A Software Pipelining Framework for Simple Processor Cores

Juergen Ributzka
University of Delaware
140 Evans Hall
Newark, DE 19716
ributzka@capsl.udel.edu

David Stephenson
PathScale, LLC
1320 Chesapeake Terrace
Sunnyvale, CA 94089-1100
dstephenson@pathscale.com

Timothy Kong
PathScale, LLC
1320 Chesapeake Terrace
Sunnyvale, CA 94089-1100
tkong@pathscale.com

Dee Lee
PathScale, LLC
1320 Chesapeake Terrace
Sunnyvale, CA 94089-1100
dee@pathscale.com

Fred Chow
PathScale, LLC
1320 Chesapeake Terrace
Sunnyvale, CA 94089-1100
fchow@pathscale.com

Guang R. Gao
University of Delaware
140 Evans Hall
Newark, DE 19716
ggao@capsl.udel.edu

Abstract

Current trends in many-core architectures show a switch from a small number of architecturally sophisticated cores (e.g. Intel Core2, IBM PowerPC) to many simple cores (e.g SiCortex and Tilera multiprocessor). These simple cores lack many of the advanced features of the complex cores (e.g. out-of-order execution, rotating register files, predication, speculation, etc.), which puts extra burden on the compiler to produce an efficient schedule, especially for loops. We provide an advanced loop scheduling framework using software pipelining, which does not rely on any special hardware support. This open-source framework makes software pipelining available to simple cores and helps to mitigate the lack of advanced hardware features. The paper also provides measurements to demonstrate the effectiveness of this framework.

General Terms compiler optimization, loop scheduling

Keywords Open64, software pipelining, modulo scheduling, register allocation, code generation, MIPS

1. Introduction

Normal straight-line scheduling techniques, like list scheduling [9] or hyper-block scheduling (HBS) [16] do a sub-optimal job when it comes to loops. The common approach is to unroll the loop several times to generate a larger loop body and then perform straight-line scheduling on it. This method has two drawbacks. Firstly, there might be not enough parallelism in the unrolled loop to fully utilize the hardware, leading to a sub-optimal schedule. Secondly, there is the draining of the pipeline at the end of every unrolled loop iteration and the refilling at the next iteration. Software pipelining is a very well known loop scheduling technique, which is able to mitigate or completely eliminate the drawbacks mentioned above. To illustrate the difference between these two scheduling techniques, consider the following reduction loop example in Figure 1. Figure 2 shows the

```
for (i = 0; i < SIZE; ++i) {
  sum += a[i];
}
```

Figure 1: Reduction Loop (C-Code)

resulting pseudo assembly code after unrolling the loop 4 times. The 4 pointer updates (daddiu) are reduced to a single pointer update and the offset of the load operations are adjusted. This is the reason why there is only one pointer update later in the displayed schedules. Figure 3 shows a

```
loop:
 TN267 :- ldc1    GTN277 (0x0)
GTN271 :- add.d    TN267 GTN271
 TN274 :- daddiu  GTN277 (0x8)
 TN268 :- ldc1     TN274 (0x0)
GTN272 :- add.d    TN268 GTN272
 TN275 :- daddiu   TN274 (0x8)
 TN269 :- ldc1     TN275 (0x0)
GTN273 :- add.d    TN269 GTN273
 TN276 :- daddiu   TN275 (0x8)
 TN270 :- ldc1     TN276 (0x0)
GTN241 :- add.d    TN270 GTN241
GTN277 :- daddiu   TN276 (0x8)
        :- bne     GTN277 GTN239 (lab:loop)
```

Figure 2: Reduction Loop - unrolled 4 times (Pseudo Assembly Code)

schedule, which has been obtained with HBS. Every iteration starts with loading the required data and finishes with

processing them. The loading and processing have been partially overlapped. This schedule uses 71% of the available hardware resources[1]. Considering that the loop was unrolled 4 times and the schedule needs 8 cycles to finish one unrolled loop iteration, each loop iteration is only 2 cycles long.

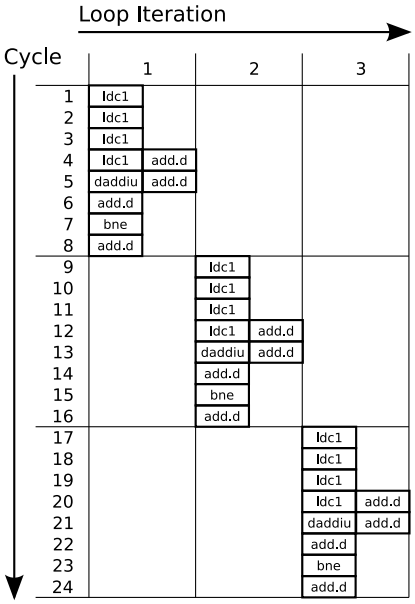


Figure 3: Trace Schedule

Figure 4 shows a possible schedule, obtained with software pipelining. Note the differentiation of the schedule into prologue, kernel, and epilogue. The prologue is necessary to fill the pipeline and is only issued once at the beginning of the loop. The kernel represents the steady-state of the schedule, which keeps the pipeline busy. The epilogue is also only issued once, but at the end of the loop to drain the pipeline. The kernel simultaneously processes the data of the current iteration and loads data for the next iteration. This schedule uses 100% of the available hardware resources. For this schedule we ignore the prologue and the epilogue for the rate calculation under the premise that we have a loop with a large number of iterations. Considering that the loop was unrolled 4 times and the schedule needs 6 cycles to finish one unrolled loop iteration, each loop iteration is now only 1.5 cycles long. The actual cost depends on the number of iterations and can be calculated with the following formula: $\frac{cycle}{iteration} = \frac{cycle_P + cycle_K \times n + cycle_E}{n}$, where $cycle_P$, $cycle_K$, and $cycle_E$ are the length of the prologue, kernel, and epilogue in cycles, respectively. This small example shows that a loop-aware scheduler is necessary to take advantage of the parallelism across iterations and to achieve better performance for in-order execution architectures. Out-of-order execution architectures are less affected, because

