Exploiting Speculative TLP in Recursive Programs by Dynamic Thread Prediction

Lin Gao [1], Lian Li [1], Jingling Xue [1] and Tin-Fook Ngai [2]

[1] University of New South Wales, Sydney, Australia
[2] Microprocessor Technology Lab, Intel

Abstract. Speculative parallelisation represents a promising solution to speed up sequential programs that are hard to parallelise otherwise. Prior research has focused mainly on parallelising loops. Recursive procedures, which are also frequently used in real-world applications, have attracted much less attention. Moreover, the parallel threads in prior work are statically predicted and spawned. In this paper, we introduce a new compiler technique, called *Speculative Parallelisation of Recursive Procedures* (SPRP), to exploit speculative TLP (thread-level parallelism) in recursive procedures. SPRP combines a dynamic thread-spawning policy and a live-in prediction mechanism in a single helper thread that executes a distilled version of a procedure on a dedicated core. It serves to predict both the invocation order of recursive calls and their live-ins in concert and dispatches these calls to the other cores in a multicore system for parallel execution. To our knowledge, SPRP is the first compiler technique to speculatively parallelise recursive procedures this way. Compared with existing static thread prediction techniques, dynamic thread prediction reduces the number of useless threads spawned, and consequently, misspeculation overhead incurred. Our preliminary results demonstrate that this technique can speedup certain recursive benchmarks that are difficult to parallelise otherwise.

1 Introduction

Parallelisation of sequential programs has been an on-going research area. Prior work has focused mainly on loops. Recursive procedures, which are also frequently used in real-world applications, have attracted much less attention.

When call sites in a recursive procedure are data-independent (as in many divide-and-conquer algorithms), techniques for their automatic parallelisation exist [20, 22, 10, 24, 21]. Such techniques have demonstrated performance advantages in achieving task-level parallelism among independent calls in regular programs and even irregular programs when they are either augmented with dependence-related programmer annotations or written in a certain programming style, e.g., component-based programming. Also, parallel programming languages such as those discussed in [5, 4] allow a concise specification of parallel algorithms on irregular data; but they rely entirely on the domain-expert programmer to expose the parallelism by identifying the tasks that can safely be executed in parallel. However, when dependence analysis is inconclusive and user/programmer involvements are unavailable, the potential presence of dependences will limit parallelism to be exploited.

Speculative multithreading (SpMT) processors [15, 18, 19, 1, 13, 9] enable the compiler to apply speculative parallelisation to optimistically create parallel threads for a sequential program without having to prove they are independent. The basic idea is to speculate on the absence of certain data/control dependences to expose more speculative TLP (thread-level parallelism) at the cost of small misspeculation penalties [30, 23, 3, 17, 27, 11, 18, 1, 26, 8, 25, 29]. So far research efforts have been largely devoted to extracting speculative TLP from loops. A few attempts have been made to speculatively parallelise whole programs [12, 15, 23, 3, 11, 6, 1]; but they are not designed to maximally exploit speculative TLP in recursive procedures. Moreover, parallel threads in all these existing approaches are either statically predicted and spawned or automatically extracted by hardware at procedures, loops or cache line boundaries.

Static (thread) prediction can be quite effective in parallelising loops because the execution order of loop iterations is statically predictable (except the last one, which needs to be control-speculated). However, this compile-time decision becomes less effective when applied to recursive procedures. The data structure operated on by a recursive procedure can vary from input to input and can also change dynamically during program execution. Therefore, when the dynamic call graph of a recursive procedure is speculated, the invocation order of recursive call instances becomes nondeterministic and the potential presence of speculation failures can severely limit parallelism to be exploited.

In this paper, we present a new compiler technique, called *Speculative Parallelisation of Recursive Procedures* (SPRP), to speculatively parallelise recursive procedures for SpMT architectures. We restrict ourselves to those irregular programs that cannot be parallelised effectively by existing techniques. Furthermore, we are particularly interested in those where recursive calls are control-dependent on some runtime values so that only a portion of their underlying data structures, which may also change at run time, may be traversed. As a result, the invocation order of recursive calls is non-trivial to predict accurately, even at run time.

For a given recursive procedure, SPRP will transform it into a helper thread running on a dedicated core and a group of worker threads running on the other cores in a SpMT multicore system. The helper thread, which is a smaller, faster version distilled from the original procedure, serves to predict both the invocation order of recursive calls made and their live-in values as well as to dynamically schedule these calls to run as parallel worker threads. The helper thread is not constrained by correctness. Thus, its predictions are validated whenever a worker thread has run to completion. When a prediction goes wrong, a recovery mechanism introduced in this paper will bring the helper thread back to the point where new predictions (for the future recursive calls) will be made. Due to dynamic thread prediction and thread spawning, SPRP is capable of exploiting more TLP in recursive procedures that is otherwise difficult to exploit in other ways as validated in our experiments.

We have evaluated SPRP using four representative irregular recursive procedures using a cycle-accurate simulator. Our preliminary results are encouraging. An average region speedup of 1.29 for recursive procedures and an average program speedup of 1.21 have been achieved by our technique on four cores. It is important to emphasise that such programs may have to be left to run sequentially on one single core otherwise (unless they are manually parallelised by domain experts). So this work demonstrates the significant performance potential achievable by automatic parallelisation of hard-to-parallelise recursive procedures, providing insights on further research in this area.

The rest of this paper is organised as follows. Section 2 reviews the related work. Section 3 introduces the basic idea behind SPRP by a motivating example. Section 4 discusses how to construct the helper thread for a recursive procedure. Section 5 describes our recovery mechanism. Section 6 presents and analyses our experimental results. Section 7 concludes the paper with some future work.

2 Related Work

Helper threads [28, 14, 7, 31, 16] have been used to speculatively execute a code region to reduce the latency of its expensive instructions. In these research efforts, a helper thread typically serves the purposes of data prefetching or branch predictions or both. In this work, the helper threads used in SPRP are required to predict quite accurately both the order of recursive calls and their live-ins in order to reduce the misspeculation overhead incurred and thus improve the overall parallelism achieved.

