Software Pipelining On Multi-core Chip Architectures: A Case Study on IBM Cyclops-64 Chip Architecture

Alban Douillet, Junmin Lin, Guang R. Gao

CAPSL Technical Memo 64
February 14, 2006

Abstract

In this paper, we propose a compiler method for software pipelining of loop nests on multi-core chip architectures. Our method is based on an extension to the *single-dimension software-pipelining (SSP)* that was proposed to apply software pipelining to any loop level (possibly beyond innermost level) in a loop nest. Using a Single-Program Multi-Data (SPMD) approach, we outline a method that takes a loop nest, without any pragmas or modifications, as an input and produces a proven correct deadlock-free multi-threaded schedule across the processing cores. Our proposed method and algorithms were implemented in an Open64 based compiler framework retargeted to the IBM Cyclops64 architecture. Experimental results show that our MT-SSP method generates correct, deadlock-free multi-threaded schedules for the Cyclops-64 chip. On a collection of benchmarks tested, the performance of our software pipelined code has demonstrated excellent scalability up to 100 cores tested[1]. The tiling factor automatically calculated by our compiler works very well in keeping the load of the cores well balanced. Our experiments has also demonstrated good efficiency of cross-iteration synchronization between thread units (cores) without using special machine atomic synchronization instructions. The register pressure measured during the experiments are well within the hardware limits of the architecture.

[1]Only 99 thread units were used in our experiments. See Section 6 for details

Contents

List of Figures

1 Introduction

The designs of high-performance microprocessor chip architectures are rapidly adopting an organization that integrates a growing number of multiple processing cores on a single chip [9]. Computer architects are now experimenting large-scale multi-core chip architectures - such as the IBM Cyclops-64 cellular architecture - where a single chip may contain more than one hundred thread units and memory banks, and beyond. However, the success of such multi-core chips is depending on the advance in system software technology (e.g. compiler technology, runtime software technology, etc.) where application programs can be effectively mapped to fully utilize the on-chip parallelism capacity.

In this paper, we are mainly interested in developing an effective compiler technology for loop nests on multi-core chip architectures. Although the technology of automatic program parallelization of loop nests has made significant and interesting progress in the past, it is yet to demonstrate that such automatic parallelization is practically and broadly applicable for multi-core chips in the near future. On the other hand, parallel programming, although an interesting alternative worthwhile to pursue, but, judged on our experience of parallel machines in the past two decades, is still a long term research task. Furthermore it will not solve the so-called dusty-deck problem - where a large number of sequential programs need to be migrated to the multi-core platform without heavy parallel programming investment.

On uniprocessor (single-core) architectures, however, software-pipelining (SWP) [10, 17, 11] is probably the most effective compiler scheduling method for single loops, and often is a primary method to enhance the performance for many loop-intensive programs. In this paper, we will demonstrate that the software pipelining technology can be extended to apply to compiling loop nest for multi-core chips. In fact, under our method, a sequential program (e.g. C, Fortran without any changes) can be automatically compiled into software pipelined code that may fully exploit the parallelism capacity of multi-core chips.

Our method is based on an extension to the *single-dimension software-pipelining (SSP)* [16] that was proposed to extend SWP to any loop level (possibly beyond innermost level) in a loop nest. Under our new SSP method - called *multithreaded SSP* or MT-SSP, loop iterations of a nest loop are mapped to processing cores across a multi-core chip, and are initiated at a regular iteration *initiation interval* (II) and their execution is overlapped to fully exploit the potential parallelism. We hope that the work proposed in this paper will demonstrate that it is indeed possible to repeat the success of software-pipelining from single-core architectures to multi-core architectures.

In summary, this paper presents the *MT-SSP* software-pipelining method to schedule loop nests on multi-core architectures. Using a Single-Program Multi-Data (SPMD) approach, we outline a method that takes a loop nest, without any pragmas or modifications, as an input and produces a proven correct deadlock-free multi-threaded schedule across the processing cores. We propose a parameterizable light-weight synchronization mechanism to coordinate the synchronization/communication between iterations across different cores. Our method automatically handles workload distribution through an automatic grouping of iteration (also called tiling in the paper), data communication and synchronization.

Our proposed method and algorithms were implemented in an Open64 based compiler framework retargeted to the IBM Cyclops64 architecture. The multi-threaded multi-core schedule are tested on a Cyclops architecture simulator and associated system software toolchain provided by ETI. Experimental results show that our MT-SSP method generates correct, deadlock-free multi-threaded schedules for the Cyclops-64 chip. On a collection of benchmarks tested, the performance of our software pipelined code has demonstrated excellent scalability up to 100 cores tested. The tiling factor automatically calculated by our compiler works very well in keeping the load of the cores well balanced. Our experiments has also demonstrated good efficiency of cross-iteration synchronization between thread units without using special machine atomic synchronization instructions. The register pressure measured during the experiments are well within the hardware limits of the architecture.

The rest of the paper is divided as follows. The next section introduces the background of the paper, describes the SSP method for single-core architectures, and the IBM Cyclops-64 architecture. Section 3 presents the problem addressed and the challenges to overcome. The next two sections explain the multi-threaded SSP solutions and its implementation for the IBM Cyclops-64 architecture. Section 6 presents the experimental results. The last two sections are dedicated toward related work and conclusion.

2 Background

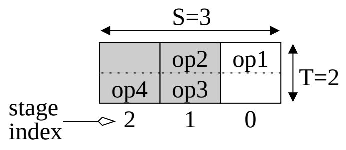
2.1 Single-dimension Software-Pipelining

Single-dimension Software Pipelining (SSP) [16] is a resource-constrained software pipelining method for both perfect and imperfect loop nests on single-core architectures. Unlike other approaches [10, 22, 11], SSP does not necessarily software pipeline the innermost loop of a loop nest, but directly software pipelines the loop level estimated to be the most profitable. From the SSP point of view, the loop levels enclosing the selected loop are ignored. Therefore, the selected loop is seen as the outermost loop. Within an outermost loop iteration, inner loops are executed sequentially.

Beside being able to software pipeline any loop level and overlap the execution of the prolog and epilog of the inner loops, SSP is able, unlike MS, to take advantage of instruction-level parallelism or data locality properties present in the outer loops. Without prior loop transformations, a faster schedule can be found. If the innermost loop level is chosen, SSP is equivalent to classical modulo scheduling. SSP retains the simplicity of modulo scheduling, and yet may achieve significantly higher performance [16].

SSP takes as input a loop nest of depth n. We note N_i the number of iterations for each loop level, level 1 being the outermost level and n the innermost. The operations of the loop nest are scheduled into a *multi-dimensional kernel* of S stages with an initiation interval T [7]. An example for a double loop nest is shown in Fig. 1(a). The kernel is partitioned into subkernels, one per loop level. The number of stages and the index of the first and last stage of the subkernel of loop level i are noted S_i, f_i, and l_i, respectively (Fig. 1(b)).

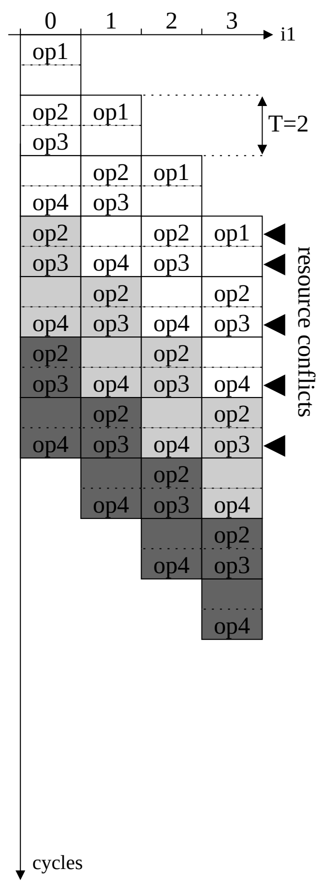
The kernel is used as a template to build the *ideal schedule*, where an outermost iteration is issued every T cycles and all the dependences are respected (Fig. 1(c). Outermost iterations (columns in the figure) are run in parallel. Within one outermost iteration, inner iterations (shades of gray) are



