Simultaneous Speculation Scheduling

A. Unger[1], Th. Ungerer[2], E. Zehendner[1]

[1] Computer Science Department; Friedrich Schiller University; D-07740 Jena, Germany
{a.unger, zehendner}@acm.org
[2] Dept. Computer Design & Fault Tolerance; University of Karlsruhe; D-76128 Karlsruhe, Germany
ungerer@ira.uka.de

Abstract—

Simultaneous Speculation Scheduling (S^3) is a combined compiler and architecture technique to control multiple path execution. It can be applied for dual path branch speculation in case of unpredictable branches and for multiple path speculative execution of loop iterations. Loop-carried dependences are handled by data dependence prediction. Architectural requirements are a minimal form of multithreaded processor architecture and three new instructions *(fork, sync, wait)*. Simulation results show performance gains of up to 40% over purely static scheduling techniques by applying the S^3 technique to branches in kernel sections of SPECint95 benchmark programs.

Keywords: instruction scheduling, multithreading, eager execution, dual path execution

I. Introduction

Future processors need new ideas to extract parallelism from program structures that are hard to parallelize. Speculative parallelism is one solution to speed up the execution of single threaded programs. Contemporary high-performance microprocessors exploit speculative parallelism by dynamic branch prediction and speculative execution of the predicted branch path. Additional speculation is provided e.g. by a speculative load instruction together with a check instruction as part of Intel's IA-64 ISA [10, 11]. The check instruction is scheduled before the use of the loaded value and prevents speculative execution using the speculatively loaded value. Research of future microcarchitectures additionally looks at the prediction of data dependences, source operand values, value strides, address aliases, and load values with speculative execution applying the predicted values [15, 14, 3].

A multithreaded processor is able to pursue two or more threads of control in parallel within the processor pipeline. Multithreading is able to increase performance of a multiprogramming workload, although it may slightly decrease single thread performance compared to a single threaded processor.

In our S^3 approach we apply multithreading to increase single thread performance by utilizing new forms of speculation. The S^3 technique can be applied to replace branches by speculative execution of both branch paths or for a speculative execution of loop iterations. The compiler employs the S^3 technique only in cases when traditional techniques probably fail and when the parallel resources of the processor cannot be usefully utilized otherwise.

Branches are usually handled by dynamic branch prediction and speculative execution of the branch path with the highest likelihood. Rerolling in case of misspeculation is expensive. However, some branches are unpredictable, nevertheless they are speculatively executed in contemporary microprocessors. One way out is shown by the Intel IA-64 approach which includes branch instructions with 'sequential prefetch hints' that allow to specify that only few instructions should be prefetched after a prediction. The latter eases the rerolling in case of misspeculation. Moreover, the instruction set is fully predicated. Predication forces loading and decoding of instructions of both branch paths, even though only one branch path is executed. Only instructions with the 'true' predicate as additional operand input are executed, but all instructions of both paths are loaded, decoded, and dispatched to the respective instruction window. Compiler scheduling is improved by larger basic blocks, but the instructions that are on the wrong path consume fetch and decode bandwidth and may clog the instruction window.

The S^3 technique is most similar to predication, but uses multithreading to execute multiple branch paths. Our technique directs the processor to stop execution of the wrong path as soon as the branch resolves. Instructions on the wrong path need not be fully loaded and decoded if the branch resolves early, which is an advantage over the predication technique. On the other hand, if the branch resolves late, then both branch paths may be executed speculatively and wrong path instructions are discarded afterwards. Rerolling is not necessary, because the correct path instructions have also been executed. Both is in contrast to predication, where instructions must be loaded and decoded even when the predicate resolves early, and where instructions are not executed speculatively when the predicate resolves late.

Loop unrolling and software pipelining are the means to extract parallelism from parallel loops by compiler. Intel IA-64 supports software pipelining by hardware by a rotation of parts of the general-purpose, floating-point, and predication registers [10, 11]. If loop-carried dependences exist, software pipelining is much harder to apply. Here our S^3 technique delivers a further possibility to speculatively execute loop iterations by controlling data dependence speculation.

Section II specifies requirements for a multithreaded base

architecture suitable for our compiler technique and architectural proposals that match the requirements. The S^3 technique is introduced in sections III and IV. We show the relation to other static techniques as well as to dynamic techniques for branch handling in section V. Section VI summarizes the results of translating a number of benchmarks using our technique.

