From Algorithm Parallelism to Instruction-Level Parallelism: An Encode-Decode Chain Using Prefix-Sum (extended abstract)

Uzi Vishkin*

Abstract

A novel comprehensive and coherent approach for the purpose of increasing instruction-level parallelism (ILP) is devised. The key new tool in our envisioned system update is the addition of a parallel prefix-sum (PS) instruction, which will have efficient implementation in hardware, to the instruction-set architecture. This addition gives for the first time a concrete way for recruiting the whole knowledge base of parallel algorithms for that purpose. The potential increase in ILP is demonstrated by experimental results for a test application.
The main technical contribution is in the form of a "completeness theorem". Perhaps surprisingly, the current abstract proves that in an envisioned system which employs parallel PS functional units, a proper use of a serial programming language suffices for the following. With a moderate effort, one can program a parallel algorithm (in a serial language), so that a parallelizing compiler (even without run-time methods!) will be able to extract the same (i.e., "complete") ILP from such serial code as from code written in a parallel language. Alternatively, rather than have the programmer produce the serial code, a precompiler could derive it from a parallel language. The most interesting idea in the proof is the reliance on the new parallel PS for circumventing collision-ambiguity in references to memory. Other new ideas in the paper include hardware-design of a prefix-sum unit and an on-line algorithm for high-bandwidth register-files.
An informal upshot of this paper is the following general insight: to accommodate parallelism in uniprocessor systems (from algorithms to ILP), it is sufficient to *only* add (and, of course, incorporate) parallel prefix-sum functional units to standard serial system designs.

*E-mail: vishkin@umiacs.umd.edu. Partially supported by NSF grant 9416890.

1 INTRODUCTION

As the number of transistors on a chip keeps doubling every 1-2 years, the challenge of making effective use of the computational power of a chip needs to be addressed in new ways. Classic performance goals such as: (i) improving completion time of a single job, and (ii) overall throughput of a computer system, make it no surprise that on-chip parallelism, in general, and instruction-level parallelism (ILP), in particular, are getting increasing attention. For instance, a considerable fraction of the standard text [HP] is devoted to that, and each among the main computer vendors came out in the last two years with processors supporting ILP (page 358 in [HP] lists Intel P6, AMD K5, Sun UltraSPARC, DEC Alpha 21164, MIPS R10000, PowerPC 640/620 and HP 8000). This text also implies that hardware capabilities will allow ILP of several hundreds by the beginning of the next decade. Unfortunately, many studies established that the main bottleneck for making this capability useful is the (rather limited) ability to extract sufficient ILP from current code. 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, but before doing that we need to articulate our goal since generally speaking parallel algorithms and programs have assumed a different architecture.

Motivating questions and a road map 1. A general motivating question for this paper has been: is there a way for extending the instruction-set architecture of standard processors only slightly, and use this extension for considerable increase in extracting ILP from current, or new, code?
2. Is there a way for designing a computer system so that, given a parallel algorithm, its full parallelism will be exposed to the execution? In other words, programming the algorithm would encode this parallelism, and later compiler and run-time methods would decode it. Such an encode-decode chain which does not lose any parallelism, and whose decode part uses only compiler methods, is called *complete for parallelism* (or just *complete*, for short). The main **concrete motivating** question which this paper (raises and) addresses is as follows. Is there a way to design a chain which is complete, on one hand, and can rely on only a slight extension of

instruction-sets on the other hand?
3. The design of an encode-decode chain and its analysis are the main concern in the main body of this abstract; figures 1 and 2 provide a **road map** for its write-up. Figure 1 illustrates a "warm-up" encode-decode chain and Figure 2 illustrates the complete encode-decode one. Although the reader may need to read on to to fully understand these figures, it would be helpful to start consulting them.

Contributions A few salient contributions are highlighted below (emphasizing new ideas). The first is a new tool (the main micro-contribution), the second is a new framework (the main macro-contribution), and the third is another application of the new tool. The first and third contributions are based on [Vi96b].

Add prefix-sum to the instruction-set architecture of a uniprocessor. While the format of a prefix-sum instruction is similar to standard uniprocessor instructions, successive prefix-sum instructions may combine into a (non-standard) multi-operand powerful parallel instruction. The following bullet points to a new idea. Subsequently, pointers to theoretical and empirical demonstrations of the usefulness of the prefix-sum addition, are provided.
• *A new hardware implementation idea for parallel prefix-sum* (see the appendix, which also demonstrates a non-conservative "relative complexity" approach for arguing about its efficiency).

An encode-decode chain which is **complete** is presented in the main body of this abstract. We summarize noteworthy elements of the encode and decode parts. Given a problem, the programmer's job will be to first design a "good" parallel algorithm for it. The encode-decode chain assumes that such an algorithm is given.
The encode part Provide the code of the parallel algorithm in a standard serial language. The programmer will be able to either code the algorithm in a parallel language, and have a pre-compiler perform the encode part below, or do it himself/herself. **The decode part** Extract parallelism from the serial code. Our decode part relies on compiler methods only. Applying the new prefix-sum instruction, the decode part uses a new idea for circumventing collision-ambiguity in memory references. The decode part employs only currently used techniques for simultaneous issue of several instructions by the hardware.
The reader may want to summarize the encode-decode chain as follows. (i) Think: parallel. (ii) Code: serial. (iii) Execute: parallel. Encode is the transition from (i) to (ii) and decode is the transition from (ii) to (iii).

A new register-file management algorithm which uses multiple prefix-sums for a drastic improvement of the best solution known is given. This on-line algorithm enables high-bandwidth fast memories - a crucial component for high ILP.

Significance Serial code is easier to write and debug, requires no changes in language standards, and is helpful for backwards (and upwards) compatibility. Also, serial code and serial instruction-code will generally enable using the same techniques that modern superscalar processors currently use. The conclusion section elaborates on why this and other properties of the ideas presented are generally "orthogonal" to important practical aspects, such as branch prediction, caching, vectoring, and out-of-order execution, and can be incorporated with them.
Identifying a restricted language (RAM-like, in this case) which still enables full retrieval of parallelism, is a fundamental problem, and is therefore of interest to the theory of the field.
I found the problem rather challenging, in part because it lies at the crossroad among algorithms, programming, compilers, and hardware, requiring to take all these specialty areas into account. Although the technology of ILP has been a reality for main-stream computer architecture for nearly a decade, it may be worth noting that this might be one of the first (if not the first) "theory" papers on ILP.
The interesting discussion of architectural trends in the forthcoming book [CS] provides relevant background for the current paper. It is also helpful to bring verbatim the following statement: "the natural question to ask is how far will ILP go and at what point will the emphasis shift to thread level parallelism". The current paper sheds new light on this natural question: using parallel algorithms an ILP processor can provide more parallelism than was known before. Comparing this extra-parallelism with expectations implied by the phrase "von-Neumann bottleneck" (see Backus's 1977 Turing Award Lecture, CACM 1978), could also be of interest.

