A Pipelining Model Which Pipelines Blocks of Code

Joachim Beer[1]

TR-90-053

October, 1990

[1]International Computer Science Institute, 1947 Center Street, Berkeley, CA 94704 USA

A pipelining model which pipelines blocks of code

Joachim Beer
International Computer Science Institute
Berkeley, CA

Abstract

This paper presents a new technique of software pipelining and an architecture to support this technique. Rather than attempting to pipeline a sequence of individual instructions, the presented technique tries to pipeline entire blocks of code, i.e. the units to be pipelined are chunks of code, instructions within each code block might or might not be pipelined themselves. In this model blocks of code are identified which can be executed in a pipelined fashion. Neighboring blocks of code do not need to be data independent; pipeline stages can feed results and/or synchronization markers on to the next pipeline stage. The architecture can be seen as an attempt to use classical pipelining techniques in a multiprocessor system. The architecture consists of a circular pipeline of ordinary microprocessors. Advantages of the architecture are: unlike supercomputers and VLIW architectures the system can be based on commercial micro-processors, it avoids the high overhead of process startup, and it is not restricted to vectorizing only inner-loops. Simulation studies show the viability of the architecture and the associated execution model.

Introduction

This paper presents a new technique to pipeline blocks of code and an architecture to support this technique. Pipelining the execution flow of a program is of course nothing new. There is probably not a single field within the realm of information processing where it has not been successfully employed. Pipelining principles can be applied at various levels:

- *Pipelining at the micro-instruction level.*
  While one micro-instruction is executing, the next micro-instruction will be fetched. *Requirements:* independent micro-instruction sequencer and execution unit.

- *Pipelining at the machine(macro)-instruction level.*
  Assembler instructions are independently fetched, decoded, and executed. *Requirements:* autonomous fetch, decode, and execution units (possibly several independent execution units).

- *Pipelining of blocks of code.*
  Requirements: a pipeline of tightly coupled independent processors.

It should be obvious that these concepts are orthogonal. Whereas nowadays pipelining at the first two levels is supported by every micro-processor, it is not so clear how large blocks of code can be overlapped in a pipeline fashion.

In traditional architectures pipelining is usually performed at the machine instruction level. Classical supercomputers achieve their performance through the use of multiple execution units capable of performing vector operations and clever instruction sequencing to keep all units as busy as possible [Sch87]. Lately VLIW architectures have entered the field [Fi83, Ra89]. VLIW architectures and the concept of software pipelining arose out of work done on automatically compacting microcode operations to schedule operations to be executed in parallel [Fi81]. VLIW architectures and their compilers try to take the use of multiple functional units and instruction scheduling one step further and to apply it to general instruction sequences and not just vector operations [El86, Ra89]. However, these pipelining schemes work on the instruction level, i.e. through scheduling of *individual* instructions.

Rather than attempting to pipeline a sequence of individual instructions, the technique presented in this paper tries to pipeline entire blocks of code, i.e. the units to be pipelined are chunks of code, instructions within each code block might or might not be pipelined themselves. In this model blocks of code are identified which can be executed in a pipelined fashion. Neighboring blocks of code do not need to be data independent; pipeline stages can feed results and/or synchronization markers on to the next pipeline stage.

Multiprocessor systems also attempt to executed independent code chunks in parallel; however, in most multiprocessor systems the high overhead of starting a process on a remote processor and the cost of subsequent inter-processor communication make these systems only a viable alternative for very large processes. The proposed architecture can be seen as an attempt to use classical pipelining techniques in a multiprocessor system. Figure 1 shows the global system architecture which consists of a circular pipeline of ordinary microprocessors. Each microprocessor executes a block of code and initiates the next block to be executed on its right neighbor. Since it is a whole block of code which is being executed on a pipeline stage this block can contain complex conditionals as long as these conditionals don't affect the execution order of the following blocks. This is in contrast to the scheduling of individual instructions in which conditional statements always cause a severe problem. Other advantages of the the presented architecture are: unlike supercomputers and VLIW architectures the system can

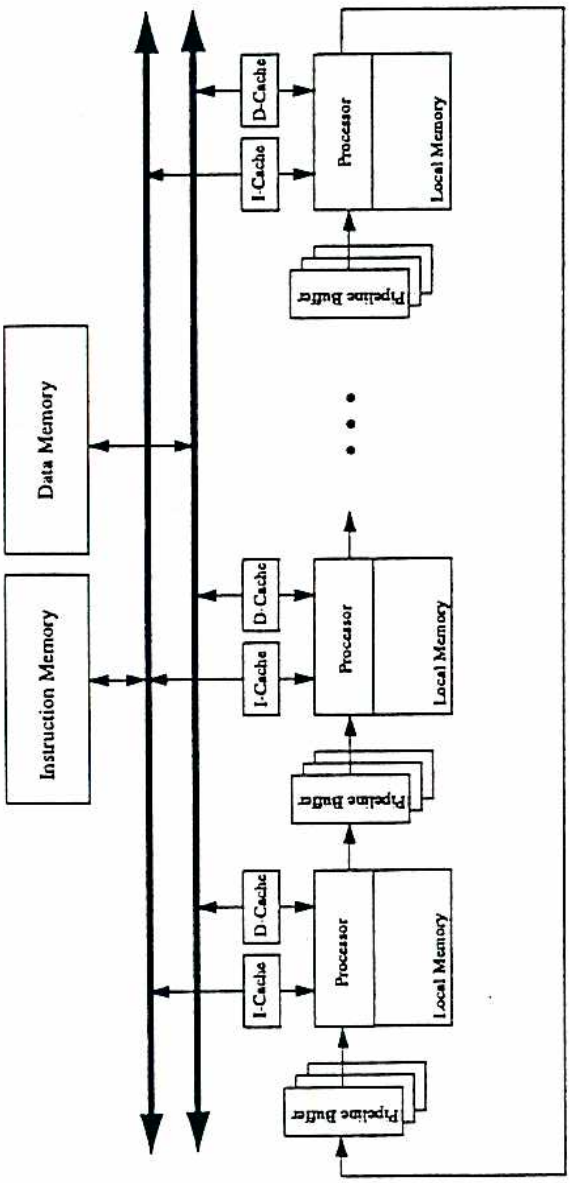

Figure 1: Global system architecture

be based on commercial micro-processors, it avoids the high overhead of process startup, and it is not restricted to vectorizing only inner-loops.

Architecture and Execution Model

The proposed architecture is shown in Figure 1 and consists of a pipeline of processors, where every processor is connected to its right neighbor via pipeline buffers. The last processor is connected with the first one, thereby creating a circular arrangement of processors. Between adjacent processors a set of pipeline buffers buffer and synchronize the data exchange of the processors. The pipeline flow is unidirectional from left to right. Hardware semaphores built into the buffer guarantee that the right processor can only access buffer cells after the left processor has transferred data objects into the respective buffer cells. In order to keep bus contention to a minimum the proposed system contains separate data and instruction buses and respective caches in front of the processors. Every processor also has a local memory which might contain instructions and/or data. However, many of these design parameters are not important for the basic execution model.