[1] we assume the SiCortex Multiprocessor for this calculation as later described in Section 4.1

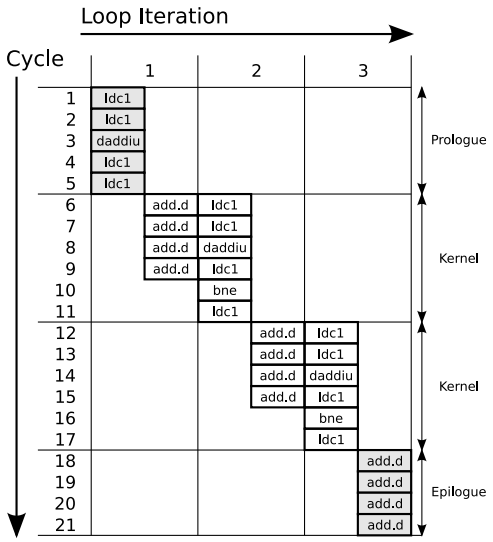


Figure 4: Modulo Schedule

their hardware will attempt to maximize pipeline utilization. In this paper

- we present our framework that implements a software pipeliner, which does not depend on any special architectural features (Section 3), including
- an optimized code generation schema for architectures with static branch prediction (Section 3.7). Moreover,
- we demonstrate its robustness and correctness in the Open64 compiler (Section 4).

2. Background

Out-of-order execution allows processor to by-pass instructions which are not ready for execution and would have otherwise stalled the processor. There have been different approaches to achieve this goal (e.g. scoreboarding, Tomasulo algorithm [25], and reorder buffer). These methods are able to increase performance, but they also require sophisticated hardware implementations. In-order execution architectures are more sensitive to a given schedule and require the compiler to carefully consider the processor's pipeline behavior. A compiler can fine tune the schedule for a specific processor, but sometimes it is not possible for the compiler to determine an optimized schedule, because necessary information may only be available during runtime. In Section 3 we introduce a scheduling technique, which allows us to narrow the performance gap between in-order and out-of-order execution architectures.

Sophisticated architectures, like the Itanium architecture from Intel and HP, have a feature called predication. Predication allows the compiler to convert control dependencies into data dependencies [11]. Predication has two distinct, but very interesting impacts on software pipelining. Firstly, it allows the compiler to generate larger basic blocks (BBs) via

if-conversion [3]. This makes more loops suitable for software pipelining. Secondly, it helps to reduce code size for software pipelined loops. The software pipeliner normally generates additional code for prologue and epilogue, which results in larger code compared to the original loop. With predication and special branch instructions it is possible to simulate the prologue and epilogue code, resulting in kernel-only code [18, 19]. Without these instructions, we have to rely on the original code generation schema of prologue, kernel and epilogue.

Speculation is another advanced feature of the Itanium architecture, which allows the processor to perform certain operations speculatively without changing the state of the memory or throwing exceptions. This is useful for software pipelining of WHILE-LOOPS, where we don't know how many iterations are to be executed. Without speculation, software pipelining these loops is very limited and deemed not profitable.

Rotating register files are one of the most important features for software pipelining. Rotating registers help to remove false dependencies (anti- and output-dependencies), allowing the scheduler to find a better schedule (see Section 3.3). Without this feature, we have to unroll the kernel and perform register rotation manually, resulting in larger code.

Since the existing software pipeliner in the Open64 compiler assumes these sophisticated features, there has not been any software pipelining support in the Open64 compiler for any processor other than Itanium. By adding our new software pipelining framework to the Open64 compiler, we hope other target architectures of the Open64 compiler can benefit from having the software pipelining feature.

3. Framework

3.1 Overview of the Framework

The software pipelining framework in the PathScale EkoPath compiler (a commercial x86 and MIPS compiler based on Open64) is part of a multi-level optimization framework for loops. Starting at the outer level we have the general loop optimization framework, which analyzes the different loops, one at a time, starting at the innermost loop level. Even though we only optimize the innermost loop level, it is also necessary to check the outer loop levels. This is necessary, because we may fully unroll an inner loop during loop optimization, which makes the innermost loop level disappear. Loops are divided into two categories - DO-LOOPS and WHILE-LOOPS. These two categories are further subdivided, depending on the properties of the loop we would like to optimize. One of the most important criteria is the number of basic blocks (BBs). Most advanced loop optimization techniques, like software pipelining, can only be performed on a single BB. It is possible to perform software pipelining on WHILE-LOOPS, but it requires special hardware support. Since the current implementation targets a processor which does not support speculation, software pipelining is not performed on those loops. Multi-BB loops are not supported by the current software pipeliner either and are therefore scheduled with the Hyper-Block Scheduler (HBS).

If the loop optimization framework finds a suitable loop for software pipelining, further optimizations are performed before the loop is finally passed on to the software pipelining framework. These include loop unrolling, recurrence breaking, induction variable removal, load store elimination and extended block optimization (EBO). After all these optimizations and transformations have been performed, the software pipelining framework is invoked to do the scheduling, register allocation and code generation of the loop.

The first step is the calculation of the data dependence graph (DDG), followed by the minimum initiation interval (MII) calculation. Then the modulo scheduler uses this information to find a schedule. If successful, modulo variable expansion (MVE) and then register allocation (RA) is performed. If register allocation fails, the framework gives up and the loop is restored to its original form. Otherwise we continue with the final step - code generation (CG) (see Figure 5).