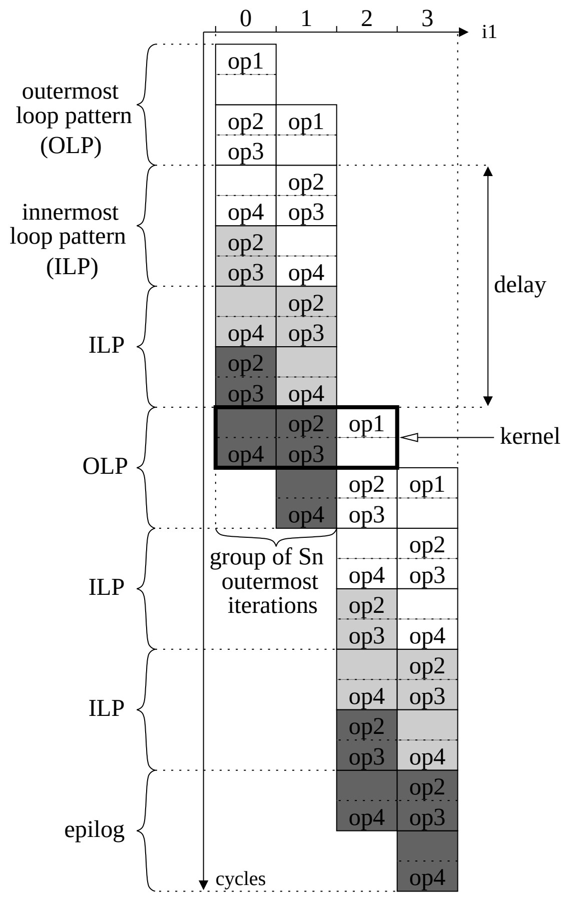
(a) Kernel

Loop Level (i)	S_i	f_i	l_i
1	3	0	2
n=2	2	1	2

(b) Subkernels Parameters



(c) Ideal Schedule



0
1
2
3
i1
outermost loop pattern (OLP)
innermost loop pattern (ILP)
ILP
OLP
ILP
ILP
epilog
delay
kernel
group of Sn outermost iterations
cycles

(d) Single-Core Final Schedule

Figure 1: Single-Core SSP Example

executed sequentially. However there may be resource conflicts if two instances of the same stage are run simultaneously.

To avoid the conflicts a delay is introduced every S_n outermost iterations to produce the *single-core final schedule* as shown in Fig. 1(d). Repeating patterns are then found: outermost loop pattern (OLP) and innermost loop pattern (ILP). prolog, draining/filling patterns (DFP), and epilog may also appear [15].

2.2 The IBM Cyclops-64 Architecture

The Cyclops-64 (C64) is the latest version of the Cyclops cellular architecture designed to serve as a dedicated petaflop compute engine for running high performance applications [4, 5]. A C64 supercomputer is attached to a host system through a number of Gigabit Ethernet links. The host system provides a familiar computing environment to application software developers and end users.

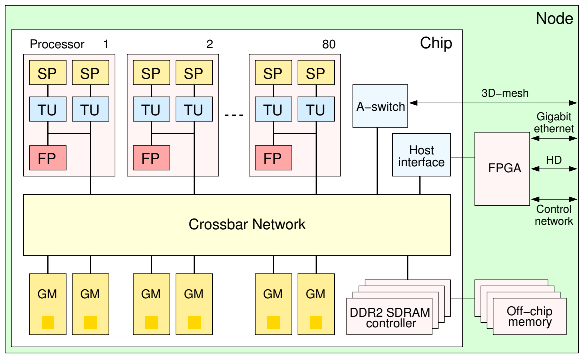


Figure 2: Cyclops-64 node

A C64 is built out of tens of thousands of C64 processing nodes arranged in a 3D-mesh network. Each processing node consists of a C64 chip, external DRAM, and a small amount of external interface logic. A C64 chip employs a many-core-on-a-chip design with a large number of hardware thread units, half as many floating point units, embedded memory, an interface to the off-chip DDR2 SDRAM memory and bidirectional inter-chip routing ports, see Figure 2. A C64 chip has 80 processors, each with two thread units, a floating-point unit and two SRAM memory banks of 32KB each. A 32KB instruction cache, not shown in the figure, is shared among five processors. The C64 chip has no data cache. Instead a portion of each SRAM bank can be configured as scratchpad memory (SP). The remaining sections of SRAM together form the global memory (GM) that is uniformly addressable from all thread units.

3 Problem Description

3.1 Problem Statement

In this apply, we apply the SSP technique to multi-threaded cellular architectures. We assume that the SSP ideal schedule has already been computed and that registers have been allocated to the loop variants. The multi-threaded final schedule must now be defined and computed. Despite the differences between the single-core final schedule and the multi-threaded final schedule, the earlier SSP steps(namely loop selection, kernel generation and register allocation), do not require any modification.

3.2 Issues

Multiple challenges must be faced to produce a multi-threaded final schedule. First, the dependence and resource constraints must be respected. Similarly to the single-core case, an operation cannot be scheduled before all the operations on which it depends are committed. However, with multi-threaded architectures, memory dependences may exist between independent thread units and synchronization is required to guarantee a sequential order between the memory accesses to the same memory location.

Second, how to make sure that a thread unit does not run ahead of the others? How to implement a light-weight synchronization scheme? How to parameterize the synchronization mechanism so that the execution time of the multi-threaded final schedule is minimized? If the synchronization occurs too often, no useful work is done. If synchronization occurs too rarely, thread units are idle and no work is done.

Third, the workload distribution over the large number of thread units available in multi-threaded cellular architectures must be fast and keep all the thread units as busy as possible to minimize the overall execution time of the multi-threaded final schedule.

Fourth, cross-iteration dependences between outermost iterations scheduled on separate thread units require the thread units to communicate the data and to synchronize each other. This exchange must occur only between outermost iterations scheduled on separate thread units.

Finally, as for every applications requiring synchronization, the multi-threaded final schedule must be deadlock-free.

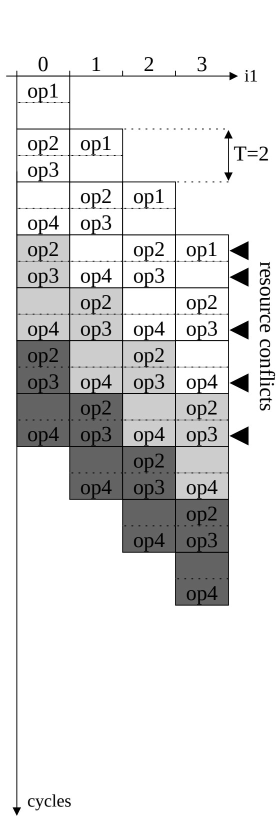
4 Multi-Threaded Single-Dimension Software-Pipelining

In the SSP framework, we assume that the ideal schedule, common to both the single-core and multi-threaded versions of SSP, has already been computed. We now present how to generate the multi-threaded final schedule from the ideal schedule.

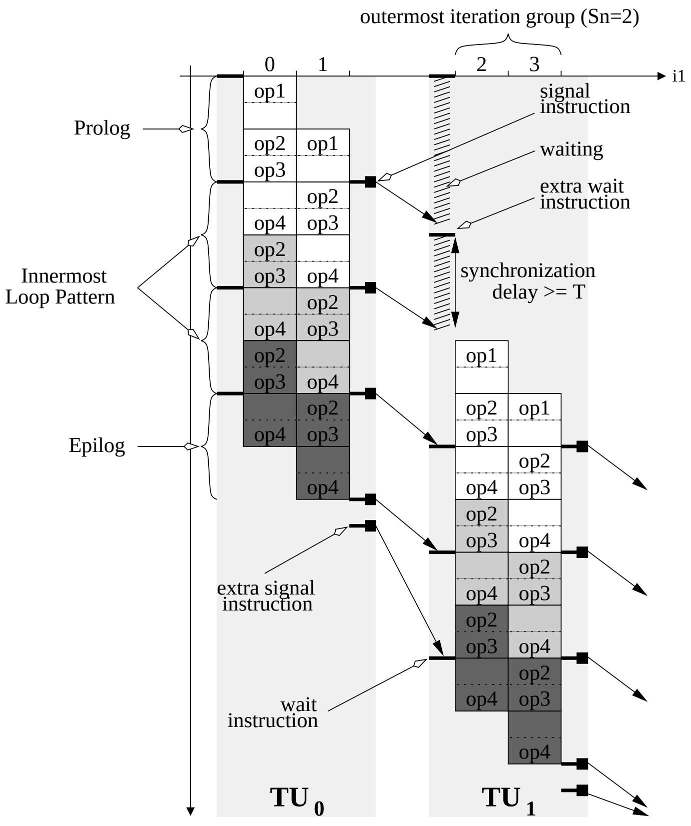
4.1 Multi-Threaded Final Schedule

In order to resolve the resource conflicts that may appear in the ideal schedule, single-core SSP adds a delay every S_n outermost iterations. Instead, in MT-SSP, we schedule each group of S_n outermost iterations on a separate thread unit, as shown in Fig. 3. The choice of S_n outermost iterations is dictated by the kernel. S_n is the maximum number of outermost iterations that can be executed in parallel on the same core without any resource conflict. Indeed, the innermost subkernel is made of S_n stages and innermost operations are scheduled in that subkernel so that there is no resource conflict. If two instances of the same innermost stage are executed simultaneously, as it is the case in the ideal schedule as soon as $S_n + 1$ outermost iterations are allowed to run in parallel, the resource conflict-free property is lost.