Extant literature Parallelizing compilers for standard serial languages, such as C, are getting a lot of attention recently: (i) current and emerging "ILP-based" processors need them to fully utilize their computation power [HP] (for example, writing programs in ordinary languages, such as C or C++, tops the "software wish list" for future processors in [So]), and (ii) in connection with the multi-institution umbrella research project entitled SUIF (Stanford University Intermediate Format) in the context of multiprocessors [SUIF]. A recent position paper, [OV95], criticizes adverse effects in current programming practice which block, and even undoes, potential benefits from parallel algorithmic thinking. The current paper binds parallelizing compiler ideas with suggestions for alleviating this critique. For concreteness, these ideas are presented as part of the proof of the Completeness Theorem. Literature related to prefix-sum is reviewed in a later comment, which follows the definition of prefix-sum.

1.1 Basics

1.1.1 The Random-Access-Machine (RAM)

Given some set of primitive operations, the generic one-processor RAM model of computation assumes that instructions are executed sequentially, one after another, with no concurrent operations and where each primitive operation takes unit time.

The RAM has been the model of choice for standard textbooks on algorithms and for standard programming. However, execution of instructions in modern off-the-shelf processors deviates from the RAM sequential abstraction in two main ways. (i) *Pipelining.* Each instruction executes in stages, where different instructions may be at different stages at the same time. (ii) *Multiple-issue.* Several instructions can be issued at the same time unit. The parallelism resulting from such overlap in time in the execution of different instructions is called *instruction-level parallelism (ILP).* The combined ILP resulting from pipelining together with multiple-issue (which may reach the product of the number of pipelining stages and the multiplicity of issue) represents a significant departure from the sequential RAM.

Preview of the remainder of this section and the "Completeness theorem" Considering these profound deviations from the RAM, the issue at hand is how to update the conceptual and actual programming models, and the execution models so that they incorporate ILP, and whether to forgo the RAM software interface. Since the programmer may need to cater for ILP, by way of seeking parallelism in algorithms and programs, we review next the basic way in which the PRAM theory of parallel algorithm taught us to think about parallel algorithms.
As a warm-up for our main discussion and result, we propose the ILP-RAM as a hardware execution model of ILP processors; we show that: given an ILP-RAM algorithm (that is, the conceptual algorithmic model is the PRAM-like ILP-RAM), it can be encoded in (that is, written in, or translated into) RAM-like code, but still be decoded, suing run-time and static methods, so that its original parallelism reaches the hardware execution model. In other words, the original parallelism is preserved through the encode-decode chain. This warm-up observation is illustrated in Figure 1; in the text it appears as Observation 1.
Then, we present the prefix-sum instruction, and finally add it to the ILP-RAM to form the PS-RAM model. In a later section, the Completeness Theorem, illustrated in Figure 2, shows that replacing the ILP-RAM by the PS-RAM (in the warm-up observation) is not only possible, but also that the decoding can be done using static methods only.
(Note, that the fact that the RAM continues to provide (standard) instruction code, does not prevent the instruction-set architecture from having an additional instruction - the prefix-sum; there will be a simple convention for representing prefix-sum using standard instruction code.)

The PRAM work-time framework A parallel algorithm consists of rounds of computation where in each round several operations, indexed by successive integers starting from 1, can be performed simultaneously. Simultaneous operations may include reading simultaneously from the same variable or even writing simultaneously into the same variable. In the latter case (which is often called the Priority concurrent-write convention) the programmer may assume that the smallest indexed among

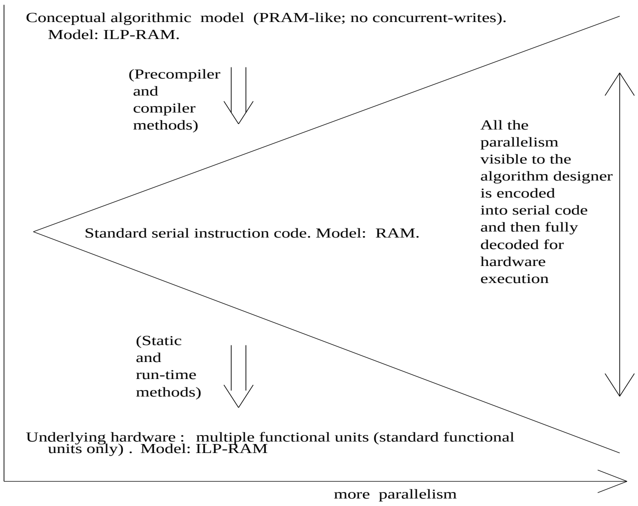

Figure 1: Encode-Decode of ILP-RAM parallelism. The X-axis grows with the amount of parallelism. The amount of parallelism is high at the computational algorithm model, low at the serial instruction code, and high again at the underlying hardware.

the writing operations succeeds. The two main parameters for measuring the efficiency of a parallel algorithm or program are: (i) *parallel work* - the total number of operations (instructions) performed in all rounds, and (ii) *parallel time* - the number of rounds.
(*Comment.* This general way of looking at parallel PRAM algorithms is represented, for example, in [Ja], and has apparently been first demonstrated as useful for describing parallel algorithms in [SV]. Since, the PRAM algorithmic knowledge base is very rich, and is probably the single richest concerning parallel algorithms, one could rely on it for providing algorithms for numerous problems.)
Next, we note the following substantial difference between the above notion of parallel algorithms and possible execution of instructions on some current ILP-rich processors: any form of concurrent-writes (i.e., simultaneous access to the same memory location by several instructions for write purposes) results in having such concurrent-writes queued and treated serially one at a time (similar to what is modeled by QW, in the QRQW PRAM model [GMR]). So, current reality guides prohibiting concurrent-write in the modeling of ILP. Leaving other aspects of a PRAM-like model intact we proceed to defining the ILP-RAM. According to PRAM folklore this is similar to the CREW PRAM.

