Explicit Instruction Parallel Bridging Models
Specifications and Assumptions

Shlomit Dascal
Advisor: Prof. Uzi Vishkin

January 23, 1998

Introduction:

Following are specifications of the extended-C necessary to write the high-level code for the explicit instruction parallel bridging models (section 1), the instruction set, e.g., assembly language (section 2) and our execution assumptions and techniques for figuring out time and work counts (section 3).

This document relies on [4] for direction and context.

These notes were written with the help of Efraim Berkovich, and under the supervision of Prof. Uzi Vishkin.

Contents

1 Extended-C Specifications

This section defines the necessary parallel extensions to the high level programming language *C* in order to implement the new ideas presented for the spawn-MT and EMT (Elastic Multi-Threading) models.

1.1 Program Control

Two new explicit control commands, spawn and spawnsync, are added in order to create respectively a parallel asynchronous block or a parallel step-synchronous block, to be executed in parallel.

There is one synchronization barrier (implicit) at the end of the asynchronous block (also referred to as "join"). Every child thread spawned at the start of the block arriving at the synchronization barrier will terminate at this point.

The synchronous block's execution is step-wise synchronous, meaning that there is a synchronization point after each instruction within the threads.

For both synchronous and asynchronous blocks there is also a "join" synchronization barrier at the end of the block. Every child thread spawned at the start of the block arriving at the "join" synchronization barrier will terminate at this point.

1. The "spawn" command:

```
    spawn(offset, size) {
        statement1;
        statement2;
        ...
        }
  where:
    offset  - is the thread-ID of the first thread
    size    - is the number of threads that will be spawned, and executed
              in parallel
```

   This command will generate the following threads:

```
    Thread-#offset:        Thread-#offset+1:  ...  Thread-#offset+size-1:
      statement1;            statement1;              statement1;
      statement2;            statement2;              statement2;
      ...                    ...                      ...

      (join)                 (join)                   (join)
```

Spawn commands can be nested within other spawn commands.

For example:

```
spawn(offset1, size1) {

    statements...;

    spawn(offset2, size2) {
        statements...;
    } /* end child spawn */

    statements...;

} /* end main spawn */
```

2. The "spawnsync" command:

```
    spawnsync(offset, size) {
        statement1;
        statement2;
        ...
        }
  where:
    offset  - is the thread-ID of the first thread
    size    - is the number of threads that will be spawned, and executed
              step-synchronously in parallel
```

This command will generate the following threads:

```
Thread-#offset:        Thread-#offset+1:  ...  Thread-#offset+size-1:
  statement1;            statement1;             statement1;
  (sync)                 (sync)                  (sync)
  statement2;            statement2;             statement2;
  (sync)                 (sync)                  (sync)
  ...                    ...                     ...
  (join)                 (join)                  (join)
```

1.2 Variables

1. For each thread we need local variables. We put the definitions of such variables within the spawn-block (i.e., thread). These variables are called thread-variables.

   Local thread variables can be directly accessed by descendent-spawns (these accesses will be resolved during compilations and/or run-time).

   Example declarations:

   ```
   int x;                          /* shared local integer variable */
   spawn(offset, size) {

       int n;                  /* thread integer variable */
       char c[10];             /* thread char array */
       register int v;         /* thread var, try to keep in register */

       statement1;
       statement2;
       ...
       }
   ```

   In the example for each thread spawned 'n' will denote a separate local variable (as opposed to 'x' which exists only once since it is declared outside the spawn scope).

2. The "thread ID" is a new variable that can be referenced by the symbol "$". It is useful inside the scope of a spawn.

   Example:

   ```
   a)  n = $ + 10;     /* set n = thread-id + 10 */

   b)  int   a[100],b[100];
       ...
       /* using the thread-id as index into arrays */
       a[$] = a[$] + b[$];
   ```

3. The "parent thread ID" is a new variable (provided by a "parent thread" service) and can be referenced by "P($)". It can be used recursively to access higher ancestors' thread-ID, but is only meaningful as deep as spawns are nested.

   Example:

```
a)  n = P($) + $; /* for example for contiguous indexing in array */

b)  int   a[1000]; /* to initialize a={1,2,3..10,1..10,1..10,...} */

    spawn (10; 1) {
      spawn(10; 1) {
          spawn (10; 1) {

            a[100*P(P($)) + 10*P($) + $] = $;
          }
      }
    }
```

1.3 New Operations

A parallel prefix-sum instruction, which is an extension of the standard binary summation, and a parallel-mark are also new.

1.3.1 Prefix-sum

Adding the element to the base for small integers, is interpreted according to its context as ordered or unordered prefix-sum, and it follows [3] and [4].

