RC 19928 (02/06/95)
Computer Science

IBM Research Report

Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-assisted Fine-Grained Multithreading

Pradeep K. Dubey, Kevin O'Brien, Kathryn M. O'Brien and Charles Barton

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

Abstract

Recent limit studies on instruction-level parallel processing, based on non-numeric applications, have reported significant performance gains from speculative execution of multiple control flows. This paper describes a new *single-program speculative multithreading* (SPSM) architecture, which can be viewed as an extension of any existing single-thread architecture. It enables speculative fetch, decode, and execution from multiple program locations simultaneously. Instruction threads are generated at compile-time using control dependence analysis. Inter-thread data dependences are also analyzed at compile-time. However, resource binding of instructions is performed only at run time, to offer binary compatibility across different implementations. New thread generation algorithms, being prototyped in a version of the TOBEY compiler, are also described. The SPSM architecture includes novel *fork/suspend* instructions which are used to identify independent instruction threads, and also to specify compile-time control flow speculations associated with inter-thread dependences.

1 Introduction

Identifying independent instructions that can be executed in parallel has always been a key to performance enhancement. Since the early days of the Tomasulo algorithm to recently emerging techniques, such as VLIW (Very Long Instruction Word), a wide variety of studies have been done on instruction-level parallel processing (ILP). The reported degree of available parallelism covers a wide range from less than two to several hundred instructions per cycle (IPC) [1, 2, 3, 4, 5, 6]. Although some of the variations are due to the parallelism inherent in the benchmarks, others are due to the architecture and organizations of the underlying machine model. Examples of such differences include sequential versus out-of-order execution, local versus global scope of concurrency detection, dependency detection and resource scheduling at compile-time versus run-time. Consider benchmarks restricted to general purpose integer benchmarks, such as those in the SPEC suite (SPEC89 or SPEC92). The ILP studies of these benchmarks which restrict the scope of concurrency detection to within basic block, i.e., the studies where branch instructions execute in-order and without any speculation, report a very low parallelism potential, less than 3 IPC, even with infinite resources [7, 8, 5]. ILP studies which allow speculation of branches but still require to in-order execution of branch instructions, have reported parallelism around 7 IPC [9, 5], with infinite resources. Even with large set of resources (such as, 8 to 16 functional units), the parallelism available through such single control flow speculation is only around 4 IPC [42, 3]. Recent ILP studies [5, 6] also based on imperfect branch speculation, have reported infinite resource performance potential of almost 40 IPC, provided branch instructions, like data instructions, are also allowed to execute out-of-order, implying speculation of multiple control flows.

1.1 Motivations and Goals

The primary motivation of this ongoing research in *Single Program Speculative Multithreading* (SPSM) architecture is the observation, based on the above mentioned limit studies, that pursuing *multiple simultaneous speculative flows of control* can lead to a significant performance improvement. Additional motivation for our research is derived from the following observation. The amount of peak parallelism supported in some recent microprocessor designs [10] is 4 IPC. It is quite conceivable that technologically it would be feasible to support even higher levels of peak parallelism, say 8 to 12 IPC. In the light of this observation, low IPC limits of single control flow speculation can also lead to very poor resource utilization of future processors.

The primary goal of SPSM research is a cost-performance study of proposed architectural extensions to enable compiler-assisted, control-dependence directed, single-program, fine-grained multithreading, with explicit compile-time speculation. The threads in this case share the same instruction set architecture and are generated at compile-time. Explicit compile-time speculation implies that if the compile-time specified speculations hold, the threads are data and control independent of each other; otherwise, if the speculations are incorrect, the threads may have inter-thread data and control dependences.

An additional goal of SPSM research is to maintain binary compatibility with previous implementations of the architecture, as well as across different new implementations

of the architecture. Therefore, similar to most current compilers, compiler enhancements described here *do not require* any knowledge of actual machine resources, e.g., maximum number of threads, number and types of functional units, etc. There is no resource binding at compile time. Compiler implementations and various machine model simulations of our research should lead to an implementable machine model proposal, supplementing the trace-analysis based limit studies of the past.
This paper describes the SPSM architecture and the associated compilation techniques for generating speculative threads. The following section provides a brief description of data and control dependence. Section 3 discusses some of the limitations of a single control thread machine model. Section 4 describes the SPSM architectural primitives which provide support for the speculative execution of multiple threads of control of a given program. These architectural primitives can be added to an existing architecture as a compatible extension. Section 5 describes how the added instructions help address the single thread performance limitations discussed before. Section 6 describes the compilation techniques being prototyped in a research version of TOBEY [11] compiler. The frontend of TOBEY accepts high-level languages such as, FORTRAN and C; the backend supports architectures such as, POWER and PowerPC [12]. Section 7 offers a brief description of the related research in the area of multithreading. A final section summarizes the status of the work and future research.

2 Data and Control Dependences

Given an initial state of program and data memory, and an initial program counter, consider the sequence in which a non-overlapping (i.e., execution of each instruction is completed before starting the execution of next instruction) and non-speculative (i.e., outcome of an instruction is not speculated prior to execution) uniprocessor executes a program. The unique complete dynamic order resulting from such an execution is referred to as *sequential trace order* (STO) in the discussion to follow. Unless otherwise specified, any reference to order among instructions implies the order determined by STO.
We refer to instructions which alter the sequential flow of control as *control* instructions; all others are called *data* instructions. A *data* (*control*) dependency exists between two instructions, if the necessary and correct execution of one instruction depends on the data value (control information) generated by the other. Such data and control dependences define a *partial order* on the set of data and control instructions. Concurrency detection and scheduling almost always involves a reordering of data and/or control instructions without violating the partial order.
Concentrating on the data dependences, the partial order implies that an instruction I that follows a data instruction D (in the STO) does not necessarily need data value from D. Data dependence analysis attempts to determine essential data dependences and an observable implication is out-of-order execution of data instructions. Similarly, concentrating on the *control dependences* [45], the partial order implies that an instruction I that follows a branch instruction B (in the STO) does not necessarily need control information from B. Control dependence analysis [44] attempts to determine essential control dependences and an observable implication is out-of-order execution of control instructions, which further leads to multiple control

flows, potentially implying multiple program counter based implementations. A machine is considered *speculative* if outcomes of certain instructions (typically, control instructions) are speculated prior to actual execution. Although many of the current commercial microprocessors allow control-flow speculation and simultaneous execution of multiple out-of-order data instructions, none to this date, allows simultaneous execution of multiple out-of-order control instructions.
A machine is said to have *single* control flow, if processing (e.g., fetch, decode, execute, etc.) of an instruction I in the STO does not begin until *all* control instructions prior to it in the STO, have either been resolved or speculated. Alternatively, a machine is said to have *multiple* flows of control if processing of an instruction in the STO does not necessarily have to wait for resolution or speculation of all control instructions prior to I in the STO. Unless otherwise stated, compiler-identified flows of control are also referred to as *threads* in this paper.

3 Background

3.1 Limitations of single control thread

All of the current generation of commercial microprocessors have a single thread of control flow. Such processors are limited in their ability to exploit control and data independence of various portions of a given program. Some of the important limitations are listed below:

- Single thread implies that the machine is limited to fetching a single sequence of instructions and is unable to pursue multiple flows (threads) of program control simultaneously.

- Single-thread control further implies that data independence can only be exploited if the data-independent instructions are close enough in the thread to be fetched close together in time and examined together to detect data independence.

- The limitation above in turn implies reliance on compiler to group together control independent and data-independent instructions.

- Speculation on control flow is already acknowledged as a necessary technique for exploiting higher levels of parallelism. Almost all of current microprocessors contain some form of branch prediction. However, due to the lack of any knowledge of control dependence, single-thread dynamic control flow speculation can only extend the ability to look ahead until there is a mis-speculation. It should be noted that run-time learning of control dependence via single thread control flow speculation is at best limited in scope, even if the hardware cost is ignored.

- Compile-time speculation on control flow can have much larger scope than run-time speculation and can also benefit from control-dependence analysis [44] However, run-time limitation of single thread, again requires the compiler to group together these speculative instructions along with the non-speculative ones, so that the parallelism is exploitable at run time.

3.2 Compile-time Control Flow Speculation

The use of compile-time control flow speculation to expose more parallelism at run time has been mentioned above. Compilers of current machines are limited in their ability to encode this speculation. Commonly used approaches such as guarding [39] and boosting [40] rely on the compiler to percolate some instructions to be speculatively executed earlier in the single thread execution. These techniques often also require the control flow speculation encoded in the speculative instruction. This approach has following important limitations:

- It is typically very difficult to find enough unused bits in every instruction to encode even shallow control flow speculations. Note that due to backward compatibility constraints (ability to run old binaries, without any translation), instruction encoding can not be arbitrarily rearranged (implying new architecture) to include the encoding of control flow speculation.
- The percolation techniques mentioned above often require extra code and/or code copying to handle mis-speculation. This results in code expansion.
- Sequential handling of exceptions raised by the speculative instructions and precise handling of interrupts are often architecturally required. However, implementing these in the context of such out-of-order speculative execution, is often quite difficult, due to the upward speculative code motion used by the percolation techniques mentioned above. Special mechanisms are needed to distinguish the percolated instructions and to track their original location.

An alternate guarding technique proposed by Pnevmatikatos and Sohi [36] groups together the guarding information of a sequence of instructions upto a basic block, into a new *guard* instruction. The mask associated with the *guard* instruction identifies the instructions that need to be squashed if the compile-time specified condition does not hold.