The basic execution model which allows to pipeline blocks of code can be explained through an example which shows how function calls can be pipelined in the proposed system. Consider the following fragment of a C program:

```
p( int A, int B, int C);

{  i = 5;
   j = A * B;
   q( C, j );
   .
   .
}


main()
{  .
   .
   p( 5, array[1,3], X );
   .
   .
}
```

A function p (with 3 scalar arguments) is called and in turn calls another function q. Even in this simple example we can identify blocks of code which can be pipelined. Consider how this little program would typically be compiled:[1]

[1]We shall assume that the actual parameters are being passed in special argument registers, A1,...,An.

```
        load #5,A1
        .
        address
        calculation for
        array[1,3]
        .
        load array[1,3],A2
        load X,A3
        call p

p:      allocate
        activation record
        i=5
        j=A1*A2
        load C,A1
        load j,A2
        call q
        .
        .
```

The prologue to the function call and the function body itself are two code segments which are, except for the function arguments, data independent. For example, why should the function *p* not allocate its environment, and maybe even perform some local computation, while the actual arguments are being set up? The function *p* can also start using those arguments that are already known; why should a procedure always have to wait until *all* its parameters have been determined? In the proposed system the caller of a function would proceed as follows:

- Load the entry and return address of the function into the pipeline buffer (typically the first and second words of the buffer are reserved for this purpose).
- Load one-by-one all the arguments of the function into the pipeline buffer.

Concurrently the callee proceeds as follows:

- Read the entry address of the function to be executed.
- Start executing the function.
- Whenever actual arguments of the function are accessed which have not been provided by the left neighbor stall the processor until they become available.

For our particular example the execution would look as follows:

```
Processor 1                                  Processor 2

call_left p,ret          ⟹
load #5,A1*              ⟹      p:  allocate
.                                    activation record
address                              call_left q,ret
calculation for                      i=5
array[1,3]                           j=A1 * ...
.                                              delay until A2 is available
load array[1,3],A2*      ⟹          A2
load X,A3*               ⟹          load j,A2
stop                                 load A3,A1
                                     stop
```

Instead of waiting until all the arguments[2] have been set up the function call is issued right away, thereby enabling the next processor to start execution of the called procedure. Note that it is necessary to explicitly name the return address since the return address is not the next instruction following the *call* instruction as in the sequential case. The return address can be determined by the compiler and the caller can keep executing code as long as it can be guaranteed that no conflicts with concurrently executing code on other processors can arise. When possible conflicts might arise the current block is terminated with a *stop* statement. When a *stop* statement is reached the processor will look in its left pipeline buffer to see if a new entry address has been provided for the next thread of code to be executed. Note that a function which has started execution on one processor does not also need to terminate on the same processor. In the example above there is, for instance, no code to deallocate p's activation record. The "stop" statement at the end does not mean it is the end of the function body, but rather serves as a barrier beyond which execution can not progress. In this particular case whenever the function q returns it will push the return address as an entry address to the next processor which will then continue with the remaining body of function p.

To minimize *busy waiting* for the next entry address each processor has been provided with a *set* of pipeline buffer blocks. The pipeline buffer blocks serve as queues which contain executable code blocks. If a program exhibits sufficient parallelism there will always be some blocks which contain executable tasks.

Of course, this execution model is not restricted to only pipeline function calls. The compiler can take any code sequence and try to partion it into code blocks which can be executed in a pipeline fashion. And this is were the real power of the execution model lies. A prime candidate for pipelining are loop iterations.[3] Consider for example the following matrix multiplication code to

[2]The argument registers have been denoted A1-A3. Register Ai* refers to the ith argument register of the right neighbor. Hence the register which the left processor refers to as Ai* is being referred to by the right neighbor as Ai.

[3]Or recusive function calls. The execution model does not make any distinction between

calculate $A * B = C$ (without loss of generality we assume A, B, C to be square matrixes with dimension *dim*):

```
for (i=0;i<dim;i++)
    for (j=0;j<dim;j++) {
        c[i][j] = 0;
        for (k=0;k<dim;k++)
            c[i][j] = c[i][j] + a[i][k]*b[k][j];
    }
```

By turning the outer-loop into a recursive loop invocation this nested loop can be transformed into:

```
Matrix_multiply: if (i<dim) {
                     PUSH_RIGHT(Matrix_multiply,i+1,a+dim,c+dim);
                     for (j=0; j<dim; j++) {
                         c[j] = 0;
                         for (k=0; k<dim; k++)
                             c[j] = c[j] + a[k]*b[k][j];
                     }
                     STOP;
                 }
Continue:        ...
```

where $PUSH\_RIGHT(label, arg_1, arg_2, ..., arg_n)$ pushes the entry address and the arguments of the code block to be executed next to the current processor's right neighbor.[4] In this particular example the entry address is the begining of the outer loop and the arguments are the loop index and pointers to the respective row vectors[5]. In this case we do not need to provide an explicit return address because the continuation is statically known.

It is immediately obvious that by unrolling the outer loop the maximal parallelism of this example is given by the dimension of the matrixes. If the dimension of the matrixes is greater than the number of processors in the pipeline computation will wrap around the pipeline, i.e. the last processor will deposit a new task into the first processor's pipeline buffer.

However, rather than unrolling the outer-loop the compiler can also choose to unroll the middle loop. This will yield a much higher degree of parallelism as there are now dim^2 iterations which can be pipelined. The following code segment shows the unrolling of the middle loop:

---

iteration and recursion.

[4]Of course, this is only a high level representation, the compiler is free to pass the arguments of PUSH_RIGHT in any order and in any non-contigous way in order to optimize the code.

[5]It would have been sufficient to only pass the loop index; however, this would have required the loop body to recalculate the row vector addresses.

```
Matrix_multiply: if (i<m) {
                    PUSH_RIGHT(Row_multiply,i,j=0);
                    STOP;
                    Row_multiply: if (j<m) {
                                    PUSH_RIGHT(Row_multiply,i,j+1);
                                    c[i][j] = 0;
                                    for (k=0; k<m; k++)
                                        c[i][j] = c[i][j] + a[i][k]*b[k][j];
                                    STOP;
                                  }
                                  PUSH_RIGHT(Matrix_multipy,i+1);
                                  STOP;
                 }
Continue:        ...
```

For clarity's sake only the loop indexes are passed as iteration parameters; however, the code can be optimized by passing row and column vector pointers through the pipeline[6]. Figure 2 gives a schematic picture of the execution when the middle loop of the matrix multiplication progam is unrolled and pipelined on n processors.

