Speculative Parallelism in Cilk++

Ruben Perez
MIT
rmperez@mit.edu

Gregory Malecha
Harvard University SEAS
gmalecha@cs.harvard.edu

ABSTRACT

Backtracking search algorithms are useful in many domains, from SAT solvers to artificial intelligences for playing games such as chess. Searching disjoint branches can, inherently be done in parallel though it can considerably increase the amount of work that the algorithm does. Such parallelism is speculative, once a solution is found additional work is irrelevant, but the individual branches each have their own potential to find the solution. In sequential algorithms, heuristics are used to prune regions of the search space. In parallel implementations this pruning often corresponds to aborting existing computations that can be shown to be pursuing dead-ends. While some systems provide native support for aborting work, Intel's current parallel extensions to C++, implemented in the `cilk++` compiler [Lei09], do not.

In this work, we show several methods for implementing `abort` as a library in the `cilk++` system. We compare our implementations to each other and quantify the benefit of abort in a real game AI. We derive a mostly mechanical translation to convert programs with native abort into `cilk++` using continuation passing style.

Keywords

Speculative parallelism, α-β pruning, Cilk++

1. INTRODUCTION

Over the years many algorithms have been devised for solving difficult problems efficiently: graph algorithms such as max-flow [EK72] and shortest path [Flo62] and numerical computations such as matrix multiplication [Str69]. Despite much success addressing many problems, fundamental limits govern the efficiency of algorithms for a range of important problems that rely on search. In these cases, parallelism can be used to mitigate some of the inefficiency of solving such problems.

Many of these algorithms come from artificial intelligence and are based around backtracking search, often optimized with heuristics. Algorithms such as these can benefit tremendously from parallel processing because separate threads can be used to search independent paths in parallel. We call this strategy *speculative* parallelism because only one solution matters and as soon as that is found, all other work can be stopped.

While simple to describe, this use of parallelism is not always easy to implement. One parallel programming platform, `cilk5` [FLR98], included language support for this type of parallelism, but it has since been removed in more recent versions of the platform adapted to C++, `cilk++` [Lei09]. Our goal is to regain the power of speculative execution in a way that is relatively natural to program with without the need to modify the runtime library.

Contributions

In this work we consider the parallel implementation of algorithms that rely on speculative parallelism in `cilk++` [Lei09]. We begin with a brief overview of the mechanisms for speculative execution in `cilk5` (Section 2.2) and then consider one prominent use of the these mechanisms for developing gam AI using α-β pruning (Section 2.2). We then consider our contributions:

- We present a technique for compositional speculative execution in `cilk++` which uses polling and can be built as a library without any special runtime support (Section 3).
- We present a mostly-mechanical translation from `cilk5` code to use our library (Section 3.1).
- We demonstrate several implementation strategies for the library needed for our technique (Section 3.2).
- We compare the performance tradeoff of several implementations of our abort library (Section 4.1).
- We determine the effect that polling granularity has on performance (Section 4.2).
- We attempt to quantify the difficulty of porting an existing program that uses speculative execution to our framework (Section 4.3).

We conclude by considering the difficulty of analyzing programs with speculative execution and then describe several avenues of future research.

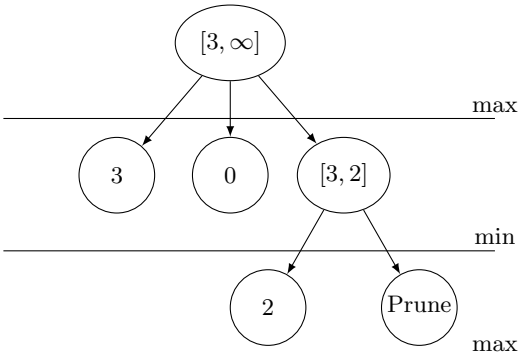

Figure 1: Example of Alpha Beta Pruning

2. BACKGROUND

We begin with an overview of our target application, the parallelization of α-β pruning, before describing how speculative execution is incorporated into the `cilk5` programming language.