As the outermost iterations are issued every T cycles in the ideal schedule and the final schedule is computed by partitioning the ideal schedule into groups of S_n iterations, all the thread units execute the



(a) Ideal Schedule



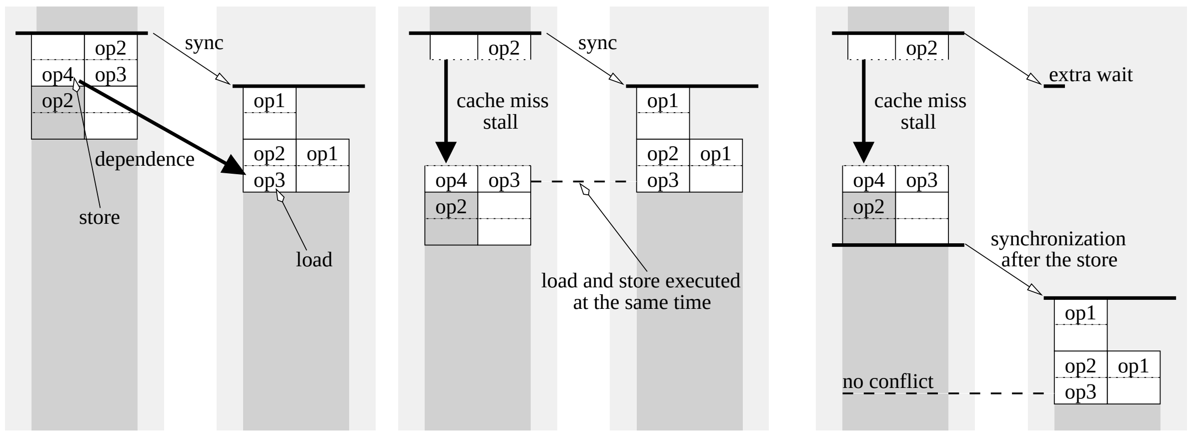
(b) Multi-Threaded Final Schedule

Figure 3: MT-SSP Final Schedule Example

same schedule, including the synchronization operations described in the next section. Therefore the MT-SSP final schedule is a Single-Program Multiple-Data (SPMD) schedule and thread units will be able to share the same instruction cache.

4.2 Synchronization

The schedule naturally requires synchronization. A stage being scheduled on one thread unit cannot be executed until the stages scheduled earlier in the ideal schedule have completed. Synchronization operations are therefore added to the MT-SSP final schedule. Their locations in the schedule are chosen to minimize code duplication during code generation. A WAIT operation is placed before each repeating pattern while a SIGNAL operation is placed after. Those repeating patterns are similar to the patterns appearing the single-core final schedule [15]. In the example in Fig. 3, three patterns appear: the prolog, the epilog and the innermost loop pattern (ILP). In the general case the repeating patterns are not necessarily of the same size. The first thread unit does not require any WAIT operations and the last thread unit does not require any SIGNAL operations.



(a) No Stall: the dependence is respected

(b) Stall without Synchronization Delay: the dependence is not respected

(c) Stall with Synchronization Delay: the dependence is respected

Figure 4: Synchronization Delay Example

In addition of the synchronization points mentioned above, a synchronization delay must be introduced to guarantee the correctness of the schedule. Let us consider the example shown in Fig. 4. Let us assume that op_4 is a store instruction to a memory location which will be accessed by op_3 two outermost iterations later. There is a memory dependence from op_4 to op_3. When generating the kernel, the dependence was taken into account. If both instruction groups are executed on the same processor or no stall occurs, the dependence is respected (Fig. 4(a)). In a multi-threaded schedule, the store and load instructions may be scheduled on two different thread units. If the execution of op_2 is stalled because of some memory port contention for instance, the second thread unit would not be affected and op_3 would be executed before op_4 has completed (Fig. 4(b). To prevent this situation, an extra WAIT operation is added. Thus the dependence from op_4 to op_3 is respected (Fig. 4(c). Symmetrically a SIGNAL operation is also added so that there are as many WAIT than SIGNAL operations.

Since the SSP loop selection step does not allow loop levels with negative dependences to be software-pipelined, there cannot be any negative dependence between outermost iterations. Therefore the synchronization signals are unidirectional, from one thread unit to its direct successor. The graph of synchronization signals between outermost iterations forms a tree. If the SIGNAL instruction is non-blocking, then the multi-threaded SSP schedule is *deadlock-free*. If an outermost dependence spans several thread units, the cascading of synchronization signals will ensure that the dependence is respected.

4.3 Innermost Loop Tiling

To reduce the synchronization costs, the execution of the $N_n - 1$ instances of the innermost loop pattern is tiled into tiles of G iterations. If G is not a multiple of $N_n - 1$, the last tile only contains the remaining instances. The WAIT and SIGNAL operations are issued at the entrance and exit of each tile, respectively.

The value of G that will minimize the length of the final schedule, G_{min}, can be approximated at compile-time using the schedule function presented in Sec. 4.4. Using the first derivative of the schedule over G, we obtain:

$$G_{min} = \sqrt{\frac{w * (N_n - 1) * \left(1 + \prod_{j=2}^{j=n-1}(N_j - 1)\right)}{T * S_n * \left(\left\lfloor \frac{N_1}{S_n} \right\rfloor - 1\right)}} \tag{1}$$

G_{min} is very close to the empiric value of G for which the execution time is minimized as shown in Sec. 6. Since the first derivative of the schedule function is used to compute G_{min}, the formula is identical for kernels with multiple initiation intervals.

4.4 Schedule Function

Given a kernel with single initiation interval, the schedule function of the final schedule can be expressed. Let us consider the instance of an operation op at iteration $\overrightarrow{I} = (i_1, \ldots, i_n)$. We note $\sigma(op, 0)$ the schedule cycle of the operation in the kennel. The schedule function of the multi-threaded SSP schedule for imperfect loop nests with a single initiation interval can be written as shown in Eqn. 2. The computation of the function is explained in Sec. 4.4.

$$\begin{aligned} f(op, \overrightarrow{I}) \quad &= \quad \sigma(op,0) + i_1 * T + \sum_{k=2}^{k=n} i_k * time_{L_k} + \left(\left\lfloor \frac{N_1}{S_n} \right\rfloor - 1\right) * (l_n * T + G * S_n * T) \\ &\quad + \left(\left\lfloor \frac{N_1}{S_n} \right\rfloor - 2\right) * 2 * w + syncsPerGroup * w \end{aligned} \tag{2}$$

where

$$\begin{aligned} time_{L_k} &= \sum_{i=k}^{i=n} \left((S_i - S_{i+1}) * T * \prod_{j=k+1}^{j=i} N_j \right) \\ S_{n+1} &= 0 \\ syncsPerGroup &= 3 + \frac{N_n - 1}{G} + \frac{N_n - 1}{G} * \prod_{j=2}^{j=n-1} (N_j - 1) + \sum_{i=2}^{i=n-1} \prod_{j=2}^{j=i} (N_j - 1) \end{aligned}$$

Theorem 1 *Given an imperfect loop nest and an SSP kernel with a single initiation interval, the schedule function proposed in Equation 2 respects both the dependencies from the n-D DDG and the resource constraints.*

Proof. To prove the theorem we must show that the dependences and the resource constraints are respected. As the single-core schedule function without the delay term already respects the dependences and the terms added in the multi-threaded function are all positive, the dependences are respected. And since only S_n outermost iterations are by construction executed on a single thread unit, there is no resource conflict. □

5 Implementation on IBM Cyclops-64 Chip Architecture

5.1 Overview

Iteration groups are executed on the thread units in a round-robin fashion. The synchronization signals from the last thread unit are redirected to the first thread unit. The number of iteration groups does not have to be a multiple of the number of thread units. Fig. 5 shows an example where only 3 thread units are available and 5 iteration groups are to be executed.

All the thread units but the first directly reach a wait instruction after their initialization and will not start executing iteration groups until the first thread unit has started. The thread unit to execute the last iteration group sends an extra synchronization completion signal to the first thread unit. When a thread unit has completed the execution of all its iteration groups and has sent all the required signals, it goes to sleep. When the first thread unit has received the completion signal from the last thread unit, it returns to the main program.

5.2 Synchronization

Synchronization is implemented using a Lamport's clock. Each thread unit has two counters. The first counter, named *synchronization counter*, is used to count the number of synchronization signals received. The second counter is the internal clock of the thread unit and is called the *clock counter*. It represents the progress of the thread unit and is incremented after each WAIT instruction. When reaching a WAIT instruction, the execution on a thread unit is allowed to continue only if the synchronization counter is greater or equal to the clock counter.