3.3 Exploiting Inter-Iteration Parallelism in Nested Loops

The lack of the knowledge of control dependence can be especially costly to performance in nested loops. For example, consider a nested loop, where outer iterations are control and data independent of data dependent inner loop iterations. If knowledge of control and data independence of outer loop iterations is not exploited, their fetch and execution must be delayed, due to the serial control flow speculation involving the inner loops. Furthermore, due to this lack of knowledge of control dependence, speculatively executed instructions from an outer loop may unnecessarily be discarded on the misprediction of one of the control and data independent inner loop iterations. Also, note that the probability of misprediction on the inner loop control flow speculation can be quite high in cases where the inner loop control flow is data dependent and hence quite unpredictable. One such example is given below.

```
/* check the environment list */

for (fp = xlenv; fp; fp = cdr (fp) )
```

```
for (ep = car (fp); ep; ep = cdr (ep) )
    if (sym == car (car (ep) ) )
       cdr (car (ep) ) = new_p;
```

This is a doubly nested loop, where the inner loop traverses a linked list and its iterations are both control and data dependent on previous iterations. However each activation of the inner loop (i.e., the outer loop iterations) is independent of the previous one. [This is a slightly modified version of one of the most frequently executed loops (*Xlgetvalue*) in one of the SPECint92 benchmarks (*Li*)].

3.4 Register Pressure

As explained above, machines with single control flow have to rely on the compiler to group together speculative and/or non-speculative data-independent instructions. However, to group together all data and control independent instructions efficiently, the compiler needs enough architected registers for proper encoding. Therefore, register pressure is increased and beyond a point such code motion becomes fruitless due to the overhead of additional spill code.

3.5 Managing Multiple Threads of Control

Some research attempts have been made to build processors with multiple threads [26, 32], primarily aimed at implementing massively parallel architectures. The overhead of managing multiple threads can potentially outweigh the performance gains of additional concurrency of execution. Some of the overheads associated with thread management are the following:

- *Inter-thread synchronization:* Maintaining and communicating the partial order due to data and control dependence, through explicit or implicit synchronization primitives.
- *Inter-thread communication:* Communicating the values created by one thread for use by another thread.
- *Thread scheduling:* Tradeoffs associated with *static*, i.e., compile-time, thread scheduling versus *dynamic*, i.e., run-time, thread scheduling. Static thread scheduling simplifies run-time hardware, but is less flexible and exposes the thread resources of a machine implementation to the compiler, and hence requires recompilation for different implementations. On the other hand, dynamic thread scheduling is adaptable to different implementations, all sharing the same executable, but it requires additional run-time hardware support.

4 Architectural Extensions for Speculative Multithreading

The SPSM architecture can be viewed as an extension of any single-thread (RISC or CISC) architecture. It adds *fork-suspend* instructions enable simultaneous fetch,

decode, and execution from multiple program locations, using multiple program counters. The Primary goal of the architecture is to fetch, decode and attempt execution of an instruction as soon as its *control dependence* is resolved (or speculated).
This is in contrast to traditional single thread superscalars where the fetch of an instruction must wait until its *control flow* resolution or speculation. For example, consider the control-flow graph shown in Figure 1, where the nodes refer to basic blocks. In a single-thread superscalar, assuming large inter-basic-block distances, blocks A, B, C, and D would all be fetched and executed in serial order, without any overlap. However, since basic blocks A and D are control-independent of each other, execution of D is imminent soon after control flows into block A. Same holds for basic blocks B and C. Therefore, in a multithreaded architecture, such as, SPSM, fetch and execution of blocks A and D (and of blocks B and C) can potentially be overlapped, as illustrated in Figure 1. Note that although control-independent basic blocks can be fetched and decoded simultaneously as different threads, their simultaneous execution further requires inter-thread data dependence analysis. In the SPSM architecture, inter-thread data dependence analysis is performed at compile-time and it often requires speculation of control-flow between the control independent blocks.

4.1 SPSM execution model

An SPSM program begins execution as a single *main* thread. This main thread can initiate one or more *future* threads, by means of the *fork* $< addr >$ instruction. [1] A future thread inherits the state of the main thread at the point of fork, and executes a piece of the sequential trace which is logically ahead (in the STO) of the main thread. The main and future thread execute in parallel. At some point the main thread catches up, ie, its flow reaches the the starting address of the future thread. The future thread's updates are then merged into the main thread's state, the future thread expires, and the main thread can then continue past the latest instruction executed by the future thread. This overlapped execution is the primary source of SPSM's performance advantage.
We describe below the fork and suspend instructions, and the process of merging with a future thread. A conditional form of the suspend instruction offers great power in speculatively avoiding the limitations of data dependence between threads.

4.2 Fork $< addr >$

When the machine encounters *fork* $< addr >$ instruction, it may (optionally) start a new thread. A new program counter is initialized with the specified address, and a (logical) copy of the current state of the forking thread becomes the initial state of the new future thread. The main and future threads then fetch and execute in parallel. The main thread continues with the instruction following the fork, and the future thread continues from the address specified in the fork. The future thread

[1] For the sake of exposition, in the rest of the paper, *forking* is restricted to main thread only. The SPSM architecture has no such restriction and future threads are also capable of forking additional future threads.

creates new register values, separate from those created by the main thread. They are assimilated by the main thread at merge (described below).
Note that this instructions merely provides a compile-time *hint* regarding the forking potential. Depending on the available machine resources at run time, a thread may or may not be forked. This optional nature of the *fork* instruction is described later in more detail.
The expected use of fork is to start a thread at an address which the main thread is certain to reach. The compiler would derive the address from control-dependence analysis. The SPSM architecture would also permit forking speculatively to an address which the main thread may or may not reach.

4.3 Unconditional suspend

An unconditional suspend instruction causes a future thread to stop fetching and executing instructions. Its program counter and its updated register values are saved, awaiting merger with the main thread. The saved program counter points to the next instruction after the suspend.
The unconditional suspend instruction is needed to prevent the future thread from executing instructions whose sources are not known by the compiler to be available at the point of fork. The conditional suspend, described below, permits speculative execution of instructions whose sources are expected to be available at the point of fork.
Note that when the machine decides not create a future thread, the main thread will encounter the corresponding suspend. A suspend has no effect when encountered by the main thread.

4.4 The *merge* action

When the main thread reaches the starting address of the forked future thread, the state of the future thread is merged into that of the main thread. This also referred to as the *merge point*. After the merge, the future thread ceases to exist.
The effect of a merge is to replace some register values in the main thread with values taken from the future thread, and then to update the main thread's program counter. The registers that are modified are exactly those that were updated by instructions of the future thread. The updated program counter points to the instruction following the last instruction executed by the future thread. After the merge, the main thread resumes execution from this point.
The description above needs to be refined in the presence of conditional suspend instructions, as described in the next section. In this case some instructions of the future thread may have been executed speculatively. At or before the merge, the speculated conditions must be resolved, and the effects of incorrectly speculated instructions must be discarded. In this case, the future thread's program counter will be reset to point to the first incorrectly speculated instruction. At this point the merge proceeds as described above.
Note that at the time of merge, the forked future thread may have already suspended, or it may still be actively executing. In the latter case, an implementation could

choose to suspend the future thread or instead choose to wait until it reached its suspend point.

4.5 Suspend $< mode\_bit >$, $< cond\_1\ cond\_2\ ...\ cond\_n >$

This is the conditional form of the suspend instruction. To simplify the discussions to follow, we define the term *dependence region* of a *suspend* instruction as the sequence of instructions in the sequential trace order that starts with the first instruction after the *suspend* instruction and is terminated on encountering any other *suspend* instruction, conditional or not.

A *suspend* instruction is associated with a set of conditions, $< cond\_1 >$ through $< cond\_n >$ which encode compile-time speculations of control flow. More precisely, the set of conditions $< cond\_1 >$ through $< cond\_n >$ associated with a *suspend* instruction in a future thread, encode compile-time speculation regarding the control path to be followed by the main thread subsequent to forking the future thread and prior to merging with the forked future thread.

Taking advantage of an innovative feature of POWER and PowerPC architectures [12], whereby multiple condition codes are scheduled and managed by the compiler (in a manner similar to general purpose register scheduling), this paper proposes associating compile-time speculations directly with different condition register bits. Consequently, the proposed control-flow speculation involves speculating on the *true* or *false* settings of a sequence of condition register bits. At run time, the speculated settings of condition register bits are compared against the actual settings of the corresponding condition register bits at the merge point to determine the validity of the compile-time speculation. This is similar to *taken* or *not-taken* assumption for the corresponding branches associated with the condition bit comparisons. This also offers potential for a very concise encoding of multiple speculations. For example a set of 8 bits in the instruction format can encode upto 8 speculations.

The mode bit is used to interpret the set of conditions $< cond\_1 >$ through $< cond\_n >$ in one of two ways. If the mode bit is set to valid (V), the thread management unit keeps the results of the set of instructions in the dependence region associated with the *suspend* instruction, provided all of the compile-time specified conditions, from $< cond\_1 >$ through $< cond\_n >$, associated with the *suspend* instruction, evaluate to *true* at run time.

If the mode bit is set to invalid (I), the thread management unit discards the results of the set of instructions in the dependence region associated with the *suspend* instruction, if all of the compile-time specified conditions, from $< cond\_1 >$ through $< cond\_n >$, associated with the *suspend* instruction, evaluate to *true* at run time.

Intuitively speaking, a compiler would use the *valid* mode setting for encoding a *good* path from the fork point to the merge point. The speculated instructions are useable only if the main thread follows this good path. The compiler would use the *invalid* mode setting to specify a *bad* path from the fork point to the merge point; unless the main thread avoids this path, the speculated instructions must be discarded.