MSSP [32] runs a distilled version of a given program on a master processor to predict the live-ins for tasks running on slave processors. Our helper threads and worker threads used in SPRP are conceptually similar to the master and slave threads in MSSP but are specifically developed to parallelise recursive procedures. MSSP skips recursive procedures when constructing distilled programs. In contrast, a helper thread used in SPRP works not only as a producer for spawning worker threads to execute recursive calls but also as a predictor for pre-computing the live-ins for worker threads.

Some compilation techniques for SpMT architectures [12, 2, 23, 3, 27, 11, 18, 15] allow threads to be formed at arbitrary control flow edges. In [12], threads are formed at loop or procedure boundaries using actual profile-run execution times. PD (Program Demultiplexing) [2] attempts to execute different procedures in a program in parallel as long as their inputs are speculatively available. The Mitosis compiler [23] encodes a P-slice – a piece of code to predict thread live-in values (similar to a distilled program in MSSP and a helper thread in SPRP) – into a speculative thread. Unlike [12, 2], thread partitioning in Mitosis is not restricted to loop or procedure boundaries. However, what differs SPRP from all these previous techniques is that SPRP embraces dynamic thread prediction while all these earlier techniques resort to static thread prediction. Furthermore, if these earlier techniques are applied to parallelise a recursive procedure, the invocation order of recursive calls and their required live-in values have to be predicted separately. Therefore, speculative TLP attainable by these techniques seems to be limited for procedures with multiple recursive call sites.

Some researchers have also proposed microarchitecture enhancements to automatically extract threads from sequential programs at run time. Capsule [20] automatically parallelises component-based programs through frequent hardware resource probing. Thread creation is by means of self-replication, and in addition, threads are allowed to commit in any order. Hence, Capsule is applicable only to certain applications that can be componentised. Instead of program structures, Atlas [6] only considers memory access instructions when partitioning threads. DMT [1] creates threads at procedure and loop boundaries. A speculative thread is always spawned at the return address of a

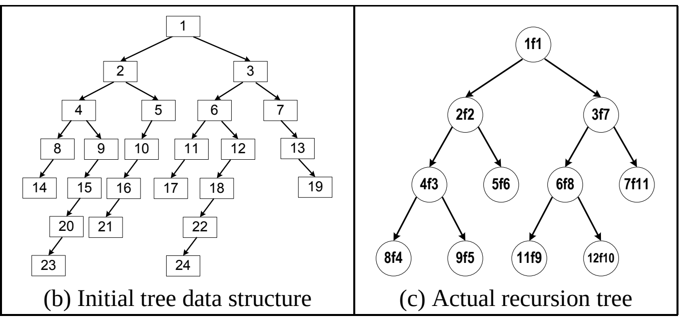
```
main() {
    f(p, d)
}

f(Node p, Data &d) {

    doit (p, d);

    if (p->c1)
       f(p->left, d);
    if (p->c2)
       f(p->right, d);
}
```

(a) A recursive procedure



(b) Initial tree data structure (c) Actual recursion tree

Fig. 1. A recursive procedure illustrated for some particular input.

```
main() {
    spawn f_help (f_work_SP-W_FS, p,d);
    f_work (p, d, f_work_SP, L1);
L1:
    kill
}
```

(a) main

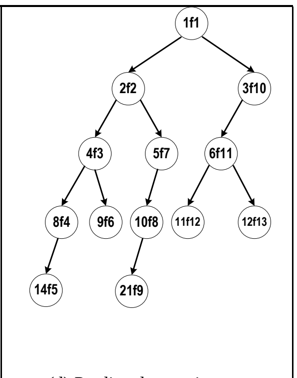
```
f_help (W_SP, p, d) {
  if (p->c1)
     spawn f_work(p->left, d, W_SP, L2);
     f_help(W_SP-W_FS, p->left, d);
  if (p->c2)
     spawn f_work(p->right, d, W_SP, L3);
     f_help(W_SP-W_FS, p->right, d);
}
```

(b) Helper thread code

```
f_work (p, d, W_SP, RA) {
    doit (p, d);
    if (p->c1)
       check (p->left, d);
L2:
    if (p->c2)
       check (p->right, d);
L3:
    return RA;
}

W_SP:  frame pointer of its
       activation record
W_FS:  frame size
RA:    return address
```

(c) Worker thread code



(d) Predicted recursion tree

Fig. 2. Speculative parallelisation of the example in Figure 1 by the SPRP approach.

call site. When DMT is applied to a recursive procedure, a speculative thread may be spawned to execute a recursive call too early to have its live-ins predicted accurately and its relevant dependences speculated successfully. This is because the spawner may later create many less speculated threads to execute some recursive calls that would have been executed earlier when the procedure were executed sequentially.

Techniques on automatic parallelisation of recursive procedures [22, 10, 24, 21] exploit task-level parallelism (i.e., coarse-grain parallelism) in embarrassingly parallel recursive calls. In [10], data speculation is said to be supported but for all benchmark applications used in their experiments, recursive calls are always independent. Irregular recursive procedures are allowed in [22] provided that all multiple recursive calls are independent and marked as such by (dependence-related) programmer annotations.