1. The "individual" prefix-sum:

   It is convenient to express the prefix-sum operation as a group of "individual" prefix-sum operations in the following manner:

   ```
     ps(X, Y)
   where:
     X - is the destination variable which contains the base
     Y - is the variable to sum into the base, and it contains
         an element
   ```

   This will perform an "individual prefix-sum" with respect to X (whose value is initially A) and Y (whose value is initially B), means the following inseparable ("atomic") outcome:

   ```
   (i)  X = A + B
   (ii) Y = A
   ```

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.

2. The multiple prefix-sum ("multiple-PS") can be achieved by one of the following:

   (a) The sequence of instructions:

```
  ps(X, Y1);
  ps(X, Y2);
  ps(X, Y3);
  ...
  ps(X, Yn);
where:
  X  - is the destination variable which contains the base
  Yi - are the variables to sum into the base, each contains
       an element
```

   This performs the ordered (order as defined by the serial semantics) prefix-sum of the input base X (whose value is initially A) and the elements Y1,Y2, ... , Yn (whose values are initially B1, B2, ..., Bn respectively) into Y1,Y2, ... ,Yn and X, respectively. This will result in:

```
Y1 = A
Y2 = A + B1
Y3 = A + B1 + B2
...
Yi = A + B1 + B2 + ... + Bi-1
...
Yn = A + B1 + B2 + ... + Bn-1
X  = A + B1 + B2 + ... + Bn-1 + Bn
```

   (b) Several "ps" instructions from different threads of a parallel synchronous block:

```
int X;              /* shared local integer variable */

spawnsync(...) {
    int     Y;  /* thread integer variable */
    ...
    ps(X, Y);
```

```
    ...
}
```

This performs the ordered (order as defined by the thread IDs) prefix-sum of the input base X (whose value is initially A) and the elements Y of \$=1 (local variable of thread number 1), Y of \$=2, ... , Y of \$=n (whose values are initially B1, B2, ..., Bn respectively) into Y of \$=1, Y of \$=2, ... , Y of \$=n and X, respectively.

This will result in:

```
Y (of $=1) = A
Y (of $=2) = A + B1
Y (of $=3) = A + B1 + B2
...
Y (of $=i) = A + B1 + B2 + ... + Bi-1
...
Y (of $=n) = A + B1 + B2 + ... + Bn-1
X  = A + B1 + B2 + ... + Bn-1 + Bn
```

(c) Several "ps" instructions from different threads of a parallel asynchronous block:

```
int X;              /* shared local integer variable */

spawn(...) {
    int     Y;  /* thread integer variable */
    ...
    ps(X, Y);
    ...
}
```

This performs the "unordered prefix-sum" of the input base X (whose value is initially A) and the local element Y of \$=1 (local variable of thread number 1), Y of \$=2, ... , Y of \$=n (whose values are initially B1, B2, ..., Bn respectively) into Y of \$=1, Y of \$=2, ... , Y of \$=n and X, respectively. Namely, this will result in some permutation-order amongst the Y variables, and in:

```
X  = A + B1 + B2 + ... + Bn-1 + Bn
```

1.3.2 Read-Mark-Write Register (mark)

Reading a variable, checking its value and according to it modifying it, follows [4]. This is used in cases where arbitrary concurrent writes might have been useful to resolve contingency.

1. The "individual" mark:

   It is convenient to express the mark operation as a group of "individual" mark operations in the following manner:

   ```
       mark(X, Y)
     where:
       X - is the base variable, for checking and marking
       Y - is the result variable of the operation (success
           or failure)
   ```

   This will perform an "individual mark" with respect to X (whose value is initially A) and Y, means the following inseparable ("atomic") outcome:

   ```
   start
   read the contents of X (which is initially A)
   if (A == 0) then
       Y = 1           (marking was successful)
       and X = 1       (mark)
   else
       Y = 0           (marking failed)
       and no write to X is done.
   end
   ```

   In and of itself, this instruction has an effect similar to a simple "test-and-set" instruction when it is tested for zero (see [2]). However, the difference between the MARK instruction and a "test-and-set" instruction is that several MARK instructions may be cascaded into a "multiple-MARK" instruction.

2. The multiple mark ("multiple-MARK") can be achieved by having several "mark" instructions from different threads of a parallel asynchronous block:

   ```
   int X;                   /* shared local integer variable */
   X = 0;                   /* initializing X to UN-MARKED! */
   ```

```
spawn(...) {
    int     Y;      /* thread integer variable */
    ...
    mark(X, Y);
    ...
}
```

only ONE of the threads will MARK the base (global) variable X, and get in return $Y = 1$ (in it's local variable Y). This thread can then proceed to do a write or other operations exclusively. The other threads will know of their failure to mark (their $Y = 0$) and will not perform their operation. This is useful to resolve contingency.

Notes:

- It is the programmer's responsibility to RESET (set to ZERO) the contents of the shared base variable BEFORE any marking is attempted, otherwise none of the marking attempts will be successful.
- After this instruction executes, X will never be zero because either the marking was successful or X wasn't zero in the first place.