Consider for example the control flow graph and the associated code sequence in Figure 2. The instructions in the dependence region associated with the *suspend* instruction are data dependent on the $r5$ register updates in basic blocks X and Y. Therefore, subsequent to forking a future thread to basic block D, if the control

flows to basic block X or basic block Y, register $r5$ gets redefined, and therefore, the speculatively executed instructions in the dependence region of the future thread *suspend* need to be discarded. This inter-thread data dependence is specified at compile-time by encoding the condition-bit settings that avoid the control flow into blocks X and Y. The assumed *true* setting of the first condition register bit implies that branch at the end of block A is assumed to be *taken*. Similarly, the *false* setting of the second condition register bit setting implies that the branch at the end of block B is assumed to *fall through*.
Here are some of the important advantages of such a speculation scheme:

- There is no assumed *order* in the set of compile-time speculations. The condition regsiter bit settings (or the associated branch resolutions) can occur in any order at run time.

- As explained above, the *invalid* setting of the mode-bit enables the compiler to encode the *bad path*, without committing itself to any of the multiple *good* paths. This addresses one of the important limitations of VLIW machines based on *trace scheduling* techniques [27], where the performance potentially suffers if there is no one *good* trace (as is typically the case with data-dependent control flows) and not all good paths are optimally compacted by the scheduler.

- Future thread size is restricted only by essential data dependence constraints, and not restricted by basic-block or procedural boundaries.

- A single *suspend* instruction encodes multiple speculations for multiple instructions in the dependence region.

- An alternative encoding of the condition field, can instead choose to associate $< cond\_1 >$ through $< cond\_n >$ with the dynamic sequence of branches. In other words, $< cond\_1 >$ would encode the speculation associated with the first branch encountered by the forking thread after the fork point, $< cond\_2 >$ would encode the speculation associated with the second branch encountered by the forking thread after the fork point, and so on. Such an encoding has the drawback of having to encode three possible branch outcomes, *taken*, *not-taken*, or *don't care*. The last outcome is needed to deal with control independent branches. Under the proposed scheme, the condition register bits associated with control-independent branches (including loops) are simply not specified in the set of conditions.

Note that a disadvantage of the proposed speculation scheme is that it does not associate each of the multiple speculations in a *suspend* instruction with the corresponding sets of dependent instructions in the dependence region. Such an information could be used to selectively discard instructions of the future thread at the time of merge with main thread. However, such a *selective* merge with arbitrary subset of future thread instructions, may not lend itself to a simple implementation.

4.6 Optional Nature of Forks

A novel characteristic of the instructions proposed in this paper is that their use at compile time does not require any assumption regarding the run-time CPU resources.

Depending on the aggressiveness of an actual implementation, a specific CPU may or may not be able to actually fork a future thread. In other words, from the CPU's point of view, an actual fork at run time in response to encountering any *fork* instruction, is entirely optional. The user of these instructions (e.g., the compiler) does not need to keep track of the number of pending future threads, and it also cannot assume any specific fork to be definitely obeyed (i.e., fork a future thread) at run time.
The compiler identifies control and data independent code regions which may be executed as separate (future) threads. However, the compiler does not perform any further restructuring or optimizations which assume that these threads will execute in parallel. For example, the compiler preserves any *spill* code that would be needed to guarantee correct program execution when any one of the inserted *fork* instructions is ignored by the CPU at run time. *Spill* code refers to the instructions which store the contents of any architecturally visible CPU register in a certain location in the instruction cache, and later reloading the contents of the same location without another intervening store. Note that the execution of spill code may be redundant during its execution as a future thread. Additional compile-time hints could reduce or eliminate this spill-code overhead during future thread execution.
As a direct consequence of the *optional* nature of *fork* instructions, there is no need to recompile for different implementations of the SPSM architecture, each capable of forking *zero* or more threads. Similarly, there is no need to recompile any old binary, which does not contain any of the new instructions.

4.7 Interpreting Multiple Conditional Suspends in a Future Thread

It is possible that a future thread which gets forked in response to a *fork* instruction, encounters a series of conditional suspends before encountering an unconditional suspend. Each conditional suspend is still interpreted in association with the common fork point and independent of other conditional suspends. Thus, it is possible to associate different control flow speculations with different portions of a future thread. Consider a *suspend* instruction A. Suppose A is followed by another *suspend* instruction B, after a few instructions other than *fork*, *suspend*, *uncond_suspend*. Instruction B would typically be followed later by an *uncond_suspend* instruction. Assume that the compile-time condition associated with the *suspend* instruction A. is determined to be false at run time. To simplify the compilation and to reduce the state keeping in future threads, an implementation can simply discard the results of all instructions between A and the *uncond_suspend* instruction, instead of limiting the discarding to between A and B.

4.8 Undisambiguable memory operations in future threads

Memory *load* (*store*) operation with undisambiguable addresses at compile-time, is commonly referred to as *speculative load* (*store*). SPSM architecture permits compiler to include speculative *loads* in future threads. For example, the *load* and *store* operations in Figure 2 have potentially ambiguous addresses, and hence *speculative*. Runtime hardware support in the form of *address disambiguation buffers* is to be provided to compare all future thread *loads* against all main thread *stores* until merge.

However, *speculative stores* are not permitted in future threads.

5 Potential Advantages

5.1 Alleviating the instruction-fetch bottleneck

As illustrated in the example above, proposed fork and suspend instructions offer a novel way of addressing the instruction fetch bottleneck of current superscalar processors. The compiler can use these instructions to point to arbitrarily far (dynamically) control independent blocks. Control independence implies that given that the program control has reached the fork point it is bound to reach these future blocks. Therefore, an instruction can be fetched as soon as the control dependence of its block is resolved (without waiting for the control flow). Also, speculatively fetched instructions should only be discarded if the branch from which they derive their control dependence (not the one from which control flow is derived) is mispredicted. For example, instructions in block D in Figure 2, can be fetched along with those of block A, soon after their shared control dependence on the branch at the end of block W is either resolved or speculated. Furthermore instructions from block D should be considered a wasted fetch or discarded only if the control dependent branch at the end of block W is mispredicted and not if the branch at the end of block B is mispredicted. A traditional superscalar without any notion of control dependence would discard its speculative execution of blocks Z (or Y) as well as D if blocks Z and D are fetched via traditional control flow speculation of the branch at the end of block B and this turns out to be a misprediction later on.

5.2 Exploiting data independence across control independent blocks

The instructions in the control independent blocks which are also data independent of all possible control flow paths leading to these blocks, can be executed simultaneously and non-speculatively, via multiple forks to these control independent blocks. For example, the first instruction in block D can be fetched and executed non-speculatively using the proposed fork and suspend instructions.

5.3 Comparison with *guarding* and *boosting*

As mentioned earlier, the net effect of the speculative execution scheme proposed in this paper is similar to that of speculatively boosting the the instructions in the dependence region of a *suspend* instruction. But unlike these previously known techniques of guarded (or boosted) instructions, which encode the control flow condition as part of each guarded (or boosted) instruction, the proposed technique encodes the condition for an arbitrary group of instructions using conditional and unconditional suspend instructions. Some of the important advantages of this approach are the following:

- *Minor architectural Impact:* As implied above, a primary advantage of the proposed scheme is its relatively minimal architectural impact. Except for the addition of fork and suspend instructions, the existing instruction encodings are unaffected. Therefore, unlike the boosting approach, the proposed mechanism does not depend on available bits in the op-code of each boosted instruction to encode the control flow speculation.

- *Simultaneous encoding of multiple speculations for multiple instructions:* A single *suspend* instruction proposed in this paper is capable of encoding multiple speculations for multiple instructions in the associated dependence region. The *guard* instruction proposed by Pnevmatikatos and Sohi [36] encodes a single condition speculation for multiple instructions, whereas, earlier schemes (such as, [39]) have proposed encoding multiple speculations with each instruction.

- *Small Code Expansion:* The typical percolation and boosting techniques often require code copying or patch-up code in the path off the assumed trace. This can lead to significant expansion of code size. The proposed technique does not have any of these overheads and the only code expansion is due to the fork and suspend instructions, which are shared by a set of instructions.

- *Simpler Implementation of Sequential Exception Handling:* There is no upward code motion in the proposed technique, and the code speculatively executed still resides in its original position only. Therefore, exception handling can be easily delayed until the main thread merges with the future thread containing the exception causing instruction. In other words, exceptions can be handled in proper order, without having to explicitly mark the original location of the speculative instructions which may raise exceptions.

- *Simpler Implementation of Precise Interrupts:* The unique main thread in this proposal, is always precisely aware of the last instruction completed in the sequential program order. Therefore, there is no need of any significant extra hardware for handling interrupts precisely.

5.4 Decoupling of Compilation and Machine Implementation

Note that due to the optional nature of the forks, as explained before, the compilation for the proposed architecture can be done assuming a machine capable of large number of active threads. And the actual machine implementation has the option of obeying most of these forks, or some of these forks, or none of these forks, depending on available machine resources. Thus compilation can be decoupled to a large extent in this context from the machine implementation. This also implies that there may be no need to recompile separately for machines capable of small or large number of active threads.

5.5 Parallel Execution of Loop Iterations

Proposed fork and suspend instructions can also be used to efficiently exploit across iteration parallelism in nested loops. For example, consider the sample loop illustrated before in Section 3.3. The inner loop iterations of this loops are both control and data dependent on previous iterations. However, each activation of the inner loop (i.e., the outer loop iterations) is independent of the previous one. Hence, it is possible for the compiler to use the proposed *fork* instruction (starting at outer loop body) to enable a machine to start many activations of the inner loop without waiting for the previous ones to complete, and without unnecessarily discarding executed instructions from the outer loop iterations on misprediction of some control and data-independent iteration of the inner loop.