1.1.2 The ILP-RAM

Suppose that some set of primitive operations, as in a standard RAM, is given. An ILP-RAM computation proceeds in steps. In each step, a sequence of instructions, each executing a primitive operation, is performed in parallel. The sequence requirement implies that these instructions are indexed by successive integers starting from 1. The ILP-RAM model of computation of a uniprocessor allows any number of instructions using

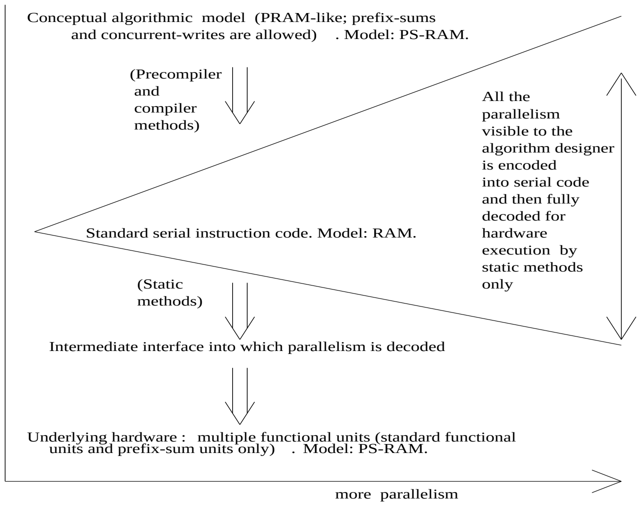

Figure 2: Encode-Decode of PS-RAM parallelism

these operations to be performed concurrently as long as no two concurrent instructions attempt to write into the same memory location (output operand). If some memory location is read by one, or more, instructions and is written into by an instruction, we assume that the reads are finished prior to the write. Each primitive operation takes unit time, and so is the concurrent execution of several primitive operations. Unlike the PRAM, the number of operations performed in different steps can be different. As in the work-time framework, the time of such a computation will be the number of rounds in which concurrent unit-time operations are performed, and the work will be the total number of such operations summed over all rounds. A simple way for describing an algorithm in an ILP-RAM would be using a sequence of commands of the form:
for $i = 1$ to k pardo { }
with varying values for k.
Another approach for "programming" an ILP-RAM is given later in this subsection in the context of mapping a RAM algorithm into an ILP-RAM algorithm. Note that empirical studies of ILP, such as [LW], have also used work and time as efficiency measures, although sometimes implicitly. Next, we relate the ILP-RAM and the RAM.

Emulating the ILP-RAM by the RAM. This emulation is known and trivial. One parallel round is emulated after the other. Consider the emulation of a single ILP-RAM round. Since the RAM performs instructions serially, rather than concurrently, finishing all reads of a memory address prior to a write to the same address requires special care. We sketch a solution for this problem, which needs two stages in the RAM emulations. In the first stage, store in temporary values those values which are read in the instructions of the ILP-RAM round (or at least, those values which are read and into which a write is performed in the current round). In the second round, proceed to finish the implementation of the original instructions. An alternative two-stage solution would be to actually execute the instructions, short of performing the writes (or at least short of those writes from which a read is performed); instead, the writes are stored in temporary values, and copied , in a second stage. The converse relationship is actually more interesting and is considered next.

A RAM algorithm can have a simple mapping into a non-trivial ILP-RAM algorithm. Take the sequence of operations in an execution of a RAM algorithm, then "pack" them inductively into the ILP-RAM, as follows. The first RAM operation will be at the first round of the ILP-RAM. For every subsequent RAM operation, do the following: if that operation is not (data- and control-)dependent on operations already placed in the first ILP-RAM round, then it is also placed in the first ILP-RAM round. Otherwise, this RAM operation is placed in the second ILP-RAM round. In general, assume inductively that using such a process the first s operations in the RAM execution have been placed into ILP-RAM rounds, and, say, that (the last) operation s was placed in round t. Then, for determining whether to place operation $s+1$ in round t, or round $t+1$, we do the following. If that operation is not dependent on operations already placed in round t, then it is also placed in round t. Otherwise, this RAM operation is placed in round $t+1$.
Comment. Placing an instruction in some round t above does not mean to prevent putting it for execution earlier than preceding instructions of round t,earlier rounds, in a later optimization.
Observation 1. Emulating an ILP-RAM algorithm by a RAM, as above, and then the resulting RAM code by an ILP-RAM, will produce the same parallel work and time as the original ILP-RAM algorithm, up to a constant factor. This shows, that expressing in serial code an ILP-RAM algorithm need not lead to loss of ILP.
A memory disambiguation caveat. The emulation which derives this parallel efficient ILP-RAM execution from the RAM code can be done on-line, but not by static (compiler) methods. The reason that a compiler may not be able to derive the same efficient execution as a dynamic (on-line) emulation is that in case indirect input-dependent memory accesses occur, the memory disambiguation needed to conclude that no two commands in some sequence of RAM commands may attempt to write into the same memory locations must wait for the operand to be available, which is not the case at compile time.

It is important to study this parallelizing emulation of a RAM computation by an ILP-RAM, and the memory disambiguation determinations required, since the inductive process used in that emulation for determining the instructions that can be simultaneously issued next is *similar to current typical superscalar processors* (using a single program counter, as per [HP], page 279). For the purpose of the current paper, this shows that the RAM can provide a useful platform for describing parallel algorithm. The proof of our Completeness Theorem, later in the abstract, essentially revisits the above back and forth emulations to achieve a considerable enhancement.

1.1.3 Prefix-sum

An extension of standard binary summation, the *prefix-sum* operation is defined as follows. Given an input *base*

x and k elements y_1, y_2, ... , y_k, the prefix-sum of the base and the k elements are the following $k+1$ elements:

$z_0 = x$

$z_1 = x + y_1$

$z_2 = x + y_1 + y_2$

...

$z_k = x + y_1 + y_2 + ... + y_k$

Similar to the known NYU-Ultracomputer Fetch-and-Add instruction (and synchronization tool), it will be convenient to express the prefix-sum operation as a group of "individual" prefix-sum operations in the following manner. $PS(A, B)$, an *individual prefix-sum* with respect to variables A and B, means the following inseparable ("atomic") outcome:
(i) add the value of variable A to the value of variable B and store the result in A, and
(ii) store the original value of A in B.

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, observe that the sequence of instructions:

$PS(A, B_1)$

$PS(A, B_2)$

...

$PS(A, B_k)$

performs the prefix-sum of the input base A and the elements $B_1, B_2, ..., B_k$ into $B_1, B_2, ..., B_k$ and A, respectively. Namely:

$B_1 = A$

$B_2 = A + B_1$

$B_3 = A + B_1 + B_2$

...

$B_k = A + B_1 + B_2 + ... + B_{k-1}$

$A = A + B_1 + B_2 + ... + B_{k-1} + B_k$

Comments:

1. In RISC-type instruction code (where arithmetic is done only with respect to registers), an individual prefix-sum instruction could look like:
$PS\ R_i\ R_j$
where PS is the instruction name and R_i and R_j are registers and the semantics is identical to the individual prefix-sum instruction above (with R_i and R_j replacing variables A and B, respectively). Multiple prefix-sum will then be a sequence of the form:
$PS\ R_0\ R_1$
$PS\ R_0\ R_2$
...
$PS\ R_0\ R_k$
where, $R_0, R_1\ ...\ R_k$ are registers and again the semantics should be obvious.
2. Prefix-sum is very similar to the Fetch-and-Add instruction; see [AG] 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 [B90], and (ii) the multiprefix as an operator and instruction in [K-96] in the context of the SB-PRAM multiprocessor. 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, as the only addition to a standard instruction-set architecture.