II. Target Architectures

S^3 is only applicable to architectures that fulfill the following requirements of a multithreaded base architecture:

- First, the architecture must be able to pursue two or more threads of control concurrently—i.e., it must provide two or more independent program counters.
- All concurrently executed threads of control share the same address space, preferably the same register set.
- The instruction set must additionally provide three thread handling instructions:
  - `fork` *label descriptor*
    The *fork* instruction creates a new thread. Execution of the instructions starts at the label *label*. A thread descriptor is returned in the register *descriptor*.
  - `sync` *condition desc1 desc2*
    The *sync* instruction conditionally terminates the execution of its own thread or the execution of some other threads. Depending on the content of the register *condition*, either execution of the threads referred to by descriptor *desc1* (*condition*==0) or the threads referred to by the descriptor *desc2* (*condition*!=0) is cancelled. All instructions of threads to be cancelled are discarded from the pipeline.
  - `wait` *descriptor*
    The *wait* instruction stops execution of the thread until the thread referred to by *descriptor* has been terminated. The *wait* instruction is necessary for the speculative execution of loop iterations—as will be demonstrated below.

  Creating a new thread by the *fork* instruction and joining threads by the *sync* instruction must be extremely fast, preferably single cycle operations. Details of the implementation of the thread handling instructions strongly depend on the target architecture.

The primary target architectures of the proposed compiler technique are simultaneous multithreaded [20], microthreaded [2], and nanothreaded architectures [7], which can be classified as explicit multithreaded architectures, because the existence of multiple program counters in the microarchitecture is perceptible in the architecture. However, implicit multithreaded architectures that spawn and execute multiple threads implicitly—not visible to the compiler—can also take advantage of a modified version of S^3. Examples of such implicit multithreaded architectures are the multiscalar, the trace processor, and the datascalar approaches (see [24]).

III. Simultaneous Speculation Scheduling—Branches

Instruction scheduling techniques [18] are of great importance to expose ILP contained in a program to a wide-issue processor. The instructions of a given program are rearranged to avoid underutilization of the processor resources caused by dependences between the various operations (e.g. data dependences, control dependences, and the usage of the same execution unit).

Branches seriously prevent a compiler-based instruction scheduling from moving instructions to unused instruction slots before the branch instruction. Global scheduling techniques as for instance PDG Scheduling [1] or Selective Scheduling [16] use various approaches to move instructions across branches to execute these instructions speculatively. The increase of performance gained by these speculative extensions is limited either by the rollback overhead of misprediction or by chosen restrictions to speculative code motion.

S^3 can be seen as an advancement of global instruction scheduling techniques. We relax the restrictions to speculative code motion and reduce the penalties for misspeculation by generating separate threads to be executed in parallel for alternative program paths.

As with most global instruction scheduling techniques, S^3 attempts to enlarge the hyperblocks. Branches are removed by the following approach: Each selected branch is replaced by a *fork* instruction, that creates a new thread, and by two conditional *sync* instructions—one in each thread. The two threads active after the execution of the *fork* instruction evaluate the two program paths corresponding to both branch targets. The *compare* instruction attached to the original *branch* remains in the program. It now calculates the condition for the *sync* instructions. The thread which reaches its *sync* instruction first either terminates, or it cancels the other thread.

After removing one or more branches by generating speculative threads, the basic scheduling algorithm continues to process the new hyperblocks. Each thread is considered separately. The heuristic used by the scheduling algorithm is modified to keep the speculative code sections small. Therefore the *sync* instruction is moved upwards as far as the corresponding *compare* allows. The speculative sections are further reduced by combining identical sequences starting at the beginning of speculative sections and moving them across the *fork* instruction.

