Reduced Code Size Modulo Scheduling in the Absence of Hardware Support

Josep Llosa[1], Stefan M. Freudenberger[2]
HP Laboratories Cambridge
HPL-2002-239
August 26th , 2002*

E-mail: josepll@ac.upc.es, stefan.freudenberger@hp.com

Modulo scheduling is a very effective instruction scheduling technique that exploits Instruction Level Parallelism (ILP) in loop bodies by overlapping the execution of successive iterations. Unfortunately, modulo scheduling has been shown to cause heavy code expansion. To avoid the penalties of code expansion, some processors have dedicated hardware support for modulo scheduled loops. However, this dedicated hardware support has a cost in chip area, cycle time, processor complexity, and compiler complexity.

This paper shows that the right combination of scheduling heuristics combined with speculative modulo scheduling can significantly reduce code expansion. In addition, several code generation schema heuristics are proposed to further reduce code expansion. The evaluations show that loops can be effectively modulo scheduled with an average code expansion only 1.5 times the original loop size. Compared with a state of the art modulo scheduler, our code size sensitive heuristics reduce the size of embedded domain benchmarks binaries by 30% on average. While performance is mostly unchanged, some applications show speed-ups up to 20% due to a reduction in instruction cache capacity misses.

* Internal Accession Date Only Approved for External Publication
To be published in and presented at the 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35), 18-22 November 2002, Istanbul, Turkey
[1] Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Barcelona, Spain
[2] Hewlett-Packard Laboratories, Cambridge, MA

Reduced Code Size Modulo Scheduling in the Absence of Hardware Support

Josep Llosa[1,3] and Stefan M. Freudenberger[2]
[1]*Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Barcelona, Spain.*
[2]*Hewlett-Packard Laboratories, Cambridge, Mass.*
josepll@ac.upc.es, *stefan.freudenberger@hp.com*

Abstract

Modulo scheduling is a very effective instruction scheduling technique that exploits Instruction Level Parallelism (ILP) in loop bodies by overlapping the execution of successive iterations. Unfortunately, modulo scheduling has been shown to cause heavy code expansion. To avoid the penalties of code expansion, some processors have dedicated hardware support for modulo scheduled loops. However, this dedicated hardware support has a cost in chip area, cycle time, processor complexity, and compiler complexity.

This paper shows that the right combination of scheduling heuristics combined with speculative modulo scheduling can significantly reduce code expansion. In addition, several code generation schema heuristics are proposed to further reduce code expansion. The evaluations show that loops can be effectively modulo scheduled with an average code expansion only 1.5 times the original loop size. Compared with a state of the art modulo scheduler, our code size sensitive heuristics reduce the size of embedded domain benchmarks binaries by 30% on average. While performance is mostly unchanged, some applications show speed-ups up to 20% due to a reduction in instruction cache capacity misses.

[3]This work has been performed while Josep Llosa was a Faculty Visitor at Hewlett-Packard Laboratories, Cambridge, Mass.

1. Introduction

VLIW architectures are widely used in the design of embedded/DSP processors [7][8][20], and in the design of some general-purpose microprocessors [11]. Statically scheduled processors, and VLIWs in particular, require efficient compiler technology to extract ILP from applications. Instruction scheduling plays a critical role in ILP exploitation. Software pipelining [1] is a very effective instruction scheduling technique for loops that overlaps the execution of successive iterations. Modulo Scheduling [18] is a class of software pipelining algorithms that is very cost effective and has been implemented in several production compilers [5][16].

One of the drawbacks of modulo scheduling (and software pipelining in general) is that it incurs significant code expansion. Although this code expansion is generally smaller than unrolling the loop multiple times and scheduling it using conventional techniques, it is still an important problem. In particular, there are some application areas in the embedded/DSP arena where code size is critical. From the general-purpose processor perspective, although code size can be tolerated more easily, it nevertheless has a negative effect on instruction cache performance.

Code expansion in modulo scheduled loops is a consequence of the need to deal with two independent problems: the generation of prologues and epilogues to fill and drain the software pipeline, and the replication of the kernel, known as Modulo Variable Expansion (MVE) [12], to deal with register lifetimes that are longer than the loop Initiation Interval (II). To deal with such problems, some processors [2][11] have dedicated modulo scheduling hardware support. Rotating register files [4] allow the lifetime of a value generated in one iteration to overlap the lifetimes of corresponding values generated in previous and subsequent iterations without requiring MVE. Full predication with rotating predicates [4] permits the generation of “kernel-only” code by selectively disabling the execution of operations during prologue and epilogue execution phases.

Architectural support for modulo scheduling has several costs associated with it that, although some general-purpose processors may be willing to pay, can be prohibitive for low cost embedded/DSP processors. Both rotating registers and predicated execution add extra complexity to the processor, and require additional chip area. Rotating registers require an adder in the, usually critical, decoding path, leading to longer cycle time or longer instruction pipelines. In addition, they require special register allocation techniques, thus increasing the complexity of the compiler. Full predication requires a predicate field for each instruction in the ISA. If instruction encoding is tight, extra bits are required to encode the predicate field, having a negative effect on overall code size that might even negate the potential benefits.

There have been several proposals to address code generation for modulo scheduled loops. Modulo Schedule Buffers (MSB) [15] allow the generation of “kernel-only” code with a more relaxed predication mechanism, but they still require rotating register files plus the cost of the MSB. In [21] schemas for efficient modulo scheduling of loops with early exits are proposed using hardware support. [19] proposes several code schemas for modulo scheduled loops with and without hardware support. Finally, [13] shows how to modulo schedule loops with multiple exits.

Modulo scheduling code size expansion has been addressed from a theoretical point of view [19], with a special emphasis in code schemas for processors with dedicated hardware support. However, little additional attention has been paid to the problem other than to observe it [22].

This paper makes several significant contributions regarding code generation schemas in the absence of dedicated hardware support:

- The impact of modulo scheduling heuristics on code expansion is analyzed. The results show that making the right choice at scheduling time can significantly reduce code size requirements. In particular, bi-directional schedulers [10][13] perform very well, with the added benefit that they reduce register pressure without sacrificing performance.
- Code expansion in the absence of hardware support is analyzed using three different code schemas: non-speculative modulo scheduling, loop preconditioning and speculative modulo scheduling. We show that speculative modulo scheduling produces less code expansion. Besides, it enables while-loops (i.e., loops whose trip count is unknown at the time the loop is entered) to be modulo scheduled.
- Two new code generation schemas are proposed to further reduce the overall size of epilogues, by collapsing several epilogues into one.
- Finally, we propose the insertion of copy instructions in unused slots to split the longest lifetimes. By splitting lifetimes that are longer than II into several segments, we are effectively rotating these lifetimes using spare processor resources. This has the benefit that MVE is not required, or that it is required to a lesser degree, reducing the code size requirements for the loop kernel.

The combination of these techniques leads to modulo schedules with very small code expansion factors (ranging from 1.5 to 2 on average, depending on the processor configuration and benchmark set). This leads to an average code size reduction of 30% for benchmarks in the embedded domain. On average, the performance gain is small (0 – 5%), but it can range in some favorable cases up to 20%. However, the main contribution of the paper is that modulo scheduling is not necessarily expensive in terms of code size, and that it can be effectively performed without requiring special hardware support.

The rest of the paper is organized as follows. Section 2 introduces basic modulo scheduling concepts; Section 3 explains our experimental framework; Section 4 discusses the impact of modulo scheduling heuristics on code size; Section 5 looks at the impact of loop preconditioning and speculative modulo scheduling; Section 6 explains our code size reduction schemas; Section 7 presents a detailed evaluation; and Section 8 presents conclusions.