5.6 Easing of Register Pressure

Instructions in the control independent basic blocks which are also data independent of each other can be not only fetched but executed as well. The obvious question one might ask is why were these data and control independent instructions not percolated up enough to be together in the same basic block? Although a good compiler would try its best to achieve such percolations, it may not always be able to group these instructions together. As mentioned before, to be able to efficiently group together all data and control independent instructions, the compiler needs to have enough architected registers for proper encoding. A compiler cannot simply group any number of control and data independent instructions without inserting additional spill code. The *fork* mechanism allows the compiler to convey the underlying data independence without any *additional* spill code. As alluded to before, some of the single-thread spill code may be redundant during a future thread execution. However, due to the simultaneous execution of main and future threads, one may not gain significant additional performance by optimizing away some of this spill code.

5.7 Speculating across control dependent blocks

In the preceding discussion, *forks* have only been used for parallel execution of control independent blocks. One can further extend the notion to include control dependent blocks. This further implies the ability to do both branch paths speculatively with appropriate condition bit speculations in the *suspend* instruction. None of these speculations require further impact on the architecture, although there are additional implementation costs involved.

5.8 Simplified thread management

- *Inter-thread synchronization:* The notion of a unique *main* thread and remaining *future* threads, offers a simplified mechanism of inter-thread synchronization, implying low overhead. At explicit suspension points, future threads simply suspend themselves and wait for the main thread control to reach them. Alternatively, at different points during its execution, a future thread can attempt explicit inter-thread synchronization with any other thread. But this more elaborate inter-thread synchronization implies more hardware/software overhead.

- *Inter-thread communication:* The notions of forking with a copy of the architected machine state and the merge operation explained before, offer a mechanism of inter-thread communication with relatively low overhead. Alternative mechanisms with much higher overhead can offer explicit communication primitives which provide continuous communication protocol between active threads, for example, via messages.

- *Thread scheduling:* The mechanisms proposed in this paper which result in the *optional* nature of the *forks* (as explained before) also simplify dynamic thread scheduling, as the run-time thread scheduling hardware is not required to schedule (fork) a thread in response to a *fork* instruction. Hence, the thread-scheduling hardware does not need to be burdened with queueing and managing the future thread(s) implied by every *fork* instruction. This lowered hardware overhead of dynamic thread scheduling makes it more appealing with respect to the static thread scheduling, due to its other benefits, such as, its adaptability to different machine implementations without recompilation.

5.9 Compile-time prefetching

The primary focus in the SPSM architecture is on simultaneous and speculative fetch, decode, and execution of different regions of a program. However the implicit early fetch of code and data can be of great benefit in itself. For example, suppose the future thread *load* in Figure 2 is a cache-miss, In which case, the future thread execution may not make much progress by the merge time. The *fork* should still be considered very valuable as it provided an overlap of the associated memory access delay with useful main thread execution. In the extreme case of very poor data or instruction cache performance, each forked future thread may simply stall for memory access. But the overlap of these memory accesses should still offer significant performance improvement over the single-thread execution.

6 Compiling for Speculative Multithreading

From a compiler perspective the detection of parallelism in user programs is orthogonal to the architectural exposition of parallelism in the underlying machine. Essentially the task contains two components: program analysis to expose existing parallelism; and program restructuring to exploit the particular hardware. The same analysis techniques may be employed to identify vectorizable code, sections of code which may be run in parallel on multiple processors, or loop bodies which may be blocked and tiled to exploit memory hierarchy. In this section we describe a compiler research project which employs existing analysis techniques to produce code which exploits the SPSM architecture described previously. The goal of this research is to exploit the existing techniques and intermediate representation to extract the control and data dependence information necessary for identifying independant threads of control. We will assume that the reader is already familiar with the theoretical underpinnings of control and data dependence, and the large body of literature discussing their derivation [45, 44, 49], so the discussion will merely give a brief overview of their application in our compiler framework. For the most part the discussion will focus on the techniques and algorithms which are currently being developed for identifying and enhancing instruction level parallelism.

6.1 Program Analysis in the existing framework

The key to generating multiple threads in a single program stream, is the ability to identify control and data independant instructions and to partition the program ac-

cordingly. The compilation part of the SPSM project is a research prototype built on a version of the TOBEY compiler (Fig 3). The SPSM component of the compiler is placed at the same level as the HPF and loop transformation components. The input to this component is an abstracted form of XIL known as YIL; both the intermediate representations have been extended to incorporate the *fork-suspend* instructions described earlier.
The classical optimizations in the compiler are performed on a low level register based representation known as XIL [11]. The high order loop transformations are performed on an abstraction of this intermediate form which encapsulates precisely the information required for deriving control and data dependences [51]. Specifically, the higher level representation, known as *YIL*, represents the program as a statement graph, (Fig 4) wherein control flow, loop and array indexing information, and Static Single Assignment form are all encapsulated [52]. Most pertinent however, is the fact that each assignment statement represented in *YIL*, has a pointer to the *Computation Table* entry for the series of computations which form the lower level representation of the statement. It is this link which is the key to the success of the *YIL* abstraction, in that it provides the ability to exploit or ignore the lower level computation details as appropriate to the type of transformation or optimization being performed. Although the SPSM compiler component fits nicely into the existing framework, alongside the other high order transformation components, there are some significant differences between them which have necessitated some modifications and extensions to it. Specifically, while the SPSM component will expect considerable speedup on numeric intensive applications, its main target is commercial applications written in C and C++. This implies that the compiler can no longer focus on intra loop optimization of nicely written Fortran type do-loops. Rather we must expect to handle unstructured branching code and loops with irregular entry and exit points. Just as the *assign* statement is an abstraction of much lower level detail, so too, the *loop* statement replaces entirely the body and control elements of an *XIL* loop. Loop bodies are hierarchically linked to their corresponding loop statements, permitting easy traversal both inwards and outwards. This high level loop construct has a single successor, namely the next statement after the loop. In addition we record whether or not a particular loop exhibits *nice* properties as is the case with a stuctured *do-loop*. The *if* statement record in *YIL* encapsulates the control flow information of the program in addition to abstracting single entry single exit (*sese*) regions [50]. As with the loop statement, we have two types of *if* statement; one representing a nicely structured *if-then-else* constuct, and having a single successor which is the merge point of the paths which forked at the branch. The other *if* statement records the divergence and confluence of multiple flows of control by means of successor and predecessor fields. At some level of course, these unstructured *if* statements will be enclosed in a structured *sese* region.

6.2 Thread Extraction

Starting from a program represented in *YIL*, and its dependance graph, the process by which threads are discovered and exposed can be considered in several parts. To begin, the dependance graph is partitioned hierarchically into regions containing statements which are equivalent with respect to their control dependances. The statements within any one of these regions will all be executed when some set of predicates is satisfied. In the absence of constraints imposed by data dependances, this means that each statement within one of these partitions could be executed in its own thread. In general data dependances between the statements prevent such unfettered parallelization from taking place, but in many cases the compiler is able to find two or more groups of instructions which are independent. Threads are formed within a partition by scheduling the nodes in the partition's subgraph. This is the basic technique used by the compiler. More advanced transformations allow for the exploitation of speculative parallelism and for parallelization of the types of loop amenable to vectorization or SMP techniques.

6.2.1 Partitioning the Dependance Graph

The aim of partitioning the dependance graph is to extract regions, or partitions containing statements which are executed under precisely the same set of control predicates. That is to say, each statement within a partition control depends on exactly the same set of nodes with the same edge labels as every other statement within the partition. It is clear that for arbitrary programs this is quite a complex task, and beyond the scope of the present discussion. However, in the case of well structured programs, the problem is much more tractable. The control dependance graph, once cycles have been reduced, forms a tree in this instance. At each level in this tree all subtrees reached by edges labeled identically are members of the same partition. In more general programs, a decomposition into single-entry single-exit (*sese*) regions of the sort proposed by Johnson and Pingali ([50]) will be needed. In any case, we require that the predicates which generate a partition, together with the set of all partitions generated by these predicates, together represent a *sese* in the program. This restriction stems from the need to be able to manipulate the complete set of statements as a single unit for the purposes of scheduling. Such a unit will be referred to as a control region and can be thought of as representing a compound statement such as a *do loop* or an *if-then-else* or *case* structure (Fig 5)

6.2.2 Scheduling within a partition

Given a decomposition as described earlier, within a single partition each statement (including the compound statements represented by control regions) can, in the absence of any data dependance, be allocated to a separate thread. This simple state of affairs is seldom encountered however and so it behooves us to consider the constraints imposed on the schedule by the presence of data dependance edges. Our analysis proceeds as follows. We note that edges from other partitions to nodes within the partition being scheduled can be ignored. These edges will constrain the placement of the partition as a whole when its parent is processed. We can also dismiss loop carried edges, since these affect the ordering of temporally distinct copies of the partition, rather than imposing an order relation within a single execution of the loop. Having disposed of these, the remaining edges impose a topological order on the nodes. Assuming that all π blocks have been reduced, the subgraph under consideration must be acyclic, and thus some number of nodes have no predecessors. These nodes represent the potential start nodes of threads.

Subject to trade-offs related to the length of threads, the hardware cost of thread dispatch and other considerations, several nodes are selected to become thread headers. The remaining nodes in the region are considered in topological order and assigned to one of the threads. If a node depends on nodes from a single thread, it is assigned to that thread. A node that depends on more than one thread may be assigned to any of them, but will cause the thread that it is placed in to suspend before execution of the statements it represents.