The Prefix-Sum Random-Access-Machine (PS-RAM) is an extension of the ILP-RAM model. The model assumes that a multiple-PS instruction takes unit time. Several multiple prefix-sums (with different bases) can execute concurrently. Having the same variable participate in two simultaneous multiple-PS whose base variable is different is "undefined". Also undefined is the case where with respect to some variable both a multiple-PS instruction and either a read command, or a write command, are performed performed simultaneously. In this paper any value will be acceptable in the undefined case, and our statements will not apply to those cases. The model's name does not include the substring ILP for the sake of brevity. In case some memory location is simultaneously accessed by read commands and write commands, we assume that the read commands are performed first, and then the PS and write commands. The cases resulting in undefined values will not restrict our expression power. Some thinking may show that the only case which needs explanation is where one might have wanted to intermix PS commands with read commands. However, if an intermediate value of the base variable (denoted x) in a multiple-PS instruction needs to be read into a variable y, we can do that by initializing y to 0 and then apply $PS(x, y)$, rather than using a read command.

This paper suggests adding this (individual and multiple) prefix-sum instruction to the instruction-set architecture.

Prefix summation may be implemented by existing instruction-set architectures, using a standard balanced binary tree, as presented, for example, in [Ja]. Preferred embodyments for the theme of the current paper, however, implement prefix summation in hardware. One idea is to have special hardware for "small-integer" (where the base A could be any integer, but each B_i is limited to small integers only) prefix summation which could be completed faster than where no such constraints are placed on the size of integers, and use it for a variety of tasks as elaborated in later sections. Proposed embodyments are reviewed in the appendix.

1.2 Overview of the Rest of the Paper

The main thrust of the current paper is a suggestion to add a parallel prefix-sum operation to instruction-set architectures, and use this as a bridgehead in a comprehensive approach for the purpose of increasing instruction-level parallelism (ILP). This would involve incorporating some needed changes into the various components of a computer system ranging from compilers to algorithms and their programming.

Assuming the PS-RAM abstraction for a machine, a new completeness theorem is proved. The theorem asserts that an algorithm for the PS-RAM (the algorithm can actually use also Priority concurrent-writes) can be written in code using standard serial (namely, RAM) programming so that its PS-RAM implementation will not be less efficient than in case it had been directly expressed using PS-RAM programming. (In other words: think PS-RAM, write code for RAM, and execute in PS-RAM.)

Following are several possible uses for our observations for the purpose of bringing more ILP to the execution level. (i) Have a (pre)compiler translate code in a high-level serial, or parallel, programming language into the PS-RAM. (ii) Have prefix-sum as an instruction accessible to the programmer in a high-level programming language. (iii) Have prefix-sums as an instruction in an instruction-set architecture into which static and dynamic methods optimizing the run of a program could aim. (iv) As an instruction available to the computer system for its operation, such as for supporting simultaneous execution of several computational threads (multi-threading), and load balancing, or as a key step in a high-bandwidth register-file system, and (v) Implement prefix-sum in hardware. (vi) Have parallel PS functional units.

Important properties of the approach are that: (i) Although to get the full benefit in terms of increased ILP the approach relies on cooperation from the programmer, limited improvement can potentially be obtained, even with no involvement from the programmer. And (ii) the resulting computer systems will be backwards compatible; that is, they will run currently available code and binaries.

2 USES OF PREFIX-SUM

Use of prefix-sum for several different purposes is suggested: (i) for bringing down to the execution the parallelism of an algorithm, (ii) as a target for code analysis, (iii) for system support, (iv) as a key step in a high-bandwidth register-file algorithm, and even (v) as a paradigm in designing parallel algorithm.

2.1 For Preserving Algorithm Parallelism

Below, we refer to a parallel PS-RAM algorithm, in which Priority concurrent-writes are allowed, as a PS-RAM+ algorithm. The following theorem assumes that in the given PS-RAM+ code, parallelism is expressed using, possibly nested, PARDO commands. A PARDO command is given in the following form
for $i = 1$ to k pardo "statement"
where the statement (which can actually be a sequence of statements) depends on the index i, and is referred henceforth as a *thread*. (Note that often the notion of threads refers to parallelism which is more coarse-grained than what is meant here.) The threads corresponding to all values of i between 1 and k are executed concurrently. We refer the reader to books on parallel algorithms, e.g., [Ja], to see how PARDO commands are used for code-like description of PRAM algorithms. It appears that using PARDO (as described above) for expressing parallel algorithms is without loss of generality, since other forms of expressing parallelism can be efficiently emulated using PARDO commands (using time and total number of operations as the efficiency parameters). Indeed, books on parallel algorithms (again, we take [Ja], as an example) use primarily PARDO commands for code-like description of algorithms.

In order not to hide the simplicity of the main ideas in the proof of the completeness theorem, the presentation is rather informal, demonstrating feasibility, rather than provide the most general way, or present the most attractive implementation. For instance, executing (successively or at once) several instructions of the same thread may sometimes exhibit better locality of reference than mixing instructions from different threads; still, we intentionally suppressed the potential performance trade-offs involved. I do not see a reason for having problems with incorporating solutions for such important practical problem within our general framework.
For simplicity, we assume that the execution of each instruction takes one round.

Completeness Theorem Given a PS-RAM+ algorithm, appropriately programming it using a serial RAM programming language (e.g., C) for a PS-RAM machine will yield the same parallel time, and total number of operations, in order of magnitude, as manifested in a parallel (PS-RAM+) programming language.
Comments (i) The theorem means that the value of each of the two efficiency parameters, time and total number of operations, is preserved up to a constant factor. That constant factor happens to be a small integer. (ii) The translation from the RAM code to a PS-RAM machine, provided in the proof below, can be done by a compiler.

Proof of the Completeness Theorem (sketch) Suppose that a parallel algorithm is given as PS-RAM+ code. It is shown how to transform it into a RAM code and then how a compiler can translate the RAM code into execution on a PS-RAM machine. In the remainder of this subsection, we show that the code analysis will always be able to extract from the RAM code the same level of parallelism as from a PS-RAM+ code. This means that it never runs into a situation where it cannot reconstruct that parallelism in full. Note that the code analysis will have a problem reconstructing that parallelism if this is in contradiction to preserving the semantics of the RAM code.

Using a known procedure, we will aim to replace a PARDO command, where no nesting of PARDO commands occur, by, one or more, simple FOR commands of the form
for$(i = 1; i \leq k; i{+}{+})$.
It is also worth noting that using standard loop-unrolling (see, e.g. Chapter 4 in [HP]), a compiler will be able to: (i) increment the FOR loop counter (i.e., generate the required intermediate values for i), and (ii) remove branch instructions at the end of unrolled iterations, thereby removing data and control dependencies.