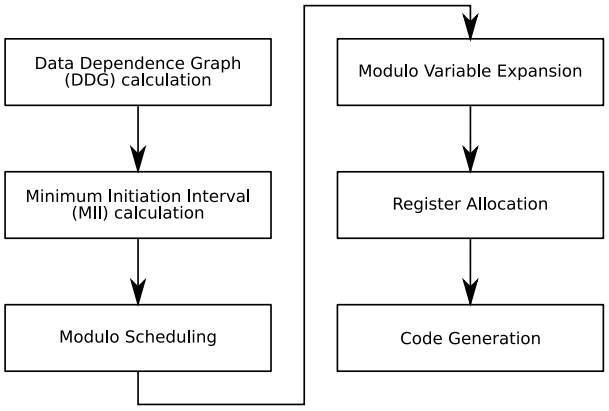
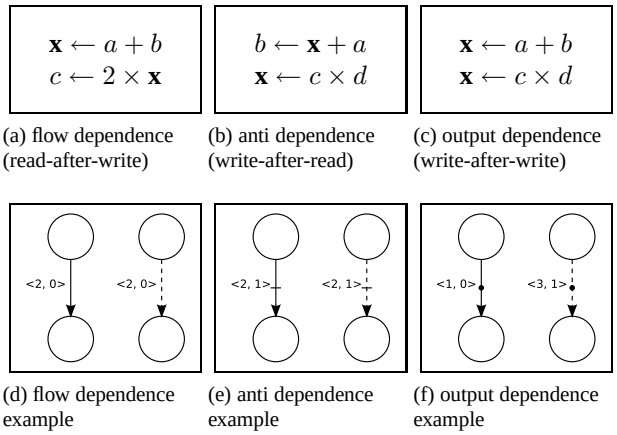


Figure 5: Software Pipelining Framework

3.2 Data Dependence Graph (DDG)

The data dependence graph (DDG) is a representation of the various data dependencies between a given set of instructions. The data dependencies are called flow, anti, and output. Flow dependence is also known as true dependence and the remaining two as false dependence [1]. In the context of a hardware pipeline, these dependencies correlate to read-after-write (RAW), write-after-read (WAR) and write-after-write (WAW) hazards, respectively. Figure 6 shows an example for each dependence type. Dependencies between registers are indicated with a solid edge. Dependencies between memory locations use a dashed edge instead. Each edge has two values associated with it (e.g. $<2,1>$). The first value is called δ and represents the latency between two instructions. The second value is called ω and represents the iteration distance.



(a) flow dependence (read-after-write)
(b) anti dependence (write-after-read)
(c) output dependence (write-after-write)
(d) flow dependence example
(e) anti dependence example
(f) output dependence example

Figure 6: Data Dependencies

Definition 3.1. Let $DDG = G(V, E, \delta, \omega)$ be a cyclic directed graph, where

- V is the set of vertices of the graph G. Each vertex $v \in V$ represents one instruction of the loop L.
- E is the set of dependence edges. Each edge $e_{k(u,v)} \in E$ represents one dependence between the vertices $u \in V$ and $v \in V$. There may be more then one edge $e_{k(u,v)} \in E$ between the same two vertices u and v.
- $\delta_{k(u,v)}$ is the latency in processor cycles between the two vertices u and v, and is associated to the corresponding edge $e_{k(u,v)}$. The value of δ depends on the architecture and which instructions u and v represent. It is a non-negative number for RISC like architectures. Negative numbers are possible for VLIW and EPIC architectures [21].
- $\omega_{k(u,v)}$ is the iteration distance between two vertices u and v, and is associated with the corresponding edge $e_{k(u,v)}$. This means that v depends on a value from u, which has been produced $\omega_{k(u,v)}$ iterations before. Therefore, ω is a non-negative number, since we can't depend on values which will be produced in future iterations.

The current DDG framework can calculate non-cyclic DDGs for single- and multiple-BBs. Cyclic DDGs on the other side can currently only be generated for single BBs. The DDG, which is generated for software pipelining does not have anti- and output-dependencies for registers, because they can be removed by register renaming (see section 3.3 and 3.5). Since DDGs are an integral part of every compiler, there was no need to reimplement it in the context of this paper.

3.3 Minimum Initiation Interval

Modulo scheduling requires a fixed initiation interval (II), for which it tries to find a valid schedule. The lower the II, the shorter is the resulting runtime. The lower bound for the II is called the minimum initiation interval (MII). Two factors determine the MII. One is the critical resource usage, the other is the critical recurrence circuit in the DDG. They are called resource MII (ResMII) and recurrence MII (RecMII), respectively. The maximum of both yield a possible, but not necessarily feasible, lower bound. Complex interactions between data dependencies and resource restrictions may prevent a valid schedule at MII. Nevertheless, it is a good starting point to find a schedule.