Inside the code blocks which execute on individual processors any code that does not affect the execution order of the other blocks can be executed. This is in sharp contrast to conventional pipeline/vector processors in which complex conditionals can prevent any parallelism.

However, loop iterations might exhibit data dependency across iterations. If the data dependency exists between *successive* iterations the dataflow through the pipeline buffers will synchronize the iterations. If data dependencies across several iterations exist we can use "strip mining" techniques to bring the iterations together. Consider the following example:

```
for (i=5, i < 100, i++ )
    a[i] = a[i-5];
```

This loop can be transformed into:

```
for (i=5, i <= 95, i+5 )
    for (j=0, j < 5, j++)
        a[i+j] = a[i+j-5];
```

The data dependency is now across successive outer loop iterations and the outer loop iterations can again be pipelined. Note that if the data dependency between iterations is of distance greater than the length of the pipeline no special measures need to be taken. This is due to the wrap around pipeline and the fact

[6]The Appendix gives the assembler listing of the optimized version.

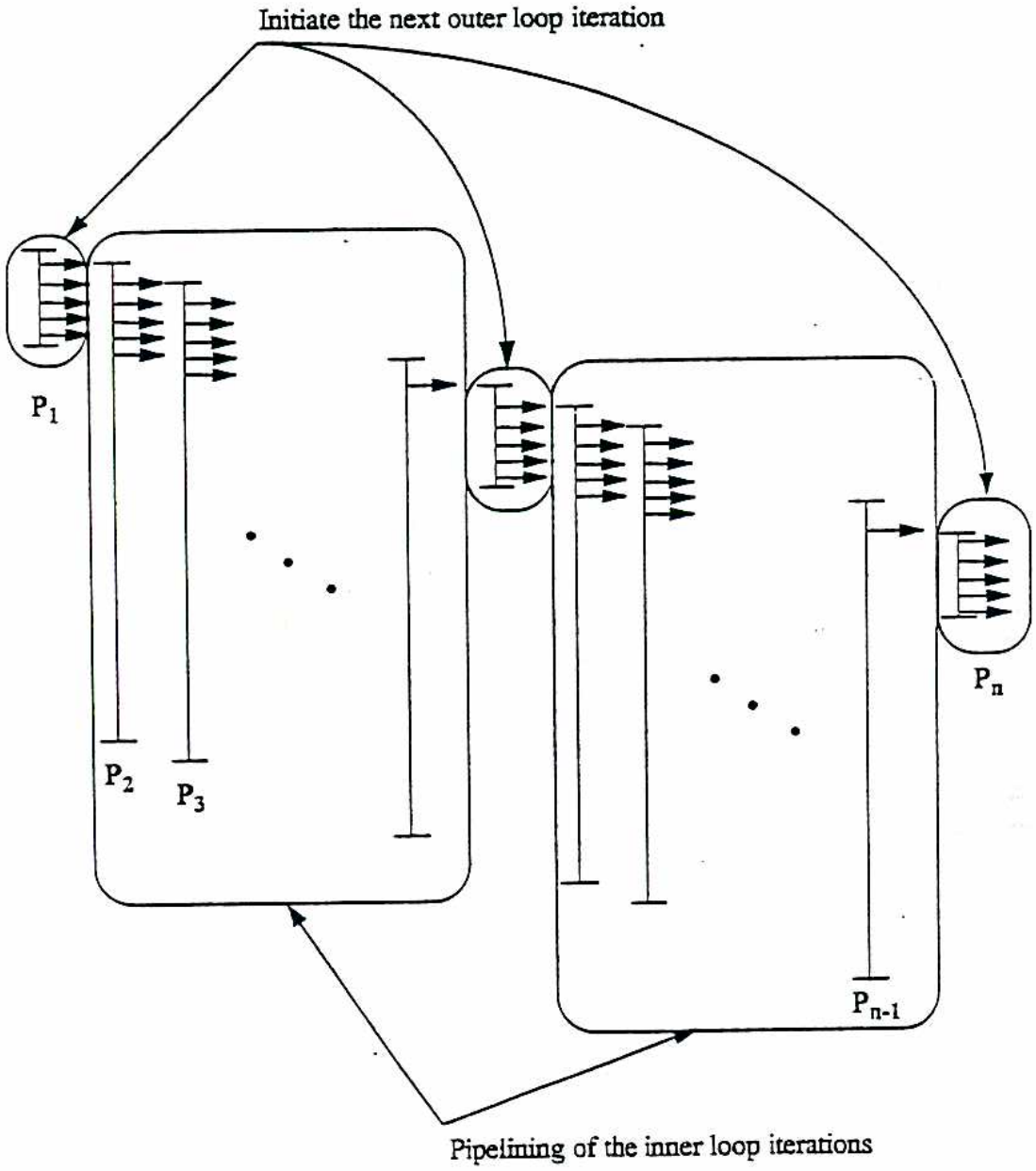

Figure 2: Schematic picture of execution when the middle loop iterations are pipelined

that every single iteration is executed until termination. Whenever an iteration i is executed it can be guaranteed that all iterations $< i - n$ have terminated, where n is the length of the pipeline.

Research on compilers for supercomputers has provided a wealth of algorithms on how to detect data dependencies and how to perform code transformations to eliminate them; the techniques are of course applicable to the proposed execution model as well [Po88, Wo89]. If everything else fails the shared memory can of course be used to synchronize execution explicitly.

Simulation

We used the matrix multiplication example to simulate the execution model. The simulation was done on a register transfer level. It was assumed that every instruction executes in one cycle except for branch instructions which took 2 cycles. The simulation took into account bus conflicts and wait states due to the memory latency. Furthermore, it was assumed that the code was fetched from local memory so that instruction fetches did not contribute to the bus conflicts. The data caches were assumed to be fast enough to not cause any wait states. Loading a cache word from memory took between 1 and 4 processor cycles. The bus arbitration cycle could be overlapped with data movements. For simplicity the cache line size was assumed to be one word. After the initial loading of the caches the cache hit ratio was 100%.

All results are normalized with respect to the execution time of the matrix multiplication program on a single processor. The single processor machine uses the same instruction set and the code is optimized for the sequential execution. The single processor is assumed to operate with zero-wait-state memory.

Figure 3 shows the speedup attainable by unrolling the outer loop of a 30x30 matrix under various cache configurations. The different configurations were:

1. Cache disabled, i.e. all data is fetched from shared main memory. Memory access time was assumed to be one processor cycle. This curve shows the effect of bus contention even if the memory is fast enough not to cause any wait states.

2. A simple data cache where every processor had to fill its own cache. Loading the cache took again only one cycle.

3. Pre-loading cache. The compiler knows that data objects are needed by other processors as well. The cache controllers are set to accept whatever is on the bus even if the current bus transaction was initiated by another processor. This is basically a broadcast operation in which all caches read what is on the bus.