2.1 Alpha-Beta Search

Alpha-Beta (α-β) search is an algorithm commonly used to search a 2 player game tree [KM75]. A game tree is a tree where nodes represent states of a game, and edges represent the transitions from one game state to another, i.e. legal moves in the game. Game states are ranked by a heuristic *static board evaluator* which assigns a number that is meant to encode a player's preference for being in the board configuration.

A naive game tree search algorithm, such as minimax, considers possible game states at each level of the tree. It assumes that one player, the maximizing player, wins the game by getting to the game state with the highest value. The other player, the minimizing player, wins by forcing the maximizing player to a game state with a low value. The player will always make the best moves possible; the maximizing player will choose game states with high values, and the minimizing player will choose games states with low values.

The Alpha Beta algorithm keeps track of two values, alpha and beta for each game state in the tree. The alpha value represents a score that the maximizing player is guaranteed to do better than, a lower bound on his possible score. The beta value represents the lowest score the minimizing player can force the other player to get, an upper bound on the maximizing player's score. With these values, we can tell that a game state where alpha is greater than beta would never be reached. Such a game state would mean that one player is not playing the best moves possible, and allowed the maximizing player to reach a game state where he is guaranteed a higher score than another available move, the move that yielded the beta value. For these game states, there is no need to further explore its subtree, as we know the move will never be reached. This elimination of work is called alpha-beta pruning, as we essentially prune away branches of the search tree. An example of such a pruning is shown in figure 1 In figure 1 we see a small game tree where a prune would occur. The left two leaves have been evaluated to have values of 3 and 0, while the alpha beta values, contained in a tuple, for the the root's third child are being determined. Having evaluated the other leaves, we know that the the maximizing player can get at least a three, hence that game state's alpha value is three. Once we evaluate the game state's left child, with a value of 2, we can say that the beta value is 2, as that now is an upper bound on the maximizing player's score at that game state. However, now we see a contradiction, where alpha is greater than beta. Given that this state is reached, the maximizing player is guaranteed to get less than or equal to 2, because the minimizing player will always choose that particular game state (or one that has a lower score). It makes no sense for the maximizing player to ever choose to go to this game state, since he has a better move available, the root's left most child with a value of 3. This we can abort exploration of the rest of root's right most subtree, and save ourselves some work.

A simple parallelization of the algorithm would be to simply search the children of a game state in parallel. However, such an implementation could execute work that a serial implementation would not, by exploring branches of the tree that serial version would have pruned away. At the time we spawn, we cannot know whether this is the case, so we speculate that such work will be useful, and search anyway. If any spawned search does invoke the cutoff condition, then we must terminate all of the other spawned threads, as they are now performing useless work by exploring game states that we will never reach. Thus we see why an abort mechanism is essential to an efficient parallel alpha beta search algorithm, and speculative parallel programming in general.

2.2 Speculative Execution in Cilk5

MIT `cilk5` [BJK+95, FLR98] includes an abort primitive to support early termination of speculative work. To properly explain how these are used, we must introduce another feature, inlets. An inlet is a local C function which shares stack space with its parent function. It is declared inside its parent function and is local to the scope of the parent function. `cilk5` allows an argument to a call to an inlet to be a spawned computation. When the spawned computation returns, the inlet is called with the result. The `cilk5` runtime guarantees that all inlets associated with a single activation record run sequentially. This allows us to conveniently access local data to accumulate a result without the need for explicit locking. Figure 2 shows an example of using inlets to implement non-deterministic choice. The code interleaves two long computations and returns the value returned by one of them.

The inlet (lines 7-10) takes the return value of one of the speculated computations and stores it in the parents local variable `x` (line 8). The inlet can then abort the remaining computation because a value has been computed (line 9). After the `sync`, we know that the inlet will have run at least once, so the value in `x` is valid and we can return it.