Definition 3.2. $MII = \max(ResMII, RecMII)$

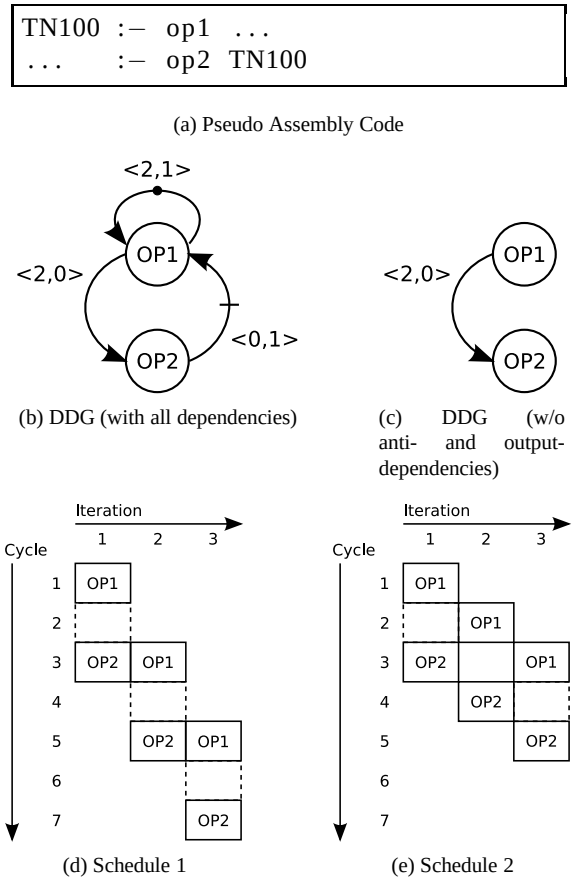
The ResMII is determined only by the critical resource usage of the functional units. Dependencies between instructions are ignored during this calculation. RecMII calculation on the other side assumes infinite resources.

Definition 3.3. $RecMII = \max_{\forall c \in C} \left\lceil \frac{delay(c)}{distance(c)} \right\rceil$, where $delay(c)$ is the sum of all δ's and $distance(c)$ is the sum of all ω's in the elementary circuit c.

Anti- and output-dependencies have a negative effect on RecMII, because they create elementary circuits in the DDG. For every flow dependence between a producer and consumer operation, there exists an anti-dependence between the consumer and the producer operation. Moreover, every operation (which produces a result) is output dependent onto itself. This creates unnecessary dependence cycles in the DDG and limits the execution rate of the loop. Register anti- and output-dependencies can be eliminated by register renaming. In hardware this can be done with a rotating register file [18]. If we do not have the hardware support, we can simulate it with modulo variable expansion (MVE) (see Section 3.5). Figure 7 shows a simplified example how these false dependencies can hinder a more efficient schedule. Figure 7b shows all the dependencies for the given pseudo assembly code in Figure 7a. Figure 7c shows the pruned DDG without anti- and output-dependencies for registers. Figures 7d and 7e show the resulting schedule for each case. The false dependencies create the recurrence circuits in the DDG, which are clearly the limiting factor in this example. By removing these dependencies, it is now possible for the modulo scheduler to overlap more instructions and therefore increase instruction level parallelism (ILP). On the other hand this might also increase register pressure. The RecMII calculation in this compiler is based on Monica Lam's method [13], by finding first all the elementary circuits (also known as strongly connected components (SCC)) in the DDG using Tarjan's algorithm [24] and then solving the all-points longest path problem for each SCC with Floyd's algorithm [8].

3.4 Modulo Scheduler

The modulo scheduler is the most important and also the most compile time consuming part of the framework. In general, a modulo scheduler tries to place the instructions of



(a) Pseudo Assembly Code

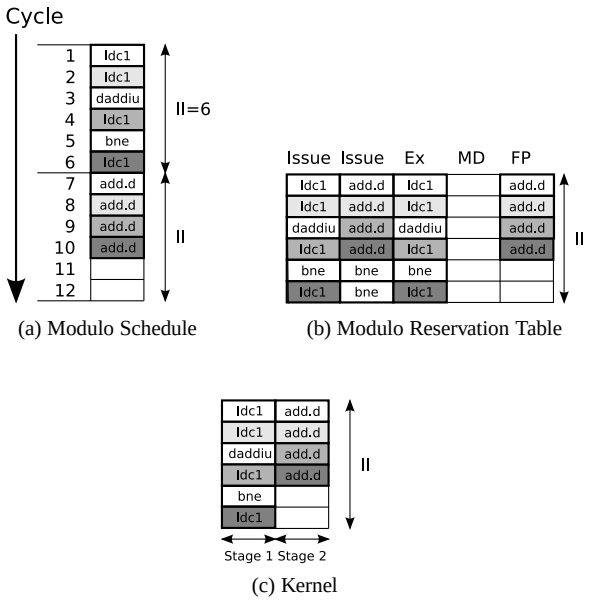
(b) DDG (with all dependencies)

(c) DDG (w/o anti- and output-dependencies)

(d) Schedule 1

(e) Schedule 2

Figure 7: Recurrence Example



(a) Modulo Schedule

(b) Modulo Reservation Table

(c) Kernel

Figure 8: Modulo Scheduler

one loop iteration, one at a time, into free issue slots, considering the dependence and resource constrains. In Figure 8a you can see one possible schedule for the reduction loop example from Section 1. The corresponding modulo reservation table (MRT) is shown in Figure 8b. Since the processor is dual-issue capable, we have two issue slots displayed in the MRT. Ex represents the integer unit, MD the integer multiply-divide unit and FP the floating-point unit. The branch instruction of this architecture can not be dual-issued and also can only have one instruction in the delay slot. Therefore, the branch instruction occupies both issue slots when it is issued and one issue slot in the cycle after. The resulting kernel has 2 stages and is shown in Figure 8c. The current modulo scheduler implementation is based on Huff's Lifetime-Sensitive Modulo Scheduler [10]. First we calculate the earliest and latest possible starting time for every instruction and the slack, which is just the difference of the former two. Furthermore, we calculate which hardware resources are in high demand. Then we start scheduling one instruction at a time. A heuristic decides which instruction needs to be scheduled next. This is done by assigning a priority to each instruction, which depends on the current slack of the instruction and on the hardware resource requirements. After an instruction has been chosen by the heuristic, we need to check if there are free resources in the modulo reservation table, between the latest and earliest starting time. The search direction, which decides if we want to place an instruction later or earlier is based on heuristics, which tries to minimize register lifetime. More details about the heuristic can be found in the paper [10]. If we can't find free resources, we just pick a time slot. Instructions, which already have been scheduled and violate any dependence constrain, will be unscheduled, including all it's successors and predecessors. If there are not enough free resources to schedule the selected instruction, other instruction, which use the same resources or relevant resources, will be unscheduled until we can place the new instruction. After an instruction has been placed, we recalculate the earliest and latest starting time and the slack, because the new placed instruction might have affected the earliest and/or latest starting time of other instructions. We repeat these steps until we were able to schedule all instructions. This is not always possible for a given II. Every instruction has a budget. Everytime an instruction is scheduled, the budget is decreased. If the budget of one instruction is depleted, the modulo scheduler gives up and tries another II. To reduce compilation time, the search space for a schedule is first pruned by searching for a valid schedule with exponential increasing steps for II. Then we perform a binary search in this pruned search space.