2. Basic concepts

2.1. Modulo scheduling

In a modulo scheduled loop, the schedule of an iteration is divided into stages so that the execution of consecutive iterations overlaps. The number of stages in one iteration is called Stage Count (SC). The number of cycles between the initiation of successive iterations determines the execution rate and is called the Initiation Interval (II). The II is bound either by recurrence circuits in the dependence graph of the loop (RecMII) or by resource constraints of the target architecture (ResMII). The lower bound on II is called the Minimum Initiation Interval (MII = max(RecMII, ResMII)).

The execution of a loop can be divided into three phases: a ramp-up phase that fills the software pipeline, a steady-state phase where maximum overlap of iterations is achieved and a ramp-down phase that drains the software pipeline. During the steady-state phase, the same pattern of operations is executed at each stage. This is achieved by iterating on a piece of code, called the kernel.

The scheduling step of a modulo scheduler builds the schedule progressively by adding instructions to a partial schedule. Sometimes the schedule reaches a partial schedule in which the remaining instructions cannot be placed. In this case, there are two alternative solutions: increase II [12][13], or apply backtracking [10][17]. Backtracking involves un-scheduling some operations in the partial schedule in order to make room for other operations, and then schedule them again later. Techniques that use backtracking are also called iterative techniques. In order to limit the scheduling time, the amount of backtracking is limited by the Budget Ratio, which determines how many times an instruction can be rescheduled before increasing II.

2.2. Basic code generation schema

When no hardware support is available, the ramp-up and ramp-down phases of the modulo schedule must be implemented using two pieces of code named prologue and epilogue, respectively. In that case SC – 1 iterations are executed by the prologue and epilogue, therefore the trip count of the kernel is N – SC + 1, where N is the number of iterations in the original loop. In addition, if the branch of the loop is not scheduled in the last stage, there may be potential exit points in the prologue, each requiring an epilogue to finish the iterations that have been initiated up to this point. In particular, SC – BS – 1 epilogues are required for early prologue exits, where BS is the stage where the exit branch is scheduled.

An additional problem that must be solved is the presence of loop variants with a lifetime longer than II. In this case, a new value of the loop variant is generated before the value generated in the previous iteration is consumed. One approach to fix this problem is by renaming the register using rotating register files [4]. In the absence of such hardware support, Modulo Variable Expansion (MVE) [12] can solve this problem. When MVE is applied, the kernel is unrolled and registers are renamed to prevent that successive lifetimes corresponding to the same original loop-variant overlap in time. The minimum degree of unroll, K_{min}, is determined by the longest lifetime, as:

$$K_{\min} = \left\lceil \frac{LifetimeLength}{II} \right\rceil$$

Figure 1 shows the code schema to be generated for a loop with SC = 4, BS = 1, and K_{min} = 2. This schema can be simplified [19]; however, we choose not to do so to improve readability.

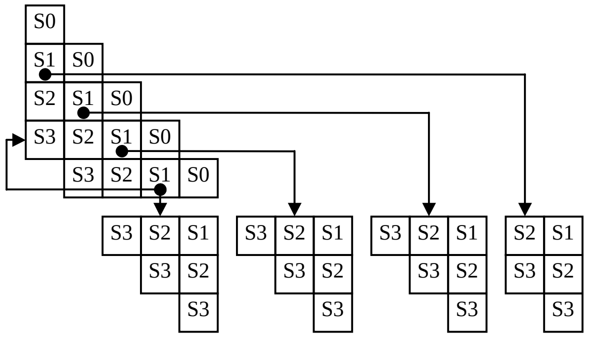

Figure 1: Basic Code Generation Schema

3. Experimental framework

3.1. Architecture

We have implemented and evaluated our heuristics in the compiler for the Lx architecture [8][9]. Lx is a scalable and customizable VLIW processor technology platform designed by Hewlett-Packard and STMicroelectronics that allows variations in instruction issue width, the number and capabilities of structures, and the processor instruction set. A first implementation within this architecture platform is the ST210, a 250-MHz VLIW developed by STMicroelectronics.

This architecture consists of a single-cluster 4-issue VLIW core composed of four 32-bit integer ALUs, two 16x32-bit multipliers, one Load/Store Unit, and one Branch Unit. The cluster also includes 64 32-bit General-purpose registers and 8 1-bit branch registers (used to store branch conditions, predicates, and carries). Instructions allow two 32-bit long immediates per cycle (which use up one issue slot). The ISA is a very simple integer RISC instruction set with minimal "predication" support through a *select* instruction. The memory repertoire includes *base+displacement* addressing, and allows for speculative execution of *dismissible loads*, which are handled by the protection unit. The data cache is a 32KB, 4-way associative, write-back array with load/store allocation. The instruction cache is 32KB direct mapped. There are no L2 caches. The control unit decouples the compare and branch operations and the compare-to-branch latency is exposed to the compiler. There are no architecturally visible delay slots after a taken branch; however, a mispredicted (taken) branch incurs a 1-cycle penalty.

Most integer operations have unit latency; multiplies and 32-bit load operations have a latency of two cycles, as have compares that are read by a branch.

3.2. Compiler platform

The Lx compiler is a newly developed state-of-the-art commercial-quality compiler that finds ILP in the embedded space for a wide range of instructions set architectures and that is capable of supporting the anticipated Lx platforms; at the same time, this compiler is flexible enough to be used in compiler-research. It is designed to deliver the highest-possible ILP in the presence of scaling (variations in the number of available resources of a given kind) and customization (variations in the instruction set architecture). It is also targeted to clustered architectures with heterogeneous clusters and partitioned register files with limited inter-cluster connectivity and operations with architecturally visible non-unit latencies (without hardware to check for latency violations). Each of these degrees of freedom forces a separate level of complexity on the compiler's design.

3.3. Benchmarks

We evaluated our heuristics using two sets of benchmarks. The first set comprises a set of programs that are representative for the Lx target domain and that includes audio manipulation, printing pipelines, color processing, cryptography, video, and still image compression and decompression. This set, which we call the BenchSuite, includes the following benchmarks:

adcmp	ADCMP audio encoder/decoder
bmark	Printing imaging pipeline
copymark	Color copier pipeline
crypto	Cryptography code (ECC, RSA and DES)
csc	Color-space conversion
mp2audio	MPEG-1 Layer 2 encoder
mp2vloop	MPEG-2 video loop (subset of MPEG-2 standard w/o IDCT or reconstruction)
mp2avswitch	MPEG-2 System Layer de-multiplexor
mpeg2	MPEG-2 decoder
mp4dec	MPEG-4 decoder
opendivx	Public MPEG-4 decoder
tjpeg	JPEG-like encoder/decoder

The second set consists of 9 of the 12 SPEC2000 integer benchmarks: we omitted 186.crafty because it uses 64-bit integer arithmetic (which our compiler currently does not support); 252.eon because it is a C++ benchmark (and we currently don't have a C++ compiler available); and 254.gap because it uses *ioctl* system call features that our simulation environment does not support. We have included the SpecInt2000 benchmarks because they are widely known. However, these benchmarks represent atypical workloads for which the Lx family has not been optimized.

These two sets represent quite different workloads: the first set spends most of its execution time in big loops with a large amount of ILP (on average, there are 45.85 operations/loop in this set), while SpecInt2000 has many small loops with a limited amount of ILP (on average, there are only 10.72 operations/loop). In total, these benchmarks contain 2014 loops that our compiler can modulo schedule, 500 in BenchSuite and 1514 in SpecInt2000.

