Appeared in the 4th Annual Workshop on Interaction between Compilers and Computer Architectures, January 2000.

Limits of Task-based Parallelism in Irregular Applications

Barbara Kreaseck   Dean Tullsen   Brad Calder

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114
{kreaseck, tullsen, calder}@cs.ucsd.edu

Abstract

Traditional parallel compilers do not effectively parallelize irregular applications because they contain little loop-level parallelism due to ambiguous memory references. We explore a different source of parallelism, namely Speculative Task Parallelism (STP), where tasks are full procedures and entire natural loops. Through profiling and compiler analysis, we find tasks that are speculatively memory- and control-independent of their neighboring code. We assume a hypothetical speculative machine that parallelizes the tasks via speculative futures, allowing the tasks to be executed in parallel with preceding code when there is a high probability of independence, but no guarantee. We estimate the amount of STP in irregular applications by measuring the number of memory-independent instructions these tasks expose. Across a variety of speculation assumptions, an average of 7 to 22% of dynamic instructions are found within memory-independent tasks.

1 Introduction

Today's microprocessors rely heavily on instruction-level parallelism (ILP) to gain higher performance. Flow control imposes a limit to available ILP in single-threaded applications [11]. One way to overcome this limit is to find parallel tasks in these applications and employ multiple flows of control (*threads*). Task-level parallelism (TLP) arises when a task is independent of its neighboring code. We focus on finding some of these independent tasks and exploring the performance gains they offer.

Traditional parallel compilers exploit one variety of TLP, loop level parallelism (LLP), where loop iterations are executed in parallel. LLP can overwhelming be found in numeric, typically FORTRAN programs with regular patterns of data accesses. In contrast, general purpose integer applications, which account for the majority of codes currently run on microprocessors, exhibit little LLP as they tend to access data in irregular patterns through pointers. Without pointer disambiguation to analyze data access dependences, traditional parallel compilers cannot parallelize these irregular applications and ensure correct execution.

In this paper we explore task-level parallelism in irregular applications by focusing on *Speculative Task Parallelism* (STP), where tasks are speculatively executed in parallel under the following assumptions: 1) tasks are full procedures or entire natural loops, 2) tasks are speculatively memory-independent and control-independent, and 3) our architecture allows the parallelization of tasks via speculative futures (discussed below). Figure 1 illustrates STP, showing a task Y in the dynamic instruction stream of an irregular application that has no memory access conflicts with a group of instructions, X, that precede Y. The shorter of X and Y determines the overlap of memory-independent instructions as seen in Figures 1(b) and 1(c). In the absence of any register dependences, X and Y may be executed in parallel, resulting in shorter execution time. It is hard for traditional parallel compilers of pointer-based languages to expose this parallelism.

The goals of this paper are to identify such regions as X and Y within irregular applications and to find the number of instructions that may thus be removed from the critical path. This number represents the maximum STP when the cost of exploiting STP is zero. To facilitate our discussion, we offer the following definitions.

task A task, exemplified by Y in Figure 1, is a bounded set of instructions inherent to the application. We specifically focus on full procedures and entire natural loops.

memory-independent Two sections of code are memory-independent when neither contains a store to a memory location that the other accesses. When all load/store combinations of the type [load,store], [store,load] and [store,store] between X and Y, in Figure 1, access different memory locations, X and Y are said to be memory-independent. Thus, [ld#2, ld#3] may access the same

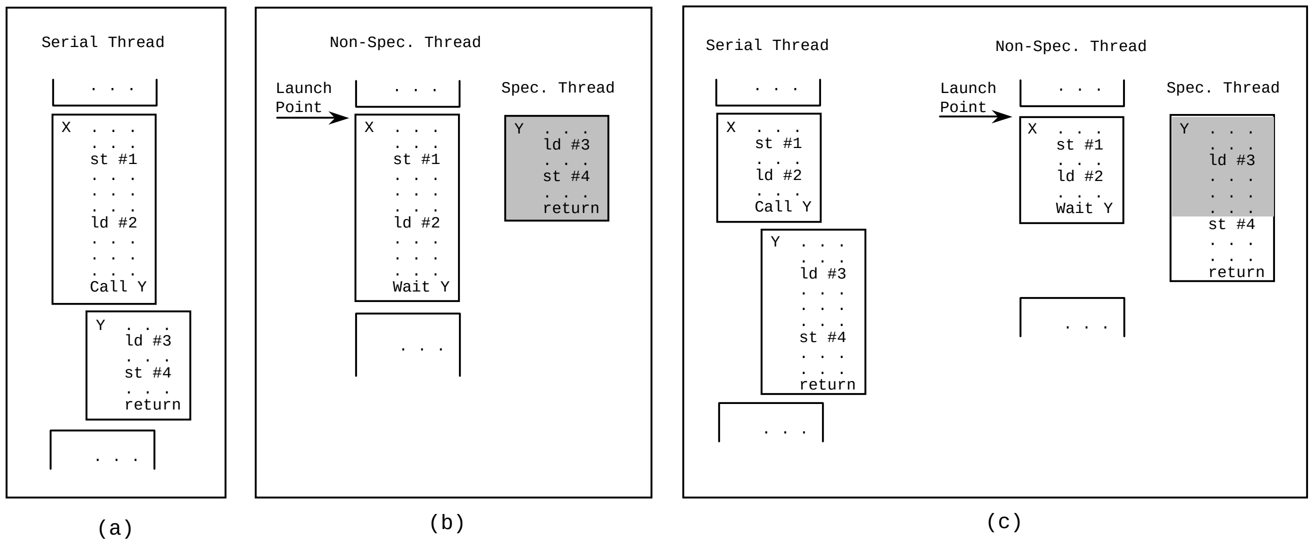

Figure 1: *STP example: (a) shows a section of the dynamic instruction stream where the task Y is known to be memory-independent of the preceding code X. (b) the shaded region shows memory- and control-independent instructions that are essentially removed from the critical path when Y is executed in parallel with X. (c) when task Y is longer than X.*

memory locations, but [st#1, ld#3] may not access the same memory locations if X and Y are to be memory-independent.

launch point A launch point is the point in the code preceding a task where the task may be initiated in parallel with the preceding code. This point is determined through profiling and compiler analysis. As the instructions in X are memory-independent of task Y in Figure 1, the launch point for Y may be at the beginning of X.

launched task A task that begins execution from an associated launch point on a different thread.