3.5 Modulo Variable Expansion

The modulo scheduler scheduled the instructions on the premise that we don't have any anti- or output-dependencies for registers. To guarantee this, we need to provide an unique

Temporary Name (TN)[2] set for every iteration. We perform register renaming according to [12, 13]. First we calculate the lifetime of every TN. The longest lifetime determines how often we have to unroll the kernel.

Definition 3.4. $k_{unroll} = \max_{\forall lt \in LT} \left\lceil \frac{lt}{II} \right\rceil$, where LT is the set of lifetimes of the modulo scheduled loop L.

Then, the kernel is unrolled k_{unroll} times and every loop iteration becomes a new TN set assigned (see Figure 9). TNs in bold indicate loop-invariants and are therefore not renamed.

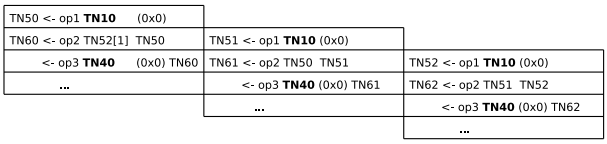


Figure 9: Modulo Variable Expansion Example

3.6 Register Allocation

After modulo variable expansion, the lifetime of every TN is calculated. The register allocator, which is based on [18], is provided with the start cycle, end cycle, ω value and α value of every TN. The start cycle defines when the TN has been defined by an instruction. The end cycle is determined by instructions which use the TN and the ω value. ω defines the iteration distance as described in Section 3.2. α defines if an value is live-out of the loop. The α value shows, as for ω, at which iteration the value has been produced. First, all loop invariant TNs are register allocated, thus reducing the available register set. Then we allocate the loop-variant TNs. The lifetimes are sorted by start and end cycle. Then an interference matrix of the lifetimes is calculated. Every lifetime is initialized with the remaining free register set. Lifetimes are allocated by picking the first free register in the remaining register set. Then the register is removed from all interfering lifetime's register set. This process is continued until all TNs are register allocated (unless we run out of registers). Registers are chosen by the "First Fit" approach as described in [18], but with the extension that caller-save registers are used first, and only if necessary callee-save registers. This register allocator is only applicable to software-pipelined kernels and is independent of the code generator's normal register allocators, which are the local register allocator (LRA) and the global register allocator (GRA).

Minor changes in LRA and GRA are needed to support software pipelining. In particular, LRA and GRA must preserve the register assignment made by the software pipeliner. In the case of LRA, LRA is simply not run for the software-pipelined basic blocks. For GRA, the problem is harder. In the past, a non-open-sourced version of software pipelining used regions to delimit basic blocks which are register-allocated by the software pipeliner. GRA would allocate all basic blocks except those in the region. "Glue copies" are inserted at region boundaries to reconcile register assignment differences across the boundary. These copies are normal register copies that have partial register assignment. It is up to GRA to make the copy redundant by allocating the same register to both sides. Redundant copies are removed afterwards by the extended block optimizer (EBO). If GRA cannot allocate the same register to both sides, the glue copy becomes a real copy.

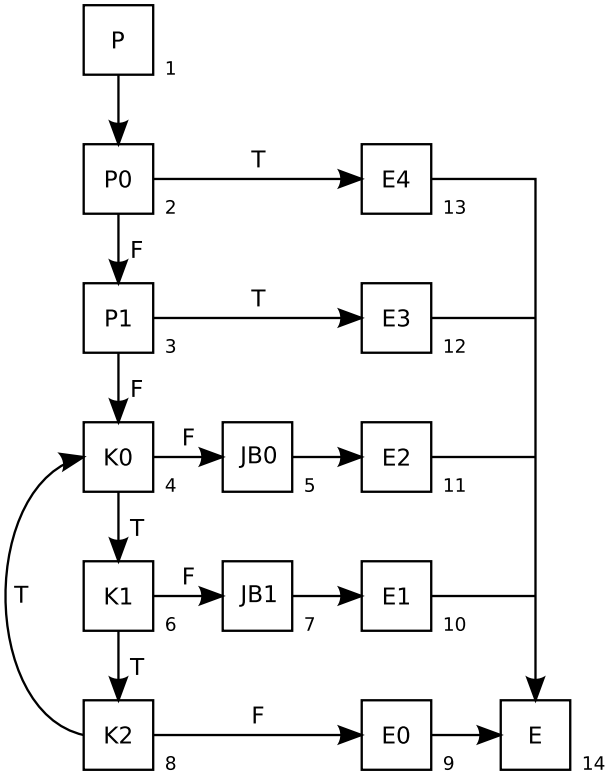
In the current implementation, instead of using regions, GRA is modified to handle partial register allocation made by earlier phases such as software pipelining. In a partial allocation, some but not all variables have assigned registers. When GRA runs, it builds live-ranges for all global variables as usual. In the coloring step, GRA detects those live-ranges that have assigned registers and prioritizes them first, so that when they are colored, GRA can always color them with their assigned register. In addition, for local variables (those spanning one basic block) with assigned registers, GRA removes their registers from the allocation set for that basic block, thus preserving the allocation to these local variables. With these modification, GRA can handle partial allocations with minimal changes to the GRA algorithm.