6.2.3 Speculative Execution

It frequently happens that a node depends on data computed on one leg of a conditional statement, but would be independent if execution arrived at it via another leg. In this case when assignment of a node to a thread results in suspension, the compiler may predict that a dependance free path is taken. The suspend will then become conditional, and the thread will remain active.

It may be instructive to consider an example. A short program is shown in Fig 7. The partitioned dependance graph is given in Fig 5. All the nodes which are control dependant on **P1** (the predicate $z > 50$ is true), form a partition. All of these statements could be executed in parallel were it not that some of them consume data produced by others. The two *if* statements (the subgraphs headed by **P2** and **P3**) form control regions which are regarded as compound statements at the level of **P1**. The assignment of the nodes of this partition to threads is shown in Fig 6. Nodes **S1, S4** and **S10** data depend on no other nodes. Thus, these statements may each be placed in a different thread. The diagram shows a later stage in the scheduling process. Node **S5** has just been assigned to *Thread 2*. Since this statement depends on **R2**, the thread must be suspended prior to execution of **S5**. This is the case shown in the assembly listing (Fig 8). Another possibility in this instance would be to conditionally suspend *Thread 2* at this juncture, by speculating on the outcome of **P2**.

6.2.4 Loop Scheduling

In cases where loop parallelism exists, the compiler will perform transformations aimed at exposing this. While traditionally this type of parallelism is exposed as inner-loop parallelization (vector processors) or outer loop parallelization (SMP), the extraction of thread parallelism can be accomplished at any level within a loop nest. The compiler uses the framework of unimodular transformations [48], [46, 47] to exploit loop level opportunities.

6.2.5 Interaction with Classical Optimization

As mentioned already, the research is being done within the context of an existing established compiler. The compiler uses robust and powerful classical optimizations and sophisticated global scheduling and software pipelining techniques. The major benefit accruing from this is the ability to evaluate continually, the interaction between the new compilation techniques and existing optimizations and restructuring techniques. This is important, not only from the compiler's perspective (correctness and performance), but is useful in validating architectural concepts. Traditional optimizations can complicate the dependance structure of a program, lengthening the span of dependance arcs and merging separate dependance sources or sinks into singletons. These transformations must be inhibited where they affect correct parallel execution of the program, but in doing this we typically make the scalar execution of the program worse. In a similar vein, both global and local scheduling may have to be modified in order to preserve the viability of the generated threads, and this may decrease potential performance. The interaction of scheduling and thread creation poses many interesting questions both for the compiler and the architecture. The guidelines for making tradeoffs in all of these areas are not yet understood. The current direction of the research in progress is to explore these issues in depth.

6.3 Exposing more fine grained parallelism

Node Splitting is a technique which we employ to increase the the availability of instructions for placement in threads and thereby increasing the parallelism. Consider the following example:

```
S1 if x > 10 then
S2   c = x+y
     endif
S3 a = b+c+d
```

Ideally we would like to fork a thread to execute statement S3 while the main thread continues to execute the if block beginning at statement S1. However, since there is a true data dependance arc carried on the use of variable c in statement S3, from its definition in statement S2, we are unable to create a thread beginning at S3. Moreover, were S3 imbedded within a block of statements which formed an independant thread, we would have to insert an unconditional SUSPEND instruction before it. By splitting out the offending variable from statement S3 we find that we can increase the length of an existing thread or make available a statement (S3a) for thread creation as follows:

```
S1 if x > 10 then
S2   c = x+y
     endif
S3a t1 = b+d
S3b a = t1+c
```

We can further extend this technique to incorporate the arithmetic rules of associativity to reorder components of statements and so make instructions available for thread creation or expansion. In the following example, thread creation is hindered by the data dependance arc carried from S2 to S3 on variable b.

```
S1 if x > 10 then
S2   b = c*d+x
```

```
       . . .
       endif
  S3  a = (b+e)-f+(g*h)
```

Clearly, the node splitting transformation will increase the available parallelism to some extent, but by reassociating the components of S3 as shown below, we get much greater improvement.

```
  S1 if x > 10 then
  S2   b = c*d+x
       . . .
       endif
  S3a  t1 = g*h
  S3b  t2 = e-f
  S3c  a = b + t2+t1
```

The above examples also highlight the powerful nature of YIL alluded to previously. In essence, we are doing the analysis at a very high, almost source level, and yet, because of the underlying XIL availability, we are able to manipulate low level instructions. In fact the thread that would be created from statements S3a and S3b will contain no store instructions but only loads and arithmetic operations.

7 Previous Work

7.1 Independent Instruction Threads

A large body of research in the area of multithreading has been primarily aimed at improving the processor throughput, by sharing the processor resources with independent instruction threads, which are typically derived from independent programs or processes. The single-program response-time in such cases either stays the same, or worsens due to inter-program interferences, such as, working-set interference. This work can be classified into following two categories:

1. Only one active thread: switch to an independent thread when the current thread blocks on some long latency operation, such as, a remote memory access. Examples include APRIL [41], Weber and Gupta [13], Boothe and Ranade [14], Iannucci [15], and, Thekkath and Eggers [16].

2. Multiple active interleaved threads: Examples include, Flynn and Podvin [43], Laudon et. al. [17], Yamamoto et. al. [18], Farrens and Pleszkun [19], Daddis and Torng [20], DART processor [21], and, Park et. al. [37].

7.2 Dependent Instruction Threads

Remaining research in the area of multithreading has had a similar primary focus such as that in this paper, namely, single-program multithreading. In a more general sense, this class of research can be extended to include all work where the instruction threads are not always independent. Therefore some form of synchronization is provided for inter-thread communication.

Other architecture extension proposals: Architectural extensions, such as, *fork* and *suspend* have been proposed in the past also. For example, Prasadh and Wu [22] mention *fork-join* instructions for thread synchronization and *spawn-sync* instructions are proposed by Hum et. al. [24]. Hirata et al. [23] and Tyson et. al. [25] have proposed architectural extensions for compile-time management of special hardware queues which handle inter-thread communication. Both these proposals employ static scheduling techniques for improving loop-level parallelism. Conditional join of threads is proposed in MTA [24] and START [26] using *sync* and *cpost* instructions, both of which provide thread synchronization based on comparisons with count values. A set of additional instructions are also proposed to be added to a RISC instruction set in the START multithreaded computer project. Since the threads in START are more coarse-grained, process-level threads, these new instructions are aimed at accomplishing inter-thread communication without kernel intervention. Some of the related research is further discussed below.

- **Statically Scheduled Machines:** In contrast to the dynamic scheduling proposed in this paper where compiler has no necessary notion of machine resources, VLIW machines have traditionally been statically scheduled. Earlier VLIW machines, e.g., Multiflow [27], are single threaded. However, some recent VLIW machines have attempted to pursue multiple threads of execution. In a static scheduling technique, called selective scheduling, Moon and Ebcioglu [3] proposed simultaneous execution of both paths of independent, non-speculative, conditional branch instructions. Wolfe and Shen [28], Prasadh and Wu [22], and, Keckler and Dally [29] have both proposed adding multiple contexts to a VLIW architecture. The XIMD architecture proposed by Wolfe/Shen provides static interleaving of multiple statically scheduled threads, whereas, Prasadh/Wu and Keckler/Dally propose dynamic interleaving of instructions from multiple statically scheduled threads.

- **Dataflow-model based machines:** Several of the previously proposed multithreaded architectures are derived from or based on dataflow model of execution rather than the von Neumann control-flow model assumed in this paper. Examples include, MTA [24], Monsoon [34], [30], START [26]. There is no thread interleaving support in MTA. Support for speculative threads is stated to be under consideration for the MTA project.

- **Memory-based thread synchronization:** Denelcor HEP [31], its followup Horizon [32], and, MASA [33] provide memory-based inter-thread synchronization. Each thread has its own independent register set and there is no sharing of registers.

- **Expandable Split Window paradigm:** In this paradigm, proposed by Franklin and Sohi [35], there is a central control unit which uses dynamic branch prediction to assign basic windows (threads) to different execution units for fetch, decode, and execution. Thus thread generation in this case can be entirely at run time and inter-thread data dependencies are also handled entirely in hardware. Pnevmatikatos et. al. [38] discuss compiler-assisted thread generation for this paradigm.

None of the previously published reports in the multithreading area, to the best of our knowledge, has proposed synchronization instructions which include explicit information on compile-time control-flow speculation, as proposed in this paper. In fact, the speculation support proposed in the SPSM architecture at the thread level is similar to that proposed at the instruction-level in previously published compilation techniques called *guarding* [39, 36] and *boosting* [40], which have been discussed previously in the paper. Furthermore, none of the previous research in this area deals directly with single-program, control-dependence directed, compile-time thread generation.

8 Conclusion

Almost all current commercial processors are based on a single thread with control flow speculation. Recent limit studies on instruction-level parallel processing, based on non-numeric applications, have reported significant performance gains from speculative execution of multiple control flows, in the range of 4 to 6 times that of single thread speculation. Also note that with microprocessor transistor count doubling almost every two years, a potential exists for designing machines capable of as many as 8 to 12 instructions per cycle. However, the low IPC limits of single thread speculation would imply very poor resource utilization for such highly parallel designs. Therfore, it is worth exploring means of going past the limits of single thread execution.

In this paper we have described a research effort to explore architectural and finely tuned compilation techniques for exceeding these limits. The (SPSM) architecture, which can be viewed as an extension of any existing single-thread architecture, includes novel *fork-suspend* instructions and requires the compiler to identify the control and data independent threads which can exploit them.