4. All data is stored locally. This is the ideal case without any bus conflicts and a zero-wait-state memory.

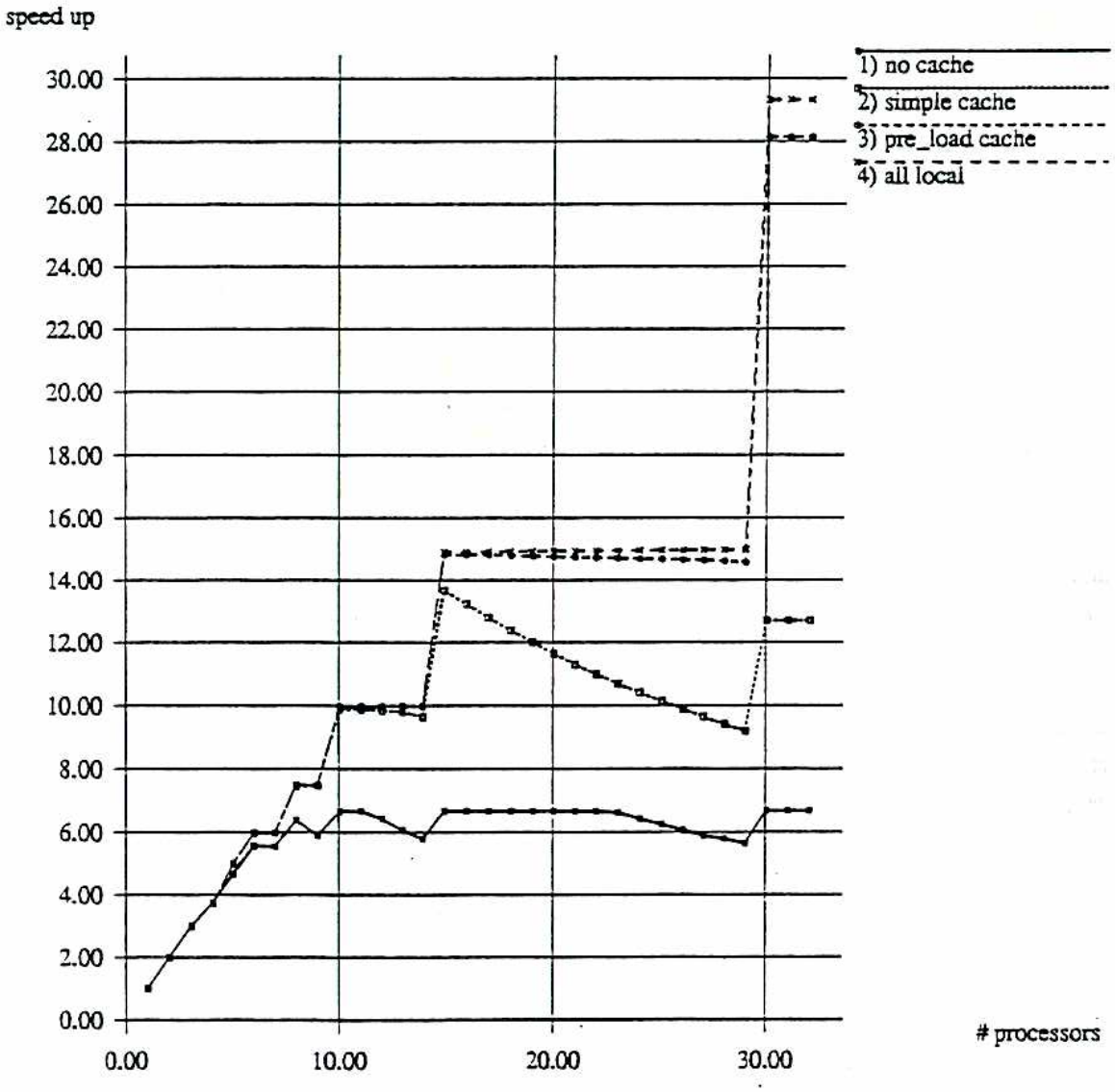

Figure 3: Effects of various cache configurations on a 30x30 matrix multiplication program with outer loop iterations pipelined.

It is interesting to see how the performance goes down when we reach 15 processors in configuration 2. Whenever the number of tasks is a multiple of the number of processors we can utilize all processors. Adding more processors will not increase performance until all tasks can again be evenly matched with the processors. As case 2 in Figure 3 shows it can even be detrimental to add more processors.

Figure 4 shows a comparison between the performance gained be unrolling the outer or inner loop of the matrix multiplication program. Both cases were simulated with pre-loading caches as in case 3 of Figure 3. Unrolling the middle loop has more overhead associated with it; however, we get a much smoother performance increase because we now have 900 tasks to hand to the processors.

Figure 5 shows the effect of the four different cache configurations of Figure 3 for the case when the middle loop is unrolled. If the caches are not pre-loaded performance will decrease beyond 15 processor due to the bus saturation. The spike in the simple cache case is caused by a lucky allignement of tasks and cache data. When the middle loop is unrolled each processor calculates one element of the result matrix by multiplying a row and a column vector. Ordinarily every iteration needs to load this row and column vector. However, if the length of the pipeline is a divisor of the dimension of the problem the tasks will, when they wrap around the pipeline, fall on processors which still have the needed row or column vector from a previous iteration.

Figure 6 shows the performance gained by unrolling the middle loop for different memory latency values. The case where all data is stored locally in zero-wait-state memory serves as the reference curve. Note the sharp cutoff of this curve at 43 processors. This is the length of the pipeline at which the first processor becomes available again when execution has reached the last processor. The other curves show the performance under increasing memory latency. The notation "1_3cycle" means 1 arbitration cycle (can be overlapped with data movements) + 3 memory cycles to load the cache word.

The question remains how much of the performance is attributable to the pipeline scheme. After all, rather then using pipeline buffers one could have used shared memory in order to create new tasks, i.e. everything that is written into the respective pipeline buffers could be written into dedicated memory regions. However, if a task block is set up in shared memory processing of this task block cannot commence until the task block creation has been completed. This is because main memory typically does not provide support for synchronized one-word message passing. Furthermore, setting up task blocks in main memory will increase the bus load. Figure 7 shows what happens when main memory rather then pipeline buffers are used to create new tasks. In Figure 7 very favorable assumptions were made for the case of creating task blocks in main memory. It is assumed that only the writing of task blocks consumes bus bandwidth. This means that any processor that will read this task block has the associated memory region already in its cache. In other words, when a task block is written it is copied directly into some processor's cache via the shared bus. Additional