The program transformation described above is implemented by the following

Algorithm

1. Determining basic blocks.
2. Assessing the execution probabilities of the basic blocks, combined with a confidence estimation.
3. Selection of the hyperblocks by the global instruction scheduling algorithm.
4. Selection of the conditional branches that cannot be handled by the basic scheduling technique and that dynamically have a low confidence, but can be resolved by splitting the thread of control; concatenation of the corresponding hyperblocks.
5. Generation of the required operations for starting and synchronizing threads; if necessary, modification of the existing *compare* instructions.
6. Further processing of the new hyperblocks by the global scheduling algorithm.
7. Scheduling of the operations within the hyperblocks, using a modified heuristic.
8. Minimizing the speculatively executed program sections by moving up common code sequences starting at the beginning of the sections.
9. Calculating a new register allocation; insertion of *move* instructions, if necessary.

The steps 1, 3, 6, and 7 can be performed by a commonly used global scheduling technique. For our investigations we use the PDG scheduling technique [1]. A simple way to implement the modifications of the scheduling heuristic (step 7) is to insert a number of artificial edges into the dataflow graph and to adjust the weights assigned to the operations by the static scheduling algorithm. This allows to almost directly use the formerly employed local scheduling technique *(List Scheduling)* that arranges the instructions within the hyperblocks. Assigning a larger weight causes an instruction to be scheduled to execute later. Therefore the attached values of the *compare* and the *sync* instructions are decreased. Since these modifications directly influence the size of the speculative section they must correspond to the implemented properties of the processor that executes the generated program. The exact weights that are assigned are not fixed by this algorithm but are parameters that have to be determined for each processor implementation.

Step 2 collects information about branches that will be replaced by the S^3 technique. Selection criteria are a low confidence for the branch prediction and the availability of hardware threads assuming that only a single program is running on the multithreaded processor. The compiler may use several techniques to decide when to apply S^3 scheduling and when to use normal branch instructions (assuming the branch instructions will be executed on a processor with dynamic single path branch prediction). The compiler may:

- examine the program structure (branches at the end of loop bodies should be dynamically predicted; branches resulting from conditional statements may be good candidates for S^3),
- use profiling (prior runs of the program), or
- relegate prediction to the programmer by compiler directives.

To gather the confidence information by profiling, the program is instrumented with additional code that collects data about the outcome of the branch instructions and thus about the execution probability of the basic blocks and that second simulates the branch predictor of the target processor.

Generating separate threads of control for different program paths causes the duplication of a certain number of instructions. The number of redundant instructions grows with the length of the speculative sections, but remains small in the presence of small basic blocks. Since the number of speculatively executed threads is already limited by the hardware resources, only a small increase in the program size is caused for integer programs.

For a more detailed description see [23].

IV. Simultaneous Speculation Scheduling—Loops

We have shown the basic concepts of the S^3 technique in the previous section. In this section we show an extension of the S^3 technique that allows to speculate on data dependences and that speculatively executes sequences of instructions in parallel even if these sequences could not be proven to be independent during compile time. Sequences under consideration are especially consecutive iterations of a loop. The compiler/parallelizer performs a static data dependence analysis. The information about the data dependence of the instructions that is found by this analysis can be classified into three categories:

1. proven dependence or,
2. proven independence or,
3. no information available.

In the first two cases the static analysis could provide sufficient information to generate appropriate code. This means a sequential program in the first case and a parallel program in the second case. Unfortunately, there exist numerous situations in which the techniques applied by a static data dependence analysis cannot provide data dependence information or in which dependences are not determined at compile time. In these situations, i.e., the third case in the above enumeration, the static analysis returns a conservative approximation. We assume the analysis tool not only to return the 'no information' approximation, but also information why it could not prove dependence or independence. This means the static analysis transfers a condition to the S^3 technique that must be proven to ensure independence, for example the addresses of two pointers that must be tested whether they alias each other. The S^3 technique generates code to test the

condition at runtime of the program. The consecutive iterations of the loop are speculatively started in paralliel. Then the condition is calculated, using information available during runtime, and depending on the outcome, speculatively executing threads are stopped or continue to execute.

A program generated for the first case of the above enumeration executes using only one thread, a program fitting into the second case executes using more than one 'safe' thread. A program generated by the S^3 technique executes always one 'safe' thread and one or more speculative threads. In the following we restrict S^3 to use only one speculative thread.