Simplifying the problem . *Getting a single operation per thread.* The PARDO-based code enables dividing the operations in the threads into parallel "rounds", preserving its semantics. Each thread has one operation per round. So, for the sake of having an easy proof, break the PARDO commands into a successive sequence of PARDO commands each of whose threads consisting of a single operation only.

How to emulate the simplified PS-RAM+ code by RAM? We now split each round into two subrounds: first reads are performed, storing values of variable which are written into temporary values, and second prefix-sums and writes. The reason for doing that is similar to the emulation of the ILP-RAM by the RAM in the previous section.

Next we show how to handle concurrent-writes. Suppose that all writes are performed into disjoint variables (i.e., no concurrent-writes occur). Had memory addressing been disambiguated at run-time, we could have simply used the relevant pieces of the PS-RAM+ threads in a RAM code, and then extract back the original parallelism at run-time (since no write conflicts occur). However, it is possible that a compiler could not have predicted that no concurrent writes actually occur (perhaps because for other runs of the same program on a different input concurrent-writes are possible). As we shall see the PS command enables extracting back parallelism by the compiler even for that case.

Suppose now, that several threads in a PARDO command of a PS-RAM+ code could write into the same variable. Suppose further that our PS-RAM+ code assumes the so-called priority-concurrent-write convention where the smallest indexed among the writes is executed. For this presentation, let us number potential write commands into a variable A, by j_1 through j_k, where $j_1, j_2, ..., j_k$ is an increasing sequence of thread numbers. This means that the programmer wants the smallest index write to be executed. Hence, once a write has been executed the other writes are not needed. A simple, but general, way for implementing this, in a FOR loop of an RAM code, uses an auxiliary "gatekeeper" variable G which is initialized to zero, and for each write command an auxiliary variable B_i, $1 \leq i \leq k$, which is initialized to 1. Then, prior to write command i, $1 \leq i \leq k$, we add a prefix-sum instruction of the form $PS(G, B_i)$. Exactly one (the one occurring in the smallest indexed thread) prefix-sum command will result in the value 0 assigned to a B_i variable; its write command into G will be the one which will be executed. Observe that the computation can be enhanced by **preaborting** subsequent prefix-sum commands with respect to G, as well as their respective write commands into A. This completes implementing concurrent-writes using a RAM code which will be properly executed on a PS-RAM machine.

Readers who have dealt with parallelizing compiler issues, should be able to understand why our approach is novel and powerful. First, note that the above gatekeeper solution treated also the case where it is not feasible to determine at compile time whether more than one write into the same variable can occur. By doing so, we allow all the threads in the RAM code to be designated (scheduled) by the compiler for concurrent execution! Recall that, to guarantee correctness, compilers must take into account any memory access that cannot be excluded at compile time. So, using current methods (i.e., without the gatekeeper solution) the compiler would have had to fully serialize memory accesses that with our solution can be done in parallel, giving a big advantage to our solution. This reliance on parallel PS for resolution of some data dependences in parallel, rather than serially, is the main new idea in this proof.

It remains to consider nested PARDO commands. Prefix-sum can be used to flatten the nested structure into a succession of (single) PARDO commands, so that in each PARDO command the threads are indexed by successive integers starting from 1. Then the above arguments can be applied.

To finish the proof, we show how to dispense with the need of calling the PS command by name. It is shown how to replace each PS command of the PS-RAM+ code by a small piece of RAM code. For each $PS(A, B)$ command in the PR-RAM+ code, add an auxiliary variable C (defined within a limited scope only) and replace $PS(A, B)$ by the following 3-command sequence:

$C = B$
$B = A$
$A = A + C$

(Alternatively, $PS(A, B)$ can be replaced by the following 2-command sequence: $A = A + B$; $B = A - B$, adding and arithmetic operation but avoiding the need for an auxiliary variable.) The compiler can now be "trained" to retrieve such a sequence as $PS(A, B)$. This completes the proof outline of the Completeness Theorem.

Using computer architecture terminology these show how to use the parallel prefix-sum instruction in the assumed PS-RAM machine for reducing read-after-write (RAW) hazards and write-after-write (WAW) hazards, as defined in [HP].

It has already been emphasized that the above presentation aims to convey the new concept rather than attempting the most efficient worst-case implementation, or any improvement at all beyond the worst-case. For instance, the explicit use of "gatekeeper" is often not needed. In the test example, the problems for which we suggested above to use gatekeepers arise, but gatekeeper are actually not used.

2.1.1 Example and empirical results

In the version of this paper available through my home page, `http://www.umiacs.umd.edu/~vishkin/`, some high-level PS-RAM code and a corresponding RAM code is presented. Although the RAM code is shorter than the general method used to prove the Completeness Theorem, this provides an example for a relationship between the two kinds of code. The codes implement breadth-first search (BFS) on undirected graphs.

To demonstrate the overall advantage of using parallel algorithms and RAM code to implement them, we empirically compared the amount of instruction-level-parallelism coming out of two different codes: (i) *the first code:* a standard serial (RAM) code for BFS, and (ii) *the second code:* a RAM code which implements a parallel (PS-RAM+) version of BFS.

For generating inputs, we used [Knuth] which generates random graphs.

For the first code the amount of ILP was measured by a simulator due to Lam-Wilson [LW]. That simulator allows several alternative "abstract machine models" (each representing some potential power of compilation and run-time methods). Given an abstract machine model and the sequential trace of an execution of the program, the platform determines for each instruction in the trace the earliest time in which it could be been executed, assuming that as many instructions as needed could have been executed in parallel.

For the second code, we used an enhanced version of the simulator, which allows to execute prefix-sum commands in one round.

The table below presents the amount of parallelism derived from two of Lam-Wilson's machine models: (i) CD-MF and (ii) SP-CD-MF. Both models assume perfect memory disambiguation (or memory aliasing), perfect register renaming, full control dependence analysis and the ability to follow unlimited multiple flows of control. The SP-CD-MF model also assumes the ability to predict branches and to speculatively execute instructions according to these predictions.

Machine model	Nodes & Edges	First code	Second code	Factor Improved
(i)	100 & 750	26.59	216.74	8.15
	500 & 2500	20.77	248.24	11.95
(ii)	100 & 750	41.43	351.81	8.49
	500 & 2500	31.65	443.29	14.01

Recall that the amount of parallelism is defined as the total number of instructions divided by the parallel time. As the table indicates, the amount of parallelism which was derived from the first code was by a factor of between 8.15 to 14.01 larger than from the serial code. This was obtained without any increase in the total number of instructions!

2.2 Through Analysis of Code

Through dynamic and static analysis and through permuting the order of instructions a processor could recognize a plurality of individual prefix-sum instructions which can comprise a multiple prefix-sum instruction. In case several iterations of a loop can be performed in parallel, a prefix-sum instruction could be used to spawn and maintain a thread per each active iteration.