The WAIT instruction is implemented with an active loop, although a SLEEP instruction could have been used. Since a single thread is executed by each thread unit, active loops were chosen. The SIGNAL instruction is a non-blocking atomic add-in-memory instruction.

In order to minimize the execution time of the WAIT instruction, the synchronization counter is placed in the scratch-pad memory of the thread unit being signaled. The value can then be quickly read by the receiving thread unit without using the crossbar network. Although the signal instruction travels over the crossbar network, the signal instruction is non-blocking. Therefore, the sending thread unit does not pay the cost of accessing the scratch-pad memory of the receiving thread unit. The clock counter, which is not accessed by the other thread units, is placed in a dedicated register of the thread unit for fast access.

The first outermost iteration group, executed on the first thread unit, does not require any synchronization signal to execute. Therefore the synchronization counter of the first thread unit is initialized to the number of synchronization signals that an outermost iteration group needs to receive to run to completion, $syncsPerGroup$. Thus the WAIT instructions of the first group will never block.

The synchronizations signals sent by the thread unit that executes the last outermost iteration group are received by the next thread unit, even if that thread unit is not required to execute any other outermost iteration. Afterward the completion signal is sent to the first thread unit.

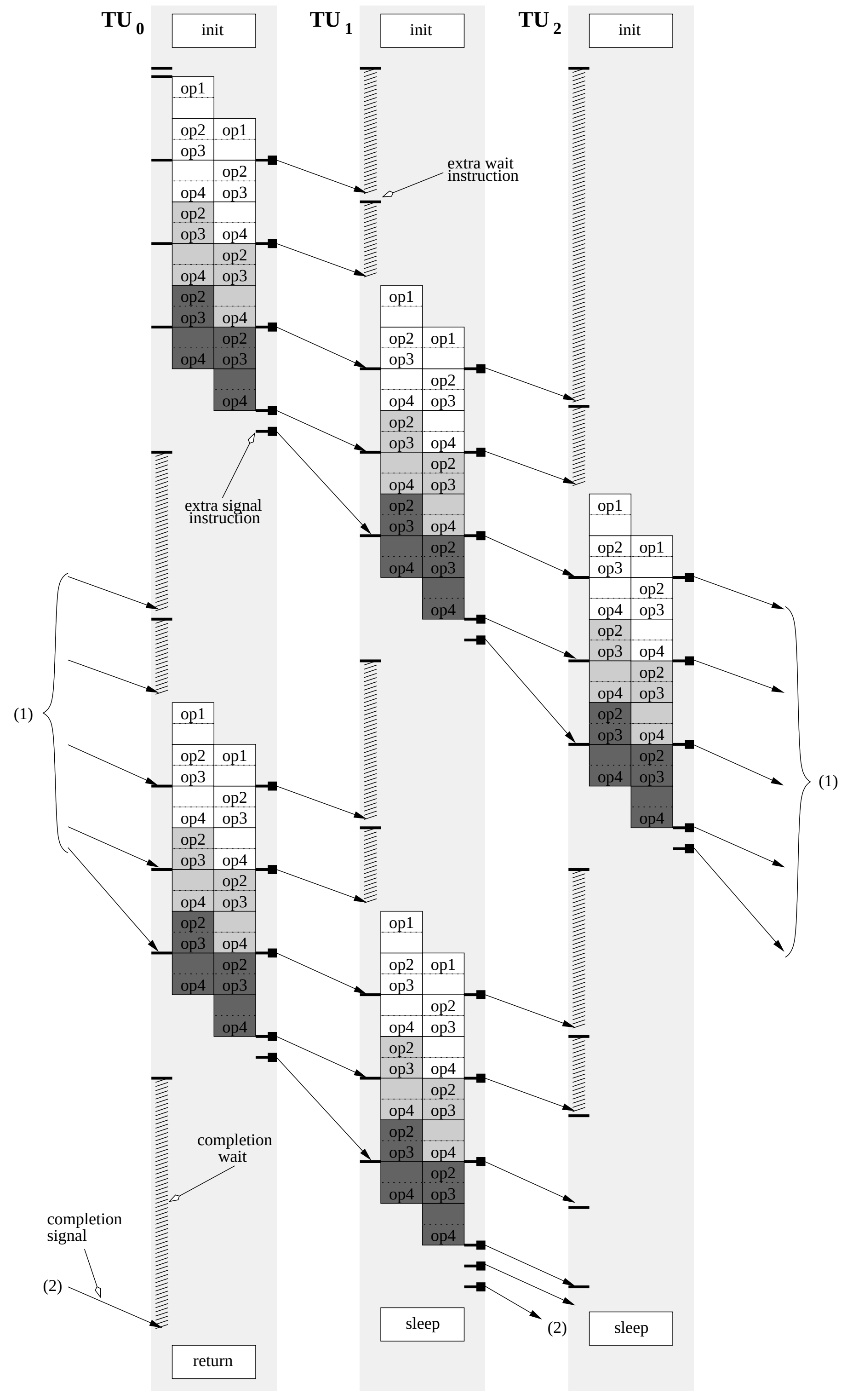


Figure 5: Execution of the Multi-Threaded Final Schedule on an IBM Cyclops-64 chip

5.3 Cross-Iteration Register Dependences

In the IBM Cyclops-64 architecture where threads units do not share registers, cross-iteration register dependences (dependences between outermost iterations whose values are stored in registers, not memory) require the insertion of copy operations to copy the value from one thread unit to the next. On single-core architectures, the problem does not exist as there is a single register file.

During the compilation the register dependence is transformed into a memory dependence. We issue memory spill instructions to copy the value from the register to a buffer in the scratch-pad memory of the destination thread unit. The value is then restored using a single memory load. The scratch-pad memory of the receiving thread unit was chosen because memory spills are non-blocking and memory restores from the local scratch-pad memory are fast because of the direct access of a thread unit to its scratch-pad memory. As the cross-iteration register dependences are known at compile-time, the buffer and offset to its respective values are statically allocated.

Memory spill instructions only need to be issued by the last outermost iteration of an iteration group and memory restore instructions by the first. Within an instruction group, the value is transferred from one outermost iteration to the next using registers as usual. If the value is to be used by another outermost iteration than the next (meaning that the distance of the cross-iteration register dependence is greater than 1), register copies and memory spills/restores will bring that value to the recipient outermost iteration in a cascaded fashion.

The mechanism is implemented by adding memory spill and restore instructions at the ends of each cross-iteration dependence during the loop selection phase. The kernel generator then produces an SSP kernel which contains those extra operations. While emitting the assembly code, the memory spill operations are removed from every iteration but the last of an iteration group, and the memory restore operations are removed from every iteration but the first of an iteration group. The removal of those operations is accompanied by a register renaming transformation to take the change into account.

Fig. 6 shows an example of a cross-iteration register dependence. Register R is used and then incremented by one in the first outermost iteration. The next outermost iteration uses the incremented value and increments again the value in the register. Because the two outermost iterations are executed on the same thread unit, the register is accessible to both iterations. However, the third iteration cannot access that register. Instead, the value is spilled into a known location by the second iteration. The third iteration retrieves the value from the buffer before using it. The spill and restore instructions only appear in the first and last outermost iterations of an iteration group.

5.4 Code Generation Algorithms

The pseudo-code skeleton of the multi-threaded final schedule is shown in Fig. 7(a). The details of the repeating loop patterns are shown in Fig. 7(b). The code is common to all thread units. The first thread unit initiates the execution of the final schedule when sending its first signal to its direct neighbor. The main loop iterates over the iteration groups that each thread unit must execute. The synchronization delay is implemented by artificially incrementing the clock counter by 1.