2 Instruction Set Specifications

This section defines the extensions and changes to the assembly language *MIPS* for the new spawn-MT and EMT (Elastic Multi-Threading) models.

2.1 Introduction

1. Standard instruction-code is executed using a program-counter (PC) which selects the next instruction to be issued. The new instruction code replaces the PC or uses multiple-PCs, by using a Spawn-Join mechanism ("Spawn Control Units", SCUs) in instruction code and "Thread Control Units" (TCUs) during execution, possibly in conjunction with an "enabled-instructions (EI)" memory.
2. At every point in time, instructions that can all be executed concurrently are duplicated at run-time into the "Thread Control Units" (TCUs).
3. This document presents the definitions for a set of instructions that implement these new ideas and was appropriate for the problems we studied.
4. A parallel-block, whose contents will be executed asynchronously, is defined between a "spawn-join" pair (described in the "new instructions" section of this document). The "join" is also referred to as a "synchronization barrier", since the program does not proceed beyond the "join" until after all threads have reached it.
5. A parallel-block, whose contents will be executed synchronously, is defined between a "spawnsync-join" pair (described in the "new instructions" section of this document).
6. A parallel prefix-sum instruction, which is an extension of the standard binary summation, a parallel-sum-immediate and a parallel-mark are also added to the machine. We also proposed to implement the "join" instructions by using the parallel-sum-immediate hardware.

2.2 General Comments

1. The instructions set is based on the MIPS instruction set, presented in appendix A of [1].
2. Instructions NOT mentioned here are the same (except maybe in the bit-count of each field) as in MIPS
3. Comments in the assembler files begin with a sharp/pound (#) sign.
4. The instruction size has to be larger than 32 bits (as used in MIPS), due to additional information that has to be included in it. We do not think this should be a concern since 64 bit architectures are already available at this point.

2.3 Notations

1. Rs, Rd and Ri- could mean any shared (general purpose) register (R1, R2, ...) or any thread register (R1\$, R2\$, ...), unless further explicit constraints are specified. Usually: Rs would mean a source-register and Rd a destination-register
2. R0 - is the register that always has the ZERO value
3. R0\$ - is the thread register that always has the value of the thread-ID (also denoted as \$)
4. RiP\$ - is the i-th thread-register of the parent-thread (i.e., spawning parent thread), for nested spawns, such as R1P\$, R2P\$, ...
   NOTE: exists only in the EMT configuration

2.4 Addressing Modes

In addition to the indirect addressing mode in MIPS, there is an array indirect addressing as follows:

```
  OFF(Rs), SZ[Ri]

where:
  OFF - is a constant offset added to the base address
  Rs  - is the source register containing the base address
  Ri  - is the index register containing the index to the array
  SZ  - is a constant size (the array entry size),
        which will multiply the index (e.g., in Ri)
```

The address is therefore:

```
(contents of Rs) + OFF + ( (contents of Ri) * SZ )
```

The load and store instructions will thus have two different instances. The “regular” instruction (lw, sw, lb, etc.) will use the simple (non-array) addressing mode. In addition there will be corresponding new instructions (lwa, swa, lba, laa, etc. - the “a” stands for “array”) which will use the array addressing mode.

Examples:

```
1) lwa   R3, 24(R1), 4[R2]  ==>  R3 = Memory[R1 + 4*R2 + 24]

2) For each thread to access its element in the array (where the
   thread-ID acts as the index):

     lwa   R2$, 24(R1$), 8[R0$]  ==>  R2$ = Memory[R1$ + 8*R0$ + 24]

3) For ``regular'' (MIPS like) indirect addressing simply use:

     lw   R2,  256(R1)   ==>  R2 = Memory[R1 + 256]
```

2.5 Load & Store Instructions

1. All the load and store commands available in MIPS, and in addition new instructions (lwa, swa, lba, laa, etc. - the "a" stands for "array") which will use the new array addressing mode, as described above.

2. Load Immediate (li) - same as MIPS:

```
li   Rd, Imm
```

2.6 Arithmetical & Logical Instructions

1. The general format for the arithmetical and logical instructions is:

```
    op   Rd, Rs1, [Rs2]

where:
    op  - is the operator (add, sub, and,  or, etc.)
    Rd  - is the destination register where the result of the
          operation will be placed
    Rs1, Rs2 - are the source registers (Rs2 is only necessary in
          binary operations)
```

2. The multiplication and division instructions use two destination registers similar to the format above, as opposed to MIPS which uses dedicated HI and LOW registers.