2.3 For System Support

As a computation proceeds, prefix-sum computation could be useful for allocation of system resources to concurrent tasks, including the allocation of registers, memories and functional units, and including load balancing among tasks coming from single or multiple computational threads.

Consider, for instance, the following *trivial example*. Suppose that we run three threads each needs three integer add operations, and we have five single clock functional units which could serve as integer adders. Taking a prefix-sum of the 3 values can be used to assign a serial number to each add operation and thereby schedule, say, the 3 additions of the first thread and two of the additions of the second thread to clock one on the five adders, and the third addition of the second thread and the three of the third to clock two on the first four adders.

Note that in concept the multi-threading in this paper is much more general than vector processing. It is more similar to [TEL-95], which advocated simultaneous multi-threading, than to [Tera], where frequent thread switching is used (but at a given clock only one thread can issue an instruction). The similarity to [TEL-95] is also limited though. In that paper each thread must be guided by completely separate control (i.e., program counter). However, in a few places in this abstract where we considered light threads occuring in the execution of the same problem, whose instructions could potentially be issued by SIMD-type control, we intentionally avoided saying whether each such light thread will be led by a separate program counter.

2.4 Conflict-Free Access to Multiple Single-Ported Memory Registers - A Dynamic Solution

ILP-based architectures require high-bandwidth register files. The continuing growth in multiple-issue of instructions per cycle makes the design of fast memories, such as register files, even more important and challenging.

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.

Basis for comparison: For comparison with our new solution, note that the number of registers is x^2, where x is the number of functional units.

This section proposes a new solution aimed at providing high register read and write bandwidth with substantially less hardware (each register file has a single read port and a single write port). The solution is dynamic in that it could be implemented at run-time, as soon as the input information each of its stages needs becomes available. Similar to a recent static (i.e., compile-time) solution, in [MV] (and [VW]), the new solution: (i) relies 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; and (ii) gets as given the allocation of instructions to steps during execution. The latter is a nice feature since the system could spare this allocation for the purpose of other optimizations rather than for satisfying the above conflict-free rule. In order to use our solution, the designer of a machine employing it will set the value of the parameter h - a "time horizon"; h will be the number of future cycle that our dynamic solution takes into account at any point in time. Next, we present an illustrated example.

Example Figure 3 demonstrates the solution, by way of example, for a time horizon of 4 cycles. For simplicity, assume that every instruction reads a single operand from a module, does some processing, and stores a value back, possibly to a different module – all of which take a single cycle. The wide columns in Figure 3 correspond to the values that can be used in a given cycle. The narrow column to its left represents the split-cycle (at the beginning of the cycle) in which the value is fetched from a module. The narrow column to the right of the wide column represents the split-cycle (at the end of the cycle) in which the value is stored into a module. Each (piecewise straight) line connecting a point in a store column to a point in a (higher numbered) read column represents a value stored at the split-cycle of the former column whose next time to be fetched is the split-cycle of the latter column. Our solution will attach a module number to each such line (see the caption of Figure 3), specifying to the execution in which module to store that value; the later fetch of that value is done from that module. Note that the whole problem we need to solve is *how to assign numbers to those piecewise straight lines.* The main idea of the new solution is as follows. *For each such value allocate to it an index (or serial number) with respect to the other values which will be needed in the same future cycle. This serial number is the number of the module into which the value will be stored. The key observation is that no two values needed in a future cycle will come from the same module.* This observation is in line with the conflict-free rule.

The connection to prefix-sums To implement this idea we use a special counter for each of the h (future) cycles in the current time horizon. The counter of a cycle maintains the number of values which have already been allocated to that future cycle. Upon the completion of each cycle, the following will be done for each such future cycle using some special hardware: (i) increment the counter for the future cycle; and (ii) allocate a serial number to each value. In effect, such a computation is similar to the "prefix-sums" problem and is assumed to be efficiently implemented in hardware. To be more precise, for each cycle i, $1 \leq i \leq h$, we are given a counter c_i and $m(i)$ values $x_{i_1}, x_{i_2} \cdots x_{i_{m(i)}}$; we wish to allocate the serial number $c_i + j$ to each value x_{i_j}, where $1 \leq j \leq m(i)$, and update the counter c_i itself into $c_i + m(i)$.

There is yet one caveat. Upon the completion of a cycle, it is possible that several values will be stored in the same module. (In the above example, this possibility is demonstrated by the assignment of lines b and d to $R7$.) However, the reason that careful hardware design should be able to cope with that is that if two values are stored in the same module their future retrievals will never be in the same cycle. (In the example, the value of line b is needed earlier than the value of line d.) So, the storage of some values in modules can be slightly delayed without causing any delay to future cycles; actually, it is not hard to figure out that a delay in storing several values into a module which increases linearly with the number of those values would not cause any delay to future cycles.

Comparison with the register-matrix solution Assume that the register files are required to enable fetching of up to y values per cycle for some integer y. Our solution would use y registers, which is the minimum possible number of single-ported register files. Note that this compares favorable with the register-matrix solu-

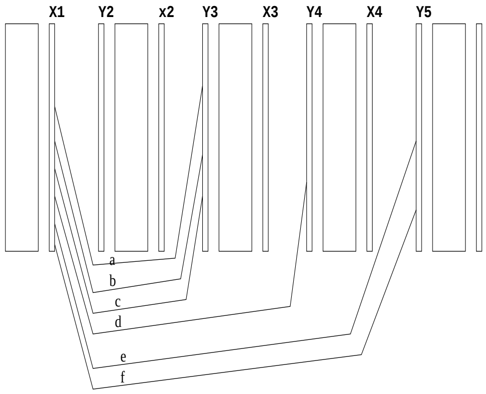

Figure 3: An illustration of an input for the new solution; x_1 to x_4, respectively, are the store columns of cycles number 1-4, respectively, and y_2 to y_5, respectively, are the fetch columns of cycles number 2-5, respectively. Following cycle 1, we have lines a, b, c which represent three values whose next use is in cycle 3, line d represents a value whose next use is in cycle 4, and lines e, f represent two values whose next use is in cycle 5. Suppose that prior to cycle 1 the counters for cycles $2, 3, 4$ and 5 are $5, 5, 6$ and 0, respectively. Then, upon the completion of the prefix-sum computation at the end of cycle 1, the counter for cycle 2 remains 5; line a will be assigned the number 6 (which means that its value will be stored in module $R6$ till cycle 3); line b the number 7 (its value stored in $R7$); line c the number 8 (its value stored in $R8$); the counter for cycle 3 increased to 8; line d the number 7 (its value stored in $R7$) and the counter of cycle 4 increases to 7; line e the number 1 (its value stored in $R1$); line f the number 2 (its value stored in $R2$); and the counter of cycle 5 increases to 2.

tion which employs x^2 register for feeding x functional units.