Because the biggest barrier to detecting independence in irregular codes is memory disambiguation, we identify memory-independent tasks using a profile-based approach and measure the amount of STP by estimating the amount of memory-independent instructions those tasks expose. As successive executions may differ from the profiled execution, any launched task would be inherently speculative. One way of launching a task in parallel with its preceding code is through a parallel language construct called a *future*. A future conceptually forks a thread to execute the task and identifies an area of memory in which to relay status and results. When the original thread needs the results of the futured task, it either waits on the futured task thread, or in the case that the task was never futured due to no idle threads, it executes the futured task itself.

To exploit STP, we assume a speculative machine that supports speculative futures with mechanisms for resolving incorrect speculation when the futured task is not independent. In this way, we can speculatively parallelize code when there is a high probability of independence, but no guarantee. Our work identifies launch points for this speculative machine, and estimates the parallelism available to such a machine. By varying the levels of control and memory speculation, 7 to 22% of dynamic instructions are within tasks that are found to be memory-independent. This was on the SPECint95 benchmarks, a set of irregular applications for which traditional methods of parallelization are ineffective.

In the next section we discuss the motivations for investigating STP and related work. In Section 3 we describe how we identify and quantify STP. Section 4 contains our experiment methodology and Section 5 continues with our results. Implementation issues are highlighted in Section 6, followed by our summary in Section 7.

2 Motivation and Related Works

In order to exploit Speculative Task Parallelism, a system would minimally need to include multiple flows of control and memory disambiguation to aid in mis-speculation detection. A number of these constructs are discussed below along with other research in the area of speculative parallelism.

Memory Disambiguation

There are a number of proposed structures that aid in dynamic memory disambiguation [5, 7, 6]. All minimally allow loads to be speculatively executed above stores and detect write-after-read violations that may result from that speculation.

The Address Resolution Buffer (ARB) [5] is proposed for Multiscalar processors [19] and is implemented in hardware alone with no reliance on a compiler. It allows both loads and stores to be speculatively executed by enforcing a temporal order amongst references to the same address. By storing reference data by address, the disambiguation process may be decentralized, allowing multiple disambiguation checks per cycle. The ARB stores the values of uncommitted stores, and forwards them to subsequent loads within the instruction window. Values are written in the order given by the sequential semantics of the program.

The Advance Load Address Table (ALAT), proposed in the IA64 [7, 9] architecture, allows a compiler to hoist loads above stores. The advanced load instruction inserts the source address into the ALAT. A subsequent store to that address causes the source address to be removed from the ALAT. The check load instruction is placed at a location where the load is known to be non-speculative and will issue a re-load if the source address is not found in the ALAT. An advanced load check instruction is executed immediately before each memory-ambiguous dependent instruction that has also been hoisted. If the source address is not found in the ALAT, it will branch to recovery code which issues a re-load and re-executes all the dependent instructions. The Memory Conflict Buffer [6] used in the IMPACT EPIC [3] architecture is similar to ALAT.

Multiple Flow Machines

Some multithreaded machines and multiprocessors facilitate multiple flows of control from a single program. Threaded Multiple Path Execution [25] generates threads from a single program based upon control speculation and is implemented on a Simultaneous Multithreading [23] architecture. Fast communication through a shared first level cache and fast, non-spin synchronization through hardware-based blocking locks [22] provide opportunities to exploit fine-grain thread level parallelism. The Tera MTA [2] is another multithreaded architecture that facilitates threads introduced by both the programmer (through parallel assertions), and the compiler. These threads mainly execute inductive loop iterations. It supports non-speculative futures, but currently they must be placed in the code by the programmer [4]. Tera has no memory hierarchy, and synchronization is through full/empty bits on main memory.

Hydra [8], a chip multiprocessor, uses data speculation as a source of parallel threads beyond those traditionally available in parallel codes, especially for certain loop iterations and post-procedure call codes. MIT's Multi-ALU Processor [10] uses threads garnered from examination of inner- and outer-loop iterations. Its register-based mechanisms provide improved communication and synchronization to exploit fine-grain thread level parallelism for tasks as small as 20 cycles in length. All of these architectures could exploit non-speculative TLP if the compiler exposed it, but only Hydra could support STP without alteration.

Speculative Parallelism

Our paper examines speculatively parallel tasks in non-traditionally parallel programs. Other proposed systems, displaying a variety of characteristics, also use speculation to increase parallelism. They include Multiscalar processors [19, 15, 24], Block Structured Architecture [12], Speculative Thread-level Parallelism [18, 17], Thread-level Data Speculation [21], Dynamic Multithreading Processor [1], and Data Speculative Multithreaded hardware architecture [14, 13].

In these systems, the type of speculative tasks include fixed-size blocks [12], one or more basic blocks [19], dynamic instruction sequences [21], loop iterations [18, 14], instructions following a loop [1], or following a procedure call [1, 17]. These tasks were identified dynamically at run-time [14, 1], statically by compilers [24, 12, 17], or by hand [21]. The underlying architectures include traditional multiprocessors [18, 21], non-traditional multiprocessors [19, 12, 13], and multithreaded processors [1, 14].

Memory disambiguation and mis-speculation detection was handled by an Address Resolution Buffer [19], the Time Warp mechanism of time stamping requests to memory [12], extended cache coherence schemes [17, 21], fully associative queues [1], and iteration tables [14]. Control mis-speculation was always handled by squashing the mis-speculated task and any of its dependents. While a few handled data mis-speculations by squashing, one rolls back speculative execution to the wrong data speculation [17] and others allow selective, dependent re-execution of the wrong data speculation [12, 1].

Most systems facilitate data flow by forwarding values produced by one thread to any consuming threads [19, 12, 21, 1, 14]. A few avoid data mis-speculation through synchronization [15, 17]. Some systems enable speculation by value prediction using last-value [1, 17, 14] and stride-value predictors [17, 14].

STP identifies a source of parallelism that is complimentary to that found by most of the systems above. Armed with a speculative future mechanism, these systems may benefit from exploiting STP.

3 Finding Task-based Parallelism

We find Speculative Task Parallelism by identifying all tasks that are memory-independent of the code that precedes the task. This is done through profiling and compiler analysis. We run the original executable on the profiling dataset, collecting data from memory access conflicts and control flow information. These conflicts determine proposed launch points that mark the memory dependences of a task. Then for each task, we traverse the control flow graph (CFG) in reverse control flow order to determine launch points based upon memory and control dependences. Finally, we estimate the parallelism expected from launching the tasks early. The following subsections contain the details of our approach to finding STP.