3.7 Code Generator

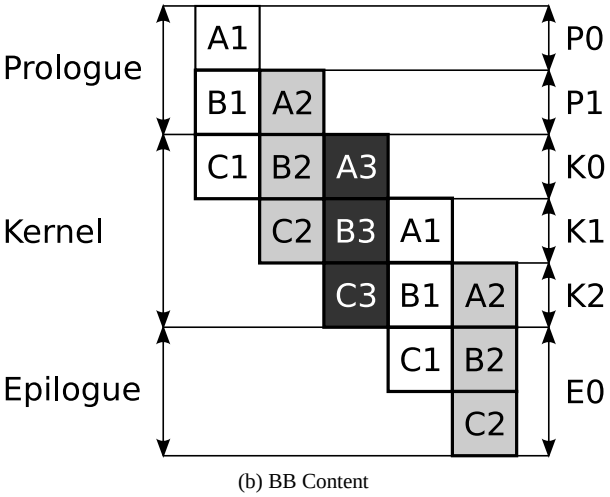
The code generator performs the final step in the software pipelining framework. There are several ways how the final code can be generated. A list of several code generation schemas can be found here [19]. Our approach is based on the paper mentioned with one additional optimization for architectures without branch prediction. We explain our approach on the example in Figure 10a. The figure shows the control flow graph (CFG) of the several BBs, which need to be generated. Prologues, kernel and epilogues are marked with a P, K and E, respectively. Jump blocks are marked with JB. Arrows show the control flow between the BBs. T or F on the arrows indicates if the branch is taken (T) or if it is a fall-through (F). The small number next to the BB indicates the actual order of the BBs in the assembly file. Lets assume we have a kernel with 3 stages and the kernel needs to be unrolled 3 times for MVE. Furthermore, we calculate that we need 2 prologue stages and 2 epilogue stages.

The first BB (P) contains copies for loop-invariant variables and falls-through to the BB (P0), which starts the first loop iteration. If the loop has just one iteration, then we jump to P0's designated epilogue BB (E4). Otherwise we continue with P2. P2 starts the second loop iteration. P2 also has its designated epilogue E3. The BBs K0, K1 and K2 represent the unrolled kernel. Each BB uses a different register set. The targeted architecture uses static branch prediction, which assumes that every branch is taken. That is the reason why the branches between the kernel BBs are for-

[2] TNs are the internal representation of the compiler for any type of operand of an instruction



(a) Control Flow Graph



(b) BB Content

Figure 10: Code Generation Schema

ward jumps instead of fall-throughs. Every kernel BB follows a jump block (except the last one). If the loop is finished, then the kernel BBs K0 and K1 fall-through to their jump blocks, which just contain an unconditional branch to the kernels designated epilogue BB. Every kernel BB needs its own epilogue, because every kernel uses a different register set. Instead of the jump blocks we could have inserted the epilogues itself, but for cache performance reasons we introduced the jump blocks.

Figure 10b shows the content of the BBs in more detail. A, B and C represent the 3 different stages of the kernel. The number indicates which register set is used.

4. Experiments

4.1 Testbed

We tested our framework on the SiCortex Multiprocessor system [22]. We used a small development system, which is not available on the market, but the SiCortex Multiprocessor, which is the integral part of the system, is the same. The SiCortex Multiprocessor was designed by SiCortex from the silicon up and is based on the MIPS 5KF IP core. Each chip has six cores. The MIPS64 5KF Core's register file is comprised of 32 64-bit integer registers, 32 64-bit floating-point registers, and several special purpose and control registers. The L1 Data - and L1 Instruction cache have been configured for 32 KB each, with 4-way set associativity and 32 byte cache lines. Each core is directly connected to a 256 KB L2 unified shared cache segment - totaling to 1.5 MB L2 shared cache for the whole chip. The L2 cache is 2-way set associative and has a line size of 64 byte. A central cache switch keeps the L2 cache segments coherent and provides access to the memory system, I/O system and the DMA engine. The core is compatible with the MIPS ABI's n32 and n64 and the MIPS V ISA. The MIPS64 5K manual [17] describes the instruction set, conventions and ABI, which are used in this paper. The timing of certain floating-point instructions differ from the one described in the manual, due to enhancements of the floating-point unit by SiCortex. The timing of the integer instructions has not changed. The MIPS core is an in-order, limited dual-issue processor. It can simultaneously issue one integer instruction and one arithmetic floating point instruction, where as any kind of memory operation can be seen as an integer instruction, because memory operations are handled by the execution unit. The first version of the chip was running at the 500 MHz. Later versions of the chip are running at 700MHz.