Finally, we would like to emphasize that the SPSM research is an ongoing project and many aspects of compiler and machine design are still being worked out. We are carrying out cycle-by-cycle, trace-driven simulations of the proposed machine designs. The traces are derived from runs of the executables produced by the experimental compiler. Thus we can, in multiple iterations, study the performance and interaction of both the compiler and machine designs.

References

[1] G. S. Tjaden and M. J. Flynn, "Detection and parallel execution of independent instructions", *IEEE Trans. Computers*, Vol. C-19, Oct. 1970, pp. 889-895.

[2] M. D. Smith, M. Johnson, and M. A. Horowitz, "Limits on multiple instruction issue", *Proc. of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, April 1989, pp. 290-302.

[3] S. Moon and K. Ebcioglu, "On performance and efficiency of VLIW and superscalar", *Proc. International Conference on Parallel Processing*, Aug. 1993, Vol II, pp. 283-287.

[4] A. Nicolau and J. A. Fisher, Measuring the Parallelism Available for Very Long Instruction Word Architectures, *IEEE Transactions on Computers*, **C-33**(11), pp. 968-976, November 1984.

[5] M. S. Lam and R. P. Wilson, "Limits of Control Flow on Parallelism", *Proc. of the 19th Annual International Symposium on Computer Architecture*, May 1992, pp. 46-57.

[6] K. B. Theobold, G. R. Gao, and L. J. Hendren, "On the Limits of Program Parallelism and its Smoothability", *Proc. of the Twentyfifth Annual International Symposium on Microarchitecture*, December 1992, pp. 10-19.

[7] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines", *Proc. of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989, pp. 272-282.

[8] W. M. Johnson, *Superscalar Microprocessor Design* Prentice Hall, 1991.

[9] D. W. Wall, "Limits of Instruction-Level Parallelism", *Proc. of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp. 176-188.

[10] L. Gwennap, "PowerPC Becomes the Best-Selling Desktop RISC," *Microprocessor Report*, Vol. 8, No. 17, Dec. 1994, pp. 1-10.

[11] K. O'Brien, B. Hay, J. Minish, H. Schaffer, B. Schloss, A. Shepherd, and M. Zaleski, "Advanced Compiler Technology for the RISC System/6000 Architecture," *IBM RISC System/6000 Techonology*, Order No. SA23-2619, IBM Corporation, Austin, TX 78758-3493, pp. 154-161.

[12] "PowerPC and POWER2: Technical Aspects of the New IBM RISC System/6000," Order No. SA23-2737-00, IBM Corporation, Austin, TX 78758-3493.

[13] W. Weber and A. Gupta, "Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architectures: Preliminary Results," *Proc. of the 16th Annual International Symposium on Computer Architecture*, Jerusalem, Israel, June 1989, pp. 273-280.

[14] B. Boothe and A. Ranade, "Improved Multithreading Techniques for Hiding Communication Latency in Multiprocessors," *Proc. of the 19th Annual International Symposium on Computer Architecture*, May 1992, pp. 214-223.

[15] R. A. Iannucci, "Toward a Dataflow/von Neumann Hybrid Architecture," *Proc. of the 15th Annual International Symposium on Computer Architecture*, June 1988, pp. 131-140.

[16] R. Thekkath and S. J. Eggers, "The Effectiveness of Multiple Hardware Contexts," Multiprocessors and Workstations" *Proc. of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1994, pp. 328-337.

[17] J. Laudon, A. Gupta, and M. Horowitz, "Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations," *Proc. of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1994, pp. 308-318.

[18] W. Yamamoto, M. Serrano, A. Talcott, R. C. Wood, and M. Nemirovsky, "Performance Estimation of Multistreamed, Superscalar Processors," *Proc. of the 27th Hawaii International Conference on System Sciences*, Hawaii, Jan 1994, pp. 195-204.

[19] M. K. Farrens and A. R. Pleszkun, "Strategies for Achieving Improved Processor Throughput" *Proc. of the 18th Annual International Symposium on Computer Architecture*, May 1991, pp. 362-369.

[20] G. E. Daddis Jr. and H. C. Torng, "The Concurrent Execution of Multiple Instruction Streams on Superscalar Processors" *Proc. of the International Conference on Parallel Processing*, Aug. 1991, vol. I, pp. 76-83.

[21] J. Shetler and S. E. Butner, "Multiple Stream Execution on the DART Processor" *Proc. of the International Conference on Parallel Processing*, Aug. 1991, vol. I, pp. 92-96.

[22] R. G. Prasadh and C. Wu, "A Benchmark Evaluation of a Multithreaded RISC Processor Architecture" *Proc. of the International Conference on Parallel Processing*, Aug. 1991, vol. I, pp. 84-91.

[23] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and Teiji Nishizawa, "An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads," *Proc. of the 19th Annual International Symposium on Computer Architecture*, May 1992, pp. 136-145.

[24] H. H. J. Hum, K. B. Theobald, and G. R. Gao, "Building Multithreaded Architectures with Off-the-Shelf Microprocessors", *Proc. of the 8th International Parallel Processing Symposium*, Apr. 1994, pp. 288-294.

[25] G. Tyson, M. Farrens, and A. R. Pleszkun, "MISC: a Multiple Instruction Stream Computer," *Proc. of 25th Annual International Symposium on Microarchitecture*, Dec 1992, pp. 193-196.

[26] M.J. Beckerle, "Overview of the START(*T) Multithreaded Computer," *Proc. of COMPCON, Spring '93*, Feb 1993, pp. 148-156.

[27] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodham, "A VLIW Architecture for a Trace Scheduling Compiler," *IEEE Trans. on Computers*, vol. C-37, Aug. 1988, pp. 967-979.

[28] A. Wolfe and J. P. Shen, "A Variable Instruction Stream Extension to the VLIW Architecture," *Proc. of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April, 1991, pp. 2-14.

[29] S. W. Keckler and W. J. Dally, "Integrating Compile Time and Runtime Scheduling for Parallelism," *Proc. of the 19th Annual International Symposium on Computer Architecture*, May 1992, pp. 202-213.

[30] P. Lenir, R. Govindrajan, and S. S. Nemawarkar, "Exploiting Instruction-Level Parallelism: The Multithreaded Approach," *Proc. of 25th Annual International Symposium on Microarchitecture*, Dec 1992, pp. 189-192.

[31] B. Smith, "Architecture and Applications of the HEP Multiprocessor Computer Systems," *SPIE 298*, 1981, pp. 241-248.

[32] M. R. Thistle and B. J. Smith, "A Processor Architecture for Horizon," *Supercomputing '88*, Florida, Oct. 1988, pp. 35-40.

[33] R. H. Halstead Jr. and F. Fujita, "MASA: A Multithreaded Architecture for Parallel Symbolic Computing," *Proc. of the 15th Annual Int. Symp. on Comp. Arch.*, Honolulu, Hawaii, June 1988, pp. 443-451.

[34] G. M. Papadopoulos and David E. Culler, "Monsoon: An Explicit Token-Store Architecture," *Proc. of the 17th Annual Int. Symp. on Comp. Arch.*, Honolulu, Hawaii, June 1990, pp. 82-91.

[35] M. Franklin and G. Sohi, "The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism," *Proc. 19th Annual Symposium on Computer Architecture*, June 1992, pp. 58-67.

[36] D. Pnevmatikatos and G. Sohi, "Guarded Execution and Branch Prediction in Dynamic ILP Processors," *Proc. 21st Annual Symposium on Computer Architecture*, June 1994, pp. 120-129.

[37] W. W. Park, D. S. Fussel, and R. M. Jenevein "Performance Advantages of Multithreaded Processors" *Proc. of the International Conference on Parallel Processing*, Aug. 1991, vol. I, pp. 97-101.

[38] D. Pnevmatikatos, M. Franklin, and G. Sohi, "Control Flow Prediction for Dynamic ILP Processors," *Proc. of 26th Annual International Symposium on Microarchitecture*, Dec. 1993, pp. 153-163.

[39] P. Y. T. Hsu and E. S. Davidson, "Highly concurrent scalar processing", *Proc. 13th Annual Symposium on Computer Architecture*, June 1986, pp. 386-395.

[40] M. D. Smith, M .Horowitz, and M. Lam, "Efficient superscalar performance though boosting", *Fifth Int. Conference on Architectural Support for Programming Languages and Operating Systems*, Sep. 1992, pp. 248-159.

[41] A. Agarwal, B.-H. Lin, D. Kranz and J. Kubiatowicz, "APRIL: A Processor Architecture for Multiprocessing," *Proc. of the 17th Annual International Symposium on Computer Architecture*, May 1990, pp. 104-114.

[42] M. Butler, T. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, "Single Instruction Stream Parallelism Is Greater Than Two", *Proc. of the Eighteenth Annual Symposium on Computer Architecture*, May 1991, pp. 276-286.

[43] M. J. Flynn and A. Podvin, "Shared Resource Multiprocessing," *IEEE Computer*, Vol. 5, Mar./Apr. 1972, pp. 20-28.

[44] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and, F. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 4, Oct. 1991, pp. 451-490.

[45] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program dependence Graph and Its Use in Optimization," the Control Dependence Graph," *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 3, July 1987, pp. 319-349.

[46] M. E. Wolf and M. S. Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism," *IEEE Trans. on Parallel and Distributed Systems*, Vol 2, No. 4, October 1991, pp. 453-471.

[47] M. E. Wolf and M. S. Lam, "A Data Locality Optimizing Algorithm," *Proceedings of ACM SIGPLAN Conference*, Toronto, June 1991

[48] U. Banerjee, "Unimodular Transformations of Double Loops," *3rd Workshop on Languages and Compilers for Parallel Computers*, Aug. 1990.