3. COMPOSITIONAL SPECULATION

For many applications, such as exhaustive search, it is acceptable to have a single level abort that allows the entire computation to be aborted; however, the structure of α-β search requires the ability to abort individual subtrees of

```
1 int f_1(void* args);
2 int f_2(void* args);
3
4 int first(void* args) {
5   int x;
6
7   inlet void reduce(int r) {
8     x = r;
9     abort;
10  }
11
12  reduce(spawn f_1(args));
13  reduce(spawn f_2(args));
14  sync;
15
16  return x;
17 }
```

Figure 2: Using speculative parallelism in `cilk5` to implement non-deterministic choice.

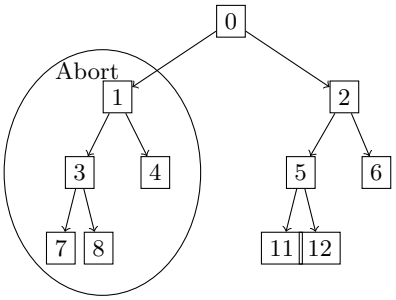

Figure 3: Using hierarchical abort, we can abort any subtree without affecting the rest of the tree.

the search tree. We refer to this as hierarchical abort. Figure 3 demonstrates how this type of abort works. While the algorithm explores node 3, it notes that it can prune the entire branch below node 1, but it can't prune the computation below node 2.

In order to implement abort without modifying the runtime system, we implement abort via polling. The granularity of polling will control the tradeoff between the responsiveness of aborts and the overhead of supporting abort in the code. We explore this relationship in greater detail in Section 4.2. In this section begin by describing an algorithm for translating programs that use inlets into programs that do not use abort (Section 3.1). We then discuss a variety of implementation strategies for implementing the interface for polling (Section 3.2).

3.1 Translating Inlets

We translate inlets using continuation passing style, a common code transformation in functional language compilers [AJ89]. The basic idea behind continuation passing style is to pass the continuation (a function pointer) that should be used to return the value to the caller. When we CPS-convert a function of type:

```
T foo(...);
```

we rewrite it into a function of the following type:

```
T foo(..., void* env, T& (*cont)(void*,T&));
```

Where `cont` is the pointer to the continuation and `env` stores any state needed by the continuation.

Strictly speaking, this translation is not the full CPS-translation. A full CPS-translation would change the return type to **void** [1] and would ensure that the function never returns. We skip this to simplify the translation of code and localize the necessary code changes.

In order to convert the body of `foo` to use continuation passing style, we convert **return** statements of the form:

```
return x;
```

into **return**s of function calls of the passed continuation:

```
return cont(env, x);
```

A good C++ compiler should be able to tail-optimize the extra function call ameliorating some of the overhead incurred by the translation. This translation allows us to run the inlet in the same thread that runs `foo` regardless of whether a steal occurs.

In order to describe our translation in greater detail, we consider the simple implementation of the non-deterministic choice code presented earlier. Figure 4 shows the code and its translation. In the code, the functions `f_1` and `f_2` are meant to compute the same value with algorithms that are efficient only only part of their domain. The `first` function speculatively evaluates both `f_1` and `f_2` in parallel and returns one of their return values [2].

Since `first` contains an inlet, we need to convert the functions that it calls that use it into continuation passing style. We elide the bodies of these functions since this translation was described previously. As can be seen from the `cilk5` code, the return values should be passed along to the `reduce` inlet, so we must factor out `reduce` as the continuation. Naïvely lifting this function is problematic however because the variable `x` is scoped in the `first` function. Thus we need the `InletEnv` structure to carry this value as well as the mutex to ensure mutual exclusion and a handle to our abort mechanism, embodied in the `Abort` class which we describe in the next section.