4.2 Results

The experimental results show that even though the framework has a maximum improvement of 15% for the 32bit CG benchmark, it does also have some performance degradation for other benchmarks. However, this degradation is not significant. In general, the improvement from SWP depends on how much time each program spends in loops, and what

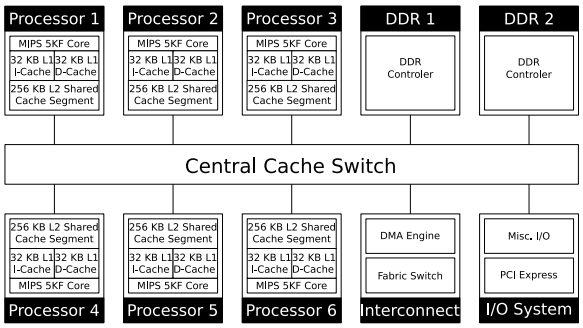


Figure 11: SiCortex System-on-Chip Multiprocessor

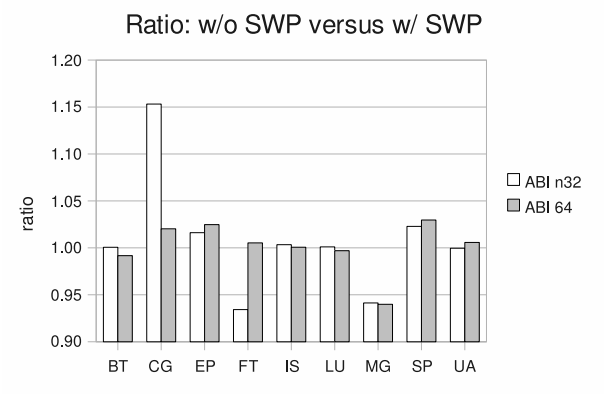


Figure 12: NAS Parallel Benchmark

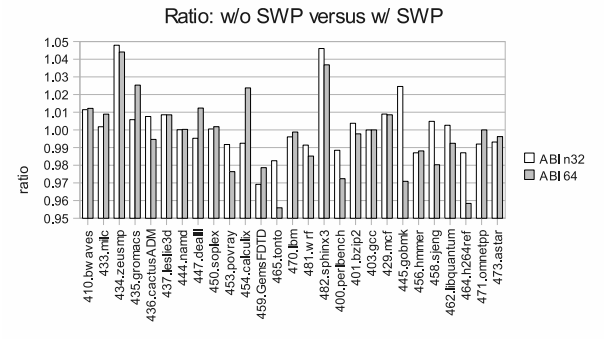


Figure 13: SPEC 2006

percentage of those loops can be software-pipelined. Even among software-pipelined loops, small loops tend to exhibit larger percentage improvement than large loops. Loops with a small number of iteration are not screened out during runtime and exhibit therefore the overhead of the prologue and epilogue. There are three main reasons why the results for the other benchmarks are not better.

First, some of the main loops in these applications have calls to mathematical operations, such as logarithms, exponentiation, etc, which are not inlined by the Inter-Procedural Optimizer (IPO). This results in loops which have multiple basic blocks for which the software pipeliner has no support. Normally, such functions are intrinsics or macros in different architectures and libraries. In these cases, the actual operation code is replaced by a short sequence of ISA instructions or with the body of the operation itself (in the case of macros or inline functions).

Second, some of the kernels in these applications have a very high register usage which in turn increases register pressure. Since register spilling has not been implemented yet for the software pipeliner, this limits the number of loops which can be successfully software-pipelined. Hence, many optimizing opportunities are lost due to the high register pressure being prevalent in the bigger kernels.

Third, the target architecture supports only one outstanding L1 cache miss. All other following loads or stores must hit in L1 cache, otherwise the processor will stall until the data of the first cache miss has been transfered to L1 cache. The current target description does not model this behavior correctly, resulting in a non-optimal schedule with unexpected stalls. Furthermore, prefetch instructions, which should prevent or reduce L1 cache misses, are ignored by the target architecture if there is already an outstanding L1 cache miss. Due to this, wrong latencies and resource requirements are passed on to the software pipeliner, preventing it from generating a better schedule.

Even with these limitations, the current implementation still delivers some marginal performance improvement without degrading the overall performance picture. The maximum performance gain was seen to be 15 percent. On the other hand, the maximum performance degradation was seen to be 7 percent. The software pipeliner has reached production-quality and will be released by SiCortex this April.

5. Related Work

There are several scheduling techniques under the umbrella of software pipelining. One of the best known and most researched is modulo scheduling. Optimal methods [2, 6, 23] have been researched and proposed, but their high computational complexity, due to NP-completeness, prevents their use in mainstream compilers. Nevertheless, they are an important instrument to validate heuristic based modulo scheduler. Well known heuristic methods are Iterative Modulo Scheduling (IMS) [20, 21], Slack Modulo Scheduling (Slack) [10], Swing Modulo Scheduling (SMS) [14], Hypernode Reduction Modulo Scheduler (HRMS) [15], and others [7, 23, 5, 12, 13]. A comparison of several heuristic based modulo scheduling techniques can be found here [4].

Iterative Modulo Scheduling (IMS) schedules instruction iteratively in order given by the priority function which considers the height of the instruction in the DDG. If an instruction can't be placed in the partial schedule, the algorithm backtracks, unscheduled already placed instructions

and tries a different placement of the instructions. This approach does not try to shorten lifetimes and may produce schedules with higher register pressure then other lifetime-sensitive methods.

Slack Modulo Scheduling (Slack) schedules operation also based on a priority function. The priority function considers the slack of the instruction and if it using a critical hardware resource. The slack is simply the difference of the earliest and latest starting time of a given instruction, which does change during the scheduling process. Furthermore it uses additional heuristics and a bi-directional scheduling approach to shorten register-lifetime. If an instruction can't be placed in the partial schedule, then the conflicting instructions and its successors and predecessors are removed from the schedule.

Swing Modulo scheduling (SMS) schedules a sorted list of instructions without any backtracking, making this method less computationally expensive. The instructions are sorted depending on the recurrence circuit they belong to and the RecMII which is associated with it. Additional heuristics are applied to produce a schedule with low register pressure.

Hypernode Reduction Modulo Scheduling (HRMS) uses a preordering phase which sorts the instructions before scheduling. Elementary circuits are converted during this process to hypernodes, starting with the circuit with the largest RecMII. Nodes which are converted to hypernodes are added to the scheduling list. After the preordering phase the instructions are scheduled without backtracking.