4. The impact of modulo scheduling heuristics on code size

4.1. The quality of the schedules

In a study like this one, special care must be given to the fact that code size expansion is in part influenced by the amount of overlapping achieved by the scheduler. In general, a scheduler that produces worse schedules (in terms of II) might require less code expansion. Therefore, it is very important to use a state-of-the-art modulo scheduler in order not to underestimate the code size expansion. In a recent study [3], Iterative Modulo Scheduling (IMS) [17] and Swing Modulo Scheduling (SMS) [13] have been shown to be at the head of the most advanced modulo schedulers from the point of view of minimizing II. We have implemented both IMS and SMS in our compiler, and compared them to the Lx compiler's modulo scheduler (LxMS), in order to show that our results are not distorted by a low-quality scheduler.

Table 1 shows a comparison of the MII/II ratio for the three modulo schedulers. Both IMS and LxMS use a Budget Ratio of 3, which we found to be a good performance/compile-time trade-off. Although it is out of the scope of this paper to perform a modulo scheduler comparison, we can make a few interesting observations.

Table 1: Average II/MII ratio

Scheduler	BenchSuite	SpecInt2000
SMS	1.059	1.028
IMS	1.054	1.026
LxMS	1.034	1.014

Both SMS and IMS obtain a worse II ratio than what is claimed in their corresponding papers and in [3]. A careful examination of some dependence graphs, and the access to the SMS authors' implementation, shows that our C benchmarks have more recurrences, and therefore are more difficult to schedule than the Fortran benchmarks used in both papers. This is due to the increased complexity of disambiguating memory references in C programs as compared to Fortran programs.

SMS is a non-iterative modulo scheduler that gives special priority to recurrence circuits. However, SMS behaves slightly worse than the rest of the schedulers because of the particular resource usage of instructions in the Lx processor. SMS tends to greedily fill the four issue slots of some cycles with arithmetic instructions. If the loop is limited by memory operations, it needs to increase II to schedule the loads and stores. In addition, it failed to schedule some of the loops because of a rare pathologic situation with certain recurrence combinations that never arises in the authors' benchmarks.

IMS is an iterative modulo scheduler. This feature allows it to deal very effectively with the resource problem stated above. Backtracking also deals quite effectively with recurrence circuits, but not as well as a dedicated priority heuristic for recurrences.

LxMS is an iterative modulo scheduler (like IMS) that gives special priority to recurrence circuits (like SMS). In addition, it gives a higher priority to the critical path than the other two schedulers do. This results in excellent schedules in the presence of either complex resource usage patterns and/or complex recurrence circuits.

4.2. Code size sensitive scheduling heuristics

For a given II, there are many different (usually infinite) schedules corresponding to that II. Each one of these different schedules can produce a different code expansion. In this section, we analyze which factors contribute to code size, and which heuristics can help to minimize code size in a modulo scheduler.

Table 2: Loop expansion ratio

Scheduler	BenchSuite	SpecInt2000
SMS	3.14	4.22
IMS	4.22	3.39
LxMS	2.48	2.67

Table 2 shows the loop expansion ratio for the three modulo schedulers described in the previous section. We define the code size expansion ratio as:

$$ExpansionRatio = \frac{\sum instrs\_in\_ms\_loops\_before\_ms}{\sum instrs\_in\_ms\_loops\_after\_ms}$$

In Figure 1, it can be seen that there are three factors contributing to code expansion. The number of stages determines the size of the prologue and the size of epilogues. A reduced number of stages will have a smaller prologue and smaller epilogues. The stage in which the loop branch instruction is scheduled partially determines the number of epilogues, since each early exit in the prologue requires a partial epilogue. The later the branch is scheduled the fewer epilogues are required. Finally, the K_{min} degree of MVE determines how many copies of the kernel are required. A smaller K_{min} will require less code expansion due to the kernel. In addition, by reducing the number of kernel replications we are also reducing the number of epilogues.

Table 3 shows the average K_{min}, the average Stage Count, and the average number of early exits for the three schedulers considered. LxMS, despite producing schedules with smaller IIs, results in the smallest loop size expansion because it succeeds in minimizing all three factors, while SMS results in an intermediate loop size expansion because it fails to reduce the stage count.[1]

[1] SMS performs worse on SpecInt2000 due to one pathological case (see comment on early/late start limits in Stage Count section) ; if this one case were removed, the code size would also be between the code size for IMS and LxMS.

Table 3: Average K_{min}, average Stage Count, and average number of Early Exits

Scheduler	BenchSuite			SpecInt2000		
	K_{min}	SC	Exits	K_{min}	SC	Exits
SMS	1.57	2.33	0.17	1.44	2.08	0.04
IMS	1.91	2.23	0.91	1.71	1.97	0.36
LxMS	1.54	1.97	0.20	1.44	1.85	0.07

Next, we outline the characteristics that modulo scheduling heuristics must have in order to improve each one of these factors. The above modulo schedulers are used as practical examples.

K_{min}

A good strategy to minimize K_{min} is to minimize the length of the lifetimes. Both SMS and LxMS reduce the length of the lifetimes by scheduling operations as close as possible to their predecessors and successors. To do so, both schedulers use a bi-directional strategy in which nodes are scheduled as early as possible if they have predecessors in the partial schedule and as late as possible if they have successors in the partial schedule. In both schedulers node priority is computed using an adjacency ordering so that the situation in which a node has both predecessors and successors in the partial schedule is minimized. This heuristic not only helps to reduce code size, but also to reduce register pressure. Although register requirements are outside the scope of this paper, they are an important factor to reduce in order to improve the effectiveness of modulo scheduling.

Early Exits

The number of early exits is determined by how late the branch instruction is scheduled. In the case of architectures with low-overhead loop instructions in which a single instruction decrements the loop counter, compares it to zero, and jumps back to the loop if necessary, it is easy to reserve a slot for the branch and allocate it as late as possible.

The Lx ISA, as many other processors, has no special hardware support for loops. Therefore, three instructions are required to modify the induction variable, compare, and branch. In addition, in the case of loops with conditional exits, which are also modulo scheduled by our compiler, the expression required for computing the branch condition might be very complex and depend on other computations in the loop. In that case, the best alternative is to schedule the branch as any other operation in the loop.

IMS has a high number of exits because it greedily schedules operations top-down, which tends to schedule the branch too early. On the other hand, both SMS and LxMS schedule operations as close as possible to successors/predecessors. Since the branch instruction usually has a control dependence with some other operations in the loop, both schedulers tend to schedule it as late as possible, minimizing the number of early exits, and therefore the number of epilogues.

Stage Count

In order to minimize the stage count, the schedule length must be minimized. A good heuristic to minimize the schedule length consists of giving priority to the critical path of the loop.

IMS uses height as priority, which, to some extent, gives more priority to the nodes on the critical path. However, if two nodes have the same height, there is no provision for selecting one over the other. This leads to the possibility that nodes belonging to a non-critical path are scheduled with higher priority. In addition, nodes on non-critical paths with greater height than nodes on the critical path are always assigned a higher priority.

SMS also uses height/depth (depending if the node is ordered in a top-down/bottom-up pass) as priority. In addition, it breaks ties by considering the mobility of the node (i.e., the criticality of the path the node belongs to) as a secondary heuristic. However, nodes with greater height/depth are always assigned a higher priority independently of their criticality.