3 The SPRP Approach

Consider an irregular procedure given in Figure 1(a) with two recursive call sites. To make this example concrete, let us assume that the data structure operated on is a tree. The tree initially looks like what is shown in Figure 1(b) but may grow and shrink at run time. Whenever a tree node is visited, the core computations abstracted by doit(p, d) in line 5 are performed. This statement accesses two live-ins p, a pointer to a tree node, and d, some global data. Inside doit(p, d), all objects pointed to by p directly or indirectly and d may be modified. Therefore, in any recursive call, d in lines 7 and 9 may have different values since it may be modified in the first call made in line 7. The two call sites in lines 7 and 9 are control-dependent on p. Hence, two successive call invocations may be control-dependent or data-dependent. Figure 1(c) gives the dynamic call graph, known as the *recursion tree*, for some input. Note that not all tree nodes in Figure 1(b) may be visited. Each node in the recursion tree represents a recursive call invocation. The two children of a parent node are the two calls invoked directly inside the parent. The notation xfi shown inside a call node indicates that xfi is the i-th recursive call applied to the tree node x in the data structure. (This tree node may be one created at run time!) Sequential execution imposes a total ordering of all dynamic call invocations.

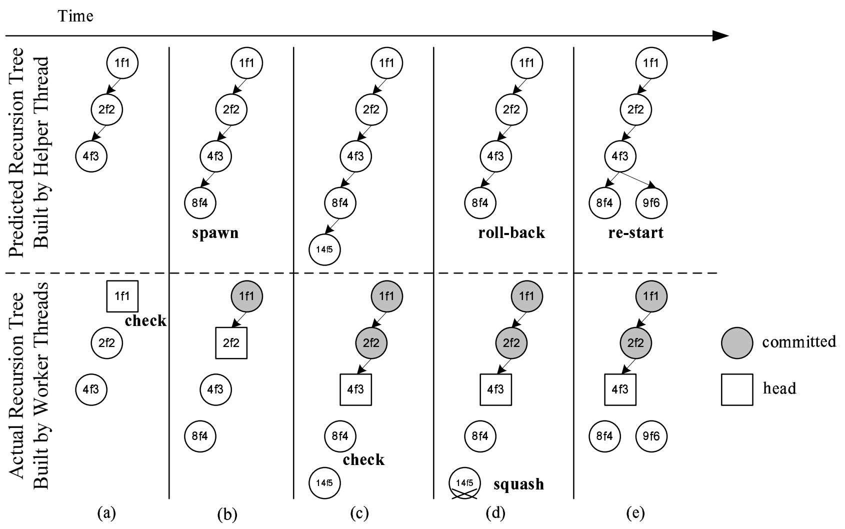
Figure 2 shows the parallelised code for the example. The *helper thread* running on a dedicated core, say, core 0, serves to predict the recursion tree and the live-ins for each recursive call and to dispatch these calls to run as worker threads on the remaining cores (numbered from 1) in parallel. The helper thread is a sequential program running in its own address space with its own runtime stack. All parallel worker threads run in a shared address space by sharing a common runtime stack (starting from f_work_SP). The meanings for W_SP, W_FS and RA are defined in Figure 2(c) and referred to later.

The execution starts from main (Figure 2(a)), which is spawned as the first worker thread to execute on a core. First, the spawn instruction is executed so that the helper thread (Figure 2(b)) is spawned to execute on its dedicated core. Second, the call f_work(p, d, f_work_SP, L1) (Figure 2(c)) is made to start the recursion. This first worker thread is the *head* thread. In speculative execution, the head thread is the only non-speculative worker thread that is allowed to commit. All other currently active worker threads are speculative. Each active worker thread represents the execution of a recursive call and thus runs in an activation record described in Section 5. Figure 2(d) depicts the recursion tree predicted by the helper thread (if being allowed to run alone to completion). However, the predicted recursion tree at run time may not be like this since it will adapt itself according to the validation outcomes from worker threads.

Figure 3 illustrates our approach by giving a snapshot of all key activities involved during program execution. In Figures 3(a) and (b), the head thread $1f1$ has committed and validated that the next call $2f2$ predicted by the helper thread is correct. So $2f2$ becomes the new head thread. Let us look at how roll-back is performed when a misspeculated call is detected as illustrated in Figures 3(c) – (e). In Figure 3(c), the speculative worker thread $8f4$ is validating if the execution of the next call predicted for $8f4$ is correct or not. The answer is negative since the next node to be visited should be node 9 rather than 14 as shown in Figure 1(c). So $14f5$ is squashed and the helper thread is instructed to roll back its state to spawn the next recursive call, $9f6$ (Figure 2(d)).

3.1 Helper Thread

In the helper thread given in Figure 2(b), the instructions abstracted by doit(p, d) happen to be all pruned according to our construction algorithm described in Section 4. The helper thread dynamically schedules worker threads by simulating the execution



Time
Predicted Recursion Tree Built by Helper Thread
Actual Recursion Tree Built by Worker Threads
1f1
2f2
4f3
8f4
14f5
9f6
spawn
roll-back
re-start
check
squash
committed
head
(a)
(b)
(c)
(d)
(e)

Fig. 3. An illustration of SPRP. An arrow linking two calls represents their caller-callee relation. For the helper thread, all calls are part of its predicted recursion tree. For worker threads, the recursion tree is dynamically constructed consisting of committed threads and the head thread.

of the given recursive procedure: it spawns a new worker thread by executing a spawn instruction whenever it reaches a call site. The helper thread is stalled if there is no free core for a new worker thread and resumed when a free core becomes available. For each predicted call, the predicted live-ins, the corresponding stack pointer and the return address for the call must be communicated to its executing worker thread. These data are passed as the arguments to f_work.

3.2 Boundaries or Lifetimes of Worker Threads

The actual execution of a procedure is done by worker threads. Every call invocation has a unique activation record. A worker thread T is executed in its own activation record if it represents a non-leaf call. For a leaf call, T initially executes in its own activation record and later in some of its callers. Let $T.SP$ be the stack pointer associated with the (current) activation record of T. Let Thread_List be the list of all committed and currently active worker threads in increasing order of their spawn times. Thread_List is the preorder traversal of the currently predicted recursion tree. All currently active worker threads are ordered from least to most speculative in Thread_List. The caller of an active worker thread T, denoted Caller(T), is the last thread T' preceding T in Thread_List such that $T.SP = T'.SP -$ W_FS. This means that the call executed by T would be made directly in the call executed by T' (during sequential execution).