2.7 Jump & Link Instructions

1. Jump and Link Register (jalr) - same as MIPS, where the default for the destination register (Rd) is R31 (which means shared register number 31):

```
        jalr  Rs, Rd
```

2. Jump and Link (jal) will now be of the format:

```
  jal   label, Rd
where:
  label - is the Target address to jump to (and is >= 26 bits)
  Rd    - is the Destination Register to save the instruction to
          jump back to in (also called link-register),
          and its default is R31.
```

2.8 New Instructions

1. **Spawn** (of the spawn-join pair) - entering the asynchronous parallel mode by creating various asynchronous parallel threads.

   In general, the SPAWN instruction, of the format:

```
  spawn    Rd, OFF, Rs, REGS
where:
  Rs    - is the number of threads to be spawned
  OFF   - is the thread-ID for the ``first'' thread generated (Immediate)
  Rd    - is the ``spawning'' register (keeps count of the threads),
          and is implicitly initialized to zero at the beginning
  REGS  - is the number of local registers a thread needs (Immediate)
```

   will have the following effect:

   (a) For some integer k (a machine parameter), threads 1,2, ... , k will be initiated, in the asynchronous mode, e.g. each thread will advance independently of the others until it reaches the join instruction.

   (b) If k is less than the value of Rs, the following instruction is generated into the “Spawn Control Unit” (SCU):

```
                SPAWN-RECUR   k+1, Rs, Rd
```

   (c) These k+1 commands are executed (the k threads and once the first among the threads is done the spawn-recur).

   Notes:

- The hardware implementation of thread-registers could be:
  (a) using a fixed-local set of registers for each thread
  OR
  (b) using a shared set of registers for all threads, with portions of it dynamically allocated to each running thread
  In case of (a) the "REGS" (i.e., the number of registers a thread needs) argument is not necessary in the "spawn" instruction. In order to maintain an implementation-free definition at this stage, we include the "REGS" in the "spawn" instruction.

- Extra care must be taken when using the "jal" and "jalr" instructions with the default register within threads if there is more than one instance of it in the same parallel block. In this case all threads will be looking at the same shared register to link back from the function call, and it must be correct for each of the threads, executing at its own pace.

  For example, when the following code appears inside a thread section:

  ```
  spawn ...{
      ...
      function1(...);
      ...
      function2(...);
      ...
  }
  ```

  the compiler (while generating the assembler), cannot use the default link-register (R31) in both function calls. In order for the execution to be correct, and since all threads are using the SAME link-register, different link-registers should be used, as follows:

  ```
  spawn ...
  ...
  jal    function1, R31
  ...
  jal    function2, R30
  ...
  join ...
  ```

  It should also be observed that (if the write collision is resolved) another way to use the registers in jump-and-link situations is by using shared registers (as opposed to thread-registers), since all threads will need to link (i.e., jump back) to the same program address, as shown in the example above.

2. **SpawnSync** (of the spawnsync-join pair) - creating various step-wise synchronous parallel threads.

   In general, the SPAWNSYNC instruction, of the format:

   ```
     spawnsync    Rd, OFF, Rs, REGS
   where:
     Rs     - is the number of threads to be spawned
     OFF    - is the thread-ID for the ``first'' thread generated (Immediate)
     Rd     - is the ``spawning'' register (keeps count of the threads),
              and is implicitly initialized to zero at the beginning
     REGS   - is the number of local registers a thread needs (Immediate)
   ```

   will have the following effect:

   (a) For some integer k (a machine parameter), threads 1,2, ... , k will be initiated, in the step-wise synchronous mode, e.g., the order of threads will be preserved, and each instruction will be executed across threads before continuing to the following instruction.

   (b) If $k <$ value of Rs, the following instruction is generated into the "Spawn Control Unit" (SCU):

   ```
   SPAWNSYNC-RECUR   k+1, Rs, Rd
   ```

   (c) These k+1 commands are executed (the k threads and once they are all done, the spawnsync-recur).

3. **Join** (of the spawn-join pair) - synchronization barrier at the end of the parallel block (asynchronous or synchronous mode threads) started with the "spawn" or "spawnsync" instruction.

   In general, the JOIN instruction, of the format:

   ```
       join    Rd, Rs
   where:
       Rs   - is the number of threads that were spawned (to wait for)
       Rd   - is the ``spawning'' register (keeps count of the threads)
   ```

   and the terminating condition is:

   ```
   if ( sumi(Rd, 1) != Rs) TERMINATE $
   ```

will have the following effect:

(a) The instruction `sumi(Rd, 1)` will be executed. This will cause the following:

```
If A was the contents of Rd then Rd = A + 1.

If there are several individual sumi computations, all with
respect to Rd, then their outcome will be as if they were done
in some serial order.
```

(b) Thread $ will terminate (according to the condition described above)

4. **Parallel Sum Immediate (sumi)** - adding the immediate value to the base, in parallel.