LxMS uses the criticality of a node as its primary heuristic, and height/depth as a secondary heuristic. The only exceptions are the sub-paths that have both predecessors and successors on the critical path, which are scheduled right before the predecessor/successor (again depending if the nodes are ordered in a top-down/bottom-up pass).

As expected, LxMS has, on average, a smaller Stage Count than the other two schedulers, because of the higher priority to operations on critical paths. Surprisingly, and contrary to what we expected, SMS has a slightly higher Stage Count than IMS. This occurs because critical path priority is not the only factor contributing to schedule length.

Another factor is what we call “Early/Late Start Limit”. In some cases, the dependences allow a node to be scheduled in an earlier cycle than its predecessors. Assume, for instance, two nodes A and B connected by a dependence of distance 1, latency 2 and II = 4. If A is scheduled at cycle zero, B can potentially be scheduled as early as $0 + 2 - 1*4 = -2$. This limit determines how early in a top-down pass (or late in a bottom-up pass) a node can be scheduled (assuming, of course, that other dependences allow this).

Since IMS is a unidirectional top-down scheduler, the Early Start Limit is cycle zero (i.e., no node can be scheduled earlier than this cycle). Since no node will be bottom-up scheduled there is no Late Start Limit.

However, with bi-directional schedulers we need to set both limits: if they are too close together, the scheduler will fail because it cannot accommodate the critical path between the two limits; if they are too far apart, the schedule will be longer than necessary.

Since SMS has no Early/Late Start Limits, a node can be scheduled as early/late as its dependences allow it. This has the benefit that if the corresponding dependence is a register flow dependence, register pressure is minimized. Unfortunately, this is done at the expense of schedule length and code size. We have observed that this has a very small impact on register pressure, but, as can be seen in Table 3, the effect on schedule length is quite significant.

LxMS uses flexible Early/Late Start Limits. Initially both limits are set to zero. When a node is scheduled in a top-down pass, it is scheduled as soon as possible, but never earlier than the Early Start Limit, so that we never generate a schedule longer than necessary. The Late Start Limit is

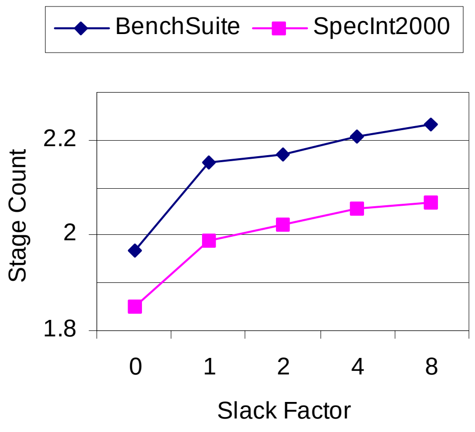

Figure 2: Effect of Slack Factor

ignored for nodes scheduled top-down, so that the limits do not constrain the schedule. In addition, whenever a node is scheduled later than the Late Start Limit, the limit is updated to the cycle the node is scheduled, i.e., the distance between the limits is always the partial schedule length. The symmetric mechanism works for bottom-up scheduled nodes.

There is an additional option in the Lx compiler that allows nodes like B in our example to be scheduled at most "slack" cycles before/after the early/late start limit. Figure 2 shows the effect on the stage count as we increase the slack. Although not shown, increasing the slack also produces a very small reduction in both K_{min} and Early Exits, which for some benchmarks compensates the extra stage count, and reduces code size. However, as the slack is increased, code size keeps growing, and the result is so unpredictable that the default for the compiler is set to zero.

5. Loop preconditioning and speculative modulo scheduling

Two other code generation schemas have been proposed in [19] in order to reduce the epilogue code size. However, to the best of our knowledge they have not been evaluated in a production environment.

Preconditioning the loop can eliminate the multiple epilogues in Figure 1. A non-software pipelined version of the loop is first executed until the number of remaining iterations for the loop kernel is a multiple of K_{min}. In that way only one epilogue is required at the exit of the kernel. Likewise, the preconditioning loop can also be used when the trip count of the loop is smaller than SC, so that the early exit epilogues can also be removed. Figure 3 shows the code schema to be generated for our hypothetic loop. However, notice that in terms of code size the preconditioning loop contributes a full copy of the original loop body that must also be considered. The main drawback of this schema is that the first few iterations executed by the preconditioning loop are executed at a lower efficiency rate, reducing the overall performance.

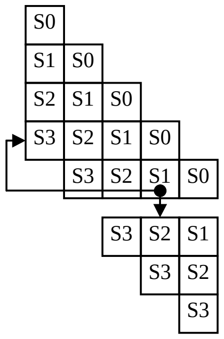

Figure 3: Preconditioned Code Generation Schema

The code schemas in Figure 1 and Figure 3 require the loop trip count to be known before entering the loop. This is acceptable for numeric applications; however, for general applications it is interesting to be able to modulo schedule loops with conditional exits. This can be achieved if speculative modulo scheduling is implemented. In that case iterations are started speculatively before knowing that they must be executed. Once the exit branch is executed, the iterations started speculatively are not completed in the epilogue, since they should never be executed. As can be seen in Figure 4, this leads to a code schema where the epilogues have reduced size, since only the non-speculative iterations must be completed.

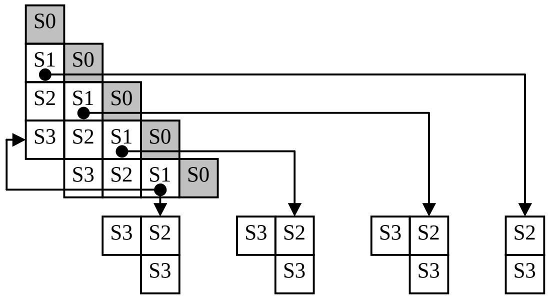

Figure 4: Speculative Code Generation Schema

Figure 5 compares the loop expansion ratio of the basic code schema of Figure 1 (Basic) with loop preconditioning (Prec) and speculative modulo scheduling (Spec). We observed that for some loops, the code size contribution of the preconditioning loop was larger than the size of the epilogues it removed. We also compare the loop expansion ratio of applying loop preconditioning only when it will help to reduce code size (Prec++).

For the default Lx scheduler (LxMS), speculation is consistently superior to all other code schemas in terms of code size. This is because, as mentioned in the previous section, LxMS tends to schedule the branch as late as possible in the schedule. In other words, LxMS performs aggressive speculation minimizing the size of the epilogues.

When IMS is used, speculation has a minimal effect on code size. As previously mentioned, IMS is a greedy top-down scheduler that tends to schedule all operations as early as possible, minimizing the amount of speculation, and maximizing the size of the epilogues. A curious result is that speculation with IMS works much better for SpecInt2000 than for BenchSuite. Recall that, on average, the loops in SpecInt2000 are much smaller than those in BenchSuite. The minimum control sequence for a loop requires three instructions, increment, compare, and branch, which have a minimum latency of 4 cycles. The fact is that many loops in SpecInt2000 have an II of less than 4 cycles, thus, even a greedy top-down scheduler schedules the branch in a relatively late stage of the schedule. However, with the BenchSuite, 4 cycles is negligible for the big loops (which incidentally dominate the code expansion ratio) with IIs in the order of tens and even hundreds of cycles, and IMS places the branch in the early stages of the schedule.

6. Code size reduction schemas

In this section, we propose several code generation heuristics to further reduce loop code expansion in modulo scheduled loops. These heuristics can be applied to any modulo scheduling technique, and can be combined with the previous code generation schemas (non-speculative, speculative, or preconditioned).