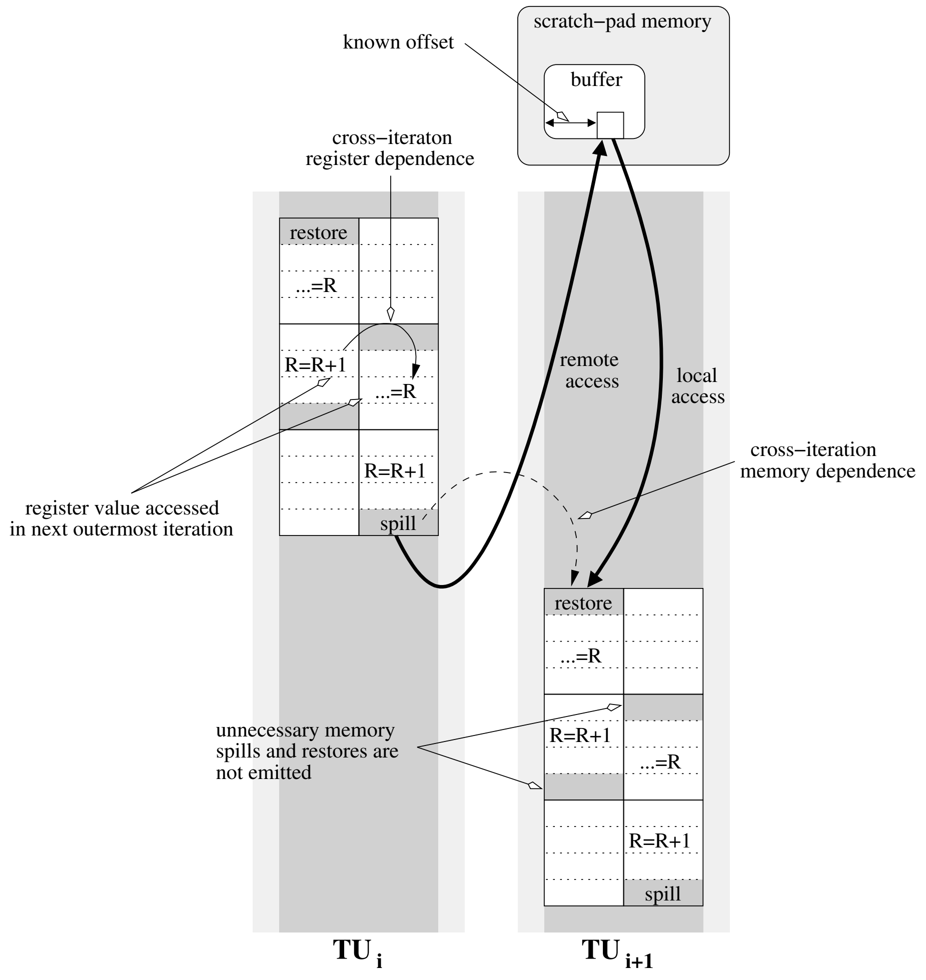


Figure 6: Cross-Iteration Register Dependence Example

Compared to the single-core processor code for the Intel Itanium architecture [15], there is no outermost loop pattern anymore and the innermost loop pattern is now tiled. Register rotation is still required in the prolog and epilog. The register rotation emulation technique used for the other patterns is similar to the Itanium version and will not be described here. The patterns are now surrounded by synchronization instructions.

The stage emission routine, $Emit\_Stages()$, shown in Fig. 8, takes into account the features of the multi-threaded schedule and the absence of predicate registers in the IBM Cyclops-64 architecture. The register offset is passed as a parameter along $total\_height$, the number of instances of the kernel in the pattern being emitted. Given the level $level$ of the stages, the operations are emitted in the order of their scheduling cycle. $stage\_count$ stages are emitted starting from stage $first\_stage + stage\_offset$ to stage $last\_stage$. If the number of emitted does not reach $stage\_count$, then the emission continues, starting from stage $first\_stage$. This cyclic emission is required for the draining/filling patterns and innermost loop patterns. The cross-iteration register dependence memory spill/restore operations are conditionally emitted as explained in Section 5.3.

```
[Initialization]
$i_1 \leftarrow my\_thread\_id$
while $i_1 < N_1$ do
    $i_1 \leftarrow i_1 + num\_TUs$
    $clock\_counter \leftarrow clock\_counter + 1$
    [Prolog]

    for $i_2 = 0, N_2 - 1$ do
        for $i_3 = 0, N_3 - 1$ do
            ...
                for $i = 1, (N_n - 1)/G$ do
                    $wait()$
                    for $g = 1, G$ do
                        [ILP]
                    end for
                    $signal()$
                end for
            ...
            if $i_3 < N_3 - 1$ then
                [DFP$_3$]
            end if
        end for
        if $i_2 < N_2 - 1$ then
            [DFP$_2$]
        end if
    end for

    $rotate\_registers()$
    [Epilog]
    $signal()$
end while
[Conclusion]
```

(a) Multi-Threaded Code Skeleton

```
[Prolog] =
    $wait()$
    for $i = f_1, l_n - 1$ do
        $Emit\_Stages(f_1, l_1, S_n, 1, 0,$
                $i - f_1 - (S_n - 1), l_n - f_1)$
        if $i < l_n - 1$ then
            $rotate\_registers()$
        end if
    end for
    $signal()$

[ILP] =
    for $i = 0, Sn - 1$ do
        $Emit\_Stages(f_n, l_n, S_n, n, -i - 1, i, S_n)$
    end for

[DFP$_{lvl}$] =
    $wait()$
    for $i = 0, l_{lvl} - f_{lvl}$ do
        $Emit\_Stages(f_{lvl}, l_{lvl}, S_n, lvl, -i - 1,$
                $f_n - f_{lvl} + i, l_{lvl} - f_{lvl} + 1)$
    end for
    $signal()$

[Epilog] =
    $wait()$
    for $i = l_n, l1 + Sn - 1$ do
        $Emit\_Stages(f_1, l_1, S_n - max(i - l_1, 0),$
                $1, 0, i - S_n + 1, l_1 - l_n + 1)$
        $rotate\_registers()$
    end for
    $signal()$
```

(b) Loop Patterns Expansion

Figure 7: Code Generation Algorithms

An operation is emitted using the $emit\_ops()$ routine. If the operations is a memory spill/restore operation, then the address register must be switched to the register containing the address of the buffer in the next thread unit. That information is only known at code-emission time and a dummy register had been used so far. Then, the register indexes must adjusted according to the reg_offset value.

For initialization each thread unit must compute the address of the buffer and synchronization counter in the next thread unit. The synchronization counter is then initialized to 0. The clock counter is set to 1 for all the thread units but the first so that the thread units do not start until told so by the previous thread unit. The clock counter of the first thread unit is initialized with $syncsPerGroup$ to be able to execute the first iteration group without requiring any synchronization signal. The live-in values are copied in the local buffer of the first thread unit to bootstrap the execution.

For conclusion the thread unit to execute the last iteration group signals the first thread unit that the

```
EMIT_STAGES($first\_stage$, $last\_stage$, $stage\_count$, $level$, $register\_offset$,
            $stage\_offset$, $total\_height$)
  for $cycle = first\_cycle[level], first\_cycle[level] + T[level] - 1$ do
    $stage\_counter \leftarrow stage\_count$

    $reg\_offset \leftarrow register\_offset$
    $stage \leftarrow first\_stage + stage\_offset$
    while $stage \leq last\_stage$ and $stage\_counter > 0$ do
      if (operation is memory spill and $stage\_counter \neq stage\_count$)
        or (operation is memory restore and $stage\_counter \neq 1$) then
          do not emit this operation
      end if
      $emit\_ops(level, stage, cycle, reg\_offset)$
      $stage\_counter \leftarrow stage\_counter - 1$
      $stage \leftarrow stage + 1$
    end while

    $reg\_offset \leftarrow register\_offset + total\_height$
    $stage \leftarrow first\_stage$
    while $stage\_counter > 0$ do
      if (operation is memory spill and $stage\_counter \neq stage\_count$)
        or (operation is memory restore and $stage\_counter \neq 1$) then
          do not emit this operation
      end if
      $emit\_ops(level, stage, cycle, reg\_offset)$
      $stage\_counter \leftarrow stage\_counter - 1$
      $stage \leftarrow stage + 1$
    end while
  end for
```

Figure 8: Stage Emission Algorithm

schedule is completed. All the thread units, but the first, then go to sleep (or terminate). The first thread units waits for the completion signal to arrive and returns.

In order to reduce the execution time of the schedule, the loop control instructions, such as iteration index increment and trip count comparison, have been added to the operations of the loop nest and therefore scheduled in the kernel. As such, the register offset has been applied to the loop counter registers. The only instruction that has not been scheduled in the kernel is the branch instruction. Therefore, the register offset must also be applied to the branch instruction. The loop control register used should correspond to the one last defined in the last outermost iteration in an iteration group.

5.5 Correctness

We present here two properties that go toward proving that the IBM Cyclops-64 schedule is correct. First the IBM Cyclops-64 implementation of the multi-threaded final schedule is also *deadlock-free*. Indeed the signal instruction is a non-blocking atomic operation. Moreover, despite the round-robin execution of the outermost iteration groups on the thread units, the graph of synchronization signals between outermost iterations still form a tree.

Second the synchronization signal guarantees that the memory accesses preceding it on the same thread unit have been committed. Indeed the accesses to the crossbar network are managed in first-in first-out order at the sending network port. A memory access will not travel across the network until the receiving side can handle the memory access atomically. Therefore the memory accesses issued over the network are guaranteed to be executed in sequential order from the point of the view of the sending thread unit. In consequence, this property also guarantees that when a signal instruction is issued, the preceding memory accesses have already been committed.