   In general, the SUMI instruction, of the format:

```
  sumi   Rd, Imm
where:
  Rd  - is the destination register which contains the base
  Imm - is the immediate value we are adding, and should be a small
        integer
```

will have the following effect:

(a) It is convenient to express the parallel-sum-immediate operation as a group of "individual" parallel-sum-immediate operations in the following manner:

"sumi Rd, B" - an "individual parallel-sum" with respect to Rd (whose value is initially A) and B (which is an immediate), means the following outcome:

```
Rd = A + B
```

In and of itself, this instruction has an effect similar to a simple "add immediate" instruction. However, the difference is that several SUMI instructions may be cascaded into a "multiple-SUMI" instruction.

(b) The "multiple-SUMI" can be achieved by one of the following

   i. The sequence of instructions:

```
sumi   Rd, B1
sumi   Rd, B2
sumi   Rd, B3
...
sumi   Rd, BN
```

performs the parallel-sum of the input base Rd (whose value is initially A) and the B1,B2, ... , BN (which are immediate values) This will result in:

```
Rd  = A + B1 + B2 + ... + BN-1 + BN
```

ii. Several "sumi" instructions from different threads of a parallel asynchronous or synchronous block:

```
spawn/spawnsync ...
sumi  Rd, 1
join  ...
```

performs the parallel-sum of the input base Rd with a "1" contributed by each of the threads (or another small integer immediate value). This will result in:

```
Rd  = A + 1*N    (N being the # of threads)
```

5. **Prefix-Sum (ps)** - adding the element to the base for small integers, is interpreted according to its context as ordered or unordered prefix-sum, and is based on [3].

   In general, the PS instruction, of the format:

```
    ps   Rd, Rsi
where:
    Rd  - is the destination register which contains the base
    Rsi - is the source register which contains the i-th element,
          and should be a small integer
```

   will have the following effect:

   (a) It is convenient to express the prefix-sum operation as a group of "individual" prefix-sum operations in the following manner:

   "ps Rd, Rs" - an "individual prefix-sum" with respect to Rd (whose value is initially A) and Rs (whose value is initially B), means the following inseparable ("atomic") outcome:

```
(i)  Rd = A + B
(ii) Rs = A
```

   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.

   (b) The "multiple-PS" can be achieved by one of the following

   i. The sequence of instructions:

```
ps   Rd, Rs1
ps   Rd, Rs2
ps   Rd, Rs3
...
ps   Rd, RsN
```

performs the ordered (order as defined by the serial semantics) prefix-sum of the input base Rd (whose value is initially A) and the elements Rs1,Rs2, ... , RsN (whose value is initially B1, B2, ..., BN respectively) into Rs1,Rs2, ... ,RsN and Rd, respectively. This will result in:

```
Rs1 = A
Rs2 = A + B1
Rs3 = A + B1 + B2
...
Rsi = A + B1 + B2 + ... + Bi-1
...
RsN = A + B1 + B2 + ... + BN-1
Rd  = A + B1 + B2 + ... + BN-1 + BN
```

ii. Several “ps” instructions from different threads of a parallel synchronous block:

```
spawnsync ...
ps    Rd, Rs$
join    ...
```

performs the ordered (order as defined by the thread IDs) prefix-sum of the input base Rd (whose value is initially A) and the elements Rs$ of $=1 (local register of thread number 1), Rs$ of $=2, ... , Rs$ of $=N (whose values are initially B1, B2,..., BN respectively) into Rs$ of $=1, Rs$ of $=2, ..., Rs$ of $=N and Rd, respectively. This will result in:

```
Rs$ (of $=1) = A
Rs$ (of $=2) = A + B1
Rs$ (of $=3) = A + B1 + B2
...
Rs$ (of $=i) = A + B1 + B2 + ... + Bi-1
...
Rs$ (of $=N) = A + B1 + B2 + ... + BN-1
Rd  = A + B1 + B2 + ... + BN-1 + BN
```

iii. Several “ps” instructions from different threads of a parallel asynchronous block:

```
spawn ...
ps    Rd, Rs$
join  ...
```

performs the unordered prefix-sum of the input base Rd (whose value is initially A) and the elements Rs$ of $=1 (local register of thread number 1), Rs$ of $=2, ... , Rs$ of $=N (whose values are initially B1, B2,..., BN respectively) into Rs$ of $=1, Rs$ of $=2, ..., Rs$ of $=N and Rd, respectively. Namely, this will result in some permutation-order amongst the local thread registers.

Example: In the array compaction problem in section 3.5 we are required to copy elements of one array which are "marked" into a new array. In order to find the new index of an element which will be copied into the new array we will use the following instructions in the asynchronous mode:

```
li   R1$,  1       # R1$ = 1
ps   R1,   R1$     # each thread - to find the new index
```

Note that R1 is a shared register to which ALL threads refer, and R1$ is a (local) thread register specific to each thread. The index sought will be in R1$ after these instructions are executed. And after all the threads are done, the total number of index entries used will be in R1. Due to the semantics for the asynchronous mode we can not predict which thread will eventually write to which index in the array. The definition of the array compaction problem tolerates the inability to predict this.