The content of the inlet is mostly unchanged except that it now must reference variables through the given environment and, since `cilk5` guarantees sequential execution of the inlets for an activation record, we acquire and release the mutex in the environment. The `abort` statement is translated into an invocation of the `abort` member function in the `Abort` class which handles the abort.

The final aspect of the translation is to inject polling calls into the speculated functions, `f_1` and `f_2`. Since our implementation relies on polling, the granularity of these poll statements are exactly the granularity at which aborts can occur. Long periods between polls will result in wasting fewer cycles polling but will potentially allow more wasted

[1] Technically any type can be used since the function would never return.

[2] If abort is implemented correctly, then the result will usually be the return value of the function that returned first, though we don't bother to guarantee this in our code.

cilk5 Code

```
int f_1(void* args);
int f_2(void* args);

int first(void* args) {
  int x;

  inlet void reduce(int r) {
    x = r;
    abort;
  }

  reduce(spawn f_1(args));
  reduce(spawn f_2(args));
  sync;

  return x;
}
```

cilk++ Code

```
struct InletEnv { cilk::mutex m; Abort abort; int x; };

int f_1(void* args, int(*cont)(int,void*), void* env);
int f_2(void* args, int(*cont)(int,void*), void* env);

int first_inlet(int result, InletEnv* _env) {
  InletEnv* env = (InletEnv*)_env;
  env->m.lock();

  env->x = result;     // x = r;
  env->abort.abort(); // abort;

  env->m.unlock();
}

int first(void* args) {
  InletEnv env;
  cilk_spawn f_1(args, first_inlet, env);
  cilk_sync;
  return env.x;
}
```

Figure 4: Translation of `abort` and `inlet`.

```
class Abort {
public:
  Abort();
  explicit Abort(Abort* p);
  int isAborted() const;
  void abort();
};
```

Figure 5: The `Abort` interface.

computation to be done between aborting the child and having the child actually stop working. We investigate this trade-off further in Section 4.1.

3.2 The Abort Library

The interface to our hierarchical abort library is given in Figure 5. The default constructor is used to construct the root of an abort tree. Since the abort tree will often mirror the stack we rely on stack allocated `Abort` objects. The "copy-constructor" is used to construct a new abort tree which is a child of the given tree. We call this when we speculate on a computation. Given the structure of the tree, the interface exposes two member-functions, `isAborted` and `abort`. The first determines if the current computation has been aborted by determining if any parent in the tree has been asked to abort. The second member-function, `abort`, actually marks a tree as needing to abort.

The polling in our interface admits two general strategies for implementing the interface: 1) `abort` just sets a flag while `isAborted` polls up the tree until it reaches the root checking for abort at each level; 2) `abort` can propagate the abort flag downwards to all of its children and `isAborted` can simply check the abort flag of the current level.

We begin with the implementation that polls upward (sketch implementation given in Figure 6. The implementation is completely straight-forward. In `isAborted` we or the current abort flag with the result of polling up the tree (C++ short-circuit semantics of || ensure that we don't poll up if our current flag is set). The `abort` function simply sets the current abort flag. When trees are deep, we can amortize the cost of recurring all the way up the tree by lazily propagating the abort flag downward. This functionality is enabled by setting the `POLL_UP_CACHE` flag at compile time. The code is similar except that we must ensure that we only update the value of the aborted flag if we are setting it to true since this operation causes potential race conditions that can alter the desired behavior of abort.

The alternative implementation of abort is considerably more complex due to the intricacies of manual memory management in C++. Figure 7 shows a sketch implementation of the interface. Naively, `abort` simply sets its `aborted` flag and recursively calls abort on its children. When we speculate by calling the second constructor, the new Abort object is recorded in the parent. When an Abort object is destroyed, it is removed from the parent's children list. When `abort` is called, the code iterates the list of children and aborts each one. Note, however, that this can race with the children's destructor so we use a combination of compare-and-swap and mutexes to protect against accessing freed memory. While a lock-free implementation would likely be superior, the locking implementation is already fairly complicated and we were unable to finish the lock-free version.