The *boundary* or *lifetime* of a worker thread T is defined as follows. When executing f_work, T starts at its first instruction and terminates at either the first check that it *dynamically* executes or the kill instruction in main. There are three cases:

1. If p->c1 evaluates to true, T terminates at the first check.

2. If p->c1 evaluates to false and p->c2 to true, T terminates at the second check.
3. When both guards are false, T represents a leaf call. By executing the "return RA" instruction in f_work, T will continue to execute at the return address RA with the activation record of $\mathsf{Caller}(T)$ being set as its current activation record. The execution of the code of $\mathsf{Caller}(T)$ may cause T to reach the second check (where we are back to the second case) or the return RA instruction in $\mathsf{Caller}(T)$ (where we are back to the third case again) in f_work. As a result, a sequence of return instructions executed by T will take it to either a check or a kill instruction.
   To understand conceptually where a leaf call terminates, let $\mathcal{RA}(\mathsf{Caller}^m(T))$ be the return address RA in the activation record of $\mathsf{Caller}^m(T)$ at which T will continue its execution, where $\mathsf{Caller}^m(T)$ stands for m applications of the function Caller to T. Let $\mathsf{Caller}^*(T)$ be $\mathsf{Caller}^n(T)$ for some unique $n \geqslant 1$ such that $\mathcal{RA}(\mathsf{Caller}^n(T))$ is either L1 (Figure 2(a)) or L2 (Figure 2(c)), and p->c2 evaluates to true when $\mathcal{RA}(\mathsf{Caller}^n(T)) = \text{L2}$. If $\mathcal{RA}(\mathsf{Caller}^*(T)) = \text{L1}$, the dynamic last instruction of T is kill. If $\mathcal{RA}(\mathsf{Caller}^*(T)) = \text{L2}$, the dynamic last instruction of T is the second check to be executed in the activation record of $\mathsf{Caller}^*(T)$. Consider Figures 3(c) – (e), where $8f4$ is assumed to be a leaf call. Then $\mathsf{Caller}^*(8f4) = \mathsf{Caller}(8f4) = 4f3$ and $\mathcal{RA}(4f3) = \text{L2}$. So $8f4$ will terminate after it has executed the second check in the activation record of $4f3$.

3.3 Validations of Predicted Calls

Consider when a worker thread T has reached its dynamically last instruction. There are two cases. In one case, the last instruction is the kill instruction. If T is speculative, then T is stalled. If T is the head thread, then the execution of the recursive procedure has completed successfully. So the helper thread is killed. In the other case, the last instruction of T is a check instruction. T will search for the *successor worker thread* of T, denoted $\mathsf{Succ\_Call}(T)$, that is responsible for executing the next call to be made after T at the check call site in T during sequential execution. $\mathsf{Succ\_Call}(T)$ is the first thread T' following T in Thread_List such that $T.SP = T'.SP + \mathsf{W\_FS}$ and the live-outs of T are identical to the predicted live-ins used by T'.

If $\mathsf{Succ\_Call}(T)$ is found, all threads between T and $\mathsf{Succ\_Call}(T)$ in Thread_List are squashed. If T is speculative, T is stalled. Otherwise, T is the head thread. Thus, the results of the validated T' are committed and T' becomes the new head thread. If $\mathsf{Succ\_Call}(T)$ is not found, all more speculative threads than T in Thread_List are squashed. A recovery mechanism introduced in Section 5 is used to steer the helper thread back to the right track so that the successor call can be spawned at the check call site. If T is the last thread in Thread_List, T is stalled until either T is squashed or a more speculative thread T' than T is spawned (so that the validation at T can be performed). Let us consider Figures 3(c) – (e) again under the assumption that $T = 8f4$ as shown in Figure 3(c) is a leaf call. Thus, $\mathsf{Caller}^*(8f4) = \mathsf{Caller}(8f4) = 4f3$. By the time when $8f4$ reaches the second check instruction in the activation record of $4f3$, we have $8f4.SP = 4f3.SP$ and $\mathsf{Thread\_List} = \{1f1, 2f2, 4f3, 8f4, 14f5\}$. Since $14f5$ is the only worker thread following $8f4$ and $8f4.SP = 4f3.SP = (w - 160) \neq 14f5.SP + \mathsf{W\_FS} = (w - 320) + 80$ as shown in Figure 5(a), the validation performed will fail. In fact, the next node to be visited

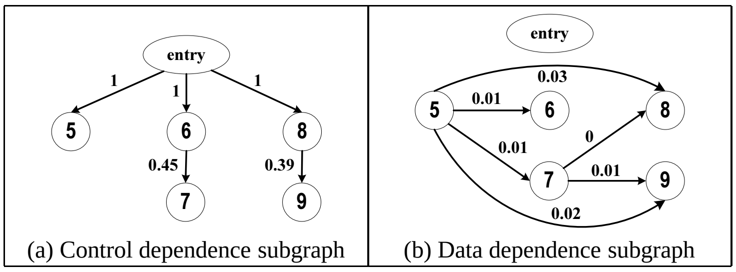


Fig. 4. Program dependence graph (PDG) of the procedure in Figure 1.

should be node 9 rather than 14 as is clear in Figure 1(c). Thus, $14f5$ is squashed and the helper thread is re-directed to spawn $9f6$ (Figure 2(d)).

3.4 Memory Dependence Speculations

Misspeculated memory dependences are handled in the normal manner [23, 27]. A misspeculation is raised if a worker thread writes into a memory location where a more speculative worker thread has already read from the same location. The misspeculated worker thread is squashed and re-started. All worker threads that depend on the misspeculated worker thread are also re-started.

4 Construction of Helper Threads

The accuracy and the size of the helper thread affect the amount of speculative parallelism SPRP can achieve but not the correctness of the SPRP execution.