The S^3 technique works for loops as follows:

1. Static data dependence analysis: Information about data dependences is calculated. If independence cannot be proven statically, the conditions to prove independence during runtime are passed to the S^3 code generation.
2. Selecting sequences for speculative execution: The candidate sequences found in step 1 are examined by a heuristic to predict if speculative execution will lead to a performance gain.
3. Copying the loop body: A number of loop iterations of the selected loops are copied in the program code. The copies of the loop body are not executed consecutively as in unrolled loops, but in parallel.
4. Calculating a new register allocation: Registers that are written to in the speculative section are mapped to different registers.
5. Generation of the condition to prove independence: The condition to prove independence is generated using instructions of the target processor's instruction set.
6. Generation of thread handling instructions.
7. Scheduling: The instructions of the new loop bodies are scheduled to achieve high ILP.

The static data dependence analysis (step 1) is performed using standard dependence analysis techniques [18].

The heuristic in step 2 uses information about the conditions to be solved, the additional effort to solve them, the length of the sequences executed in parallel, and the numbers of registers to be renamed. From the conditions itself we derive information on the probability of a successful speculation. The additional effort to solve the condition decreases the performance gain. The performance gain increases with the length of the sequences executed in parallel. Using many additional registers has the potential of introducing spill code and thus to decrease the performance. The heuristic tries to select code sequences that are likely to be successfully executed in parallel and thus to significantly improve execution speed.

The S^3 technique generates a 'safe' version and a 'speculative' version of the loop body. This is done by inserting multiple copies of the loop body into the program code (step 3) and enriching these copies with the instructions for thread control (step 6). Generating two versions is necessary for two reasons. First, one version is needed that executes always without speculation. Consequently only one thread can run a 'safe' version at the same time. The second reason for generating different versions is that in the original code the iterations running in parallel work on the same registers. As we have already discussed in section III the S^3 technique has to perform a static register renaming. Alternatively a dynamic register renaming would solve this problem. Here we assume that the target processor does not support a dynamic register renaming, because this results in a much higher hardware effort and in a significantly larger time for executing the thread handling instructions [25].

After the copying of the loop body and the insertion of the thread handling instructions all instructions in the two versions of the loop body are scheduled to improve the amount of ILP (step 7). Both versions are scheduled independently.

In the first step the instructions of the speculative version are scheduled. The heuristic places the *sync* instruction to execute early, i.e., to keep the speculative section small. This first scheduling step determines the temporal distance between the beginning of the speculatively executed iteration and the availability of the result the condition to ensure correctness evaluates to. This value is a parameter to the scheduling step that reorders the instructions of the 'safe' version. Here the heuristic tries to find a schedule that avoids always stopping the speculatively executed iteration, because the condition is never resolved before the first *sync* instruction in the 'safe' version is executed. The scheduling step of the 'safe' iteration may cause copying of instructions into the speculative version. Therefore the instructions in the speculative version need to be rearranged again. This second scheduling step may cause a change of the above mentioned distance and thus of the arrangement of the instructions in the 'safe' version. Therefore we perform the scheduling step again on both versions. If this again causes a change of the distance the scheduling step is not repeated, but either the first *sync* instruction in the 'safe' version is delayed by inserting NOPs or the speculative execution of consecutive iterations is discarded, i.e., the loop is executed sequentially using only one thread.

During the scheduling step of the 'safe' version an instruction needs to be copied into the speculative version, whenever the *fork* instruction is moved across this instruction and either it shows a loop-carried dependence to an instruction in the speculative version or the *sync* instruction depends on it. Copying is not required if both, the *fork* instruction and the first *sync* instruction, are moved across an instruction. If copying is required, the instruction is inserted in the speculative version before the *sync* instruction.