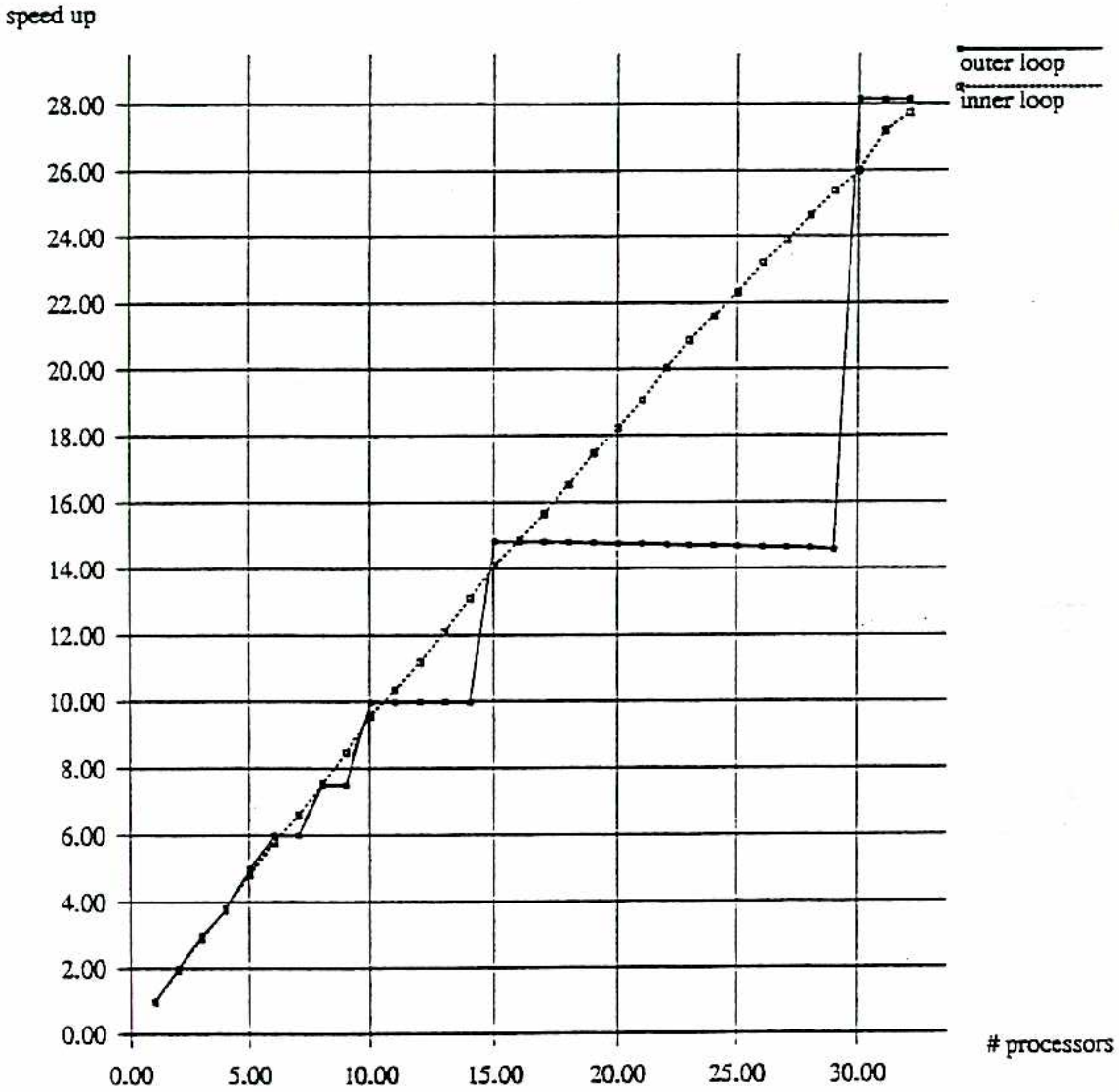

Figure 4: Comparison between pipelining the outer loop iterations and middle loop iterations in a 30x30 matrix multiplication program.

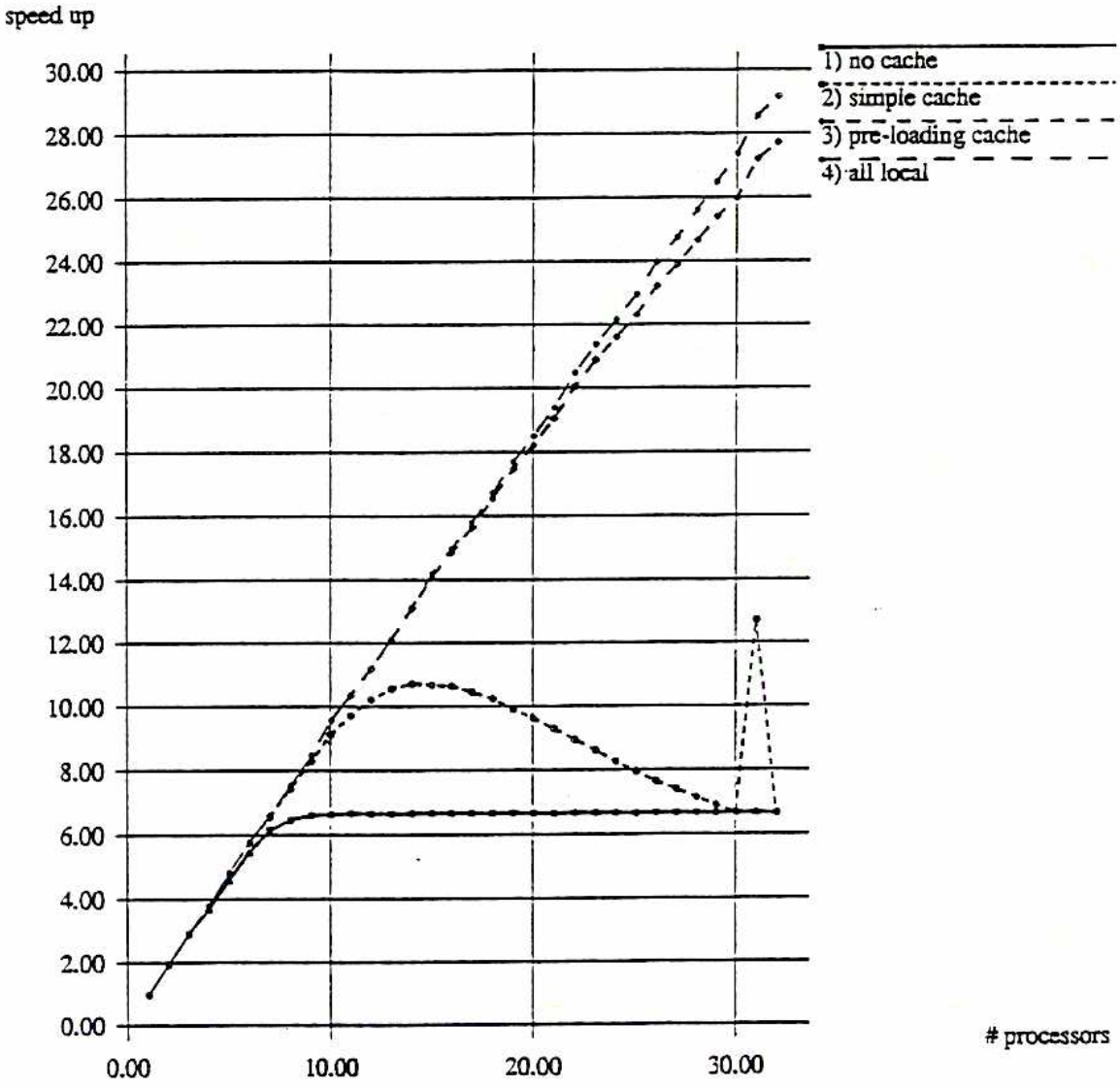

Figure 5: Effects of various cache configurations on a 30x30 matrix multiplication program with middle loop iterations pipelined.