6.1. Collapse kernel epilogues

Notice that in Figure 1 and Figure 3 the epilogues corresponding to the kernel and the last prologue exits are identical. They only differ in the registers they read. It is relatively easy to make the compiler assign the same registers in the prologue and in the different kernel

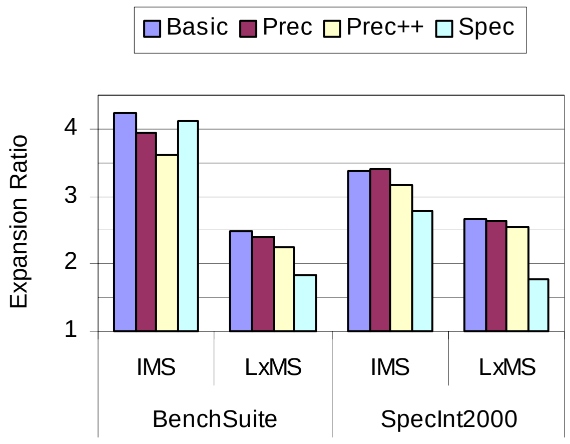

Figure 5: Code size expansion for Basic, Preconditioning, and Speculative Modulo Scheduling

replications. In that way, all epilogues will be identical and could be collapsed together into a single epilogue. However, the long lifetimes that span for more than II cycles cannot be assigned to the same register. Note that this is the reason for MVE in the first place.

One way of overcoming the long lifetimes renaming problem is to insert basic blocks between loop exits and the corresponding epilogues and to explicitly rename long lifetimes with copy instructions in these blocks. This permits to collapse all kernel epilogues into a single epilogue copy, leading to the code schema shown in Figure 6. This schema has the advantage of reducing the code size of the epilogues, but it also increases the code size by the new copy instructions. However, in the case of LxMS (and any code size sensitive modulo scheduler in general), very few copy instructions are required, since the scheduler already tries to minimize the length of the lifetimes.

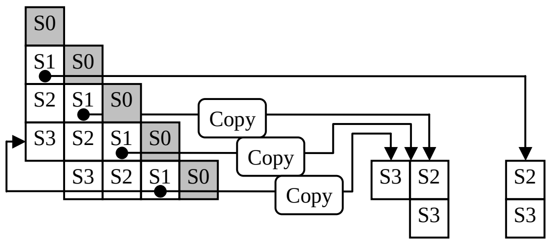

Figure 6: Collapse Kernel Epilogues

The inserted copy instructions require some extra time to execute, which can be a performance problem in short trip count loops that are executed many times. Nevertheless, all copy instructions in a copy basic block are independent and can be executed in parallel. The Lx processor can execute up to four copy instructions per cycle, and we almost never require more than four copies per basic block.

Finally, an additional problem with the Lx processor is that all taken branches have a 1-cycle penalty. That means that every copy block will execute for at least two cycles. Notice however, that this penalty is already present at the end of each of the epilogues. Therefore, in practice, we are only moving this penalty to the copy block.

The above performance problems are in part compensated by the fact that, if there are no early exits in the prologue, there is a single exit epilogue that offers an opportunity to the scalar scheduler of merging it with the basic block following the loop, generating a better combined schedule.

6.2. Collapse prologue epilogues

In Figure 6, we can see that the partial prologue-epilogue actually corresponds to a subset of the full kernel epilogue. Notice that this is also true for the non-speculative case of Figure 1. In general, any partial epilogue corresponding to an early exit is a subset of the next partial epilogue. The code generated for one partial epilogue can be reused by the next epilogue if the completion of iterations is done in a sequential way, instead of a parallel way. Of course, the long lifetimes are still a problem and copy blocks may be required for the early exits. Figure 7 shows the corresponding schema for our example.

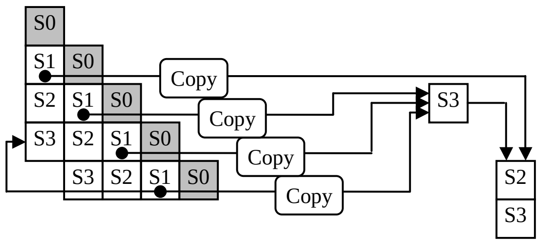

Figure 7: Collapse Prologue Epilogues

Notice that it is not always necessary to generate a copy block for the partial epilogues. For instance, if in our example the lifetime that caused MVE starts in stage S2 and ends in stage S3, no copy will be required for the first exit. Also notice that only $K_{min} - 1$ out of every K_{min} exits actually require a copy.

Collapsing the prologue-epilogues has the same code size and performance drawbacks as collapsing the kernel epilogues. In addition, extra performance penalty might be incurred by the fact that the epilogue is partially serialized, since no overlapping of iterations is exploited. However, this code schema has a single exit point, which allows the last portion of the epilogue to be scheduled jointly with the basic block following the loop.

6.3. Lifetime splitting to reduce MVE

Another source of code expansion are the multiple kernel copies required to deal with long lifetimes. Reducing MVE has the additional benefit that it also reduces the number of epilogues corresponding to kernel exits.

One way of reducing the amount of MVE is to split the longest lifetimes into multiple shorter segments by using copy instructions. This reduces K_{min} by explicitly renaming the lifetimes into multiple registers. In some sense, it behaves like rotating register files, where the registers are "rotated" by copy instructions.

Unfortunately, the extra copy instructions require resources to be executed, and add some latency to the path where they are inserted. While it can be acceptable to pay some extra cycles in the loop exit, we are not willing to pay extra cycles in the kernel to reduce code size. The solution we have adopted consists of inserting copy instructions in the kernel in a post-schedule pass using only existing free issue slots.

Using only free slots limits the number of lifetimes that can be split. However, notice that if a bi-directional scheduler that tries to schedule nodes as close as possible like LxMS is used, the number of lifetimes longer than II is significantly reduced. We have also observed that for a big K_{min}, there is usually a single lifetime causing it. For instance, in a loop with $K_{min} = 4$, it is quite frequent to have a single lifetime of length 4*II and several more of length 2*II. In this case, a single copy to split the longest one can halve the size of the kernel.

Since we only use free issue slots, lifetime splitting has no negative effects on performance. However, since we incur a 1-cycle penalty each time we traverse the loop back edge, unrolling the loop fewer times causes us to incur this penalty more often. Many embedded and general-purpose processors have some kind of branch prediction mechanism that can remove this penalty.

6.4. Evaluation of code size reduction schemas

Figure 8 shows the loop size expansion ratio with the previous optimizations applied to LxMS with a speculative code schema (Spec). The figure shows the contribution of each of the proposed techniques: collapse kernel-epilogues (+ek), collapse prologue-epilogues (+ep), which includes also +ek and MVE reduction (+mve) as well as their combined effect (+mve+ek and +mve+ep).

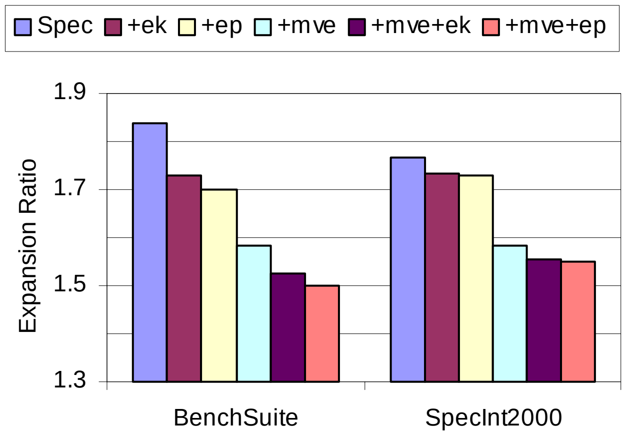

Figure 8: Code size expansion for various LxMS code generation schemas

MVE reduction using copy instructions results in the highest code size reduction, since it reduces both kernel and epilogue replications. The next most important factor is the reduction of the kernel-epilogues, since they are the most abundant. Collapsing prologue-epilogues results in the smallest code size reduction. This small contribution is not related to effectiveness of the technique, but to the fact that LxMS generates very few loops with partial epilogues. However, for the few loops where this happens, it is a very effective technique. When collapsing epilogues is combined with MVE reduction, the contribution is smaller because MVE reduction is very effective in removing replicated epilogues.