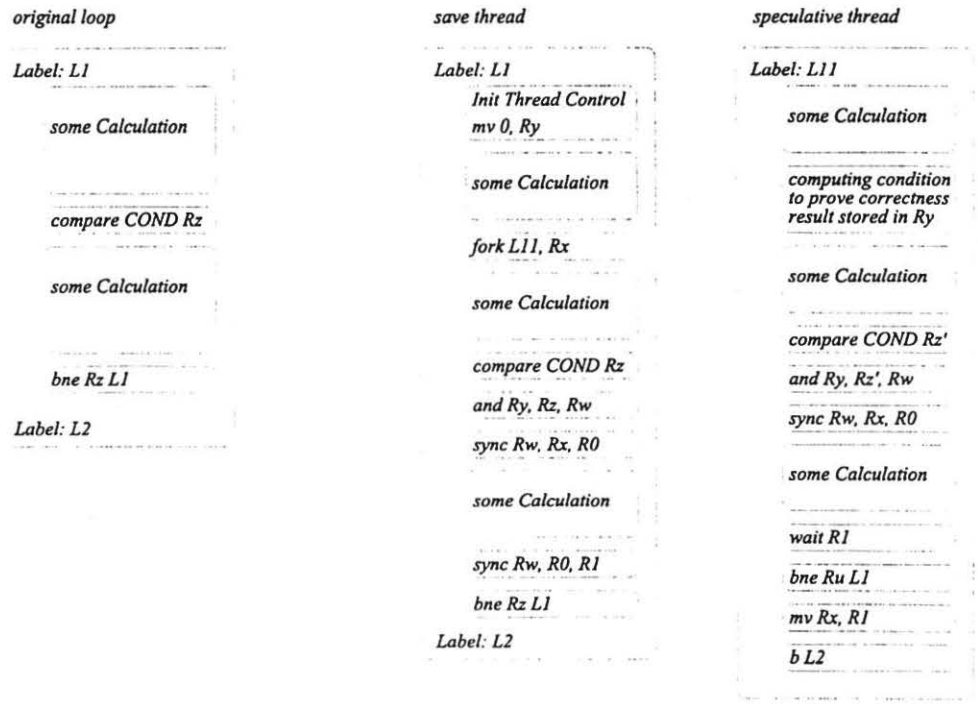

Fig. 1: Speculation on data dependences

Figure 1 demonstrates the application of the S^3 technique to speculate on potential data dependences between the iterations of a loop. The basic structure of the original loop (left hand side) and the two versions of the transformed loop body (right hand side) are shown.

The 'safe' thread starts with a sequence of instructions called *Init Thread Control*, that initializes thread interaction. In the simple version shown here it only sets register *Ry* to zero. This register is the destination of the result the condition to ensure correctness evaluates to and it contributes to the argument of the *sync* instructions. Setting this register to zero makes sure that the 'safe' thread stops the execution of the speculative thread, if the speculative thread could not prove its correctness before the first *sync* instruction in the 'safe' thread is executed. The *init thread control* section is followed by a sequence of instructions from the original loop body. This sequence includes those instructions that have to be executed before the speculative iteration is started due to dependences between these instructions and the instructions within the speculative section. Next the *fork* instruction follows, which speculatively starts the next loop iteration. The *fork* instruction returns the thread descriptor of the newly started thread. In the example it is stored in register *Rx*. The *fork* instruction is followed again by a block of instructions from the original loop body including the original *compare* instruction. The result of this *compare* instruction and the register *Ry* are combined by an *and* instruction. If this returns the value *TRUE* the speculative execution of the next loop iteration was correct. Otherwise the correctness could not be proven and thus the first *sync* instruction stops the execution of the speculative thread. The arguments of this *sync* instruction are the condition in register *Ry*, the thread descriptor of the speculative thread in register *Rx*, and register *R0*, which always returns the value zero. This value is detected as an invalid thread descriptor and therefore the *sync* instruction has no effect if the condition evaluates to *TRUE*.

The *sync* instruction is followed by a third block from the original loop body and a second *sync* instruction that prevents the 'safe' thread to jump back and re-evaluate the next iteration if the next iteration was started speculatively and has been proven to be correct. The second *sync* instruction uses the same condition, but an invalid descriptor as the second argument and the descriptor of the 'safe' thread as the third argument. Finally the 'safe' version of the loop body contains the original *branch* instruction to the begin of the loop body. This branch is executed if the speculative execution of the consecutive iteration was stopped.

All these instructions are treated together by the scheduling step and by the transformations following the S^3 technique. This may cause the sequences from the original loop body to change.

The speculative version of the loop body has a similar structure. It further contains instructions to solve the condition that dynamically ensures the correctness of the speculative execution. The arguments of the *sync* instruction are modified that in case of the result *FALSE* the speculative thread stops itself. The speculative version of the loop body is completed by a *wait* instruction, that protects a branch to the beginning of the 'safe' version if the previous iteration has not finished yet, for instance because of a cache miss. Thus, no two copies of the 'safe' version will ever run in parallel. The *wait* instruction is followed by a conditional branch to the beginning of the 'safe' version, which uses the condition from the original loop body, and an unconditional branch to the first instruction following the loop if the speculative thread was executing the last iteration.