4. ANALYSIS

In this section we analyze the trade-offs inherent in our technique. At the high level, we are interested in three aspects: First, the performance implications of the different Abort implementations. Second, the effect of polling granularity on the useful work that our system does. And third, we wish to quantify the ease with which the technique can be

```
class Abort {
  Abort *parent;
  bool aborted;

 public:
  Abort()
    : aborted(false), parent(NULL) { }

  explicit Abort(Abort* p)
    : aborted(false), parent(p) { }

  int isAborted() const {
#ifndef POLL_UP_CACHE
    return this->aborted
        || (this->parent &&
            this->parent->isAborted());
#else
    if (this->aborted) return 1;
    if (this->parent &&
        this->parent->isAborted())
      const_cast<Abort*>(this)->aborted = true;
    return this->aborted;
#endif /* POLL_UP_CACHE */
  }

  inline void abort() {
    this->aborted = true;
  }
};
```

Figure 6: Implementation of `Abort` that polls upward.

applied to existing code.

4.1 Cost of Different Aborts

In Section 3.2 we proposed three different implementations of the Abort library. In this section, we do our best to compare these implementations in as fair a manner as possible. We begin at a theoretical level by considering the amount of work performed by each of the operations. Since both `abort` and `isAborted` are sequential in all of our implementations, we consider only the amount of work done.

Consider first the push-down implementation in which the `abort` propagates the abort downward and `isAborted` simply checks a flag. It is easy to see that `isAborted` is constant time while `abort` takes time proportional to the size of the tree being aborted ($|C|$).

$$\begin{aligned} \texttt{abort}_{\downarrow} &\in \Theta(|C|) \\ \texttt{isAborted}_{\downarrow} &\in \Theta(1) \end{aligned}$$

In this implementation, `abort` will be considerably more expensive than `isAborted`; however, `abort` does work proportional to the amount of work that is saved by the abort since each node can be aborted only once and, in theory, once a node is aborted, no additional work is done at that node.

The implementation of poll-up has the opposite tradeoff. The `abort` procedure is constant time making it cheap to abort even a large subtree. Instead the cost is paid in more expensive `isAborted` calls. The `abort` procedure takes time linear in the depth of the tree ($\log |S|$) being aborted or lin-

```
class Abort {
  list<Abort*> children;
  Abort** parent;
  bool aborted;

  cilk::mutex m;

private:
  void registerListener(Abort* res) {
    m.lock();
    if (this->aborted) {
      res->parent  = NULL;
      res->aborted = true;
    } else {
      children.push_back(res);
      res->parent = &(*children.end())
    }
    m.unlock();
  }

public:
  Abort() : aborted(false), parent(NULL) { }

  Abort(Abort* parent) {
    parent->registerListener(this);
  }

  ~Abort() {
    if (this->parent &&
        !__bool_cas(this->parent, this, NULL)) {
      while (!this->aborted);
      m.lock();
      m.unlock();
    }
  }

  inline int isAborted() const {
    return this->aborted;
  }

  void abort() {
    Abort** tmp = this->parent;
    if (tmp && !__bool_cas(tmp, this, NULL))
      return;

    m.lock();

    if (this->aborted) {
      m.unlock();
      return;
    }

    this->aborted = true;

    for (iterator itr = children.begin();
         itr != children.end(); itr++) {
      Abort* r = *itr;
      if (!r) continue;
      if (__bool_cas(&*itr, r, NULL)) {
        r->abort();
      }
    }
    m.unlock();
  }
};
```

Figure 7: Sketch implementation of `Abort` that pushes the abort flag down.