Finally, notice that, although the initial code expansion of SpecInt2000 loops is smaller than that of the BenchSuite, code expansion is reduced more for BenchSuite. This is because SpecInt2000 loops are much smaller, and the proposed techniques are more effective with larger loops (which, by the way, is what really worries developers). For example, a loop with II=1 requires at least MVE=3 to accommodate the lifetime of the compare instruction, and there is no way of splitting it with a copy, since this is already the minimum lifetime.

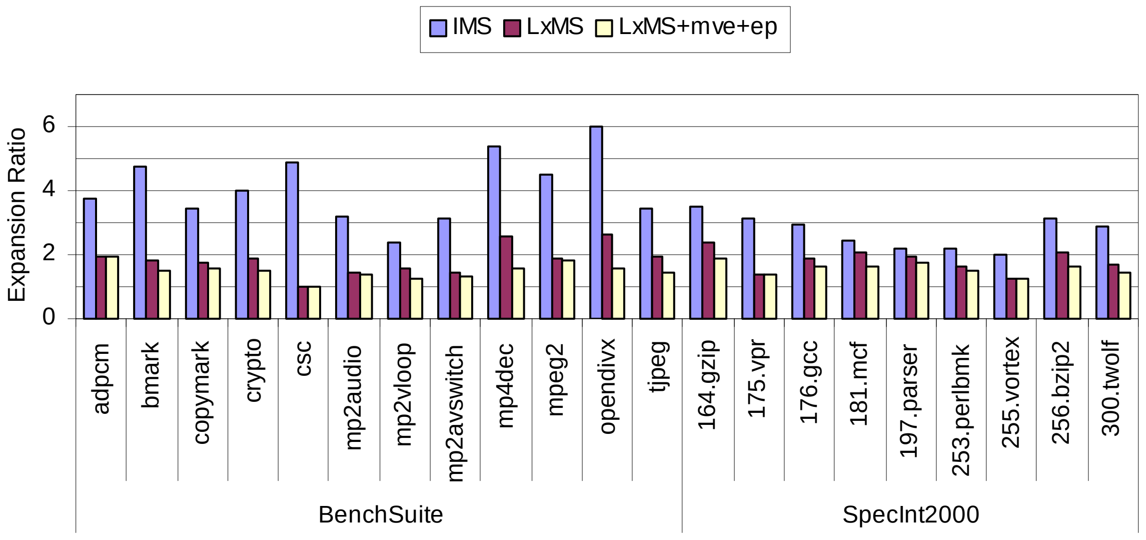

Figure 9: Code size expansion ratio

7. Evaluation

7.1. Impact of code size reduction in performance

In this section we analyze the impact of the code size reduction on performance on a detailed benchmark-by-benchmark basis. We have excluded the non-speculative schemes because the number of loops for which a valid modulo schedule can be generated with those schemas is very limited, especially in the case of SpecInt2000. We have used IMS with a speculative code generation schema as an example of a code size insensitive modulo scheduler, our default modulo scheduler (LxMS) also speculative to show the benefits of a code size sensitive modulo scheduler, and finally LxMS with all optimizations from section 6 enabled (this is the default for our compiler). All benchmarks have been run to completion and the output has been validated for all runs.

Figure 9 shows the loop expansion ratio. Notice that almost all the individual results follow the trend of the global numbers previously shown. In all cases, LxMS causes significantly less code expansion than IMS, which is more pronounced for the BenchSuite benchmarks due to the larger loops. Also, for all benchmarks except *adpcm* (in which the extra copies are worse than the epilogue reduction), the proposed schemas reduce the code expansion even more. Notice that although the code size reduction mechanisms make a small contribution on average, there are a few benchmarks (like *mp4dec* and *opendivx*) where these mechanisms make an important contribution. Finally, there is the exceptional case of *csc* where LxMS manages to schedule the three inner loops of the program with SC=1 and MVE=1, using the same II as in IMS, which results in no code expansion at all.

Table 4 shows the impact of the code size reduction heuristics (LxMS+mve+ep) in the final binary size, relative to a code size insensitive scheduler (IMS). Notice that for SpecInt2000 the

Table 4: Binary size in bytes

BenchSuite				SpecInt2000			
Program	IMS	LxMS +mve+ep	reduction	Program	IMS	LxMS +mve+ep	reduction
adpcm	2696	1856	31.2%	gzip	66448	60848	8.4%
bmark	119104	71976	39.6%	vpr	163128	154248	5.4%
copymark	79272	57040	28.0%	gcc	1725692	1689120	2.1%
crypto	105480	67432	36.1%	mcf	12592	11952	5.1%
csc	15664	5088	67.5%	parser	115280	113416	1.6%
mp2audio	68040	53160	21.9%	perlbmk	722296	715232	1.0%
mp2vloop	26640	25760	3.3%	vortex	555464	552656	0.5%
mp2avswitch	189232	157696	16.7%	bzip2	39392	34824	11.6%
mp4dec	75464	44736	40.7%	twolf	301880	273016	9.6%
mpeg2	98456	73160	25.7%				
opendivx	77056	42008	45.5%				
tjpeg	56560	52288	7.6%				
AVERAGE			30.3%	AVERAGE			5.0%

reduction is minimal because they are mostly huge programs with very small loops. However, for the BenchSuite, the binary size reduction is very significant; 30% on average and 67.5% for the best case. Recall that BenchSuite is representative of the embedded Lx target domain, where binary size is sometimes critical.

Figure 10 shows the performance relative to IMS with no stalls, i.e., the ideal execution time assuming no cache miss penalty and no branch penalty. In general, the performance improvement/degradation is almost negligible, being in the –2% +3% range in the most extreme cases. We have observed that although LxMS produces slightly smaller IIs, this is not a