The S^3 technique can be modified to execute more than one iteration speculatively and to take advantage of dynamic register renaming.

V. Relating S^3 to other multi path execution techniques

If we compare the speculative execution of alternative program paths as defined by S^3 to a dynamic branch prediction and speculative single path execution as it is done by contemporary superscalar microprocessors, the advantage of our technique is a smaller misprediction penalty, and the disadvantage is the slight overhead introduced by the execution of the new instructions that replace the branch instruction. Dynamic branch prediction and speculative execution forces a complete reload of the pipeline and possibly suffers from additional penalty cycles for canceling the mistakenly issued and executed instructions in case of a misprediction. Mispredicted branches incur in the Pentium II a penalty of at least 11 cycles, with the average misprediction penalty being 15 cycles [5]. Also, for the 6-issue Alpha 21264 an average misprediction penalty of more than 11 cycles is reported [6] resulting in potentially more than 66 unused instruction issue slots.

S^3 does not cause any penalty related to a rollback since the correct thread always proceeds while the misspeculating thread is terminated. The only possible slow-down would be due to instruction slots shared between the threads. Investigations have shown that ILP in real programs is much smaller than the usually available hardware resources. Furthermore, only part of the instruction slots are lost, i.e., the instruction slots occupied by the misspeculating thread minus the instruction slots that could never be covered by the correct thread. Finally, S^3 controls the number of threads that are concurrently executed and thus it controls the speculative use of processor resources.

A number of compiler scheduling techniques employ purely static methods of branch speculation [18]. This approach has the advantage of no further hardware requirements. For highly irregular programs the drawback of these techniques lies either in restrictions which are too strong to apply speculation or in a very large penalty in situations where the predicted path is incorrect. S^3 avoids these problems by simultaneously speculating on alternative program paths and executing the generated threads in parallel on a multithreaded architecture.

Predication techniques [9] enhance the ISA of a processor by predicated instructions and one or more predicate registers. The boolean result of a condition testing is recorded in one or two predicate registers. Predicated instructions use a predicate register as an additional input operand. The execution of a predicated instruction depends on the value of the predicate register. Predication affects the instruction set, adds a port to the register file, and complicates instruction execution. Predication is most effective when control dependences can be completely eliminated, such as in an if-then-statement with a small then-body (a so-called hammock in dynamic predication [13]), and when the condition can be evaluated early. The use of predicated instructions is limited when the control flow involves more than a simple alternative sequence. Furthermore, predicated instructions are fetched and decoded but usually not executed before the predicate is resolved. Alternatively, as envisioned for Intel's IA-64 Merced processor, a predicated instruction may be executed, but commits only if the predicate is true, otherwise the result is discarded [4].

A number of research projects survey eager execution. They extend either superscalar architectures, e.g. Disjoint Eager Execution [22], Selective Dual Path Execution [8], Limited Dual Path Execution [21], and the PolyPath architecture [12] or SMT architectures, e.g. the 'nanothreaded' DanSoft processor and Threaded Multiple Path Execution (TME) [25].

All these proposals—except for the DanSoft processor—use dynamically collected information to decide whether eager execution is applied instead of single path speculation. In contrast, S^3 statically assigns dual path execution. Advantages are larger hyperblocks for an optimized compiler-performed instruction scheduling, an optimized register mapping done at compile time resulting in less register mapping overhead, and less hardware complexity.

On the other hand, dynamically collected branch confidence information may be more accurate, and a dynamic decision on dual path execution allows load-dependent thread spawning. However, in case of fairly irregular programs the branch predictor may not be able to derive any suitable information about branch probabilities, thus speculative execution supported by hardware will not be able to gain any advantage over S^3 but still has to implement the additional hardware support.

benchmark	compress	go (mrglist)	go (getefflibs)
number of instructions in the program code	22	48	40
average number of instructions executed	10	12.6	25
number of branches	4	6	5
number of speculations	3	3	2
nest of speculation	1	1	1
SBP (average of cycles)	9.6	18.6	37.3
MBP (average of cycles)	8.8	13.3	31.4
performance gain	9%	40%	19%

TABLE I: Performance gain achieved by S^3

Memory dependence prediction [17] is a pure hardware technique that tries to use memory dependence locality to speculatively execute instruction and to prove semantic correctness later. If speculation turns out to be erroneous the effects of the speculatively executed instructions are undone. Memory dependence prediction collects information about the history of the program to predict memory dependences. Compared to S^3 memory dependence prediction will need fewer functional units, because it does not execute instructions that are predicted to be depend on other instructions. On the other hand it needs additional processor resources to implement the predictor. S^3 can use information available during compile time.