3.1 Task Selection

The type of task chosen for speculative execution directly affects the amount of speculative parallelism found in an application. Oplinger, et. al. [17], found that loop iterations alone were insufficient to make speculative thread-level parallelism effective for most programs. To find STP, we look at three types of tasks: *leaf procedures* (procedures that do not call any other procedure), *non-leaf procedures*, and entire *natural loops*. We treat loops as pseudo-procedures, allowing them to have call sites and calling routines. When profiling a combination of task types, we profile them concurrently, exposing memory-independent instructions within an environment of interacting tasks.

Although all tasks of the chosen type(s) are profiled, only those that expose at least a minimum number of memory-independent instructions are chosen to be launched early. We use this task selection threshold to identify the tasks that exhibit the most promising levels of STP. The final task selection is made after evaluating memory and control dependences to determine launch points.

3.2 Memory Access Conflicts

Memory access conflicts are used to determine the memory dependences of a task. A *memory access conflict* occurs when two load/store instructions access the same memory region. Many memory access conflicts occur during a profile, but only a subset of these are useful for calculating launch points. Useful conflicts span task boundaries and are of the form [load, store], [store, load], or [store, store]. We disregard [load, load] conflicts as they are non-destructive. We also disregard stores or loads due to register saves and restores from procedure calls. We call the conflicting instruction preceding the task the *conflict source*. The conflicting instruction within the task is called the *conflict destination*.

Specifically, when the conflict destination is a load, the conflict source will be the last store to that memory region that occurred outside the task. When the conflict destination is a store, the conflict source will be the last load or store to that memory region that occurred outside the task.

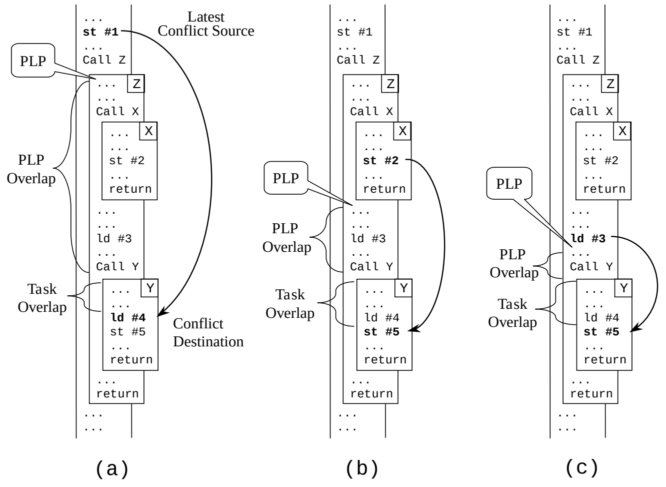

Figure 2: *PLP Locations: conflict source, conflict destination, PLP candidate, PLP overlap, and task overlap, when the latest conflict source is (a)* before the calling routine, *(b)* within a sibling task *(c)* within the calling routine.

3.3 Proposed Launch Points

The memory dependences for a task are found through profiling analysis and are marked by proposed launch points. A *proposed launch point* (PLP) represents the memory access conflict with the latest (closest) conflict source in the dynamic code preceding one execution of that task. In the absence of any intervening control dependence, a PLP may become a launch point. Exactly one PLP is found for each dynamic task execution.

In our approach, launch points for a task occur only within the task's calling region. In this way, we limit the amount and scope of executable alterations that would be needed to exploit STP. Thus, PLPs must also lie within a task's calling routine – directly before the first calling routine instruction that follows the latest conflict source.

Figure 2 contains an example that demonstrates the latest conflict sources and their associated proposed launch points. Task Z calls tasks X and Y. Y is the currently executing task in the example and Z is its calling routine. When the conflict source occurs before the beginning of the calling routine, as in Figure 2(a), the PLP is directly before the first instruction of the task Z. Although this point does not reflect a true memory dependence, it keeps a task's launch point within the calling routine. When the conflict source occurs within a sibling task or its child tasks, as in Figure 2(b), the PLP immediately follows the call to the sibling task. In Figure 2(c), the conflict source is a calling routine instruction and the PLP immediately follows the conflict source.

Two measures of memory-independence are associated with each PLP. They are the PLP overlap and the task overlap, as seen in Figure 2. The *PLP overlap* represents the number of dynamic instructions found between the PLP and

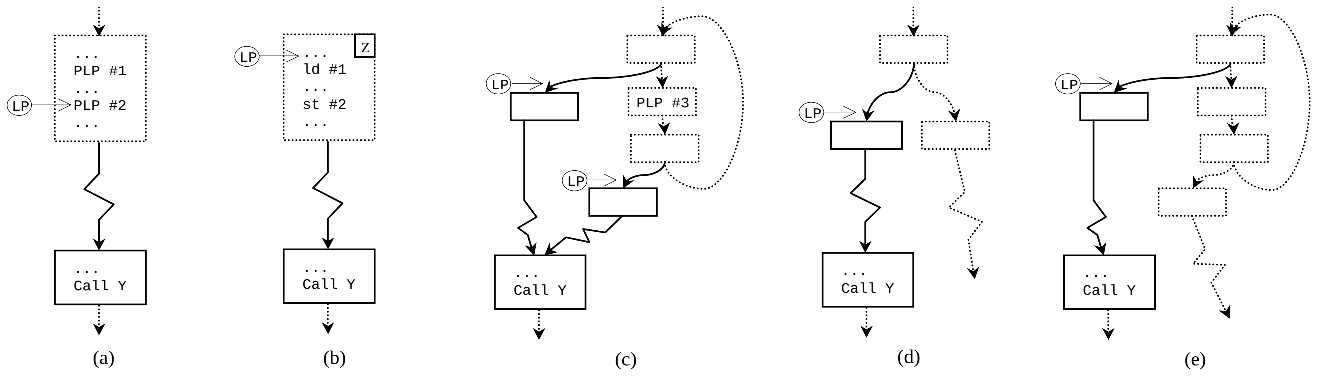

Figure 3: *Task Launch Points: Dotted areas have not been fully visited by the back-trace for task Y. (a) CFG block contains a PLP. (b) CFG block is calling routine head. (c) Loop contains a PLP. (d) Incompletely visited CFG block. (e) Incompletely visited loop.*

the beginning of the task. The *task overlap* represents the number of dynamic instructions between the beginning of the task and the conflict destination. With PLPs determined by memory dependences that are dynamically closest to the task call site, and by recording the smallest task overlap, we only consider conservative, safe locations with respect to the profiling dataset.