6. Conclusion and Future Work

We have laid out the foundation of the software pipelining framework and run it through our test harness to provide a robust implementation. We hope other target architectures of the Open64 compiler will benefit from having the software pipelining feature and we welcome any contributions from the Open64 community to further enhance our open-sourced SWP framework.

An important area of improvement is to increase the percentage of loops that can be software-pipelined. Many important loops could not be software-pipelined, because we were running out of registers. We hope to achieve this by adding register spilling to the software pipelining framework.

An other importand problem we need to address is the correct scheduling of cache misses. Currently, the Open64 compiler assumes that every load is a hit in L1 cache. We need to identify already outside of the software pipelining framework which loads are likely to miss and adjust their load latency and resource requirements. In this way, not only the modulo scheduler, but also the normal list scheduler can take advantage of this information to generate a better schedule. Only minor changes to the software pipelining framework will be necessary in this case.

Further steps include the generation of a preconditioning loop to filter out loops with small number of iterations, to reduce the overhead of SWP for these loops. This also enables new code generation schemas, which may reduce the code size of the generated SWP schedule.

We also plan to implement other modulo scheduling techniques and verify the results against an integer linear programming based scheduler, as it has been done before for the MIPSpro compiler [23].

References

[1] ALLEN, R., AND KENNEDY, K. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.

[2] ALTMAN, E., AND GAO, G. Optimal Modulo Scheduling Through Enumeration. *International Journal of Parallel Programming 26*, 3 (1998), 313–344.

[3] CHOI, Y., KNIES, A., GERKE, L., AND NGAI, T. The Impact of If-Conversion and Branch Prediction on Program Execution on the Intel® Itanium® Processor. In *International Symposium on Microarchitecture: Proceedings of the 34 th annual ACM/IEEE international symposium on Microarchitecture: Austin, Texas* (2001), vol. 1, pp. 182–191.

[4] CODINA, J., LLOSA, J., AND GONZÁLEZ, A. A Comparative Study of Modulo Scheduling Techniques. In *Proceedings of the 16th international conference on Supercomputing* (2002), ACM New York, NY, USA, pp. 97–106.

[5] DANI, A., RAMANAN, V., AND GOVINDARAJAN, R. Register-Sensitive Software Pipelining. In *Procs. of the Merged 12th International Parallel Processing and 9th International Symposium on Parallel and Distributed Systems, april* (1998).

[6] EICHENBERGER, A., DAVIDSON, E., AND ABRAHAM, S. Optimum Modulo Schedules for Minimum Register Requirements. In *Proceedings of the 9th international conference on Supercomputing* (1995), ACM New York, NY, USA, pp. 31–40.

[7] FEAUTRIER, P. Fine-Grain Scheduling under Resource Constraints. *LECTURE NOTES IN COMPUTER SCIENCE* (1995), 1–1.

[8] FLOYD, R. Algorithm 97: Shortest Path. *Communications of the ACM 5*, 6 (1962).

[9] HU, T. Parallel Sequencing and Assembly Line Problems. *Operations Research 9*, 6 (1961), 841–848.

[10] HUFF, R. A. Lifetime-Sensitive Modulo Scheduling. In *PLDI* (1993), ACM, pp. 258–267.

[11] INTEL CORPORATION. *Intel® Itanium® Architecture Software Developers Manual*, 2006.

[12] LAM, M. S. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *PLDI* (1988), pp. 318–328.

[13] LAM, M. S. *A Systolic Array Optimizing Compiler*. Kluwer Academic Pub, 1989.

[14] LLOSA, J., GONZALEZ, A., AYGUADE, E., AND VALERO,

M. Swing Modulo Scheduling: A Lifetime-Sensitive Approach. In *PACT*, vol. 96, pp. 20–23.

[15] LLOSA, J., VALERO, M., AYGUADÉ, E., AND GONZÁLEZ, A. Hypernode Reduction Modulo Scheduling. In *Proceedings of the 28th annual international symposium on Microarchitecture* (1995), IEEE Computer Society Press Los Alamitos, CA, USA, pp. 350–360.

[16] MAHLKE, S., LIN, D., CHEN, W., HANK, R., AND BRINGMANN, R. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proceedings of the 25th annual international symposium on Microarchitecture* (1992), IEEE Computer Society Press Los Alamitos, CA, USA, pp. 45–54.

[17] MIPS TECHNOLOGIES, INC. *MIPS64™5K™Processor Core Family Software Users Manual*. 1225 Charleston Road, Mountain View, CA 94043-1353, May 2002.

[18] RAU, B., LEE, M., TIRUMALAI, P., AND SCHLANSKER, M. Register Allocation for Software Pipelined Loops. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation* (1992), ACM New York, NY, USA, pp. 283–299.

[19] RAU, B., SCHLANSKER, M., AND TIRUMALAI, P. Code Generation Schema For Modulo Scheduled Loops. In *Microarchitecture, 1992. MICRO 25., Proceedings of the 25th Annual International Symposium on* (1992), pp. 158–169.

[20] RAU, B. R. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *MICRO* (1994), ACM/IEEE, pp. 63–74.

[21] RAU, B. R. Iterative Modulo Scheduling. Tech. rep., Hewlett Packard, November 1995.

[22] REILLY, M., STEWART, L., LEONARD, J., AND GINGOLD, D. SiCortex Technical Summary.

[23] RUTTENBERG, J., GAO, G., STOUTCHININ, A., AND LICHTENSTEIN, W. 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* (1996), ACM New York, NY, USA, pp. 1–11.

[24] TARJAN, R. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing 1* (1972), 146–160.

[25] TOMASULO, R. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development 11*, 1 (1967), 25–33.