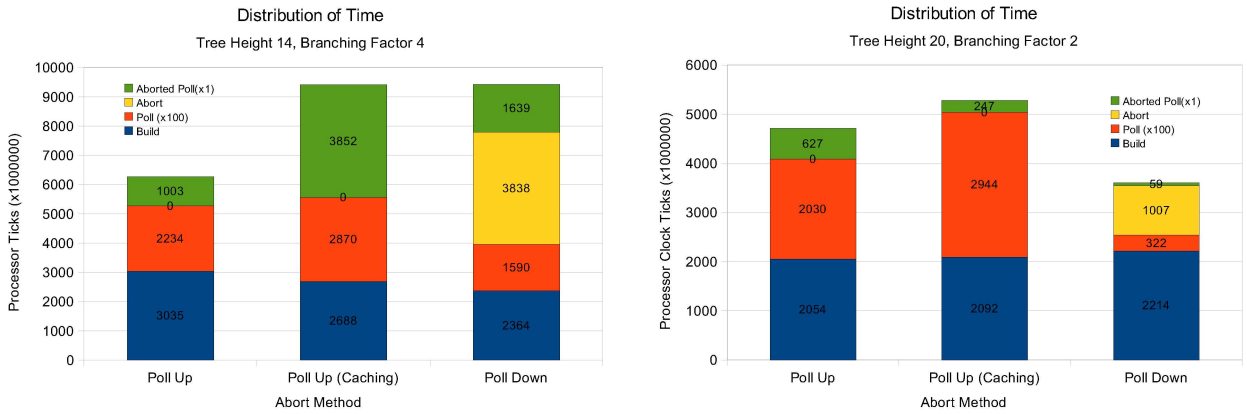

Figure 8: Time for building, polling and aborting a tree.

ear in the depth of the entire tree if the node is not aborted.

$$\begin{aligned} \texttt{abort}_{\uparrow} &\in \Theta(1) \\ \texttt{isAborted}_{\uparrow} &\in O(\log |S|) \end{aligned}$$

The benefit of using one algorithm or another will depend on a variety of factors including: the amount of polling that the system does and the size of trees being aborted. Therefore, we consider the following experimental setup. We perform, in parallel, a breadth-first construction of an abort tree of height h and branching factor b, gathering the leaves into a list (**Build**). We then randomly permute the list in order to avoid some of the cache-effects. Considering all of the leaves in parallel, we poll 100 times at each leaf (**Poll (x100)**). Note that these nodes are not aborted yet. We then abort the root node (**Abort**). Finally, we poll a single time at each leaf, noting that this time the node is aborted. Figure 8 shows the time that each of these phases takes in processor clock ticks.

The difference in tree size and shape drastically changes whether the poll-up or push-down implementation is more efficient. When the tree height is 14 with a branching factor of 4, poll-up is considerably faster than poll-down, but this changes completely when increasing the tree height to 20 and decreasing the branching factor to 2. It appears that the cost of polling repeatedly doesn't overcome the cost of the sequential abort until the tree height is quite large.

Note also the difference between the orange and green segments. Even on the naive poll-up strategy polling up 100 times only takes twice as much time as polling up once when the tree height is 14. This suggests that subsequent poll-ups are essentially free which is most likely due to caching effects. Traversing the tree upward a second time simply results in 14 more hits to the L1 cache which is very cheap on current hardware.

Finally, we note that while the caching is meant to be an optimization on the poll-up strategy, it actually degrades performance considerably. This is likely due to the added complexity of the code which requires several jumps in addition to the recursion. The graph of height 20 suggests that this overhead is proportional to the size of the tree, since it is not significant for this smaller test, but predominates in the larger tree test case.

4.2 Varying the Frequency of Polls

One additional variable common to all three polling methods is how often polling should be performed. The cost of polling, as elaborated on in the previous section, is not trivial, but polling less frequently means aborted threads waste time by performing useless work. To measure this effect we ran the ported Pousse code using our poll-up strategy without caching and vary the frequence that client code poll.

The implementation uses the iterative deepening search strategy which performs the standard α-β search to a given level d and iteratively increases d until time runs out. This strategy ensures that there is always a decent, valid move. Because aborts that occur closer to the leaves result in less work being saved, there should be a tradeoff between the depth of polling and the amount of work that is done.