3.4 Task Launch Points

Proposed launch points mark profiled memory dependences. Task launch points need to be placed in consideration of both memory dependences and control dependences. Our initial approach to finding STP determines task launch points that provide two guarantees. First, *static control dependence* is preserved: all paths from a task launch point lead to the original task call site. Second, *profiled memory dependence* is preserved: should the threaded program be executed on the profiling dataset, all instructions between the task launch point and the originally scheduled task call site will be free of memory conflicts with the launched task. Variations which relax these guarantees are described in Section 5.2.

For each task, we recursively traverse the CFG in reverse control flow order starting from the original call site, navigating conditionals and loops, to identify task launch points. We use two auxiliary structures: a stalled block list to hold incompletely visited blocks, and a stalled loop list to hold incompletely visited loops. There are five conditions under which we record a task launch point. These conditions are described below. The first three will halt recursive back-tracing along the current path. As illustrated in Figure 3, we record task launch points:

a. when the current CFG block contains a PLP for that task. The task launch point is the last PLP in the block.

b. when the current CFG block is the head of the task's calling routine and contains no PLPs. The task launch point is the first instruction in the block because task launch points are only within the task's calling routine.

c. when the current loop contains a PLP for that task. Back-tracing will only get to this point when it visits a loop, and all loop exit edges have been visited. As this loop is really a sibling of the current task, task launch points are recorded at the end of all loop exit edges.

d. for blocks that remain on the stalled block list after all recursive back-tracing has exited. A task launch point is recorded only at the end of each visited successor edge of the stalled block.

e. for loops that remain on the stalled loop list after all recursive back-tracing has exited. A task launch point is recorded only at the end of each visited loop exit edge.

Each task launch point indicates a position in the executable in which to place a task future. At each task's original call site, a check on the status of the future will indicate whether to execute the task serially or wait on the result of the future.

3.5 Parallelization Estimation

We estimate the number of memory-independent instructions that would have been exposed had the tasks been executed at their launch points during the profile run. Our approach ensures that each instruction is counted as memory-independent at most once. We use a two-pass system in which, for each task, memory-independent instructions are first reserved and then possibly claimed. When a task's reservation exceeds the task selection threshold, the task is marked for STP, and the reserved memory-independent instructions are claimed. Otherwise, the reservation is forfeited, as the task exhibits little STP. We use the total number of claimed memory-independent instructions as an estimate of the limit of STP available on our hypothetical speculative machine.

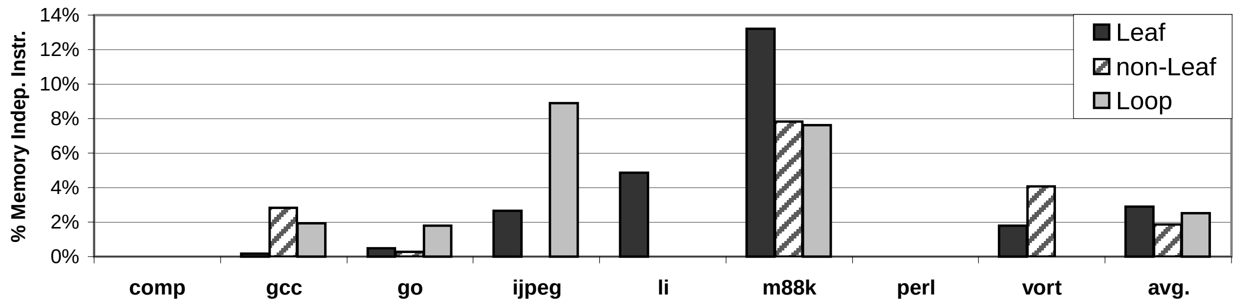

Figure 4: *Task Types: Individual data points identify memory independent instructions as a percentage of all instructions profiled and represent our starting configuration.*

4 Methodology

To investigate Speculative Task Parallelism, we used the ATOM profiling program [20], identifying natural loops as defined by Muchnick [16]. We profiled the SPECint95 suite of benchmark programs. Each benchmark was profiled for 650 million instructions. We used the reference datasets on all benchmarks except compress. For compress, we used a smaller dataset, in order to profile a more interesting portion of the application.

We measure STP by the number of memory-independent task instructions that would overlap preceding non-task instructions should a selected task be launched. We report the percentage of dynamically executed instructions that were found to be memory-independent using the following formula.

$$\text{\% Memory-Independent Instructions} = \frac{\text{number of overlapping task instructions}}{\text{number of dynamic instructions executed}}$$

The task selection threshold comprises two values: an average number of memory-independent instructions exposed per task execution and a total number of memory-independent instructions exposed across all executions of the task. For all runs, the task selection threshold was set at 25 memory-independent instructions per task execution and a total of 0.2% of instructions executed. We impose this threshold to compensate for the expected overhead of managing speculative threads and to enable allocation of limited resources to tasks exposing more STP.

Our results show a limit to STP exposed by the launched execution of memory-independent tasks. No changes, such as code motion, were made or assumed to have been made to the original benchmark codes that would heighten the amount of memory-independent instructions. Overhead due to thread creation, along with wakeup and commit, will be implementation dependent, and thus is not accounted for. Register dependences between the preceding code and the launched task were ignored. Therefore, we show an upper bound to the amount of STP in irregular applications.

5 Results

We investigated the amount of Speculative Task Parallelism under a variety of assumptions about task types, memory conflict granularity, control and memory dependences. Our starting configuration includes profiling at the page-level (that is, conflicts are memory accesses to the same page) with no explicit speculation and is thus our most conservative measurement.

We first look to see which task types exhibit the most STP. We then explore various explicit speculation opportunities to find additional sources of STP. Finally, we investigate any additional parallelism that might be exposed by profiling at a finer memory granularity.

5.1 Task Type

We specifically chose to investigate leaf procedures, non-leaf procedures, and entire natural loops. We profiled these three task types to see if any one task type exhibited more STP than the others. As Figure 4 shows, Li and m88ksim get their largest amount of parallelism from leaf procedures, gcc and vortex from non-leaf procedures, while loops account for the most parallelism in go and ijpeg. On average, a total of 7.3% of the profiled instructions were identified as memory independent, with task type contributions differing by less than 1% of the profiled instructions. This strongly suggests that all task types should be considered for exploiting STP. Each of the succeeding experiments include all three task types in their profiles.