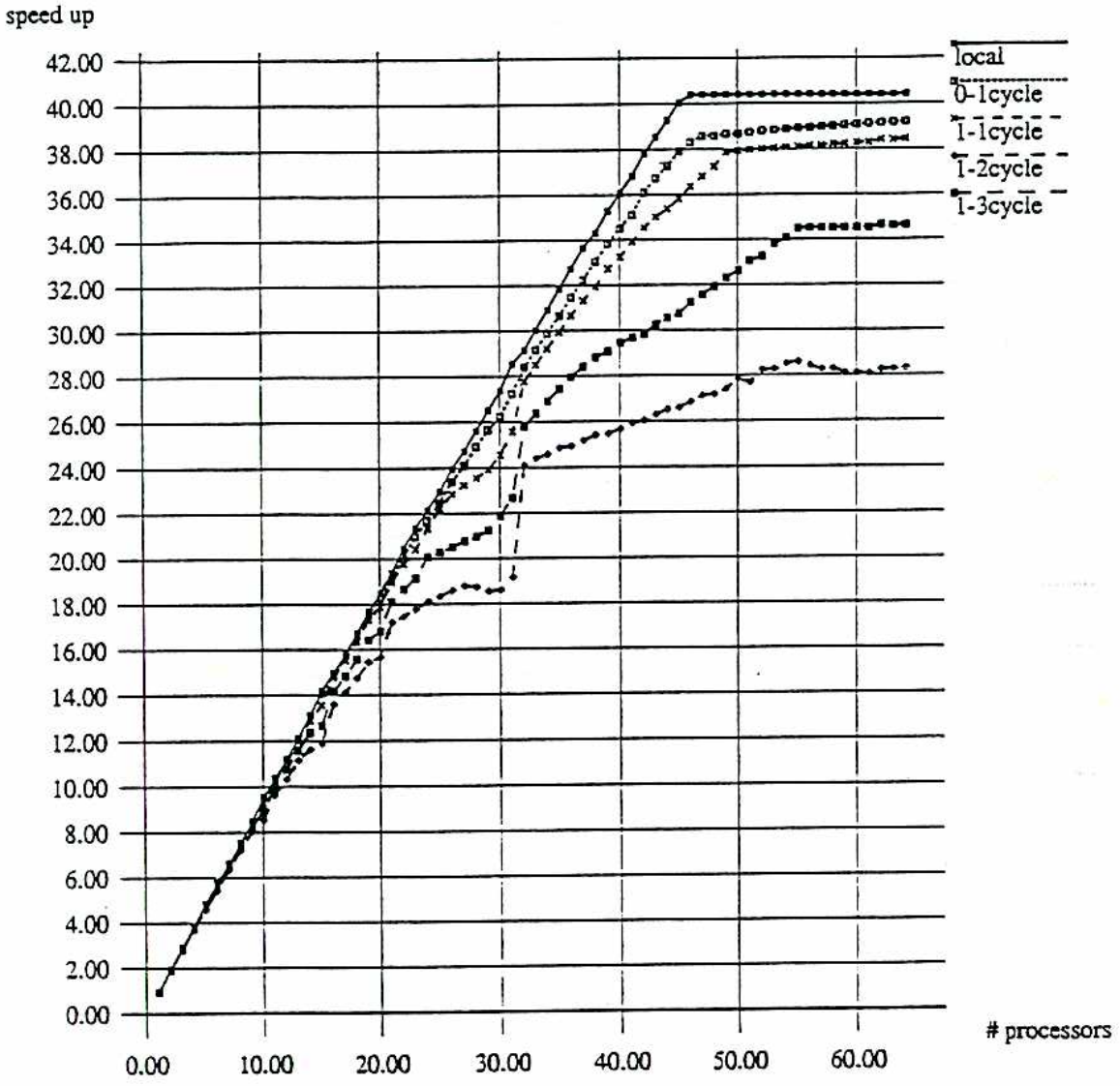

Figure 6: Effect of memory latency on the pipelined execution of the middle loop iterations of a 30x30 matrix multiplication program.