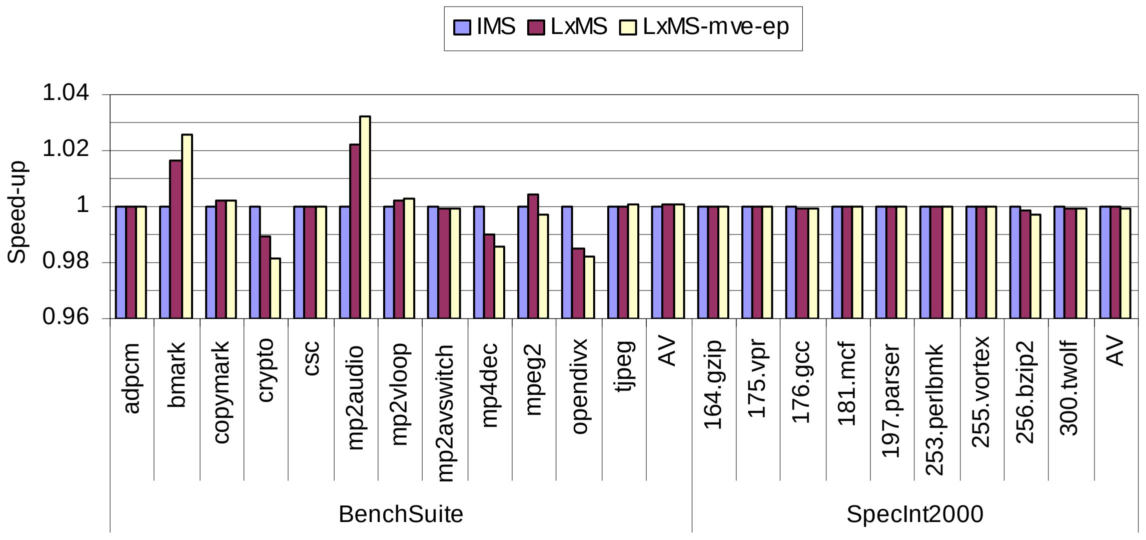

Figure 10: Performance relative to IMS without stalls

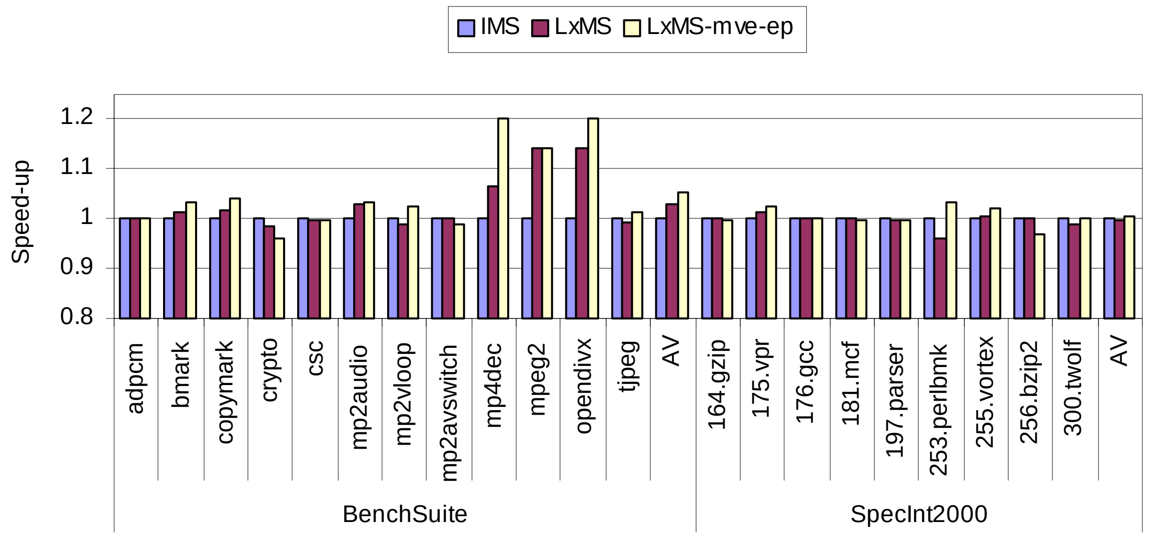

Figure 11: Real execution times relative to IMS

determinant factor on performance. The two cases (*bmark* and *mp2audio*) in which LxMS improves performance are due to the fact that loops having a single epilogue allow the epilogue to be scheduled in combination with the succeeding basic block. When MVE reduction and epilogue collapsing is applied, this is emphasized. The three cases (*crypto*, *mp4dec* and *opendivx*) where LxMS degrades performance are due to loops with short iteration counts that are executed many times. Our compiler schedules prologues and epilogues using a different pattern than in the kernel, benefiting from the lower resource requirements of this two pieces, to generate more efficient schedules than what can be achieved with kernel-only code. If the loop performs very few iterations, the code in the prologue-epilogue can be more efficient for schedulers that speculate less (IMS). This situation is aggravated by the copies in copy blocks and the serialization of the epilogue to reduce its size. However, on average, the performance is not affected.

Finally, Figure 11 shows the real execution time. We have observed that the D-Cache stall cycles are practically the same for the three cases. Branch penalty cycles are slightly higher for LxMS than for IMS (due to smaller MVE) and even higher for LxMS-mve-ep; however, they have a minimal impact on performance. The most important factor that varies significantly with code size reduction is I-Cache Stalls. There are three benchmarks where performance is significantly improved by reducing loop code expansion (*mp4dec* +20%, *mpeg2* +14% and *opendivx* +20%). Of these, in *mp4dec* and *opendivx* the code reduction schemas play a critical role. There are several other benchmarks where reducing loop expansion produces a small improvement in I-Cache stalls. Finally, there are a few benchmarks where reducing code size actually increases I-Cache stalls. The most notable case is *perlbmk* where the code size reduction produced by LxMS over IMS reduces I-Cache capacity misses, but causes two critical functions to conflict in the I-Cache, thus increasing the conflict misses; when code size is further reduced they do not conflict any longer, and we benefit from reduced I-Cache capacity misses. A number of

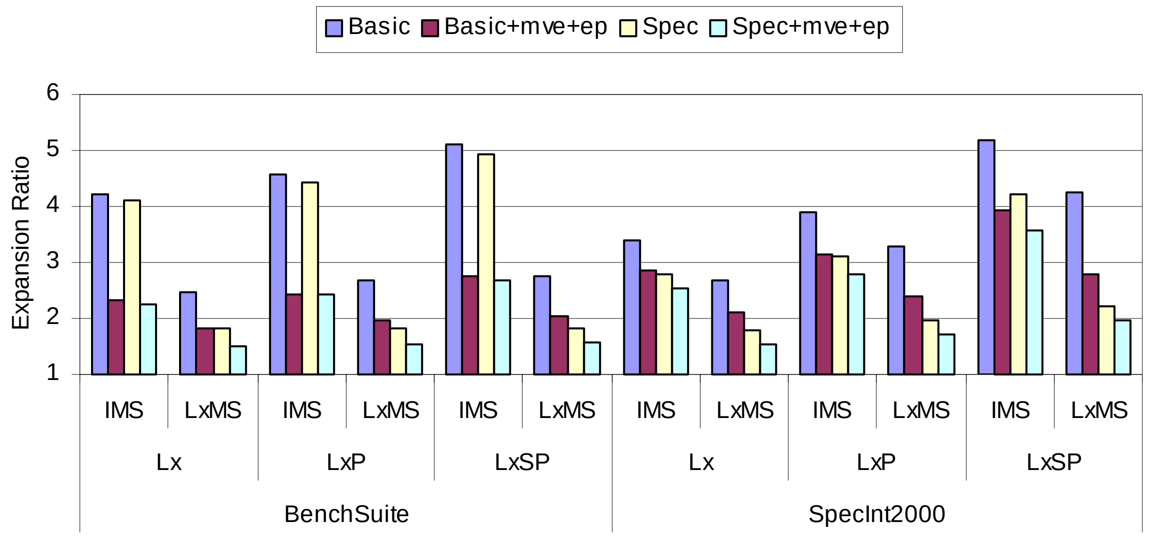

Figure 12: Evaluation with other architectures

approaches that address the I-Cache conflict problem have been published, but are beyond the scope of this paper.

7.2. Evaluation with other architectures

Finally, we have evaluated the proposed techniques to reduce code expansion with different schedulers, with and without speculation and with different machine configurations. Lx corresponds to the ST210 processor used throughout this paper. LxP corresponds to a processor where load latency (32, 16 and 8 bit) has been increased to 4 cycles, multiplies take 3 cycles, and the compare to branch latency is 3 cycles. LxSP is like LxP with 8 cycle loads.