Software based run-time parallelization techniques [19] employ an inspector-executor scheme to test, if the iterations of a loop can be executed in parallel and consecutively run a sequential or a parallel version of the loop. More advanced techniques [26] start the parallel version speculatively and in case of a misspeculation roll back to a 'safe' state and restart the loop sequentially. Compared to S^3, the technique requires a higher regularity in the program to achieve a performance gain. When applied to irregular programs these techniques will lead to an increase of the execution time.

VI. Performance Evaluation

In this section, we show a number of experimental results to examine the performance of S^3 applied to branches only. We used benchmarks from the SPECint95 benchmark suite. Presently, we cannot translate the complete programs but have to focus on frequently executed functions, for two reasons. First, we currently do not have a compiler implementation of the algorithm that would allow to translate arbitrary programs. Second, the architectures under consideration are subjects of research. For some of them simulators are available, for others only the documentation of the architectural concepts is accessible. This means that both the process of applying our technique to the programs and the calculation of the execution time must be partially done by hand.

Since this is a time-consuming process we currently can present only results for three kernel sections from the SPECint95 benchmark suite. The translated program sections cover the inner loops of the functions `compress` from the *compress* benchmark and `mrglist` and `getefflibs` from the *go* benchmark. The results are shown in Table I.

For the calculation of the execution time we use two processor models. The first one—the *superscalar base processor* (SBP)—implements a simplified SPARC architecture. The SBP has the ability to execute up to four instructions per cycle. We assume latencies of three cycles for *load* instructions, one cycle for branches, and one cycle for all arithmetic operations, except for the multiplication which takes four cycles. The second processor model—the *multithreaded base processor* (MBP)—enhances the SBP by the ability to execute up to four threads concurrently, and by the instructions *fork* and *sync*. Here we expect these instructions to execute within a single cycle. Both processor models do not include any assumptions on caching effects, because of the size of the used sample code.

The values presented in Table I were derived from counting the machine cycles, processors that match our processor models would need to execute the generated programs. The numbers shown are the number of instructions in the program code, the average number of instructions executed (instructions per thread, weighted with the measured execution probability of the thread), the number of conditional branches, the number of speculations performed by S^3, the nest of speculation, the cycles for the SBP, the cycles for the MBP, and the performance gain (calculated as (SBP cycles / MBP cycles) -1). The results show that our technique achieves performance gains up to 40% over purely static scheduling tech-

niques when generating code for simultaneous multithreaded processors as well as processors that employ nanothreading or microthreading.

VII. Conclusion

In this paper we proposed a combined compiler/hardware technique—Simultaneous Speculation Scheduling (S^3)—that targets unpredictable branches and loop iterations with loop-carried dependences. Such branches are resolved by dual path execution; the loop iterations can be executed simultaneously applying data dependence speculation. Our approach eliminates the misprediction penalty of the dynamic single path branch prediction that is applied in commodity microprocessors and it allows more speculative parallelism for loop execution. In the absence of a sufficient amount of parallelism to completely utilize all execution units of a multithreaded processor, the S^3 technique speculatively utilizes coarse grained parallelism. The threads to be executed concurrently are generated from speculative program paths and directly mapped on the hardware threads of a multithreaded processor.

Applied to branches our technique improves the capability of global instruction scheduling techniques by removing conditional branches. Compared to pure hardware dual path execution techniques, S^3 gains in presence of branches with a high misprediction rate. Furthermore, S^3 can migrate part of the functionality required for branch speculation from hardware into the compiler.