5.2 Explicit Speculation

The starting configuration places launch points conservatively, with no explicit control or memory dependence spec-

```
<< Conflict Source 100%
intsr-1
instr-2
if (cond-1) { // 90% taken
   instr-4
   instr-5
   if (cond-2) { // 90% taken
      instr-7
      instr-8
      if (cond-3) { // 100% taken
         instr-10
         instr-11
         call task A
      } else {
         instr-13
      }
   } else {
      instr-14
   }
} else {
   instr-15
}
instr-16
```

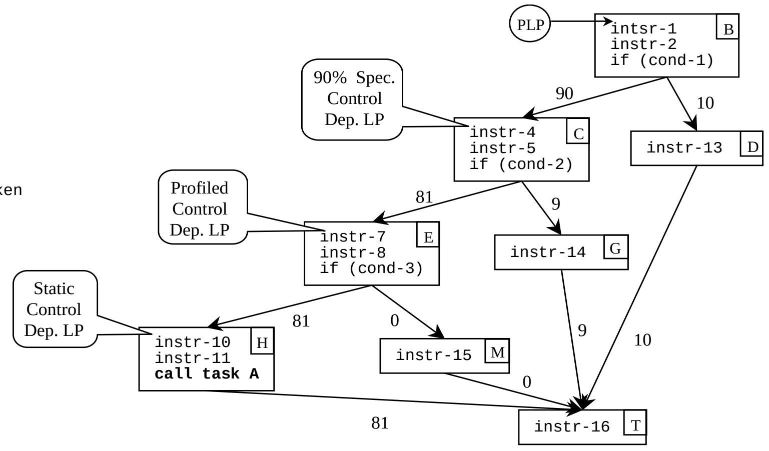

Figure 5: *Control Dependence Speculation: In the code above, the call to task A is nested within three if statements. Static control dependence will prohibit the task to be launched outside of the inner if statement. Using the profile information that the conditions are true 90%, 90%, and 100%, respectively, profiled control dependence and speculative control dependence free the task to be launched outside of the inner if statement. The corresponding CFG displays the launch points as placed by each type of control dependence. The edge frequencies reflect that the code was executed 100 times.*

ulation. Because launched tasks will be implicitly speculative when executed with different datasets, our hypothetical speculative machine needs no extra hardware to support explicit speculation. We explore the level of STP exhibited by explicit speculation, first, by speculating on control dependence, where the launched task may not actually be needed. Next, we speculate on memory dependence, where the launched task may not always be memory-independent of the preceding code. Finally, we speculate on both memory and control dependences by exploiting the memory-independence of the instructions within the task overlap.

5.2.1 Control Dependence

Our starting configuration determines task launch points that preserve **static control dependences**, such that *all paths* from the task's launch points lead to the original task call site. Thus, a task that is statically control dependent upon a condition whose outcome is constant, or almost constant, throughout the profile, will not be selected, even though launching this task would lead to almost no mis-speculations. We considered two additional control dependence schemes that would be able to exploit the memory-independence of this task.

Profiled control dependences exclude any static control dependences that are based upon branch paths that are never traversed. When task launch points preserve profiled control dependences, *all traversed paths* from the launch points lead to the original call site. A traversed path is one in which all edges along the path have a profiled frequency count greater than zero.

When task launch points preserve **speculative control dependences**, *all frequently traversed paths* from the launch points lead to the original call site. The amount of speculation is controlled by setting a minimum frequency percentage, *c*, to a value between 0 and 100. For example, when *c* is set to 90, then at least 90% of the traversed paths from the launch points would lead to the original call site.

In Figure 5, the call statement of task A is statically control dependent on all three if-statements. The profile for task A reveals a single PLP at instr-1 and a frequency count of 100 for the entire code segment. The profile also reveals that the three if-conditions had true outcomes 90%, 90%, and 100% of the time, respectively. The corresponding CFG in Figure 5 highlights the launch points as determined by the three control dependence options. All paths beginning with block H, all traversed paths beginning with block E, and 90% of the traversed paths beginning with block C lead to the call of task A. Therefore, the static control dependence launch point is before block H, the profiled control dependence launch point is before block E, and with *c* set to 90, the speculative control dependence launch point is before block C.

The price of using speculative control dependences will be the waste of resources used to speculatively initiate a launched task during an execution of the calling routine in which the executed path does not lead to the task call site. These extra launches can be squashed at the first mis-speculated conditional. In Figure 5, using profiled control dependence, squash instructions would be placed at the beginning of block M. Using speculative control dependence at *c* = 90, squash instructions would also have to be placed at the beginning of block G. Our performance estimation does not impose any penalty for mis-speculation.

The first three bars per benchmark in Figure 6 show the effect of control dependence speculation on the amount

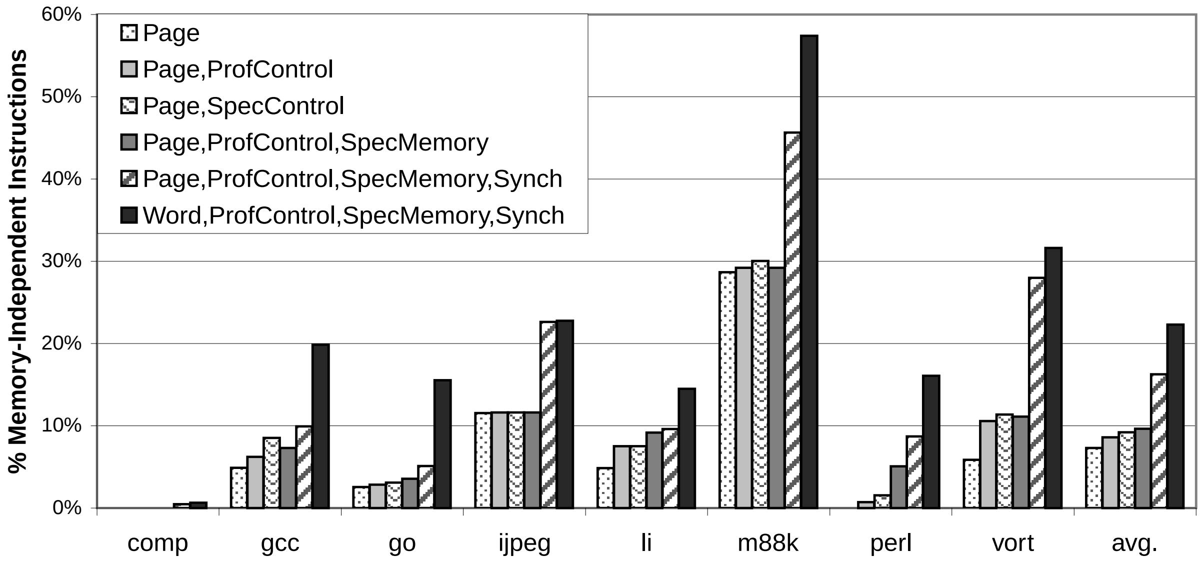

Figure 6: *Memory-independent instructions reported as a percentage of all instructions profiled.* Page = *Page-level profiling,* Word = *Word-level profiling,* ProfControl = *Profiled control dependence,* SpecControl = *Speculative control dependence,* SpecMemory = *Speculative memory dependence,* Synch = *Early start with synchronization.*

of memory independent instructions exposed. The first bar displays static control dependence. Profiled control dependence is shown by the second bar, and the third bar displays speculative control dependence at $c = 90$. Vortex and li get their greatest individual increase in parallelism from the use of profiled control dependence, while gcc benefits the most from the use of speculative control dependence. On the average, profiled control dependence exposed an additional 1.3% of dynamic instructions as memory-independent, while speculative control dependence only exposed an additional 0.6% over profiled.