Extensions For simplicity our discussion focused on how to move temporary values among registers, allowing instructions to be executed in the cycles for which they have been allocated. However, the data needed for, or produced in, a given cycle is not limited to temporary values used in previous cycles (and which reside in the register files). So, we must consider ways for fetching in time such data to avoid delaying instructions which need these data. One possibility is that the compiler would extend our solution to include prefetching of these data into registers; this should be done so that the conflict-free rule will still be satisfied with respect to all values accessed in a cycle.
During the execution of a program we will have certain future knowledge concerning its next h cycles, as explained next. We could employ a policy where following an instruction each value which is known to be needed again within the next h cycles is directed to a module. Other values will not be stored in the modular register files, but rather be directed to other (slightly slower) memories.

2.5 For algorithmic thinking

Prefix-sum could serve as an abstraction paradigm for algorithmic thinking. For example, consider the standard integer sorting algorithm and have it described, as follows:
1. Using a separate prefix-sum with respect to each key-value, find the plurality of the key-value and the index of each key relative to identical-value keys.
2. Using a single prefix-sum relative to the pluralities computed in Step 1, find for each key-value the smallest index, in the output sorted vector, of a key equal to it.
3. For each input key add its serial number, computed in Step 1, to the smallest index of its key-value in the output, computed in Step 2, to get its index in the output.

I am not aware of a more elegant and compact way for describing this algorithm.

3 CONCLUSION: ON THEORY VERSUS PRACTICE

The current paper focuses on some theoretically intriguing aspects of an envisioned system update, and is apparently one of the first to deal with the timely issues of ILP that way. However, this need not hide the possibility of putting the paper in a wider context, as done below. Observing that most recent developments in the ILP scene are aimed at improving performance only for existing binaries, it is not surprising that our new ideas are generally "orthogonal" to those developments, and perhaps the most challenging task would be to incorporate them all in one system.

Abstractions used, and what they mean to practitioners Abstractions are used: (i) To demonstrate feasibility; that is, even if a different solution for a similar problem ends up being used later, having provided the statement of a problem and a possible solution is worthwhile. (ii) To help articulate ideas and simplify the presentation. We used several kinds of abstractions. (i) *Omissions:* At the outset, ILP manifests itself through instructions whose execution overlaps in time due to pipelining and multiple-issue. However, there is much more to modern processors than these. Some major aspects include branch prediction, caching, vectoring, and out-of-order execution. Mostly, these aspects are orthogonal to the ideas presented in the current paper and therefore, for the sake of simplicity, it was beneficial to suppress them. Also suppressed is the aspect of the new processor functioning in an assemblage of processors, whether in a centralized or distributed shared-memory, or other, architectures. This should not be misinterpreted as ignoring these aspects, or the merit of overall incorporation. (ii) *Constant factors:* The proof of the Completeness theorem and other considerations were not meant to minimize constant factors. In fact the constant factors that I observed are small and should be attractive for current generation architectures. (iii) *Worst-case:* Practical consideration dictate "making the common case fast". PS provides an excellent tool for that. For instance, it is straightforward to apply it for not wasting operations on non-active threads. Another aspect of "worst-case" is anticipation of pathological cases. As the BFS code suggests, there is rarely a need in practice to fully simplify the code as was done in the proof of the Completeness Theorem. (iv) *Space complexity:* The proof of the Completeness Theorem assumes that the compiler doubles the memory to resolve write-conflicts, and implement multiple-PS. Since the extra memory is used once there is no need to use much additional memory for this purpose in practice. Sometimes extra virtual variables are needed to signal lack of dependence among instructions, while, in fact, much less extra memory is required.

The PRAM theory and its gap with multiprocessors The PRAM theory of parallel algorithms offers a rich knowledge base for the exploitation of fine-grained parallel computation. Papers such as [Vi84], [Va90] proposed ways for mapping fine-grained parallel computing into the coarse-grained reality of multi-processors. The paper [CKPSSSSvE], on the other hand, suggests that to be practical, parallel algorithms will have to be designed for coarse-grained parallel computing. As the next paragraph indicates, these problems do not exist with ILP.

PRAMs fit ILP Allowing fine-grained computing, ILP-based uniprocessors are fundamentally different than those multi-processors. The current paper implies that such uniprocessors fit fine-grained PRAM algorithmics and this ought to to be studied more.

Programming The present paper was motivated, in part, by the general thesis in [Vi96a] which calls for using parallel algorithms in order to enhance uniprocessor performance and by worrying that parallelism which is clear to the algorithm designer may be lost. Interestingly, the completeness theorem suggests that which programming language to use (and whether parallel or standard) for that purpose may not be that important; rather, "disciplined" programming suffices.

Programmability The PRAM is known to be a convenient abstraction for algorithms designers. Note that the PS-RAM allows nested PARDOs which makes it even more convenient to program than flat PARDOs as per the ILP-RAM; observe, that this important difference between the two models was not noted elsewhere in this paper. It is similar to a difference between the VRAM in [B90], and Blelloch's language NESL [BCHSZ].

Acknowledgement Helpful comments by Bob Cypher, Yossi Gil and Mooly Sagiv are gratefully acknowledged.

References

[AG] 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.

[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.

[B90] G.E. Blelloch. *Vector Models for Data-Parallel Computing.* MIT Press, 1990.

[BCHSZ] G.E. Blelloch, S. Chatterjee, J.C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *Proc. 4th ACM PPOPP*, 102–111, 1993.

[CSV] A. Chandra, L. Stockmeyer and U. Vishkin. Constant size reducibility. *SIAM J. Computing* 13, 1984, 423-439.

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

[CS] D. Culler and J.P. Singh. *Parallel Computer Architecture (alpha draft).* Morgan Kaufmann, San Mateo, California, to appear.

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

[GMR] P. B. Gibbons, Y. Matias, and V. Ramachandran. The QRQW PRAM: Accounting for contention in parallel algorithms. In *Proc. 5th ACM-SIAM SODA*, 638–648, 1994.

[H] J. Hastad. Almost optimal lower bounds for small depth circuits. *Advances in Computing Research* 5, 1989, 143-170. JAI Press, Greenwich, Conn.

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

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

[K-96] 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.

[Knuth] D.E. Knuth. *The Stanford GraphBase.* Addison-Wesley, New York, NY, 1993.

[LW] M.S. Lam and R.P. Wilson. Limits of control flow on parallelism. In Proc. 19th ISCA, 46–57, 1992.

[Ma] M.M. Mano. *Digital Logic and Computer Design.* Prentice Hall, 1979.

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

[OV95] R. Orni and U. Vishkin. Two Computer Systems Paradoxes: Serialize-to-Parallelize, and Queuing Concurrent-Writes. Preprint, September 1995.

[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.