6 Experimental Results

This section present our experimental results and analysis. Section 6.1 briefly describes the experimental framework. Section 6.2 summarizes the main results. The rest subsections explain in more details each individual results.

6.1 Experimental Framework

Our MT-SSP method has been implemented in the Open64 research compiler infrastructure as illustrated in Figure 9.

Open64 uses the Winning Hierarchical Intermediate Representation Language, or WHIRL, as the representation for all the optimizations. Most of the Multi-threaded SSP steps are applied at the Very Low level during the code generation (CG) phase, while the data dependence analysis step takes place during the LNO phase at the High level. The final step generates the IBM Cyclops-64 assembly code from the register-allocated kernel.

Fourteen loop nests from the Livermore Suite, SPEC2000 and NAS were compiled and evaluated using FAST–an architecture simulator provided by ETI that can simulate the IBM Cyclops-64 architecture [3] and provide performance information. Loop tiling factor of 1, 2, 4, 8, 16, 32, 64, and 128 were tested on the processor with 99 thread units. The execution time absolute speedup was measured with 1, 3, 7, 15, 31, 63, and 99 thread units with the best loop tiling factor measured. The issue width of a thread unit was assumed to be equal to 2. The problem size of each benchmark was chosen as large as possible.

6.2 Summary of Main Results

The main experimental results can be summarized as follow.

- Result I: Overall correctness and efficiency. For All test cases we experimented with, our MT-SSP scheduler produces multithreaded software pipelined schedules that are correct and deadlock free.
- Result II: Speedup (also see Section 6.3 for details). Our experiments demonstrate a very good scalability across all benchmark tested. For example, as the number of thread units increases, a

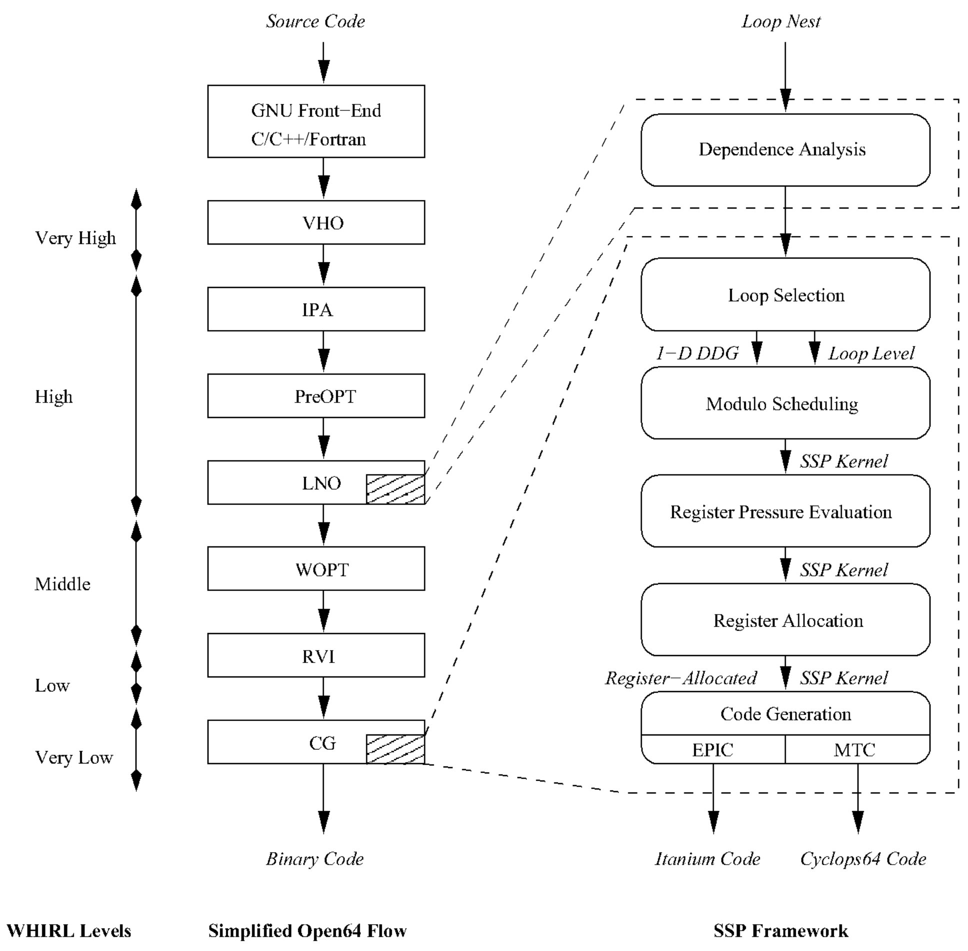


Figure 9: SSP Implementation in Open64

linear speedup has been observed and has reached beyond 81 for matrix multiply and 78 for SOR benchmarks when running on 99 thread units. Also, the speedup improves as the problem sizes increases.

- Result III: Tiling and tiling Factor (also see Section 6.4 for details). Our experiments demonstrates that the tiling factor we have propose in Section 4.3 is quite good: the best results observed with hand-tuned tiling factor matches very well with the calculated factor.

- Result IV: Efficiency of cross-iteration synchronization between thread units (also see Section 6.5 for details). Our experiments demonstrates good efficiency of cross-iteration synchronization between thread units. Although we implement in software such synchronization between thread units, the overhead is quite small and there will not be much room for further improvement even if in-memory atomic synchronization at instruction set architecture level is used.

- Result V: Register Pressure (also see Section 6.6 for details). Our experiments also show that the register pressure due to the MT-SSP scheduling is well within the hardware limits of the register resource in the target multi-core architecture we have used.

6.3 Execution Time Speedup

The scalability results for a representative set of benchmarks are shown in Figure 10. The best loop tiling factor for each benchmark was used in these experiments.

Our experiments demonstrate a very good scalability across all benchmark tested. As the number of thread units increase, a linear speedup has been observed. Also, the speedup improves as the problem sizes increases.

As the number of thread units increases, the total execution time of the benchmarks dramatically reduces. The ikj variant of matrix-multiply shows the best result with an absolute speedup of 81 for 99 thread units. The worst speedup, 57.5 for 99 thread units, was encountered when evaluating benchmark $hydro$.

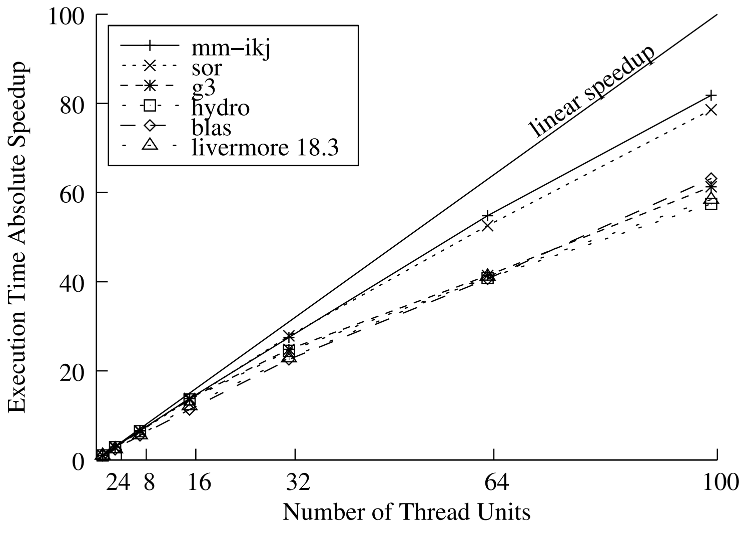


Figure 10: Execution Time Absolute Speedup

We can observe that the performance of our ST-SSP results is still at a distance from the ideal linear speedup. The difference is explained by two facts. First, cross-iteration dependences prevent the iteration group from being executed in fully parallel and achieving a linear speedup. The second explanation is the fixed cost of initializing the schedule. With 99 thread units, the cascaded initialization of all the thread units is costly: thread unit i will not start before receiving two signals from thread unit $i - 1$. Given a fixed number of outermost iterations, the more thread units are used, the higher the initialization cost becomes. If the number of outermost iterations is too small, the initialization becomes the dominant factor in the total execution time.

When the number of outermost iterations increases, so does the execution time speedup. For instance, as shown in Figure 11, when the normalized problem size of the $livermore18.3$ benchmark, is increased by a factor of only 4 (see X-axis: from 2 to 8), the speedup for 99 thread units jumps from 42 to 67.