6. **Prefix-Sum to Memory (psm)** - adding the element to the base for small integers, where the base is a memory address. Only available for usage as unordered prefix-sum in the inter-thread asynchronous parallel block context (see the ps instruction for context interpretation).

   In general, the "psm" instruction, of the format:

```
  psm   Mem, Rsi
where:
  Mem - is the memory address which contains the base, of the
        form: OFF(Rd)
  Rsi - is the source register which contains the i-th element,
        and should be a small integer
```

   will have the same effect as the (unordered) prefix-sum instruction, but using a memory location as the base-element instead of a register, and will have a special locking mechanism on the memory location.

7. **Read-Mark-Write Register (mark)** - reading a register, checking its value and according to it modifying it. This is used in cases where arbitrary concurrent writes might have been useful to resolve contingency.

The instruction format:

```
    mark    Rs, Rd
where:
    Rs     - is the base register, for checking and marking
    Rd     - is the result of the operation (success or failure)
```

The instruction will perform the following:

(a) "mark Rs, Rd" - an "individual mark" with respect to Rs and Rd, means the following inseparable ("atomic") outcome:

```
start
read the contents of Rs
if (Rs == 0) then
    Rd = 1              (marking was successful)
    and Rs = 1          (mark)
else
    Rd = 0              (marking failed)
    and no write to Rs is done.
end
```

In and of itself, this instruction has an effect similar to a simple "test-and-set" instruction when it is tested for zero. However, the difference between the MARK instruction and an "test-and -set" instruction is that several MARK instructions may be cascaded into a "multiple-MARK" instruction.

(b) The "multiple-MARK" can be achieved by having several "mark" instructions from different threads of a parallel asynchronous block:

```
move    Rs, R0
spawn   ...
mark    Rs, Rd$
  .
  .
  .
join    ...
```

only ONE of the threads will MARK the base (global) register Rs, and get in return Rd$ = 1 (in it's local register Rd$). This thread can then proceed to do a write or any other unique operation. The other threads will know of their failure to mark (their Rd$ = 0) and will not perform the unique operation. This is useful to resolve contingency.

Notes:

- It is the programmer's responsibility to RESET (set to ZERO) the contents of the register BEFORE any marking is attempted, otherwise none of the marking attempts will be successful. Reseting the register can be simply done by writing zero to it, for example:

  ```
  move    Rs, R0
  ```

- After this instruction Rs will NOT be zero in any case: if succeeded marking, due to the write; if failed marking, wasn't zero in the first place.

8. **Read-Mark-Write to Memory (markm)** - reading a memory address, checking its value and according to it modifying it in memory. This is used in cases implementing arbitrary concurrent writes. It is only available for usage in the inter-thread asynchronous parallel block context (see the mark instruction for further details on the context).

   The instruction format:

   ```
       markm    OFF(Rs), Rd
   where:
       Rs    - is the base address for the memory access
       OFF   - is a constant offset from the base address (Immediate)
       Rd    - is the result of the operation (success or failure)
   ```

   The instruction will perform the following:

   ```
   start
   read the contents of MEM[Rs + OFF]
   if (MEM[Rs+OFF] == 0) then
       Rd = 1 (marking was successful)
       and MEM[Rs+OFF] = 1 (write)
   else
       Rd = 0 (marking failed)
       and no write to MEM[Rs+OFF] is done.
   end
   ```

   Note:
   It is the programmer's responsibility to RESET (set to ZERO) the memory contents BEFORE any marking is attempted, otherwise none of the marking attempts will be successful. Reseting the memory can be simply done by writing zero to it, for example:

   ```
   sw    R0, OFF(Rs)
   ```

3 Execution Assumptions & Techniques for Figuring Out Time & Work Counts

This section explains the assumptions and techniques we used to obtain the performance values from the instruction code for the serial model and the new spawn-MT and EMT (Elastic Multi-Threading) models. The last subsection gives a simple example of performance time and work counts in the asynchronous mode.

3.1 TIME Counts

General Rules

- Each instruction is scheduled to the earliest possible cycle in which it can be executed to completion (subject to data dependences, control dependences and assumptions regarding branch prediction and speculation).
- Simple branch prediction with speculation is allowed. However, we limit that to cases where we know that the probability of success is high. For example, when checking if it's the last element (all the element-threads have to check that), in the list-ranking, all the elements except one will get a negative answer, so we allow speculation.
- To simplify treatment of memory accesses, if, due to spatial-locality, or temporal-locality, the address could be found in the cache, we assumed unit-time memory access. Otherwise, a cache miss was assumed, and we applied a penalty of 5 time cycles.