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

[So] G. Sohi. Multiscalar processors. Transparencies of a talk available from: `http://www.cs.wisc.edu /~mscalar/multiscalar\_talks.html`.

[SUIF] SUIF Compiler System: Advanced Compiler Technology for Scalable Parallel Machines. Summary of Research in 95-96. http://suif.stanford.edu/darpa/summary96.html.

[TEL-95] 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.

[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. Parallel-Design Distributed-Implementation (PDDI) General Purpose Computer. *Theoretical Computer Science* 32 (1984), 157–172.

[Vi96a] U. Vishkin. Can parallel algorithms enhance serial implementation? *CACM*, 39(9), 1996, 88–91 .

[Vi96b] U. Vishkin. U.S. Patent Application No. 08/667,554, June 1996.

[VW] U. Vishkin and A. Wigderson, Dynamic parallel memories. *Inf. and Cont.* 56, 174–182, 1983.

A A HARDWARE IMPLEMENTATION

A Hardware implementation for small-integer prefix-sum follows. Generally, one would expect each (single) prefix-sum to need an adder. see e.g. [AG]. In this section we show that hardware cost can be drastically reduced, in one case where the input values for a multiple prefix-sum are limited.

We say that a multiple prefix-sum instruction
$PS\ R_0\ R_1$
$PS\ R_0\ R_2$
...
$PS\ R_0\ R_k$
is *small-integer* if each input value in registers R_1 through R_k can only be a small integer; for example, integers between 0 and 3. No limits are placed on the input value in the base register R_0.

The number of individual prefix-sum instructions that can be combined into a compound prefix-sum hardware implementation should certainly be machine dependent. In case this number is $h(< k)$ we will first perform the first h individual prefix-sum instructions, then the next such h instructions and so on till the full k-wide multiple prefix-sum has been executed.

The proposed hardware implementation will demonstrate that a small-integer prefix-sum may not need much more hardware resources than standard binary integer addition. This, together with demonstration of usability, may support the case for implementing compound prefix-sum in hardware, or even adding one or more compound prefix-sum functional units to standard processors.

We proceed to outline an implementation of the multiple small-integer prefix-sum instruction:
$PS\ R_0\ R_1$
$PS\ R_0\ R_2$
...
$PS\ R_0\ R_h$
into
$z_0\ =\ R_0$
$z_1\ =\ R_0\ +\ R_1$
$z_2\ =\ R_0\ +\ R_1\ +\ R_2$
...
$z_h\ =\ R_0\ +\ R_1\ +\ R_2\ + ... +\ R_h$

1. Compute the base-zero prefix-sum, using dedicated hardware. By base-zero we mean that the input values of registers R_1 through R_h are taken as they are, but the base register R_0 is taken as 0. Store these prefix-sums in an intermediate array V of h elements v_1, v_2 ... v_h.
2. Compute the following "suffix sum", using dedicated hardware into an intermediate array U of h elements u_1, u_2 ... u_{h-1}.
$u_{h-1}\ =\ R_h$
$u_{h-2}\ =\ R_{h-1}\ +\ R_h$
...
$u_1\ =\ R_2\ +\ ...\ +\ R_h$

3. Compute the last output element z_h by adding the last intermediate element v_h to the original input base R_0, using an adder.

Overview of the rest of the implementation. The rest of the output elements, z_1 through z_{h-1}, are found by using arithmetic only on a small number of bits, either by additions of elements of the intermediate array V to the input element R_0 or by subtractions of elements of intermediate array U from z_h. These additions and subtractions are based on first determining the index, i, of the most significant bit at which R_0 and z_h differ, and constructing an auxiliary number, w, which has one in its i-th bit and zeroes in all bits less significant than the i-th bit, and is identical to both R_0 and z_h, for more significant bits than the i-th bit. The remaining steps of the hardware implementation are as follows:
4. Find index i as follows: Apply exclusive-or bitwise to the binary representation of R_0 and z_h. i is the index of the most significant bit for which the exclusive-or yields one.
5. Derive w.
6. Get the values of the rest of the output which are smaller than w by adding the low-index elements of the intermediate array V to R_0.
7. Get the values of the rest of the output which are equal to or greater than w by subtracting the high-index elements of the intermediate array U from z_h.

Because h is small, and all the elements in R_1 through R_h are small, the additions and subtractions of the last two steps involve only short carries. They can be implemented using short look-ahead carry generators (for look-ahead carry generators, see, e.g., [Ma]).

A.1 Relative complexity

The above presentation is rather instructive for producing an efficient hardware embodyment of prefix-sum and should suffice for arguing that its hardware cost (in gate count and delay) justifies including our prefix-sum hardware component in a processor. However, it would be interesting to advocate this inclusion in a more principled way since the underlying technology will change and the level of ILP in a processor may drastically increase (the latter would imply that h may increase more than other processor design parameters, such as the number of bits in a computer word). The unfortunate reality is that a robust notion of (absolute) hardware cost is not possible, since costs, and relative costs, change as technology changes; furthermore, each of the components may have more than one implementation (for instance, if look-up tables are used, different mixes of circuitry and loop-up tables will produce different gate counts and different delays) and their relative costs may be different for different technologies. This leads us to the rather pragmatic notion of relative complexity.

By way of example, consider a superscalar processor which enables a multiple issue of m. Such a processor will (at least) have to provide some "minimal features" for quick coordination of m instructions (that are being issued concurrently). If we are able to make concrete sense of such minimal features, this will provide us with (complexity) yardsticks for arguing whether a proposed addition or change are within the *same* budget for building the processor, or perhaps only a few per cent higher. This explains the term relative complexity.

Consider the following "binary count" problem. Input: m bits. Output: a log m - bit representation of their sum. I suggest to use this problem (or a prefix-sum version thereof) as one of the yardsticks for the cost of building such a processor since the processor will, for instance, need to count how many jobs (e.g., instructions) are, or remain, active.

The main purpose of the current discussion is to briefly demonstrate how to argue that the proposed prefix-sum construction is within budget. For that, observe the following (assuming that $h = m$). We would like to argue that each of its components is similar in cost to hardware which actually exists in a superscalar processor that allows a multiple issue of m. For instance, we relate Step 3 to a standard adder, and steps 1 and 2 to binary count.
If this discussion sounds obvious, observe that the above relative complexity argument is rather non-conservative for theory. Following the work of [CSV] and [H] the parallel circuit complexity of binary count is reasonably well understood: for unbounded fan-in parallelism, a polynomial size circuit requires depth (number of layers) which is at least proportional to $\log m / \log\log m$. This helps articulating the differences between the studies of [CSV] and [H] and the current paper: (1) As a matter of staying "within-budget" relative to the overall cost of the processor, we will be willing to regard the depth of binary count of h as constant (and even unit) time, and its cost (in terms of say gates) as acceptable. (2) In the current paper constant factors do matter.