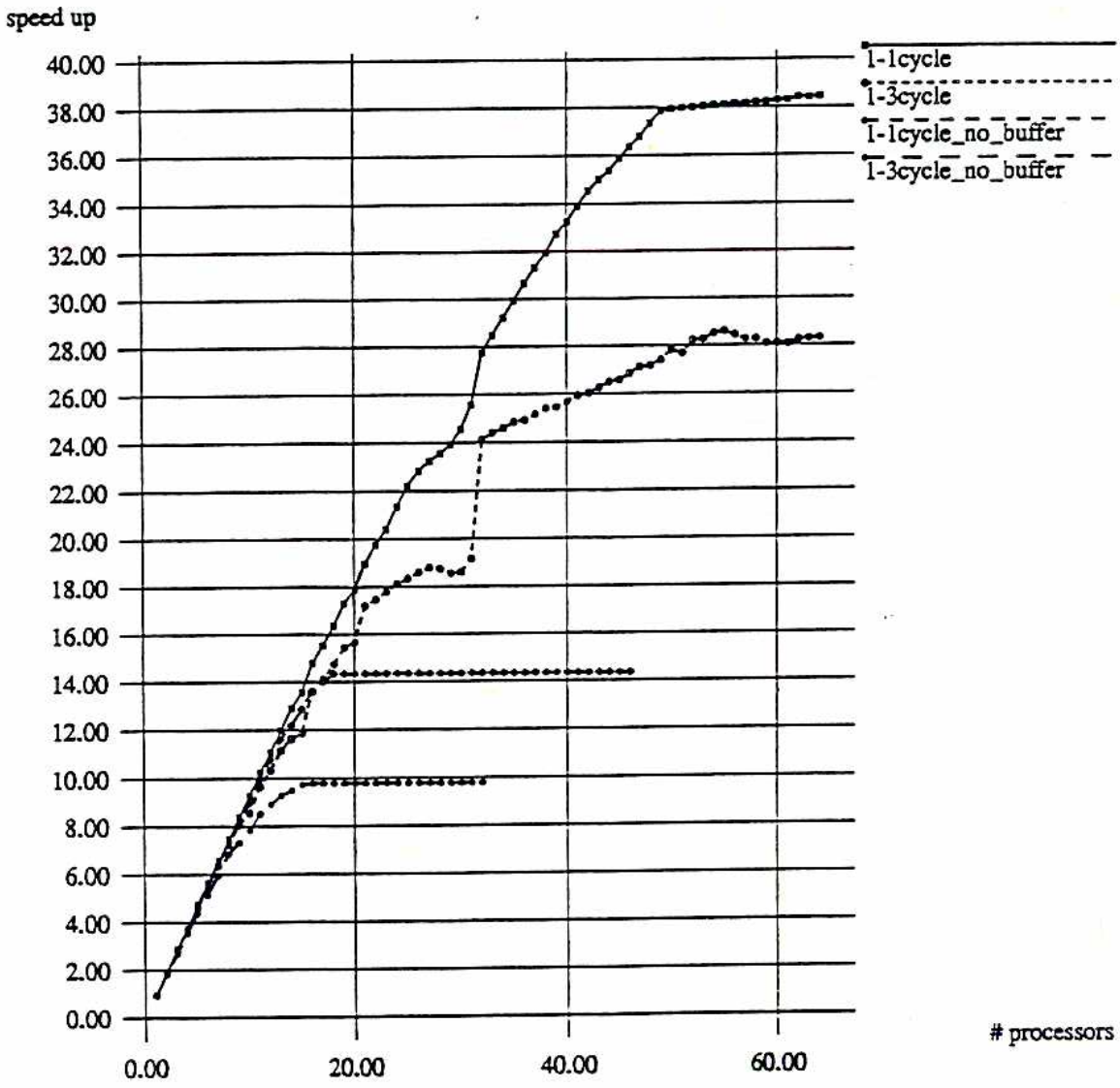

Figure 7: Effect of hardware synchronized pipeline buffers on system performance.

synchronization requirements were ignored. As can be seen from Figure 7 before the bus becomes the bottleneck the lack of pipelining limits the performance. Remember that in the fully pipelined case a new task could begin executing, and in turn create new tasks, before all the arguments for this task had been provided. In the shared memory alternative the processors have to wait until the respective task block has been set up in its entirety. This causes such a delay that after a few steps the first processor becomes idle again.

Finally, Figure 8 shows that the proposed pipeling scheme also works well with rather small problems. Problems of such small size cannot be effectively handled by explicitly creating task blocks.

Conclusion

The proposed architecture allows the pipelining of blocks of code on a circular pipeline of microprocessors. The proposed execution model is especially suitable for the dynamic unrolling of loops and/or recursive function calls. If the caches are designed in such a way that they permit to be pre-loaded with commonly used data objects this simple bus based architecture can be scaled to several dozens of processors.

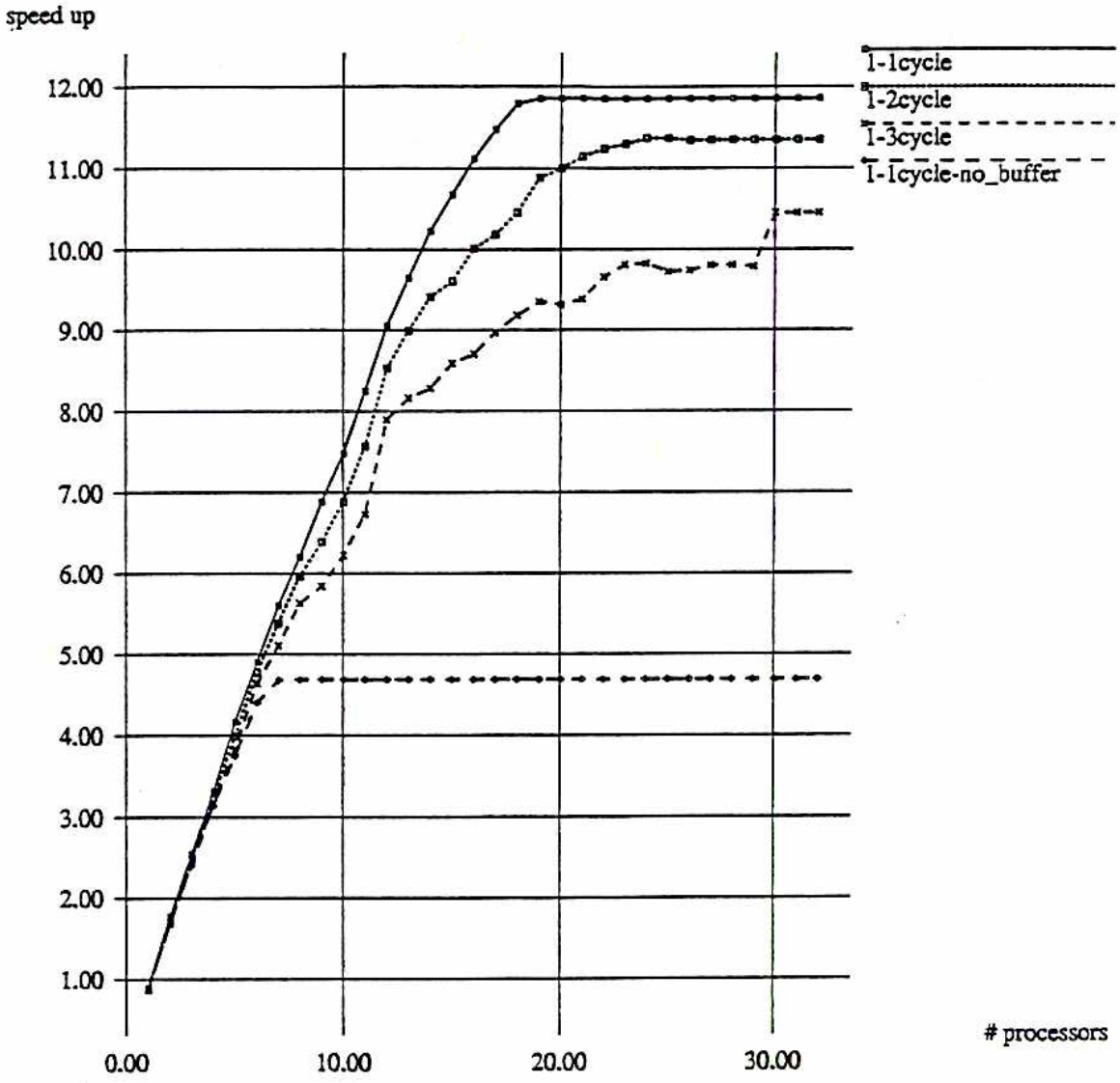

Figure 8: Performance for a 10x10 matrix multiplication with middle loop unrolled.