Constraints

- An instruction cannot be scheduled to issue prior to any data dependence resolution (either at compile time and/or run time)
- An instruction cannot be scheduled to commit prior to any control dependence resolution. For example: the earliest that an instruction which depends on a branch resolution can commit is the time cycle after the branch.
- For simplicity first-referenced data loads are considered a cache miss. That is, the first access to a "data-block" cannot be used prior to 5 time cycles from the load request.
- Prefetch of data from memory becomes possible once the memory address can be resolved, i.e., if an explicit prefetch instruction could have been inserted in that place.

  Example:

  In the second round of the loop - the load in line 65 can be prefetched without having to wait for the add in line 64, since it is a constant known offset. The "prefetch" statements below show how this could have been achieved at run-time (and they aren't part of the actual code).

```
     can prefetch element_SIZE+rank_OFF(R5) - for the first round
61: FINAL:
62:    add   R8, R8, R4
63:    sw    R8, rank_OFF(R5)
64:    addi  R5, R5, element_SIZE
65:    lw    R8, rank_OFF(R5)
     can prefetch element_SIZE+rank_OFF(R5) - for the next round
66:    blt   R5, R7, FINAL
```

- Given a branch instruction followed by a load, the commit of the load instruction (i.e., its last cycle) has to occur after the branch is resolved (unless speculation is used). This permits prefetch across branch instructions.
- Given a spawn instruction followed by a load, the commit of the load instruction (e.g., its last cycle) has to be within the spawn block. This permits prefetch across spawn instructions, if the memory address can be resolved (no additional data dependencies).
- For simplicity first reference operations to memory (which implicitly require a load), such as the prefix-sum-to-memory (psm), are considered a cache miss. That is, the instruction is counted as if it commits only after 5 time cycles.

Further Explanation Regarding Load Instructions

- Spatial locality exists within a thread. A load from a location adjacent to one which was accessed in the last few cycles (in the same thread) takes 1 cycle. All the loads from adjacent locations (in the same thread) can be done at once, thus incurring a single penalty of 5 cycles.

  NOTE: For simplicity (and robustness) we avoided specifying the size of a block (for the use of the spatial locality assumption), the time (cycle) span (for the use of the temporal locality assumption), the cache size, the number of threads allowed at once, etc. In the manual counting we will generally interpret that to mean that spatial locality exists within our threads, but not across threads or spawn-join blocks.
- Temporal locality exists within a thread. When data from a specific address is already in the cache (even if a store is performed to the same address) subsequent load requests from the same address are available in the cache (and are consistent with the recently stored information).

- Load from the stack (e.g., relative to the Stack Pointer (SP)) counts as a single cycle since it is assumed the information was stored immediately before the call, and therefore available in the cache.

Further Comments

- The separation of the instruction code into blocks helps with the counting. The numbering of the time cycles is relative to the beginning of the block which contains the instruction. There can be overlap of time cycles between consecutive blocks if all the dependencies are cleared soon enough in the preceding block. In this case time cycles within a block might be denoted with negative and/or zero values (zero being the last time cycle of the preceding block).
- Unconditional jump instructions that can be replaced by inlining (and are present for readability purposes only) are not counted as a separate time-cycle. Note that this does not include jump instructions used for controlling loops (which can never be inlined).
- In cases where loop unrolling is used the number of unrolled loops and the overhead depend on the number of registers available on the machine (denoted as "R"). The time cycle count is therefore dependent on it too. In unrolled-loops that do not need additional registers for each copy we assume, for the serial case, a lower-bound time count. This means that the k-copies of the unrolled loop will always be exactly the number of times the loop would execute (N mod k == 0).
- We used scheduling of the first and second iterations of loops for evaluating the total time per block, including time overlap between the iterations.
- A line in the instruction code which consists of a label only is not considered an instruction (and not counted in the execution).

3.2 WORK Counts

- Each instruction, except joins and jumps, is counted as one work unit.
- Spawn instructions are counted as a single work unit for all the threads, due to the necessity of instruction fetching and decoding. There is no specific work associated with each thread spawned since that is part of the control hardware, which, from the comparison point of view, exists in "serial" machines too (for ILP handling).
- Join instructions are counted as many times as the number of spawned threads – 1 work unit per thread. If N threads were spawned, then N work units are counted for the join

instruction. This is consistent with our conservative approach. Since each thread's join must be fetched and decoded we are counting this work, even though there is no specific work associated with each thread joined since that is part of the control hardware, which, from the comparison point of view, exists in "serial" machines too (for ILP handling).

- Plain jump instructions are not counted as work (since they can be replaced by inlining and are present for readability purposes). This does not include jump instructions used for controlling loops.
- In cases where loop unrolling is used the number of unrolled loops and the overhead depend on the number of registers available on the machine (denoted as "R"). The work count is therefore dependent on it too.
- An average case approach was used, taking the expected value E[p] as the factor of decrease of the input size. For example, in the list ranking problem, the factor of decrease of the list size for each iteration was: $\frac{3}{4}$ for "no-cut list ranking".