The *Program Dependence Graph* (PDG) of a recursive procedure is used to construct its helper thread. In the data dependence subgraph of PDG, only true or flow data dependences are included. Each edge $x \longrightarrow y$ in PDG is labelled with a probability value $p_{x\to y}$ in $[0, 1]$. If $x \longrightarrow y$, is a data dependence, then $p_{x\to y}$ means that for every N writes at x, only pN reads will access the same memory/register location at y during program execution. If $x \longrightarrow y$ is a control dependence, then $p_{x\to y}$ means that for every N execution of x, only pN will reach y. Figure 4 gives the PDG of the recursive procedure in Figure 1. A node is numbered using the line number of its corresponding statement.

Let $\mathcal{H}$ be the set of instructions forming the helper thread. $\mathcal{H}$ is initialised with the set of nodes in PDG that correspond to all the recursive call instructions. Next, for every node u in PDG, we add u to $\mathcal{H}$ if $\exists\ v \in \mathcal{H}$ such that (1) edge $u \longrightarrow v$ is in PDG, and (2) $p_{u\to v} \geqslant D$, where D is relatively large, if $u \longrightarrow v$ is in the data dependence subgraph and $p_{u\to v} \leqslant C$, where C is relatively small, if $u \longrightarrow v$ is in the control dependence subgraph. Here, D and C are some tunable parameters. The intention is to ignore infrequently occurring data dependences and frequently occurring control dependences. The instructions in PDG are included in $\mathcal{H}$ iteratively until a predefined size limit has been reached or no more nodes can be added.

The values of D, C and the helper thread size are likely to be application-dependent. Our experience gained in this work is that data dependences tend to be bi-modal while

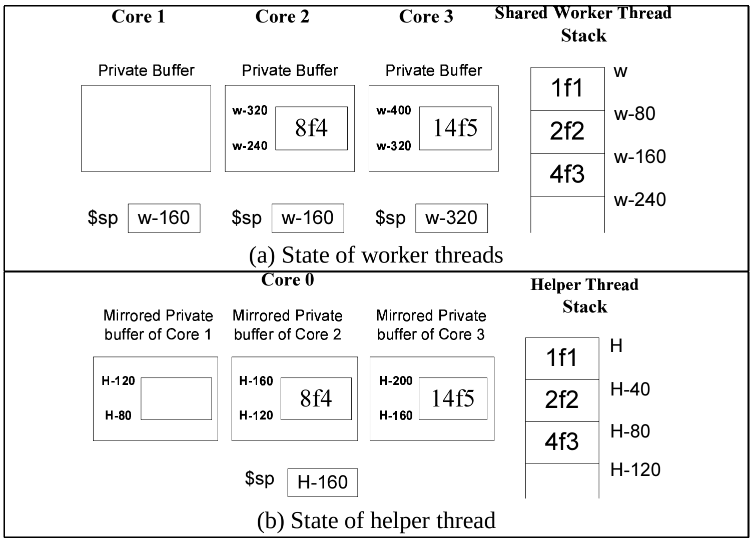


Fig. 5. Machine state when a misspeculation is detected as illustrated in Figure 3(c).

control dependences tend to be tri-modal. These parameters can be tuned by profiling and program analysis. In our experiments, $D \geqslant 0.8$ and $C \leqslant 0.6$ are reasonable.

In our example, let us assume $D = 0.8$ and $C = 0.6$. There are two call sites. So $\mathcal{H} = \{7, 9\}$ initially. Note that $6 \rightarrow 7$ is a control dependence. So node 6 is added to $\mathcal{H}$ since $p_{6\rightarrow7} < 0.6$. Node 8 is included in $\mathcal{H}$ for the same reason. The probabilities of of all data dependences are small. Finally, $\mathcal{H} = \{6, 7, 8, 9\}$. This leads to the corresponding helper thread as depicted in Figure 2(b).

5 Misprediction Recovery

This section describes our recovery mechanism developed to support the SPRP scheme using the motivating example with respect to Figure 3. As explained in Section 3.3, a misprediction is raised in Figure 3(c). SPRP will then roll back the states for the execution of both the helper thread and worker threads. Suppose that $4f3$, $8f4$ and $14f5$ run on cores 1, 2 and 3, respectively. (The helper thread runs on core 0.)

5.1 Recovering the State of Worker Threads

Figure 5(a) depicts the state of the worker threads at the time when a misprediction is detected. The activation records of all past and current head threads have already been committed to the shared runtime stack. The activation records of all speculative ones are buffered in on-chip memory. Let us assume that the frame size of an activation record is W_FS $= 80$. In the example, the activation records of $1f1$ and $2f2$ have been committed to the shared stack. The activation records of $8f4$ and $14f5$ are buffered since both are speculative. The activation record of the head thread $4f3$ is on the shared stack. As a

Parameter	Value
Fetch, Issue, Commit	bandwidth 4, out-of-order issue
L1 I-Cache	16KB, 4-way, 1 cycle (hit)
L1 D-Cache	16KB, 4-way, 3 cycles (hit)
L2 Cache (Unified)	1MB, 4-way, 12 cycles (hit), 80 cycles (miss)
Local Register File	1 cycle
Spawn Overhead	5 cycles
Commit Overhead	5 cycles
Validation Overhead	15 cycles

Table 1. SpMT multicore system simulated.

leaf call, $8f4$ branches to L2 to execute in the activation record of $\mathsf{Caller}^*(8f4) = 4f3$ where a misprediction is detected. Hence, $SP on core 2 is pointing to the activation record of $4f3$. The mispredicted thread $14f5$ is squashed (Figure 3(d)) and its buffered data discarded. The helper thread is then informed to spawn $9f6$ according to the current state of $8f4$ (Figure 3(e)), as described below.