References

[El86] Ellis, J.R.: *Bulldog: A Compiler for VLIW Architectures*, MIT Press, Cambridge, Massachusetts, 1986

[Fi81] Fisher, J.A.: *Trace scheduling: A technique for global mircocode compaction*, IEEE Transactions on Computers, C-30(7), pp 478-490, July 1981

[Fi83] Fisher, J.A.: *Very Long Instruction Word Architectures and the ELI-512*, Proceedings of the 10th International Symposium on Computer Architecture, pp. 140-150, 1983

[Po88] Polychronopoulos, C.D.: *Parallel Programming and Compilers*, Kluwer Academic Publishers, 1988

[Ra89] Rau, B.R. et al.: *The Cydra 5 Departmental Supercomputer*, IEEE Computer Magazine, Vol. 22, No. 1, January 1989

[Sch87] Schneck, P.B.: *Suppercomputer Atchitecture*, Kluwer Academic Publishers, 1987

[Wo89] Wolfe, M.: *Optimizing Supercompilers for Supercomputers*, MIT Press, Cambridge, Massachusetts, 1989

Appendix

The following listing is the machine code used in the simulation of the 30x30 matrix multiplication example. However, the code is more general in that it allows multiplication of rectancular matrixes as well, the respective dimensions are given as dim_i, dim_j, dim_k. The machine instructions are based on a *load_and_store* architecture. Actually this instruction set doesn't even allow to read directly from memory, as in read_mem(X,r1), where X is a memory location and r1 a register (all registers start with 'r'), but rather requires the following: move(address of X,r0) , read_mem(r0,r1). However, this makes simulation much easier since no decoding needs to be performed. If anything, this idiosyncracy handicaps the simulation of the pipelined execution scheme, and certainly doesn't constitute an unfair advantage.

As can be seen, the proposed execution model does not require any special instructions. All operations can be performed with any standard instruction set. In the code below stp() is a macro which just reads an entry address from the pipeline buffer and starts execution.

```
label_0:
   start       ()
label_1:
   move_val    (i_addr,r14) | i_addr is the location of
                            | the outer loop index variable
   read_mem    (r14,r1)     | get loop variable
   move_val    (dim_i,r2)   | load the dimension into r2
   cmp         (r1,r2)      | i < dim ?
   bge         (label_5)    | if i >= dim then DONE
   move_reg    (r1,rtemp)   | increment loop counter
   inc         (rtemp)
   write_mem   (r14,rtemp)  | and store it in memory
   move_val    (label_2,r3) | load the entry address into r3
   write_right(0,r3)        | and write r3 into the first location
                            | (0) of the right pipeline buffer
   clear       (r4)         | clear the middle loop counter
   write_right(2,r4)        | pass the middle loop counter to the
                            | right buffer in location 2
   move_val    (a_base,r4)  | put the base pointer to matrix A
   move_val    (dim_k,rtemp)| into r4 and determine the current
   mul         (r1,rtemp)   | row vector of A based on the current
   add         (rtemp,r4)   | outer loop index, push the row vector
   write_right(3,r4)        | pointer to the right neighbor
   move_val    (b_base,r5)  | push the base pointer of matrix B into
   write_right(4,r5)        | the right pipeline buffer (location=4)
   move_val    (c_base,r6)  | as for matrix A determine the current
```

```
   move_val    (dim_j,rtemp)
   mul         (rtemp,r1)
   add        (r1,r6)       | row vector of C and pass the row vector
   write_right(5,r6)        | pointer to the right neighbor
   stp()                    | look into the left buffer for more work

label_2:
   move_val    (dim_j,r0)   | check the middle loop index to see
   read_left   (2,r4)       | if middle loop is done. If yes then jump
   cmp         (r4,r0)      | to label 1 to start with the next outer
   bge         (label_1)    | loop iteration
   move_val    (label_2,r13)| otherwise invoke middle loop iteration
   write_right(0,r13)       | again on the next processor
   inc         (r4)         | increment middle loop index and pass it on
   write_right(2,r4)
   read_left   (3,r1)       | get row vector of A and pass it on too
   write_right(3,r1)
   read_left   (4,r2)       | get base pointer of B. Increment the base
   move_reg    (r2,rtemp)   | pointer and pass it on to the next iteration
   inc         (rtemp)      | the incremented pointer is really nothing
   write_right(4,rtemp)     | else but the column pointer of B
   read_left   (5,r3)       | to the same to the base pointer of C
   move_reg    (r3,rtemp)
   inc         (rtemp)
   write_right(5,rtemp)
   clear       (r4)         | now go for the inner loop
   clear       (rtemp)      | rtemp is c[i][j]
   move_val    (dim_k,r5)
label_3:
   cmp         (r4,r5)      | inner loop done ?
   bge         (label_4)
   read_mem    (r1,r11)     | read the elements from A and B
   read_mem    (r2,r12)     | from memory
   mul         (r11,r12)    | c[i][j] = c[i][j] + a[i][k]*b[k][j]
   add         (r12,rtemp)
   inc         (r4)         | k = k+1
   inc         (r1)         | pointer to a[i][k+1
   add         (r0,r2)      | pointer to b[k+1][j] r0 is still dim_j
   branch      (label_3)
label_4:
   write_mem   (r3,rtemp)   | write the result c[i][j] back
label_5:
   stp()
```

The following is the listing of the sequential code which was used as a performance reference. Again, this code can is not restricted to square matrixes.

```
label_0:
   start();
label_1:
   move_val    (dim_i,r0);
   move_val    (dim_j,r17);
   move_val    (dim_k,r13);
   move_val    (a_base,r1); |
   move_val    (b_base,r2); |
   move_val    (c_base,r3); |  r3=C
   clear       (r4);        |  r4=i  outer loop counter
label_2:
   cmp         (r4,r0);
   bge         (label_7);
   move_reg    (r2,r5);     |  r5=B[j]
   clear       (r6);        |  r6=j  middle loop counter
label_3:
   cmp         (r6,r17);
   bge         (label_6);
   move_reg    (r1,r7);     |  r7=A[k]
   move_reg    (r5,r8);     |  r8=B[k]
   clear       (r9);        |  temp result
   clear       (r10);       |  r10=k  inner loop counter
label_4:
   cmp         (r10,r13);
   bge         (label_5);
   read_mem    (r7,r11);
   read_mem    (r8,r12);
   mul         (r11,r12);
   add         (r12,r9);
   inc         (r7);
   add         (r0,r8);
   inc         (r10);
   branch      (label_4);
label_5:
   write_mem   (r3,r9);
   inc         (r3);
   inc         (r5);
   inc         (r6);
   branch      (label_3);
label_6:
   add         (r13,r1);
```

```
   inc         (r4);
   branch      (label_2);
label_7:
   stp();
```