The choice of using profiled or speculative control dependence will be influenced by the underlying architecture. Architectures in which threads compete for resources may be adversely affected by the unnecessary work introduced by wrong speculation. Architectures that allow threads to have dedicated resources should benefit from speculation so long as there are idle threads for speculation. Because speculative control dependence showed such a slight gain at the cost of greater anticipated overhead, we chose to proceed with profiled control dependence as the basis for the remaining results.

5.2.2 Memory Dependence

Memory dependence provides another opportunity for explicit speculation. Our starting configuration determines launch points that preserve **profiled memory dependences** such that all instructions between each launch point and its associated original task call site are memory-independent of the task for each execution of the task during the profile. This approach results in a conservative, but still speculative, placement of launch points. Thus, when a store instruction immediately before the task call site is a conflict source once out of 100 task executions, that task will not be selected even though 99 times out of 100 it would be safe to launch the task early.

We also consider the less conservative approach of determining task launch points by **speculative memory dependences**, which ignores profiled memory conflicts that occur infrequently. The amount of speculation is controlled by setting a minimum frequency percentage, m, to a value between 0 and 100. For example, when m is set to 90, then at least 90% of the traversed paths from the task launch points to the original call site would be memory-independent of the task. In other words, at least 90% of the tasks launched from these points are expected to be correctly speculated. Using speculative memory dependences is especially attractive when PLPs are far apart, and the ones nearest the task call site seldom cause a memory conflict.

We examined the effect of task launch points that preserve speculative memory dependence at $m = 90$. The fourth bar per benchmark in Figure 6 shows those results. Speculative memory dependence provides small increases in parallelism for gcc, go, li, and vortex. Perl exhibited the largest increase by exposing an additional 4% of instructions as memory-independent over profiled memory dependence. Despite the small gains, we include task launch points determined by speculative memory dependence for the remaining results.

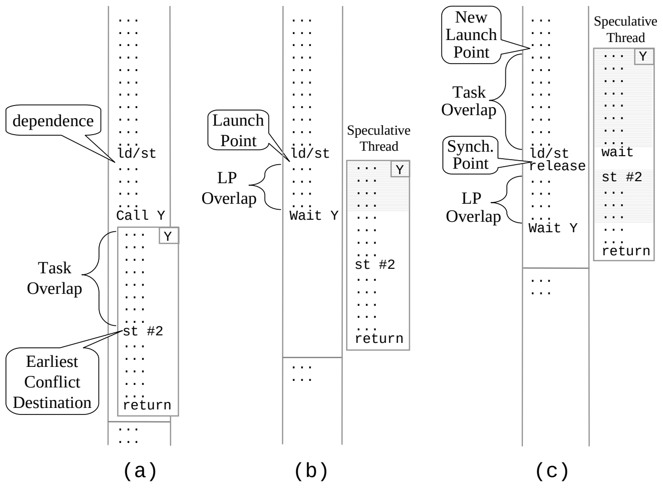

Figure 7: *Synchronization Points: Gray areas represent memory-independent instructions. (a) a task with a large amount of task overlap on a serial thread. (b) When the dependence is used as a launch point, only LP overlap contributes to memory-independence. (c) By synchronizing the dependence with the earliest conflict destination, both LP overlap and task overlap contribute to memory-independence.*

5.2.3 Early Start with Synchronization

Up to this point, by placing launch points (futures) at control dependences or memory dependences (PLPs), we have used the limited synchronization inherent within futures to synchronize these dependences with the beginning of the speculative task. This limits the amount of STP that we have been able to expose to the number of dynamic instructions between the launch point and the original task call site, which we call the *LP overlap*. The instructions represented by the task overlap, between the beginning of the speculative task and the earliest profiled conflict destination, are profiled memory-independent of all of the preceding code. By using explicit additional synchronization around the earliest profiled conflict destination, early start with synchronization enables the task overlap to contribute to the number of exposed memory-independent instructions.

Currently, a task with a large task overlap and a small LP overlap, would not be selected as memory-independent, even though a large portion of the task is memory-independent with its preceding code. By synchronizing the control or memory dependence with the earliest conflict destination, the task may be launched earlier than the dependence. Where possible, we placed the task launch point above the dependence a distance equal to the task overlap.

Figure 7 illustrates synchronization points. Figure 7(a) shows task Y with dependence, earliest conflict destination, and task overlap. When the dependence determines a launch point, in Figure 7(b), all memory-independent instructions come from the LP overlap. Figure 7(c) shows that by synchronizing the dependence with the earliest conflict destination, both the LP overlap and the task overlap contribute to the number of memory-independent instructions. When the average dynamic instruction length of task Y is larger than the LP overlap, synchronization points expose an additional number of memory-independent instructions.

Early start shows the greatest increase in parallelism so far, exposing on the average an additional 6.6% of dynamic instructions as memory-independent. The results are shown in the fifth bar per benchmark of Figure 6. M88ksim and vortex each show an increase in memory independent instructions above 16% and ijpeg over 11%. The big increase in parallelism came from tasks that had not previously exhibited a significant level of STP. With the additional memory-independent instructions that early start exposed, tasks that had not previously been selected as memory-independent rose above the threshold of 25 memory-independent instructions, and account for the extra exposed parallelism.

The extra parallelism exposed through early start will come at the cost of additional dynamic instructions and the cost of explicit synchronization. We did not impose any penalties to simulate those costs as they will be architecture-dependent.