5.2 Recovering the State of Helper Thread

As shown in Figure 5(b), the sequential execution of the helper thread is made to mirror the parallel execution of worker threads. For every worker thread running on a core, the execution results for the corresponding call invocation (including its activation record) made in the helper thread are buffered in the mirrored private buffer for the core on core 0. Whenever a private buffer on a core is committed, discarded or released, the mirrored private buffer is also committed, discarded or released in sync. Hence, the roll-back activities performed by both the helper thread and worker threads are synchronised. In general, the helper thread is smaller than a worker thread. For illustration purposes, we assume the frame size of an activation record for each recursive call to f_help made in the helper thread (Figure 2(b)) is 40 (bytes). Recall that when a misprediction is detected by the worker thread $8f4$ running on core 2, $8f4$ is pointing to the second check of f_work, causing $14f5$ running on core 3 to be squashed. Correspondingly, (1) the mirrored private buffer of core 3 is discarded, (2) the most up-to-date live-ins for the successor call after $8f4$, which is $9f6$, are passed to the mirrored private buffer of core 2, (3) $SP on core 0 is rolled back to the activation record corresponding to that of $4f3$ that $SP on core 2 is pointing to, and finally, (4) the execution of the helper thread is rolled back to point to the second spawn instruction in f_help. Therefore, the helper thread will be restarted to spawn a worker thread $9f6$ with the most up-to-date live-ins.

6 Experimental Results

To evaluate SPRP, a preliminary implementation of SPRP is built on top of GCC 4.1.1 with programmer annotations indicating which recursive procedures are to be parallelised. All benchmarks are compiled under the optimisation level “-O2”. The generated code is simulated using a detailed execution-driven microarchitectural simulator built on top of SimpleScalar. The simulator models an SpMT quad-core system. Table 1 provides the main architectural parameters, which are similar to those used in the recent work [23, 8]. Each core is capable of executing the Alpha ISA. One core is dedicated to the helper thread while the other three cores are used to execute worker threads.

Benchmark	I-size	Fan-out	W-size	H-size	H-size/W-size	#Live-ins
Bh	256	1...8	131	29	0.22	7
Bisort	8192	2	57	27	0.47	5
Knapsack	15	2	447	24	0.05	6
Queens	9	1..9	2887	68	0.02	8

Table 2. Benchmarks.

In Section 6.1, we describe the benchmarks used. In Section 6.2, we present and discuss the performance speedups achieved by SPRP. The speedups on a quad-core system may not be huge but they are close to the ideal ones attainable. Otherwise, these hard-to-parallelise may have to be either run sequentially on one single core or manually parallelised by domain experts in a case-by-case basis. In Section 6.3, we compare SPRP with two existing compiler techniques to demonstrate further the performance stability and scalability of SPRP when dealing with the same program with varying inputs and dynamically changing runtime data structures.

6.1 Benchmarks

Four benchmarks are used in our experiments: Bh (Barnes-Hut) and Bisort are taken from the Olden benchmark suite and Knapsack and Queens are from the Cilk benchmark suite. These benchmarks represent a wide spectrum of application domains. Bh solves the N-body problem using hierarchical methods on a tree. Bisort implements a recursive bitonic sorting algorithm on a tree. Knapsack is a combinatorial optimisation algorithm that solves a one-dimensional backpack problem using branch-and-bound on an array. Queens is modified from Cilk to find all solutions to the N-queens problem on an array.

To evaluate the performance of SPRP, we parallelise only recursive procedures, although selected benchmarks may have more parallelism if other program structures such as loops are also used to form threads. Table 2 provides some statistics about the four benchmarks. The input size (I-size) for each benchmark is listed in Column 2. The fan-out in Column 3 represents the range for the number of child calls invoked directly in each parent call in the recursion tree of a recursive procedure. The fan-outs of all four benchmarks are larger than 1. Therefore, these four benchmarks allow us to evaluate the accuracy of our helper threads in predicting the invocation order of recursive calls made in these benchmarks. In Column 4, W-size represents the average number of instructions executed for all committed worker threads in a benchmark (i.e., all recursive calls made in the sequential execution of the benchmark). In Column 5, H-size is the average number of instructions executed by the helper thread between two successive spawn instructions for a benchmark. Thus, the ratio H-size/W-size listed in Column 6 indicates how much faster the helper thread spawns recursive calls than if a direct execution of the original procedure would do. The lower the ratio, the faster. The ratios are very low for Knapsack and Queens. As for Bh and Bisort, the sizes of their worker threads are small. It seems to be difficult to reduce the ratios any further.

In the last column, the number of live-ins for a procedure is given. This is the size of data to be passed to a spawned thread. A maximum of 8 live-ins has been observed

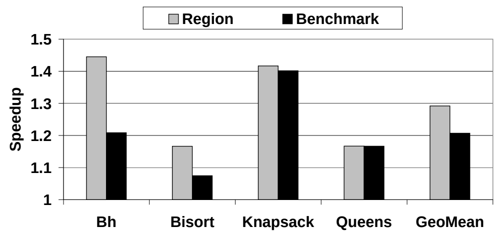


Fig. 6. Speedups of SPRP over sequential execution.

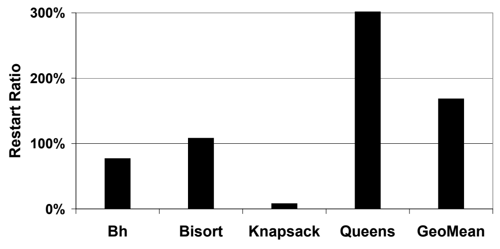


Fig. 7. Restart ratios of SPRP.

in the four benchmarks, indicating that a latency of 5 cycles for spawn overhead is adequate in our experiments, as previously demonstrated in [23, 8].

6.2 Performance and Analysis

It is important to understand the performance improvements achieved by SPRP in the context that the recursive procedures selected and used in our experiments are very difficult to parallelise by existing methods. The parallelisation-inhibiting factors are that (1) there is more than one dynamic call site (as in all four benchmarks), (2) call sites are guarded by non-trivial expressions (as in Bh, Knapsack and Queens), (3) there are memory dependences among recursive calls (as in all four benchmarks), (4) the underlying data structure may dynamically change at run time (as in Bisort) and (5) only part of the underlying data structure is traversed (as in Bh, Knapsack and Queens).