Speculating on data dependences allows to execute iterations of a loop in parallel, even if the compiler cannot prove that no data dependences will occur. Compared to pure static techniques S^3 can defer the decision if the execution was correct until runtime. On the other hand S^3 can avoid additional hardware cost by doing part of the work during compile time.

We have compared our technique to other techniques that employ speculation to improve the execution time of irregular programs, and we have evaluated our technique by translating code from the SPECint95 benchmark suite. Currently, we are translating additional programs from the SPECint95 benchmarks.

References

[1] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. In B. Hailpern, editor, *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 241–255, Toronto, ON, Canada, June 1991.

[2] A. Bolychevsky, C. R. Jesshope, and V. B. Muchnik. Dynamic scheduling in RISC architectures. *IEE Proceedings Computers and Digital Techniques*, 143(5):309–317, 1996.

[3] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proceedings of the ISCA 25*, pages 142–153, Barcelona, Spain, 1998.

[4] C. Dulong. The IA-64 architecture at work. *IEEE Computer*, 31(7):24–31, July 1998.

[5] L. Gwennap. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, 9(2):9–15, February 1995.

[6] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, 10(14), October 1996.

[7] L. Gwennap. Dansoft develops VLIW design. *Microdesign Resources*, pages 18–22, February 1997.

[8] T. Heil and J. Smith. Selective dual path execution. Technical report, University of Wisconsin-Madison, http://www.engr.wisc.edu/ece/faculty/smith_james, 1996.

[9] W.-M. Hwu. Introduction to predicated execution. *IEEE Computer*, 31(1):49–50, 1998.

[10] Intel. *IA-64 Application Developer's Architecture Guide*. http://developer.intel.com/design/ia64/index.htm, June 1999.

[11] Intel. *IA-64 Application Instruction Set Architecture Guide Rev. 1.0*. http://www.hp.com/go/ia64, June 1999.

[12] A. Klauser, P. Abhijit, and D. Grunwald. Selective eager execution on the PolyPath architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 250–259, Barcelona, Spain, June 1998.

[13] A. Klauser, T. Austin, D. Grunwald, and B. Calder. Dynamic hammock predication for non-predicated instruction set architectures. In *Proceedings of the PACT 98*, pages 278–285, Paris, October 1998.

[14] M. H. Lipasti and J. P. Shen. The performance potential of value and dependence prediction. In *Lect. Notes Comput. Sci. 1300*, pages 1043–1052, 1997.

[15] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Compilation Systems*, pages 138–147, Cambridge, MA, October 1996.

[16] S. M. Moon and K. Ebcioglu. Parallelizing nonnumerical code with selective scheduling and software pipelining. *ACM Transactions on Programming Languages and Systems*, 19(6):853–898, 1997.

[17] A. I. Moshovos. *Memory Dependence Prediction*. PhD thesis, University of Wisconsin-Madison, 1998.

[18] S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, San Francisco, 1997.

[19] L. Rauchwerger. Run-time parallelization: It's time has come. *Journal of Parallel Computing*, 24(3). Special Issue on Languages & Compilers for Parallel Computers, 1998.

[20] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202, Philadelphia, PA, May 1996.

[21] G. Tyson, K. Lick, and M. Farrens. Limited dual path execution. Technical Report CSE-TR 346-97, University of Michigan, 1997.

[22] A. K. Uht and V. Sindagi. Disjoint eager execution: An optimal form of speculative execution. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 313–325, Ann Arbor, MI, November 1995.

[23] A. Unger, Th. Ungerer, and E. Zehendner. Static speculation, dynamic resolution. *Proceedings of the 7th Workshop on Compilers for Parallel Computers (CPC '98)*, Linköping, Sweden, June 1998.

[24] J. Šilc, B. Robič, and Th. Ungerer. *Processor Architecture - From Dataflow to Superscalar and Beyond*. Springer-Verlag, Berlin, Heidelberg, New York, 1999.

[25] S. Wallace, B. Calder, and D. Tullsen. Threaded multiple path execution. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 238–249, Barcelona, Spain, June 1998.

[26] Y. Zhang, L. Rauchwerger, and J. Torrellas. Speculative parallel execution of loops with cross-iteration dependencies in in DSM multiprocessors. In *Proceedings of High Performance Computer Architecture 1999 (HPCA-5)*, pages 135–141, Orlando, FL, 1999.