We measured the tradeoff between work and polling depth by considering the number of nodes that different polling strategies were able to search. For our tests, the search algorithm was given 8 seconds per turn, a roughly constant amount of that is used for input and output. The strategies that we tested are:

- **Aggressive** polls at the beginning of and periodically throughout `search`.
- **Coarsen-n** polls at the beginning of `search` when the remaining depth is greater than or equal to n. For example, Coarsen-2 does not poll at the last two levels of the tree.
- **Spawn** polls only before a **cilk_spawn** statement and thus avoids spawning extra work, but does not abort currently running work.

The results of our trials are shown in Figure 9.

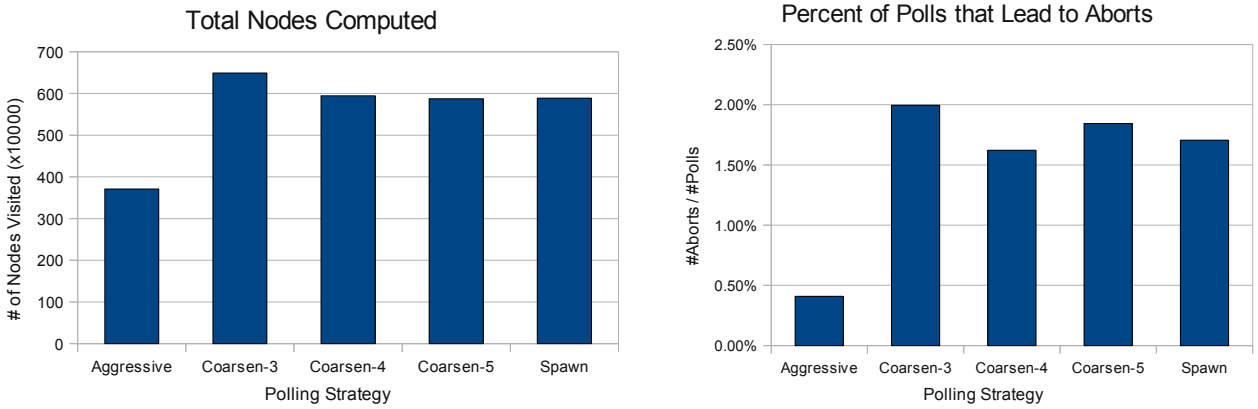

Figure 9: Pousse performance modifying the granularity of polling.

Code Type	Lines of Code
`cilk5` Pousse	956
Inlet Code	41
`cilk++` Pousse	1011
Env. Decls.	15
Inlet Code	52
Code Increase	5.07%

Figure 10: Code metrics for porting `cilk5` Pousse to `cilk++` Pousse.

We note that the more aggressively we poll the fewer nodes we are exploring. This is because we are wasting more of our time doing polling which leaves less time for the actual search. We peak at **Coarsen-3** which does significantly better than both **Coarsen-2** and **Coarsen-4**.

Since polls are purely wasted work if they don't result in an abort, we consider the percentage of polls that lead to aborts. The graph shows that this percentage peaks at 2% which agrees with the total nodes computed graph for the best strategy. The percentage of polls that lead to abort is considerably lower for the **Aggressive** polling strategy. This, at least partially, explains their low node counts because a larger portion of their polls are wasted.

A confounding factor with these tests is that the static evaluator, the function that heuristically assigns to a value to a game state, is relatively cheap for the Pousse game. For many other games, this may not be the case and could impact the effectiveness of not polling at the leaves of computation where the static evaluator is performed.

4.3 Ease of Use

In addition to quantitatively analyzing the performance of our strategy, it is also important to consider the amount of work that is required to make use of our technique. To measure this, we consider the amount of work that was required to port the existing `cilk5` Pousse code to work with `cilk++`. Figure 10 gives the quantitative difference in code metrics for porting the code. These numbers do not include the size of the Abort library.