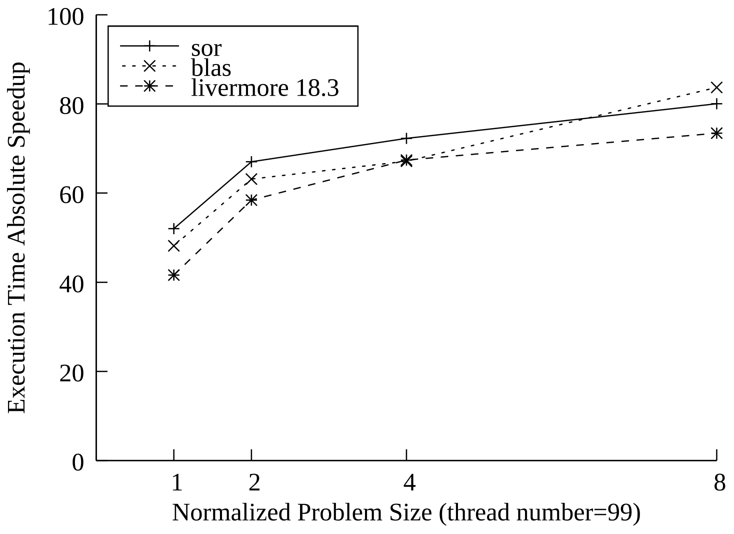


Figure 11: Scalability With Problem Size

6.4 Loop Tiling Factor

The best loop tiling factor was searched for each of the benchmarks running on the IBM Cyclops-64 processor with 99 thread units. A representative set of results are shown in Figure 12. Overall, the benchmarks can be partitioned into 2 groups. In the first group, loop tiling helps reducing the execution time of the benchmarks. For instance, the ijk variant of matrix-multiply shows a speedup of 1.29 with a tile factor of 32 or 64. The second group, which includes $blas$, $hydro$ and $livermore18.3$ in the graph, only shows deteriorating speedup as the tiling factor increases.

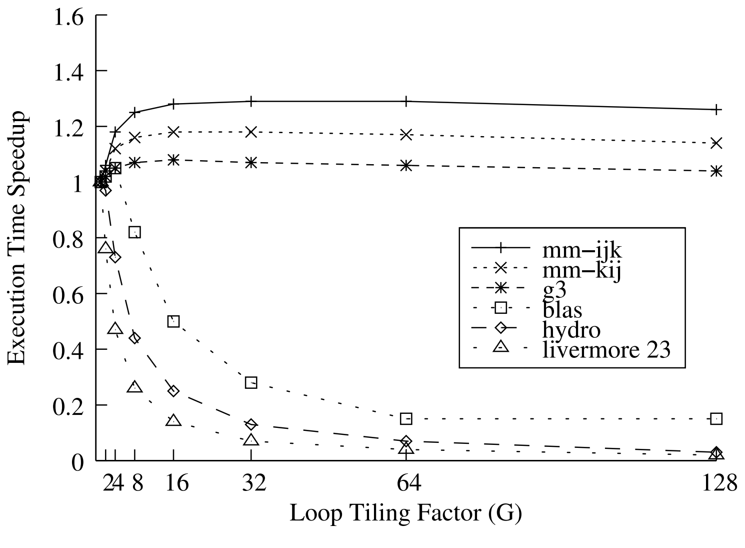


Figure 12: Loop Tiling Factor

The experimental results are in line with theoretical best value for G computed in Equation 1. The best value for G for the benchmarks in the second group happens to be 1 because the number of iterations of the inner loops is too small. Using the measured average execution time of the wait instruction (14 cycles), the difference in total execution time using the empirical best value for G and the theoretical best G value (G_{min}) was measured. The maximum recorded difference was 1.7% with an average of 0.3%. G_{min} is therefore a very accurate approximation of the best value to be chosen for G.

Note that, here we report our experiments results on smaller problem sizes to save simulation time while without loss of generality.

6.5 Efficiency of cross-iteration synchronization between thread units

To study the efficiency of cross-iteration synchronization between iterations - we report the synchronization stall cycles. The stall cycles were measured for all the benchmarks and using 99 thread units with the best loop tiling factor. The average execution time of the wait is 14 cycles (synchronization delays excluded). It is exactly the time it takes to execute the instruction when the data are already in the buffer. Therefore, the wait instruction never blocked in the tested benchmarks.

The value could be further reduced by implementing the wait instruction as an atomic instruction in the instruction set architecture. However, such a dedicated instruction would have almost no impact on the total execution time of the schedules. On average, the number of cycles to execute the wait instructions represents only 0.2% of the total execution time with a maximum of 0.7%. The cost of the wait instruction is therefore negligible.

6.6 Register Pressure

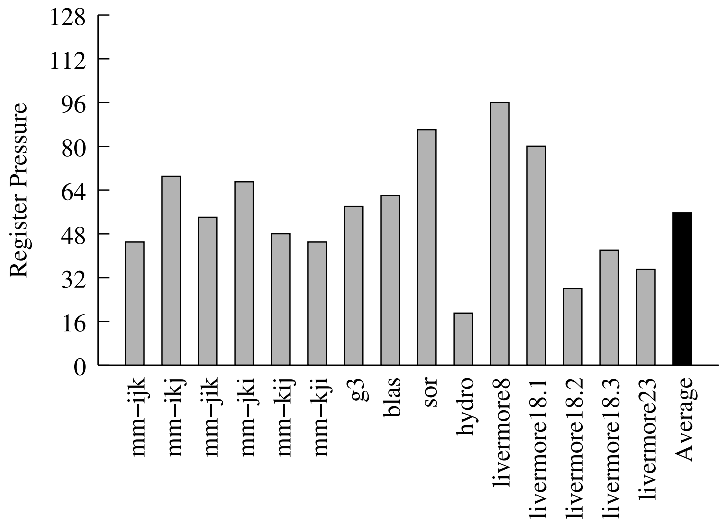


Figure 13: Register Pressure

Finally, the register pressure was measured for each of the benchmark. The average register pressure was measured at 55.1 registers with a maximum of 96 for $livermore8$. That pressure is reasonable and much lower than for the Intel Itanium architecture [14], considering that IBM Cyclops-64 registers are used both for floating-point and integer values. Such difference is explained by the reduced issue width of the IBM Cyclops-64 processor. A limited issue width increases the initiation interval of the kernel and therefore reduces the number of stages of the kernel. As a result, the number of interfering lifetimes is also reduced.

The register pressure can be further reduced by tuning the register allocator to the multi-threaded schedule. Indeed, stretched intervals [14], present in the single-core schedules, are now non-existent and do not need to be accounted for anymore.

7 Related Work

There exists a very large number of related work to schedule loops on multi-threaded or multi-core architectures. However they can only be applied to single loops. MT-SSP is applied to a loop nest and produces a compact multi-threaded final schedule with minimized synchronization costs. Some of this work is presented here.

Several software-pipelining techniques [12, 8, 18, 19] were proposed for clustered-VLIW architectures. The IBM Cyclops-64 architecture is fundamentally different as the thread units are independent from each other and interconnected via a network instead of a bus. Extra synchronization is required for the cellular architecture, which can easily step up the number of thread units to the hundreds whereas clustered architectures are limited to tens of independent compute engines.

Decoupled Software-Pipelining [13] schedules a single loop over multiple thread units. Instead of distributing the iterations over the thread units, the same thread unit always executes the same group of operations. Thus, if an iteration can be partitioned into groups, the first thread unit executes the first group of every iteration, the second thread unit the second group, etc. However the solution does not scale well when the number of thread units reaches the hundreds.

Other multi-threading techniques include speculative multi-threading with run-time analysis, also called run-time parallelization of DOACROSS loops [1, 21, 20]. Threads are speculatively issued and killed as information about the execution of loops is known. Although those methods allow for a wider range of loops to be scheduled, especially with pointer-chasing structures, the thread control overheads are very high compared to MT-SSP.

Some work was also done to efficiently port OpenMP to the IBM Cyclops64 architecture [2]. However, OpenMP is high-level language requiring extra intervention from the programmer and suffering from generic constructs overheads. MT-SSP is directly applied at the instruction level and exclusively target loop nests.

8 Conclusion

In this paper we presented a solution to software pipeline loop nests on multi-threaded cellular architectures based on SSP. The method is named Multi-Threaded Single-dimension Software-Pipelining

(MT-SSP). Given the SSP kernel, a fully synchronized multi-threaded final schedule is generated to efficiently execute the loop nest without any modification to the source code. The schedule is proven deadlock-free and respect all the dependence and resource constraints.