Figure 6 gives the speedups of SPRP over sequential execution. The region speedups (for recursive procedures only) range from 1.16 to 1.45 with an average of 1.29. The program speedups are close to the region speedups for Knapsack and Queens. But this is not true Bh and Bisortsince the recursive procedures parallelised represent only 55% and 46% of their total execution times, respectively.

Let us now analyse the performance results achieved by SPRP. First of all, SPRP can achieve a good degree of speculative TLP in our benchmarks. The average number of active worker threads per cycle for Bh, Bisort, Knapsack and Queens are 2.51, 2.14, 1.79 and 2.19, respectively. Whether this amount of speculative thread-level parallelism can translate into performance gains or not depends on how often speculated work threads succeed and how precise the predictions made by helper threads are.

Figure 7 shows the restart ratios for all four benchmarks. The *restart ratio* for a benchmark represents the number of restarted threads over the number of committed threads. A call that is restarted n times will be counted to have been restarted n times. The restart ratio of a benchmark is a rough approximation of the impact of misspeculations on performance. For example, Queens has the highest restart ratio, which is caused by excessive misspeculations of memory dependences as discussed in Section 3.4. In the parallelised recursive procedure of Queens, every call invocation may depend on the earlier calls made – they may not be the immediate predecessors, since every call uses the passed-in array a and may also update one element of a as well as pass a to the ensuing call invocation. Hence, the performance improvement for Queens is limited.

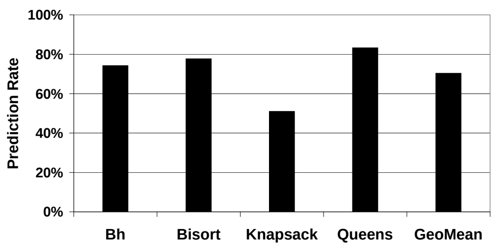


Fig. 8. Prediction rates of SPRP for correctly executed recursive calls.

Figure 8 gives the prediction rate, i.e., success rate at which the recursive calls have been correctly predicted by the helper thread for each benchmark. When constructing the helper thread for a recursive procedure, a trade-off between the prediction rate and the exposed speculative TLP has to be made. For example, Knapsack has the smallest prediction rate since most branches used to prune the searching space are not included in the helper thread. Hence, Knapsack has a very small H-size/W-size ratio as shown in Table 2 indicating a large portion of speculative TLP has been exposed by SPRP. Any further improvement on its prediction rate requires extra time-consuming computations to be included in the helper thread, resulting in a significant decrease of the exposed speculative TLP. Similarly, any further improvement on the prediction rate for Bh requires the entire subroutine `subdivp` to be included in the helper thread. As a result, very little speculative TLP could be exposed. On the other hand, as shown in Table 2, the H-size/W-size ratio of Bisort is the largest due to the strong memory dependences among the recursive calls since the underlying tree structure used by Bisort may be modified at run time. Any further reduction of its H-size/W-size ratio leads to a significant drop of its prediction rate, resulting in a performance slowdown. If we increase its H-size/W-size ratio to obtain a better prediction rate, the helper thread will be too large to expose any speculative TLP in the benchmark.

Figure 9 shows the performance gap between SPRP and what can be achieved during an ideal program execution (the H-size/W-size ratio during the ideal execution is negligible and the helper thread always makes precise prediction). On average, the execution time of SPRP is only 14% longer than the ideal execution. Hence, SPRP is potentially effective in parallelising these irregular recursive procedures.

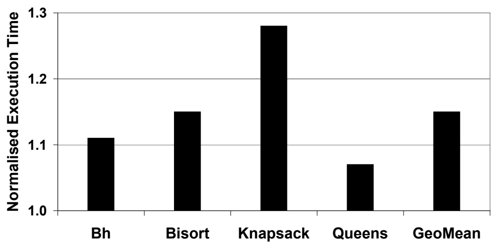


Fig. 9. Normalised execution times of SPRP with respect to ideal execution.

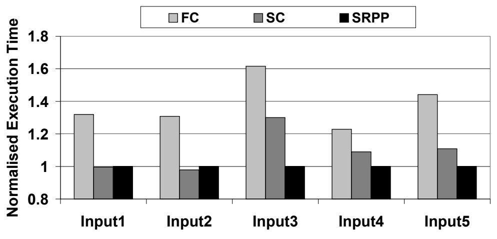


Fig. 10. Normalised execution times of FC, SC and SPRP with respect to SPRP.

6.3 Dynamic Prediction and Static Prediction

Due to space limit, we use Knapsack to demonstrate the performance advantages of SPRP over two static thread prediction and spawning schemes used for parallelising recursive procedures, SC and FC. In *subroutine-continuation (SC)* spawning scheme [12, 15, 27, 1], a speculative thread is always spawned at the return address of a recursive call site. In another scheme referred to as *First Call (FC)* in this paper, only the calls made at the first call site are control-speculated to be always invoked. Note that unlike SPRP, both FC and SC predict live-ins separately. By using a helper thread to predict both the recursion tree and live-ins required by each predicted recursive call, SPRP outperforms SC and FC almost always when different input data are used.

Figure 10 compares SPRP with SC and FC in terms of five different inputs. The search space of Knapsack is a binary tree. We have carefully selected these inputs so that five representative recursion trees are used at run time. The recursion trees exercised by Input1, Input2, Input3, Input4 and Input5 are a complete binary tree, a right-biased tree (the left child of every tree node is a leaf), a left-biased tree (the right child of every tree node is a leaf), a random tree (with its nodes randomly distributed) and a left-and-right-biased tree (a combination of a left-biased subtree and a right-biased subtree), respectively. FC is the worst performer in all cases, because it always sequentialises all leaf nodes that contain some computations. SC performs only slightly better than SPRP for Input2 (i.e., the right-biased tree) and similarly as SPRP for Input 1 (i.e., a complete binary tree). In the other three cases, SPRP significantly outperforms SC. SC is very sensitive to the shapes of recursion trees. When the underlying recursion trees are left-biased, a large number of threads created in SC are later squashed to release cores for less speculative threads.