5.3 Memory Granularity

We previously defined a memory access conflict to be when two load/store instructions access the same memory region. The memory granularity has a large effect on the amount of parallelism that is exposed. Typical memory granularities are full bytes, words, cache-lines, or pages. The larger the memory granularity, the higher the probability that two different memory addresses would access the same memory region. When a larger memory granularity is used, this may result in a conservative placement of launch points when the recorded memory access conflict only conflicts in the memory region and not the actual address. The actual granularity chosen will depend on the granularity at which the processor can detect memory ordering violations. Managing profiled-parallel tasks whose launch points were determined with a memory granularity of a page would allow the use of existing page protection mechanisms to detect and recover from dependence violations. Profiling at a small memory granularity would be much slower and involve more overhead in terms of memory. For these reasons, our starting configuration used page-level profiling. We also investigated word-level profiling.

In Figure 6, the last bar per benchmark shows the results of word-level profiling on top of profiled control dependence, speculative memory dependence and early start. M88ksim showed the largest gain in memory-independence at just about 12%, with go and gcc around 10%. The av-

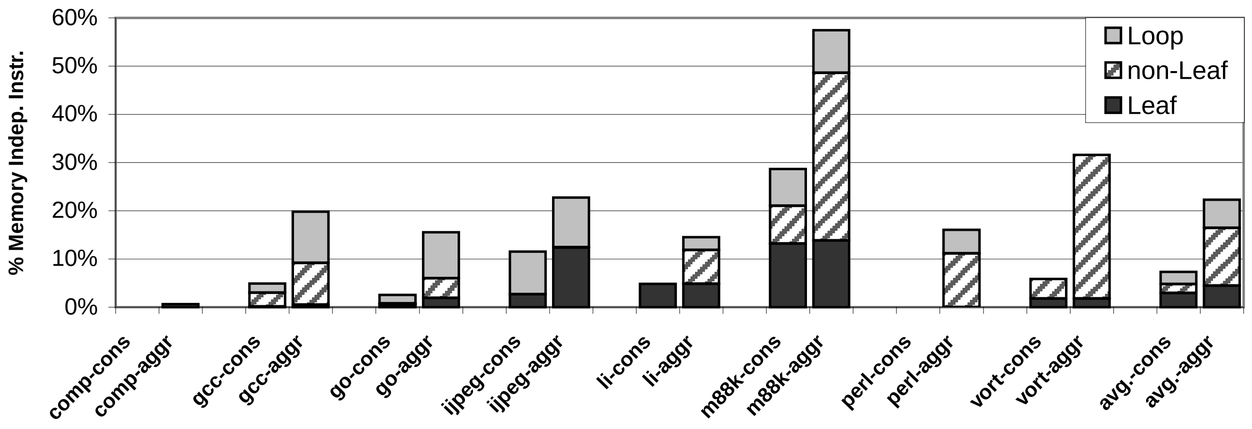

Figure 8: *Conservative vs. Aggressive Speculation: Conservative is page-level profiling on all task types with static control dependence and profiled memory dependence. Aggressive is word-level profiling on all task types with profiled control dependence, speculative memory dependence, and early start.*

	Number of Selected Tasks	Avg. Dynamic Task Length	Average Overlap
comp-cons	0	0	0
comp-aggr	1	282	42
gcc-cons	22	412	93
gcc-aggr	74	363	89
go-cons	12	546	118
go-aggr	49	334	76
ijpeg-cons	5	803	300
ijpeg-aggr	12	996	266
li-cons	3	50	50
li-aggr	9	14198	55
m88k-cons	5	34	29
m88k-aggr	20	123	43
perl-cons	0	0	0
perl-aggr	10	515	42
vort-cons	14	114	45
vort-aggr	92	215	47
avg.-cons	8	245	79
avg.-aggr	33	2128	83

Table 1: *Task Statistics (Conservative vs. Aggressive)*

erage gain in memory-independence across all benchmarks was about 6% of dynamic instructions.

5.4 Experiment Summary

Figure 8 re-displays the extremes of our STP results from conservative to aggressive speculation broken down by task types. The conservative configuration includes page-level profiling on all task types with static control dependence and profiled memory dependence. The aggressive configuration comprises word-level profiling on all task types with profiled control dependence, speculative memory dependence, and early start. M88ksim showed the largest increase in the percentage of memory-independent instructions at over 28%, with vortex very close at over 25%, and the average across benchmarks at about 14%.

Each of these increases in parallelism were largely seen in the non-leaf procedures. Ijpeg was the only benchmark to see a sizable increase contributed by leaf procedures. Loops accounted for increases in gcc, go, li and perl. Although launching loops will introduce overhead due to an additional call structure, having the ability to launch all three task types will allow irregular applications that are rich in only one type to exhibit STP.

Table 1 displays task statistics from the conservative and aggressive speculation. The number of selected tasks indicates the number of tasks that exhibited parallelism above the threshold value. The average dynamic task length is the sum of the task instruction length for each dynamic invocation of a selected task divided by the total number of dynamic invocations of the selected tasks. The average overlap is the sum of the memory-independent task instructions for each dynamic invocation of a selected task divided by the total number of dynamic invocations of the selected tasks. The number of tasks selected for STP is greatly affected by aggressive speculation. Neither the average dynamic task length nor the average overlap correlate with aggressive speculation, as they are more affected by the dynamic frequency of each task.

As seen in Figure 6, early start with synchronization provided the highest single increase among all alternatives. This may indicate that speculative systems with fast synchronization will be able to exploit STP the most effectively. Word-level profiling was not far behind, showing that a low-overhead word-level scheme to exploit STP would be profitable.

Overhead specific to any architecture exploiting STP will dictate the net gain from any of the alternatives that we presented. It could be that the extra gain in memory independent instructions from word-level profiling will, or will not, outweigh the induced complexity of implementation. In fact, every alternative that relaxes the initial configuration has a similar trade-off.

6 Implementation Issues

For our limit study of Speculative Task Parallelism, we have assumed a hypothetical speculative machine that supports speculative futures with mechanisms for resolving incorrect speculation. When implementing this machine, a number of issues need to be addressed.

Speculative Thread Management

Any system that exploits STP would need to include instructions for initialization, synchronization, communication and termination of threads. As launched tasks may be speculative, any implementation would need to include speculative thread management to handle mis-speculations.

Managing speculative tasks would include detecting load/store conflicts between the preceding code and the launched task, buffering stores in the launched task, and checking for memory-independence before committing the buffered stores to memory. One conflict detection model includes tracking the load and store addresses in both the preceding code and the launched task. The amount of memory-independence accommodated by this model will be determined by the size and access of load-store address storage, and the conflict granularity.