[49] G. Goff, K. Kennedy, and C. Tseng, "Practical Dependance Testing," *Proceedings of ACM SIGPLAN Conference*, Toronto, June 1991.

[50] R. Johnson, D. Pearson, and K. Pingali, "The Program Structure Tree: Computing Control Regions in Linear Time," *Proceedings of ACM SIGPLAN Conference*, Orlando, June 1994, pp. 171-185.

[51] K. O'Brien, and K. M. O'Brien, "The Implementation of Locality Optimizations in the RS/6000 FORTRAN Compiler," *The Proceedings of the 1992 IBM Programming Languages ITL*, Toronto, June 1992.

[52] K. O'Brien, K. M. O'Brien, M. Hopkins, A. Shepherd, and R. Unrau, "XIL and YIL: The intermediate Languages of TOBEY," *The Proceedings of the 1995 ACM Workshop on Intermediate Representation*, San Francisco, Jan. 1995.

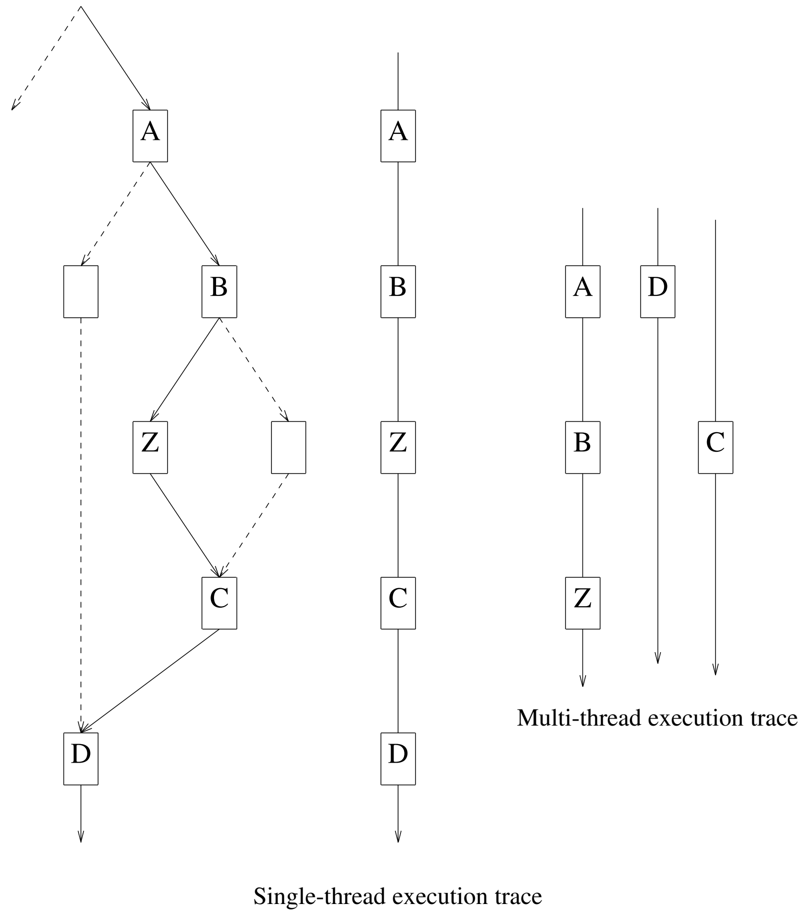

Figure 1: Single-thread versus multi-thread execution of a sample control flow graph.

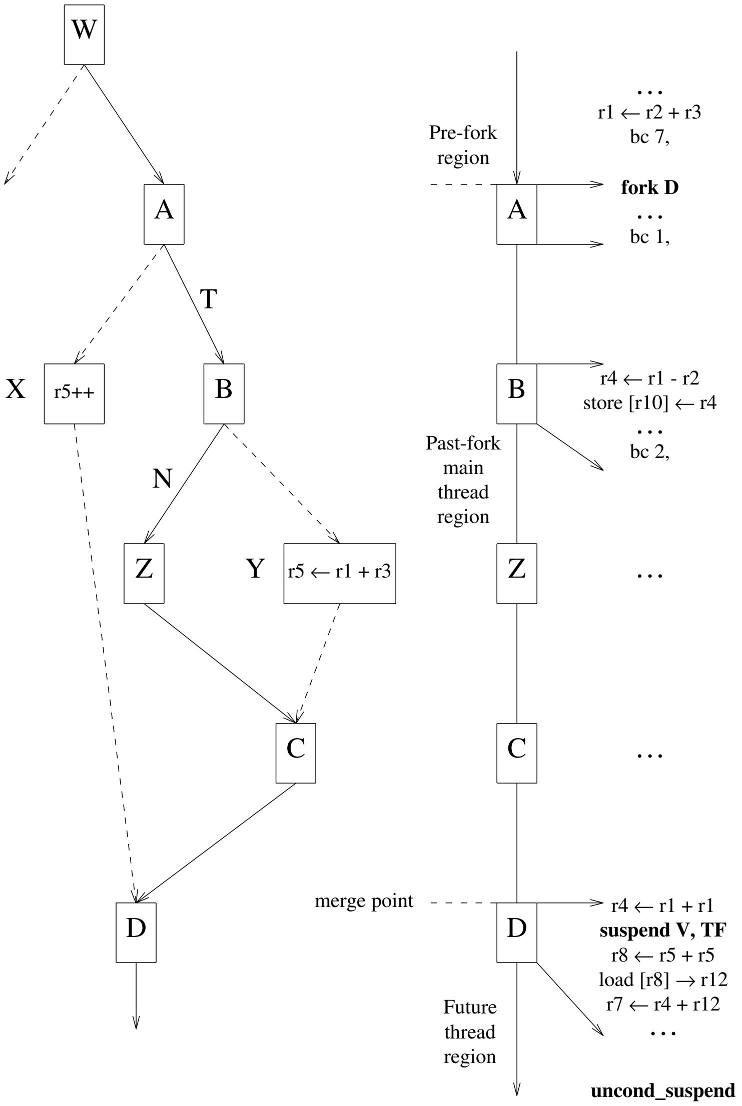

Figure 2: A sample control flow graph and associated code seqence.