Our experience shows a 5.07% increase in the number of lines of code (an increase of 63 lines). This seems fairly reasonable considering that we are implementing a primitive language construct by a code transformation. On a more qualitative note, the transformation was relatively painless, our initial translation had no bugs outside of the Abort library. In addition to making porting code easy, this also means that reading and maintaining the code is only slightly more difficult than reading and maintaining the original `cilk5` Pousse code. The largest difference here is the need to explicitly denote which variables need to be accessible in the inlet so that they can be explicitly placed within the structure passed as the continuation environment.

While the transformation is simple to perform, the problem with it is that it is not modular because it changes the signatures of functions which use the abort mechanism. This breaks the possibility for separate compilation without explicit annotations specifying which functions should be compiled to work with inlets and and abort. However, if the need to abort does not need to be exposed across interface boundaries, it can be effectively isolated.

The other problem with the polling strategy in general is determining when to poll. Our results in Section 4.2 suggest that polling relatively infrequently is quite effective, at least for our code. However, as we noted above, this might not always be the case. If static board evaluation or trying a move was more expensive it might be better to poll more often in order to avoid these overheads. In addition, as we saw in the previous section, it is often beneficial to change the polling strategy at different levels in the search tree based on how much work will be saved. Retrofitting this into some algorithms may require additional work.

5. CONCLUSIONS

We demonstrated how speculative parallelism in the style of `cilk5` using `inlet` and `abort` can be implemented as a simple CPS translation with a small amount of additional work threading through an Abort object. Our results show a performance improvement over the same α-β search algorithm that does not use abort. In addition, our experiences suggest that the translation is relatively easy to carry out when the transformation is local to a single module.

In general analyzing parallel programs is more difficult than analyzing sequential programs, but we found that analyzing programs that use speculative parallelism is even harder. The primary difficulty is because small changes in scheduling decisions can drastically affect the amount of work performed by the system if analyzing a particular node triggers a large abort. Perhaps one way to analyze this is to consider a notion of *virtual work* which is the amount of work that would be done by the mini-max algorithm if it was allowed to complete the current depth of the tree. In our analysis, we attempted to control this when we compared the abort implementation; however, we found that this strategy contributed relatively little insight into the Pousse code because the frequency of aborts is much more random.

5.1 Future Work

Our implementation focuses heavily on the porting of `cilk5` Pousse. This is good because it has the behavior of a real game, but it does not incorporate a very sophisticated/expensive static board evaluator like other games. Especially since the polling method requires making choices about how often to poll, we believe that further exploration of a game that uses a more sophisticated static board evaluator would shed additional light on problems likely to be encountered in parallelizing other code.

In `cilk5` abort is implemented in the runtime system using signals. This is nice because it avoids the need to poll which is especially painful because it obviates the need to tune an extra polling frequency parameter. In addition, implementing inlets via a compiler transformation is convenient because the compiler can mechanically generate additional code to make speculating across module boundaries more reasonable and ensure returns through continuations are compiled efficiently as tail-calls whenever possible. We believe that it would be beneficial to re-implement `abort` and `inlet` in the runtime using signals in order to compare the total performance overhead of polling. In addition to the compiler support, we belive that additional tools for gathering statistics such as the number of times aborts occurred and estimates for the amount of work that was aborted would greatly help developers in understanding their algorithms and how best to achieve efficient implementations.

6. REFERENCES

[AJ89] A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 293–302, New York, NY, USA, 1989. ACM.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Notices*, 30(8):207–216, 1995.

[EK72] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.

[Flo62] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Notices*, 33(5):212–223, 1998.

[KM75] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293 – 326, 1975.

[Lei09] Charles E. Leiserson. The cilk++ concurrency platform. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 522–527, New York, NY, USA, 2009. ACM.

[Str69] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.