Another conflict detection model uses a system's page-fault mechanism. When static analysis can determine the page access pattern of the preceding code, the launched task is given restricted access to those pages, while the preceding code is given access to only those pages. Any page access violation would cause the speculative task to fail. This model, while low in overhead, finds lower amounts of memory-independence due to a restricted area of applicability and a coarse conflict granularity.

Inter-thread Communication

Any implementation that exploits STP will benefit from a system with fast communication between threads. At the minimum, inter-thread communication is needed at the end of a launched task and when the task results are used. Fast communication would be needed to enable early start with synchronization, which exposed a significant number of memory-independent instructions. The ability to quickly communicate a mis-speculation would reduce the number of instructions that are issued but never committed. This is especially important for systems where threads compete for the same resources.

Adaptive STP

We selected tasks that exhibited STP based upon memory access profiling and compiler analysis. The memory access pattern from one dataset may or may not be a good predictor for another dataset. Two feedback opportunities arise that may be used to mitigate the vagaries of predictive profiling by allowing the membership of the launched task set to adapt with the current execution. The first feedback comes from the frequency with which a task is mis-speculatively launched. A high failure rate would indicate a poor predictive relationship between datasets. When the failure rate of launching a task passes a threshold value, further launching of that task could be suspended, allowing the original serial call to take effect, with no further need for tracking loads and stores in the preceding code.

The second feedback opportunity is found in continuous profiling. Rather than have a single dataset dictate the launched tasks for all subsequent runs, let datasets from all previous runs dictate the launched tasks for the current run. It is possible that the aggregate information from the preceding runs would have a better predictive relationship with future runs, than is available from a single dataset. Additionally, the profiled information from the current run could be used to supersede the profiled information from previous runs, with the idea that the current run may be its own best predictor.

7 Summary

Traditional parallel compilers do not effectively parallelize irregular applications because they contain little loop-level parallelism due to ambiguous memory references. We focus on a different source of parallelism, namely Speculative Task Parallelism. STP arises when a task (either a leaf-procedure, a non-leaf procedure or an entire loop) is control- and memory-independent of its preceding code, and thus could be executed in parallel. To exploit STP, we assume a hypothetical speculative machine that supports speculative futures (a parallel programming construct that executes a task early on a different thread or processor) with mechanisms for resolving incorrect speculation when the task is not, after all, independent. This allows us to speculatively parallelize code when there is a high probability of independence, but no guarantee.

Through profiling and compiler analysis, we find memory-independent tasks that have no memory conflicts with their preceding code, and thus could be speculatively executed in parallel. We estimate the amount of STP in an irregular application by measuring the number of memory-independent instructions these tasks expose. We vary the level of control dependence and memory dependence to investigate their effect on the amount of memory-independence we found. We profile at different memory granularities and introduced synchronization to expose higher levels of memory-independence.

We find that no one task type exposed significantly more memory-independent instructions, which strongly suggests that all three task types should be profiled for STP. We also find that starting a task early with synchronization around dependences exposed the highest additional amount of memory-independent instructions, an average across the SPECint95 benchmarks of 6.6% of profiled instructions. Profiling memory conflicts at the word-level shows a similar gain in comparison to page-level profiling. Speculating beyond profiled memory and static control dependences shows the lowest gain, and is modest at best. Overall, we find that 7 to 22% of instructions are within memory-independent tasks. The lower amount reflects tasks launched in parallel from the least speculative locations.

8 Acknowledgments

This work has been supported in part by DARPA grant DABT63-97-C-0028, NSF grant CCR-980869, equipment grants from Compaq Computer Corporation, and La Sierra University.

References

[1] H. Akkary and M. Driscoll. A dynamic multithreading processor. In *31st International Symposium on Microarchitecture*, Dec. 1998.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. *1990 International Conf. on Supercomputing*, June 1990.

[3] D. August, D. Connors, S. Mahlke, J. Sias, K. Crozier, B. Cheng, P. Eaton, Q. Olaniran, and W. Hwu. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, June 1998.

[4] D. Callahan and B. Smith. A future-based parallel language for a general purpose highly-parallel computer. In *Languages and Compilers for Parallel Computing*, 1990.

[5] M. Franklin and G. S. Sohi. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, May 1996.

[6] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of the 6th International Conference on Architecture Support for Programming Languages and Operating Systems*, Oct. 1994.

[7] L. Gwennap. Intel discloses new IA-64 features. *Microprocessor Report*, Mar. 8 1999.

[8] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. *ACM SIGPLAN Notices*, 33(11):58–69, November 1998.

[9] Intel Corp. / Hewlet-Packard Co. *IA-64 Application Instruction Set Architecture Guide, Revision 1.0*, 1999.

[10] S. Keckler, W. Dally, D. Maskit, N. Carter, A. Chang, and W. Lee. Exploiting fine-grain thread level parallelism on the MIT Multi-ALU processor. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, pages 306–317, June 1998.

[11] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA-92)*, May 1992.

[12] R. H. Litton, J. A. D. McWha, M. W. Pearson, and J. G. Cleary. Block based execution and task level parallelism. In *Australasian Computer Architecture Conference*, Feb. 1998.

[13] P. Marcuello and A. Gonzlez. Clustered speculative multithreaded processors. In *Proceedings of the [ACM] International Conference on Supercomputing*, June 1999.

[14] P. Marcuello and A. Gonzlez. Exploiting speculative thread-level parallelism on a SMT processor. In *Proceedings of the International Conference on High Performance Computing and Networking*, April 1999.

[15] Andreas Moshovos, Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, June 1997.

[16] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publ., San Francisco, 1997.

[17] J. Oplinger, D. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT99)*, October 1999.

[18] J. Oplinger, D. Heine, S. Liao, B. A. Nayfeh, M. S. Lam, and K. Olukotun. Software and hardware for exploiting speculative parallelism with a multiprocessor. Technical Report CSL-TR-97-715, Stanford University, Computer Systems Laboratory, 1997.

[19] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA-95)*, June 1995.

[20] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. Research Report 94.2, COMPAQ Western Research Laboratory, 1994.

[21] J. Steffan and T. Mowry. The potential of using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, Feb. 1998.

[22] D. Tullsen, J. Lo, S. Eggers, and H. Levy. Supporting fine-grained synchronization on a simultaneous multithreaded processor. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, Jan. 1999.

[23] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA-95)*, June 1995.

[24] T. N. Vijaykumar and G. S. Sohi. Task selection for a multiscalar processor. In *31st International Symposium on Microarchitecture*, Dec. 1998.

[25] S. Wallace, B. Calder, and D. Tullsen. Threaded multiple path execution. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, June 1998.