Figure 12 shows the code size expansion for these more aggressive configurations for both IMS and LxMS. For each processor configuration and each scheduler there are four columns: Basic corresponds to the basic code schema, +mve+ep corresponds to the schema with all epilogs collapsed and copies inserted to reduce MVE. Finally the Spec columns correspond to the equivalent speculative schemas.

Notice that all processor configurations follow the same trend with code size increasing with the more aggressive configurations. An interesting observation is that, with aggressive configurations, code size expansion grows faster for smaller loops (SpecInt2000) than for big loops (BenchSuite) for all schedulers and code schemas. This is caused by the fact that the minimum posible K_{min} is determined by the longest operation latency in the loop, which for SpecInt2000 loops is usually longer than the II. However, all the combined heuristics for containing code size (code size sensitive modulo scheduling with speculative and code size reduction schemas) behave much better and perform an excellent job containing loop expansion.

It is also important to notice that the proposed techniques of collapsing epilogs and inserting copies work as well for non-speculative schemas and for any modulo scheduler. In particular, for the embedded benchmarks, those optimizations reduce significantly the code size expansion of IMS.

8. Conclusions

One of the drawbacks of modulo scheduling is heavy code expansion due to explicit generation of prologues and epilogues, and the replication of the loop body to avoid self-overlapping lifetimes. This problem is so severe that some high performance processors incorporate dedicated hardware support to generate kernel-only code for modulo scheduled loops.

This paper shows that code size expansion can be significantly reduced by incorporating a few code reduction heuristics in the modulo scheduler. In particular, the following heuristics are very effective:

- Minimize the length of the lifetimes to reduce the amount of MVE. These heuristics also reduce register pressure. Our scheduler achieves this objective by scheduling all operations as close as possible to their predecessors/successors. The same objective can be achieved by using other register sensitive schedulers, or a register reduction post-pass [6].
- Minimize the number of stages. This can be done by maximizing the priority of operations on the critical paths (in terms of height) over operations on other paths.
- Schedule the loop branch instruction as late as possible in the schedule to minimize the number of exits in the prologue. Our compiler achieves this objective with the same heuristics used for reducing MVE and register pressure. A post-scheduler transformation could easily be implemented to achieve the same objective. These heuristics also increase the amount of speculation, which reduces code size in speculative schemas.

Among several code schemas (non-speculative, loop-preconditioning, and speculative), speculative code schemas are by far the most effective to reduce code size if the modulo scheduler contributes by maximizing speculation.

Finally, we have proposed code schemas where multiple epilogues are collapsed in a single one, at the expense of explicitly renaming some registers, and serializing the execution of the epilogue. In addition, we have proposed inserting copy instructions in unused issue slots to reduce the amount of MVE.

We have shown that that all the above heuristics have a great effect on code size. For our processor, code expansion is reduced from 4.2 to 1.5 with the BenchSuite benchmarks (from 3.4 to 1.5 with SpecInt2000). This reduced code expansion results in binaries 30% smaller on average for the embedded benchmarks. The code size reduction factor is even more significant with configurations that are more aggressive (5.1 to 1.6 with BenchSuite and 5.2 to 2.0 with SpecInt2000).

Moreover, although the code size reduction techniques produce a small performance degradation for some benchmarks, in general they contribute to a slight increase in performance. For a few benchmarks, reducing code expansion is critical in terms of I-Cache performance, producing speed-ups of up to 20%.

References

[1] V. Allan, R. Jones, R. Lee, and S. Allan. Software pipelining. *ACM Computing Surveys 27*,3 (Sept. 1995), pp. 367-432.

[2] G.R. Beck, D.W.L Yen, and T.L Anderson. The Cydra 5 minisupercomputer: Architecture and implementation. *J. Supercomputing 7*,1/2 (May 1993), pp. 143-180.

[3] J. Codina, J. Llosa, and A. Gonzalez. A comparative study of modulo scheduling techniques. In *Proc. Intl. Conf. on Supercomputing*, June 2002.

[4] J. Dehnert P. Hsu, and J. Bratt. Overlapped loop support in the Cydra 5. In *Proc. 3rd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp 26-38, April 1989.

[5] J. Dehnert and R. Towle, Compiling for the Cydra 5. *J. Supercomputing 7*,1/2 (May 1993), pp. 181-228.

[6] A.E. Eichenberger and E.S. Davidson. Stage scheduling: A technique to reduce the register requirements of a modulo schedule. In *Proc. 28th Intl. Symp. on Microarchitecture*, pp. 338-349, Nov. 1995.

[7] Equator technologies. MAP1000 unfolds at Equator. Microprocessor report, 129160, Dec. 1998.

[8] P. Faraboschi, G. Brown, G. Desoli, and F. Homewood. Lx: A technology platform for customizable VLIW embedded processing. In *Proc. 27th Intl. Symp. on Computer Architecture,* pp. 203-213, June 2000.

[9] J. Fisher, P. Faraboschi, and G. Desoli. Custom fit processors: Letting applications define architectures. In *Proc. 29th Intl. Symp. on Microarchitecture,* pp. 324-335, Dec. 1996.

[10] R.A. Huff. Lifetime-sensitive modulo scheduling. In *Proc. ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation*, pp. 258-267, June 1993.

[11] Intel Corporation, Intel IA-64 Architecture Software Developer's Manual Volume 1: Application Architecture, Jan. 2000.

[12] M.S.Lam, Software pipelining: An effective scheduling technique for VLIW machines. In *Proc. ACM SIGPLAN 1988 Conf. on Programming Language Design and Implementation,* pp. 318-328, June 1988.

[13] D.M. Lavery and W.W. Hwu. Modulo scheduling of loops in control-intensive non-numeric programs. In *Proc. 29th Intl. Symp. on Microarchitecture,* pp. 126-137, Dec. 1996.

[14] J. Llosa, A. Gonzalez, E. Ayguade, and M. Valero. Swing modulo scheduling: A lifetime sensitive approach. In *Proc. Intl. Conf. on Parallel Architectures and Compilation Techniques,* 1996.

[15] M.C. Merten and W.W. Hwu. Modulo Schedule Buffers. In *Proc. 34th Intl. Symp. on Microarchitecture,* pp. 138-149, Dec. 2001.

[16] S. Ramakrishnan. Software pipelining in PA-RISC compilers. *Hewlett-Packard Journal*, pp. 39-45, July 1992.

[17] B.R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc., 27th Intl. Symp. on Microarchitecture,* pp. 63-74, Nov. 1994.

[18] B.R. Rau and C. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proc. 14th Annual Microprogramming Workshop,* pp. 183-197, Oct. 1981.

[19] B.R. Rau , M.S. Schlansker , P. P. Tirumalai. Code generation schema for modulo scheduled loops. In *Proc. 25th Intl. Symp. on Microarchitecture,* pp. 158-169, Dec. 1992.

[20] Texas Instruments. TMS320C6000 CPU and instruction set reference guide. March 1999.

[21] P. Tirumalai, M. Lee and M.S. Schalansker, Parallelization of loops with exits on pipelined architectures. In *Proc. Supercomputing'90,* pp. 200-212, Nov. 1990.

[22] N.J. Warter, G.E. Haab, K. Subramanian, and J.W. Backhaus. Enhanced modulo scheduling for loops with conditional branches. In *Proc. 25th Intl. Symp. on Microarchitecture,* pp. 170-179, Dec. 1992.