Code generation algorithms were presented for the IBM Cyclops-64 architecture. Synchronization is done through the use of a Lamport's clock on each thread unit. The signal instruction is non-blocking to allow for faster execution. The synchronization counter is placed in the local scratch-pad memory of the receiving thread unit to limit network accesses and drastically reduce the execution time of the wait instruction. Cross-iteration register dependences between thread units were handled through the use of memory spill and restore operations to and from a buffer also in the scratch-pad memory of the receiving thread unit. Those operations are scheduled in the kernel.

Experimental results showed that multi-threaded SSP schedules scales up well when the number of thread units increases. The implementation uses a very light-weight synchronization method with only standard instructions of the IBM Cyclops-64 architecture. The loop tiling factor was shown to be correctly approximated using the definition of G_{best}. Finally, the register pressure appeared to be reasonable without taking any extra steps to reduce it.

References

[1] D.-K. Chen. *Compiler Optimizations for Parallel Loops with Fine-grained Synchronization*. PhD thesis, Department of Computer Science, University of Illinois, Urbana, Illinois, 1994.

[2] Juan del Cuvillo, Weirong Zhu, and Guang R. Gao. Landing OpenMP on Cyclops64: An efficient mapping of Open64 to a many-core system-on-a-chip. In *Proceedings of the ACM International Conference on Computing Frontiers 2006*, May 2006.

[3] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. FAST: a functionally accurate simulation toolset for the Cyclops64 cellular architecture. In *Proceedings of the Workshop on Modeling, Benchmarking and Simulation*, pages 11–20, June 2005.

[4] Monty Denneau and Henry S. Warren, Jr. 64-bit Cyclops principles of operation part I. Technical report, IBM Watson Research Center, Yorktown Heights, NY, April 2005.

[5] Monty Denneau and Henry S. Warren, Jr. 64-bit Cyclops principles of operation part II: Memory organization, the A-switch, and SPRs. Technical report, IBM Watson Research Center, Yorktown Heights, NY, April 2005.

[6] Alban Douillet. *A Compiler Framework for Loop Nest Software-Pipelining*. PhD thesis, University of Delaware, Newark, Delaware, USA, August 2006.

[7] Alban Douillet, Hongbo Rong, and Guang R. Gao. Multi-dimensional kernel generation for loop nest software pipelining. In *Proceedings of the 2006 Europar*, Dresden, Germany, August 29th–September 1st, 2006. Springer-Verlag, Lecture Notes in Computer Science.

[8] M.M. Fernandes, J. Llosa, and N. Topham. Distributed modulo-scheduling. In *Procs. of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 130–134, January 1999.

[9] Rakesh Kumar, Victor Zyuban, and Dean M. Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. *SIGARCH Comput. Archit. News*, 33(2):408–419, 2005.

[10] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, Atlanta, Georgia, June 22–24, 1988. SIGPLAN Notices, 23(7), July 1988.

[11] Kalyan Muthukumar and Gautam Doshi. Software pipelining of nested loops. In *Proceedings of the 10th International Conference on Compiler Construction, CC 2001*, volume 2027 in Lecture Notes in Computer Science, pages 165–181, 2001.

[12] E. Nystrom and A. E. Eichenberger. Effective cluster assignment for modulo scheduling. In *Procs. of 31st International Symposium on Microarchitecture*, pages 103–114, 1998.

[13] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, 2005.

[14] Hongbo Rong, Alban Douillet, and Guang R. Gao. Register allocation for software pipelined multi-dimensional loops. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 154–167, New York, NY, USA, 2005. ACM Press.

[15] Hongbo Rong, Alban Douillet, Ramaswamy Govindarajan, and Guang R. Gao. Code generation for single-dimension software-pipelining for multi-dimensional loops. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*, pages 175–184. IEEE Computer Society, March 2004.

[16] Hongbo Rong, Zhizhong Tang, Ramaswamy Govindarajan, Alban Douillet, and Guang R. Gao. Single-dimension software pipelining for multi-dimensional loops. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*, pages 163–174. IEEE Computer Society, March 2004.

[17] John Ruttenberg, G. R. Gao, A. Stoutchinin, and W. Lichtenstein. Software pipelining showdown: optimal vs. heuristic methods in a production compiler. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 1–11. ACM Press, 1996.

[18] J. Sánchez and A. González. The effectiveness of loop unrolling for modulo scheduling in clustered VLIW architectures. In *Procs. of the 29th Int. Conf. on Parallel Processing*, pages 555–562, August 2000.

[19] J. Sánchez and A. González. Modulo-scheduling for a fully-distributed clustered VLIW architecture. In *Procs. of 33rd Int. Symp. on Microarchitecture*, December 2000.

[20] J.Y. Tsai, J. Huang, C. Amlo, D.J. Lilja, and P.C. Yew. The superthreaded processor architecture. *IEEE Transactions on Computers, Special Issue on Multithreaded Architectures*, 48(9), September 1999.

[21] J.Y. Tsai and P.C. Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT'96)*, pages 35–46, October 1996.

[22] Jian Wang and Guang R. Gao. Pipelining-dovetailing: A transformation to enhance software pipelining for nested loops. In *Proceedings of the 6th International Conference on Compiler Construction, CC '96*, Lecture Notes in Computer Science, pages 1–17, Linkoping, Sweden, April 22–26, 1996. Springer-Verlag.

A Appendix: Mathematical Definitions

A.1 Synchronizations per Outermost Iteration Group

The number of synchronization signals sent (or received) by a group of S_n outermost iterations, $syncsPerGroup$, is equal to the number of instances of each repeating pattern in the group to which we add the extra synchronization instruction used for the synchronization delay ($extraSync = 1$). In one iteration group, the prolog is executed only P=1 time. So is the epilog ($E = 1$). The number of times DFP_i is executed, noted D_i with $i \in [2, n-1]$, is given by:

$$D_i \quad = \quad (N_i - 1) * D_{i-1} \text{ with } D_1 = P = \prod_{j=2}^{j=i} (N_j - 1) \tag{3}$$

The number of times the tiled ILP is executed, noted I, can be expressed as:

$$I = \frac{N_n - 1}{G} * (D_{n-1} + P) = \frac{N_n - 1}{G} * \left(1 + \prod_{j=2}^{j=n-1} (N_j - 1)\right) \tag{4}$$

Then we have after simplification

$$\begin{aligned} syncsPerGroup &= extra + P + I + E + \sum_{i=2}^{i=n-1} D_i \\ &= 3 + \frac{N_n - 1}{G} + \frac{N_n - 1}{G} * \prod_{j=2}^{j=n-1} (N_j - 1) + \sum_{i=2}^{i=n-1} \prod_{j=2}^{j=i} (N_j - 1) \end{aligned} \tag{5}$$

If $N_n - 1$ is not a multiple of G, $(N_n - 1)/G$ must be replaced by $\lfloor (N_n - 1)/G \rfloor + 1$ to account for the extra wait instructions. If the loop nest is a double nest, then the last two terms of Equation 5 are equal to zero.

A.2 MT-SSP Schedule Function

The multi-threaded schedule function, given in Eqn. 2, can be computed from the single-core schedule function [16, 6] by ignoring the term introduced by the delay in single-core schedules and by adding the initialization costs and the synchronization costs of the multi-threaded schedules:

$$\sigma(op, 0) + i_1 * T + \sum_{k=2}^{k=n} i_k * time_{L_k} + initCosts + syncCosts \tag{6}$$

where

$$\begin{aligned} time_{L_k} &= \sum_{i=k}^{i=n} \left((S_i - S_{i+1}) * T * \prod_{j=k+1}^{j=i} N_j \right) \\ S_{n+1} &= 0 \end{aligned}$$

The initialization costs correspond to the cascaded wake-ups of the thread units to which we add the synchronization delay. We assume that the SIGNAL operation is executed instantaneously. The last group of S_n outermost iterations is executed after $\lfloor N_1/S_n \rfloor - 1$ cascaded wake-up signals. Those signals are sent every $l_n * T$ cycles. The synchronization delay adds another $G * S_n * T$ cycle delay before the second signal is sent. Therefore the initialization cost is equal to:

$$\left(\left\lfloor \frac{N_1}{S_n} \right\rfloor - 1 \right) * (l_n * T + G * S_n * T) \tag{7}$$

The synchronization costs is the sum of the execution time of all the WAIT operations in the last group of S_n outermost iterations, $syncsPerGroup$ (defined in Appendix A.1), and those required for the cascaded wake-up calls. Let w be the time to execute one WAIT operation without stalling. For a thread unit to start running, its predecessor must execute two WAIT operations. Therefore the cascaded signals amount to $(\lfloor N_1/S_n \rfloor - 2) * 2 * w$ cycles and the synchronization costs are equal to

$$\left(\left\lfloor \frac{N_1}{S_n} \right\rfloor - 2 \right) * 2 * w + syncsPerGroup * w \tag{8}$$