling assumes (N mod 4) = 3
        ble     R4, R0, UNRL                       4
MOD:
        lw      R10, B_OFF(R3)                    1-5    11-15   21-25
        lw      R11, C_OFF(R10)                   6-10   16-20   26-30
        bne     R11, R7, MOD2                     11      21      31
        lw      R12, A_OFF(R3)                    8-12   18-22   28-32
        sw      R12, D_OFF(R1)                    13      23      33
        addi    R1, R1, ELEMENT_SIZE              14      24      34
MOD2:
        addi    R3, R3, ELEMENT_SIZE              13      23      33
        # check if reached (N mod 4)
        bne     R3, R4, MOD                       14      24      34

UNRL:   # preparations for the unrolled loop
        multi   R2, R2, ELEMENT_SIZE              35
        # check if more elements
        bne     R3, R2, END                       36

L1:     # the unrolled loop; executed N/k times
        # 1st "loads" unrolled
                                                1st iter. 2nd iter.
        lw      R10, B_OFF(R3)                    34-38   47-51
        lw      R11, C_OFF(R10)                   39-43   52-56
        # 2nd "loads" unrolled
        lw      R12, B_OFF+ELEMENT_SIZE(R3)       34-38   .
        lw      R13, C_OFF(R12)                   39-43   .
        # 3rd "loads" unrolled
        lw      R14, B_OFF+2*ELEMENT_SIZE(R3)     34-38   .
        lw      R15, C_OFF(R14)                   39-43   .
        # 4th "loads" unrolled
        lw      R16, B_OFF+3*ELEMENT_SIZE(R3)     34-38   .
        lw      R17, C_OFF(R16)                   39-43   .

        # 1st "if" unrolled
        bne     R11, R7, L2                       44      57
        lw      R10, A_OFF(R3)                    41-45   54-58
        sw      R10, D_OFF(R1)                    46      59
        addi    R1, R1, ELEMENT_SIZE              47      60
L2:     # 2nd "if" unrolled
        bne     R13, R7, L3                       45      58
        lw      R12, A_OFF+ELEMENT_SIZE(R3)       42-46   55-59
        sw      R12, D_OFF(R1)                    48      61
        addi    R1, R1, ELEMENT_SIZE              49      62
        # 3rd "if" unrolled
L3:     bne     R15, R7, L4                       46      59
```

```
        lw      R14, A_OFF+2*ELEMENT_SIZE(R3)   43-47   56-60
        sw      R14, D_OFF(R1)                   50      63
        addi    R1, R1, ELEMENT_SIZE             51      64
L4:     # 4th "if" unrolled
        bne     R17, R7, L5                      47      60
        lw      R16, A_OFF+3*ELEMENT_SIZE(R3)   44-48   57-61
        sw      R16, D_OFF(R1)                   52      65
        addi    R1, R1, ELEMENT_SIZE             53      66

L5:     # check final condition
        addi    R3, R3, 4*ELEMENT_SIZE           49
        blt     R3, R2, L1                       50
        # end of unrolled loop


END:    divi    R1, R1, 4                               ...     t+1
        sw      R1, 0(Rn)                                       t+2
        ...
```

Then: $t = 15 * (N/4) + 10 * (N mod 4) + 4$.
Critical Path Time: $T = 15 * (N/4) + 10 * (N mod 4) + 6 = 3.75 * N + 10 * (N mod 4) + 6$.
Work: $W = (3 + 3 * 0.25) * N + 2 * (N/4) + 2 * (N mod 4) + 12 <= 4.25 * N + 18$.