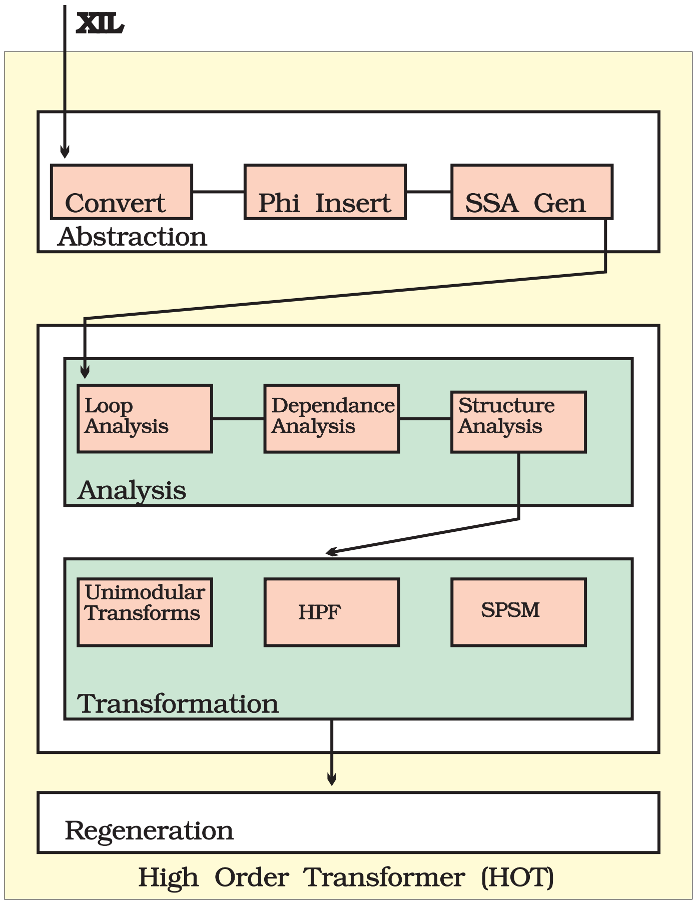

Figure 3: Structure of SPSM Compiler

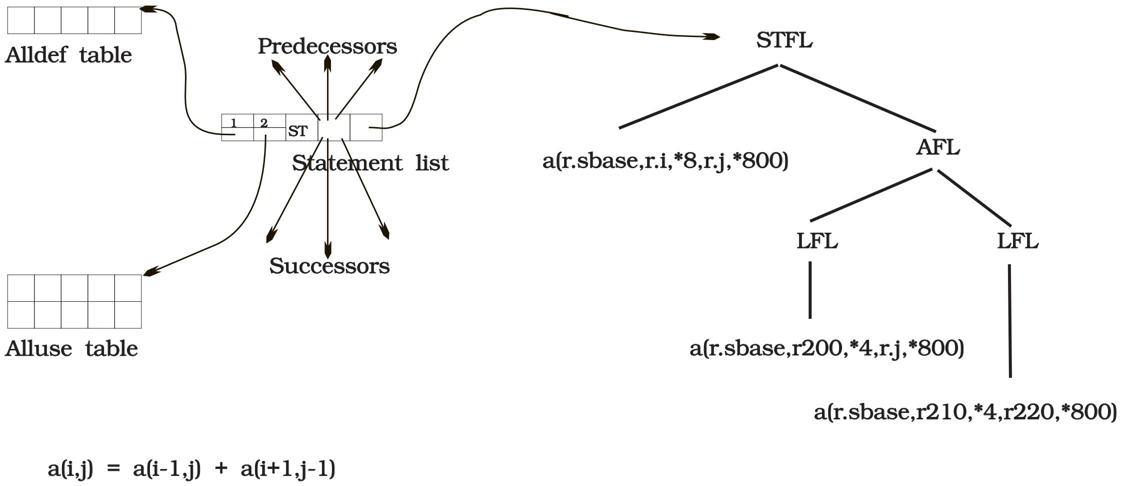

Figure 4: *YIL* Statement Graph Entry

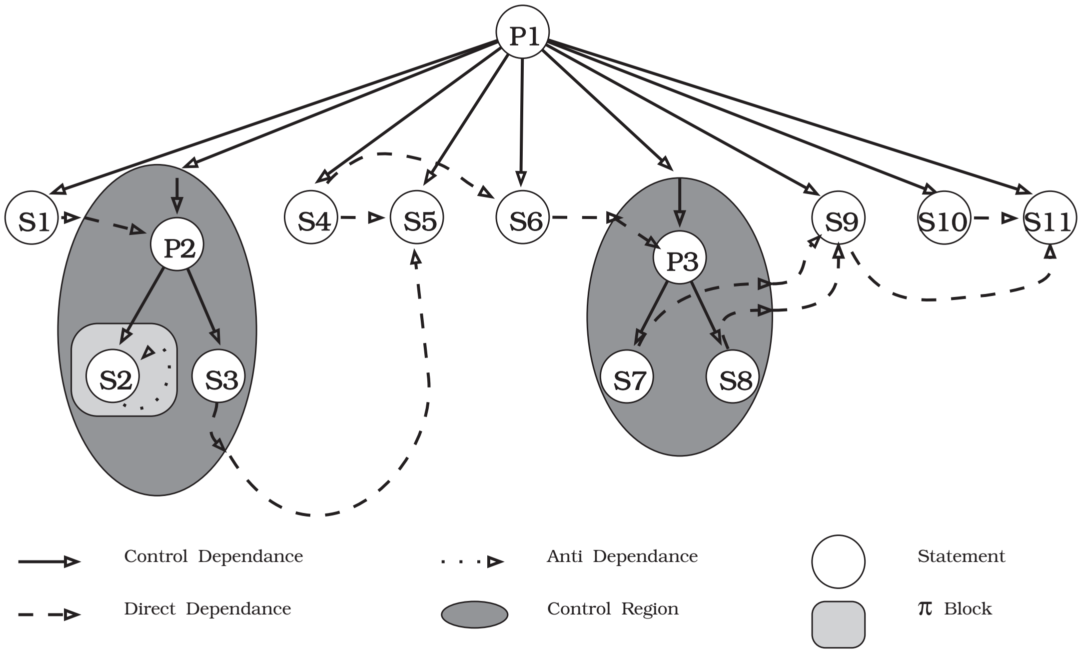

Figure 5: Control Regions and Partitions for try.c

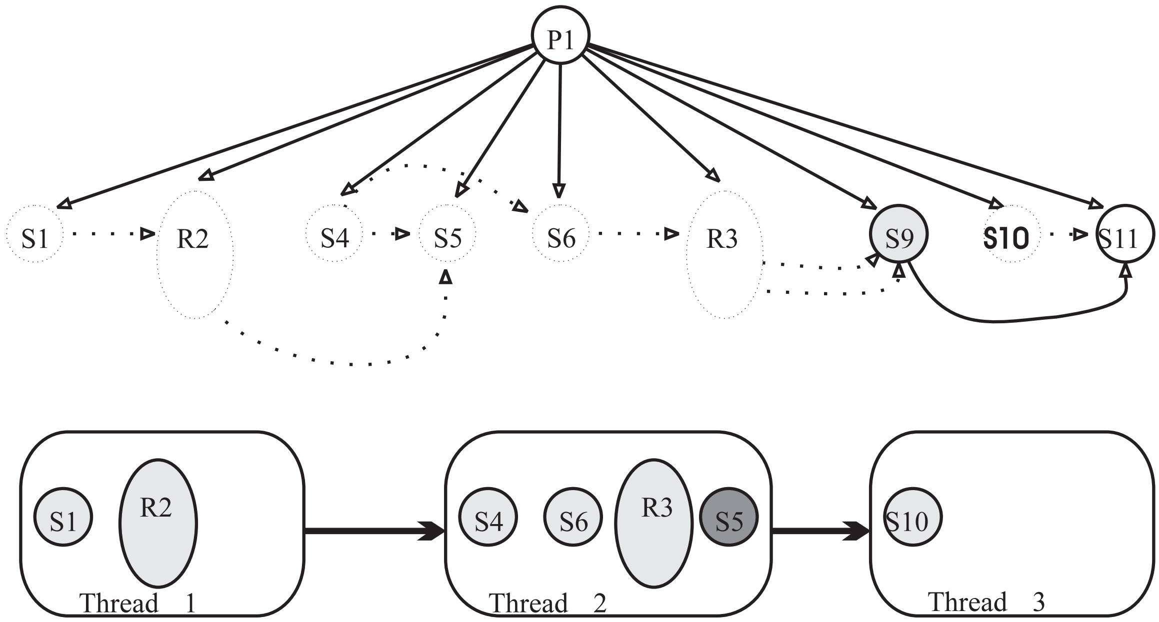

Figure 6: Assigning nodes to threads

```
if (z>50) {            P1
  a=b+c;               S1

  if (a<100)           P2
    x=x+1;             S2     R2
  else
    u=v+w;             S3

  r=s+t;  d=r+u;  e=f+r;   S4,S5,S6

  if (e>n)             P3
    g=h=i;             S7     R3
  else
    g=j+h;             S8

  m=g+l;  o=p+q;  y=m+o;   S9,S10,S11
}
```

Figure 7: Source for Try.c

```
Main Thread

   PROC
   L4A          gr3=.z(gr2,0)
   L4A          gr3=z(gr3,0)
   C4           cr0=gr3,50
   BF           CL.6,cr0,0x2/gt ,taken=50%
   FORK         CL.12
   FORK         CL.10
   L4A          gr3=.b(gr2,0)
   L4A          gr4=b(gr3,0)
   L4A          gr3=.c(gr2,0)
   L4A          gr3=c(gr3,0)
   A            gr4=gr4,gr3
   L4A          gr3=.a(gr2,0)
   ST4A         a(gr3,0)=gr4
   C4           cr1=gr4,100
   L4A          gr3=.u(gr2,0)
   BT           CL.9,cr1,0x1/lt ,taken=50%
   L4A          gr4=.v(gr2,0)
   L4A          gr0=v(gr4,0)
   L4A          gr4=.w(gr2,0)
   L4A          gr4=w(gr4,0)
   A            gr5=gr0,gr4
   ST4A         u(gr3,0)=gr5
   B            CL.3,-1
 CL.9:
   L4A          gr4=.x(gr2,0)
   L4A          gr0=x(gr4,0)
   AI           gr0=gr0,1
   ST4A         x(gr4,0)=gr0
   L4A          gr5=u(gr3,0)
 CL.3:
 CL.10:                                   Thread 1
   L4A          gr3=.s(gr2,0)
   L4A          gr4=s(gr3,0)
   L4A          gr3=.t(gr2,0)
   L4A          gr3=t(gr3,0)
   A            gr0=gr4,gr3
   L4A          gr3=.r(gr2,0)
   ST4A         r(gr3,0)=gr0
   L4A          gr3=.f(gr2,0)
   L4A          gr3=f(gr3,0)
   A            gr3=gr0,gr3
   L4A          gr4=.e(gr2,0)
   ST4A         e(gr4,0)=gr3
   UNCOND_SUSPEND
   A            gr0=gr0,gr5               dependance on u
   L4A          gr4=.d(gr2,0)
   ST4A         d(gr4,0)=gr0

   L4A          gr4=.n(gr2,0)
   L4A          gr4=n(gr4,0)
   C4           cr0=gr3,gr4
   L4A          gr3=.g(gr2,0)
   BT           CL.11,cr0,0x2/gt ,taken=50%
   L4A          gr4=.j(gr2,0)
   L4A          gr0=j(gr4,0)
   L4A          gr4=.k(gr2,0)
   L4A          gr4=k(gr4,0)
   A            gr0=gr0,gr4
   ST4A         g(gr3,0)=gr0
   B            CL.5,-1
 CL.11:
   L4A          gr4=.h(gr2,0)
   L4A          gr0=h(gr4,0)
   L4A          gr4=.i(gr2,0)
   L4A          gr4=i(gr4,0)
   A            gr0=gr0,gr4
   ST4A         g(gr3,0)=gr0              dependance on g
 CL.5:
 CL.12:                                   Thread 2
   L4A          gr3=.p(gr2,0)
   L4A          gr4=p(gr3,0)
   L4A          gr3=.q(gr2,0)
   L4A          gr3=q(gr3,0)
   A            gr4=gr4,gr3
   L4A          gr3=.o(gr2,0)
   ST4A         o(gr3,0)=gr4
   UNCOND_SUSPEND
   L4A          gr3=.l(gr2,0)
   L4A          gr3=l(gr3,0)
   A            gr0=gr0,gr3
   L4A          gr3=.m(gr2,0)
   ST4A         m(gr3,0)=gr0
   A            gr4=gr0,gr4
   L4A          gr3=.y(gr2,0)
   ST4A         y(gr3,0)=gr4
 CL.6:
   BA           lr
```

Figure 8: Assembly code for try.c, showing threads