By comparing with static thread prediction, SPRP can more precisely predict the order in which recursive calls are made and thus expose more parallelism.

7 Conclusion

We have presented a new compiler technique for speculatively parallelising irregular recursive procedures that are difficult to parallelise traditionally. These recursive procedures may sometimes be parallelised manually by domain experts in a case-by-case basis. However, the potential presence of some dependences in a program will cause even the expert programmers to be conservative, limiting the parallelism to be exploited. This works aims to make a case that these hard-to-parallelise recursive procedures can be potentially parallelised automatically. Our preliminary results using four representative benchmarks are very encouraging. Our approach is general since it can handle recursive procedures with code blocks appearing both before and after a call site by spawning threads using a combination of preorder, inorder and postorder traversals.

There are a number of interesting research issues we will pursue in the future. One is to develop good heuristics to construct faster helper threads with good prediction accuracies. Another way to improve the prediction accuracies of helper threads is to allow the helper thread to access more up to date memory variables. This means that some tradeoffs must be made between the efficiency and accuracy of a helper thread.

Acknowledgement

This work is supported the Australian Research Council Grant (DP0881330) and the UNSW Engineering-International Research Collaboration Grant (PS16380).

References

1. H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *MICRO-31*, pages 226–236, 1998.
2. S. Balakrishnan and G. S. Sohi. Program demultiplexing: Data-flow based speculative parallelization of methods in sequential programs. *ISCA '01*, 2006.
3. A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreaded processors. *IEEE Trans. Parallel Distrib. Syst.*, 15(8):713–724, 2004.
4. G. E. Blelloch, J. C. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee. Implementation of a portable nested data-parallel language. *J. Parallel Distrib. Comput.*, pages 4–14, 1994.
5. R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. L. Zhou. Cilk: an efficient multithreaded runtime system. *PPoPP '95*, 1995.
6. L. Codrescu and D. S. Wills. On dynamic speculative thread partitioning and the mem-slicing algorithm. In *PACT '99*, 1999.
7. J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y. F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: long-range prefetching of delinquent loads. In *ISCA '01*, pages 14–25, 2001.
8. Z. H. Du, C. Ch. Lim, X. F. Li, C. Yang, Q. Zhao, and T. F. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *PLDI '04*, 2004.

9. M. Franklin. *The Multiscalar Architecture*. PhD thesis, The University of Wisconsin at Madison, 1993.
10. M. Gupta, S. Mukhopadhyay, and N. Sinha. Automatic parallelization of recursive procedures. *International Journal of Parallel Programming*, 28(6):537–562, 2000.
11. T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *PLDI '04*, 2004.
12. T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Speculative thread decomposition through empirical optimization. In *PPoPP '07*, pages 205–214, 2007.
13. V. Krishnan and J. Torrellas. Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor. In *ICS '98*, pages 85–92. ACM Press, 1998.
14. S.W. Liao, P. H. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. P. Shen. Post-pass binary adaptation for software-based speculative precomputation. In *PLDI '02*, pages 117–128, 2002.
15. W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. Posh: a tls compiler that exploits program structure. In *PPoPP '06*, pages 158–167, 2006.
16. C. K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *ISCA '01*, pages 40–51, 2001.
17. P. Marcuello and A. Gonzalez. A quantitative assessment of thread-level speculation techniques. In *IPPS '00*, 2000.
18. T. Ohsawa, M. Takagi, S. Kawahara, and S. Matsushita. Pinot: Speculative multi-threading processor architecture exploiting parallelism over a wide range of granularities. In *MICRO-38*, 2005.
19. J. Oplinger, D. Heine, and M. Lam. In search of speculative thread-level parallelism. In *PACT' 99*, 1999.
20. P. Palatin, Y. Lhuillier, and O. Temam. Capsule: Hardware-assisted parallel execution of component-based programs. In *MICRO-39*, pages 247–258, 2006.
21. A. J. Piper. *Object-oriented Divide-and-conquer for Parallel Processing*. PhD thesis, University of Cambridge, July 1994.
22. L. Prechelt and S. U. Hänßgen. Efficient parallel execution of irregular recursive programs. *IEEE Transactions on Parallel and Distributed Systems*, 2002.
23. C. G. Quinones, C. Madrile, J. Sanchez, P. Marcuello, A. Gonzalez, and D. M. Tullsen. Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices. In *PLDI '05*, 2005.
24. R. Rugina and M. C. Rinard. Automatic parallelization of divide and conquer algorithms. In *PPoPP '99*, pages 72–83, 1999.
25. J. Y. Tsai and P. Ch. Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *PACT '99*, pages 35–46, 1999.
26. N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *PACT '07*, pages 49–59, 2007.
27. T. N. Vijaykumar. *Compiling for the Multiscalar Architecture*. PhD thesis, The University of Wisconsin at Madison, 1998.
28. P. H. Wang, J. D. Collins, H. Wang, D. Kim, B. Greene, K. M. Chan, A. B. Yunus, T. Sych, S. F. Moore, and J. P. Shen. Helper threads via virtual multithreading on an experimental Itanium 2 processor-based platform. In *ASPLOS-XI*, 2004.
29. S. Y. Wang, X. R. Dai, K. S. Yellajyosula, A. Zhai, and P. Ch. Yew. Loop selection for thread-level speculation. In *LCPC '05*, 2005.
30. H. T. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *HPCA '08*, 2008.
31. C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *ISCA '01*, pages 2–13, 2001.
32. C. Zilles and G. Sohi. Master/slave speculative parallelization. In *MICRO-35*, 2002.