3.3 "Conservative" Bounds on Total Execution Time

Upper Bound
Total Execution Time $\leq \frac{W}{P} + T$

Lower Bound (for serial case)
Total Execution Time $\geq T$

Where:

- W - is the work counted (as described above)
- T - is the critical time counted (as described above)
- P - is the number of instructions that can be simultaneously issued in each cycle

Note: See the scheduling lemma proved in [4].

3.4 General Optimization Assumptions

- Register usage is optimized for global, local and re-used variables. Special attention was given to minimizing loads and stores by not overwriting registers which contain variables that are frequently used. The number of registers used is virtually unlimited, but we attempted re-using registers which would not create unnecessary dependencies (thus resulting in stalls). When doing loop unrolling a parameter was used to denote the number of available registers (which is not unlimited).

- We did not take into consideration the influence of the optimization techniques (such as inlining and loop unrolling) on the size of the code (and therefore on spatial locality in the instruction-cache).

3.5 A Simple Example for the Asynchronous Mode

We use the following toy problem to demonstrate a "depth-cut" of the bridging model from algorithm through instruction code to execution scheduling.

The array compaction problem *Input.* An array $A = A(1), \ldots, A(n)$ of elements; a second array $B = B(1), \ldots, B(n)$, whose elements are integers between 1 and n; and a third array $C = C(1), \ldots, C(n)$ of bits (each valued zero or one).
The *compaction problem* is to copy each element of $A(i)$ for which $C(B(i)) = 1$, for $1 \leq i \leq n$ and $C(B(i)) = 1$, into a different entry of a fourth array $D = D(1), \ldots, D(s))$, where s is the (a priori unknown) numbers of ones in $C(B(i))$, for $1 \leq i \leq n$. 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 $j < i$.

Algorithm level A high-level program follows.

```
....
int   x;

x = 0;
Spawn(1,n);
{ int  e;

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

A verbal description follows. x initialized to 1; it is the base variable for the prefix-sum computation The Spawn command spawns threads 1 through n. Although not explicitly stated, a Join command is implied by the end of the scope of the Spawn command (the closing parenthesis }); The Join command comprises a parallel sum computation which increments and "invisible" variable (say y); this invisible y is initialized to 0 by the Spawn command. Thread $ initialize 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 in the serial state occurs and n gets the size of the compacted array.

Instruction set level For simplicity, we provide next informal instruction code.

```
      Mnemonics  Instruction             Cycle no.  Comments
      ...
      li         R1,0                        1
      lw         R2, 0(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
      sw         R1, 0(Rn)                  15
      ...
```

The load-immediate (li) command initializes $R1$ to 0. The address of n is in Rn; Load-word (lw) brings n into $R2$. Spawn into $R2$ threads indexed 0 to $R2$-1; use 4 local registers per thread; the Join command matching the Spawn command will count terminating threads into $R3$. $R0\$$ includes always the thread index $; it is a 5th local register. Using standard MIPS convention $R0 = 0$. B_OFF is the offset for array B. We use our new Load–word-array (lwa) command for directly accessing the array address; the 4 multiplier is since MIPS addressing is done in bytes. If $R3\$$ equals 1 a prefix sum is performed, and the value from array A is copied into the compacted array D using lwa and store-word-array (swa) commands. Each thread reaching Join increments $R1$ by one using a parallel sum-integer command (SUMI), which is part of Join, and once $R1$ become equal to $R3$ Join switches back to serial state and stores into Rn the size of the compacted array.

Scheduling instructions into cycles along the critical path of the dependence graph follows our general rules. Whenever cache misses are possible we used a penalty of 5; when possible we loaded this memory by a prefetch; however, since we avoided speculations, we committed to a load only after its preceding control dependence has been resolved.

Assume that $\frac{1}{4}$ of the original elements are in the compacted array. Using the framework of the Scheduling Lemma, 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 $T(n) = 15$. Assuming parallelism of p, the scheduling Lemma implies a time upper bound of $\frac{5\frac{3}{4}n + 4}{p} + 15$.

For comparison, 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 assembly code is considerably more involved and hard to follow than the parallel assembly code.

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;
```

Serial 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 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
                                            1st iter. 2nd iter. 3rd iter.
        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
```

References

[1] Hennessy J.L., Patterson D.A., *Computer Organization and Design - The Hardware / Software Interface*. Morgan/Kaufmann. 1994.

[2] Hennessy J.L., Patterson D.A., *Computer Architecture - A Quantitative Approach*, 2nd Edition. Morgan/Kaufmann. 1996.

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

[4] Vishkin U. et al, *Multi-Threading Bridging Models for Explicit Instruction Parallelism*. TBD

Index

Note: *ASM* marks items which are relevant to the assembly code specifications and *C* marks items relevant to the extended-C specifications.