Revisiting the Sequential Programming Model for Multi-Core

Matthew J. Bridges Neil Vachharajani Yun Zhang Thomas Jablin David I. August

Department of Computer Science
Princeton University
{mbridges, nvachhar, yunzhang, tjablin, august}@princeton.edu

Abstract

Single-threaded programming is already considered a complicated task. The move to multi-threaded programming only increases the complexity and cost involved in software development due to rewriting legacy code, training of the programmer, increased debugging of the program, and efforts to avoid race conditions, deadlocks, and other problems associated with parallel programming. To address these costs, other approaches, such as automatic thread extraction, have been explored. Unfortunately, the amount of parallelism that has been automatically extracted is generally insufficient to keep many cores busy.

This paper argues that this lack of parallelism is not an intrinsic limitation of the sequential programming model, but rather occurs for two reasons. First, there exists no framework for automatic thread extraction that brings together key existing state-of-the-art compiler and hardware techniques. This paper shows that such a framework can yield scalable parallelization on several SPEC CINT2000 benchmarks. Second, existing sequential programming languages force programmers to define a single legal program outcome, rather than allowing for a range of legal outcomes. This paper shows that natural extensions to the sequential programming model enable parallelization for the remainder of the SPEC CINT2000 suite. Our experience demonstrates that, by changing only 60 source code lines, all of the C benchmarks in the SPEC CINT2000 suite were parallelizable by automatic thread extraction. This process, constrained by the limits of modern optimizing compilers, yielded a speedup of 454% on these applications.

1 Introduction

Until recently, increasing uniprocessor clock speed and microarchitectural improvements could be counted upon to provide performance improvement for all programs. Today, this is no longer true. Instead, processor manufacturers now use the continuing growth in transistor count to place multiple cores on a processor die. Machines with four or more cores are already shipping, and tomorrow's machines promise still more cores. However, these additional cores only improve the performance of multi-threaded applications. The prevalence of single-threaded applications means that these cores often provide no benefit.

To encourage development of multi-threaded applications, many new languages have been proposed to ease the burden of writing parallel programs [4, 9, 11]. While these languages make parallel programming easier than in the past, the effort involved in creating correct and efficient parallel programs is still far more than that of writing the equivalent single-threaded version. Developers must be trained to program and debug with parallel concerns in mind (i.e. deadlock, livelock, race conditions). Converting an existing single-threaded application is often worse, as it was not developed to be easily parallelized in the first place.

Automatic parallelization techniques that extract threads from single-threaded programs without programmer intervention do not suffer from this cost. Because of this, these techniques should be preferred over manual parallelization. Unfortunately, except in the scientific domain, automatic parallelization techniques have not proven successful enough for broad adoption.

This paper argues that this lack of success has occurred for two reasons. First, no existing framework has brought together the many techniques in compiler analysis, compiler optimization, and hardware support to allow for extraction of parallelism. As this paper shows, many applications can be parallelized through the integration of these existing techniques in a modern compiler with whole-program scope. Second, for the many applications which are not parallelized by these existing techniques, the problem with automatic extraction of threads comes from artificial constraints imposed by sequential execution models. In particular, the programmer is often unable to specify that multiple legal outcomes of the program exist. Because of this, the compiler is forced to maintain the single correct output that a sequential program specifies, even when others are more desirable. By expressing multiple correct execution orders, a parallelization can often be achieved automatically without the cost of moving to a multi-threaded programming model. Interestingly, the extensions to the sequential model are simple and natural, minimizing the burden on programmers.

To illustrate this position, this paper explores as a case study the manual parallelization of the SPEC CINT2000 benchmarks. Where possible, the parallelization was performed, by compiler writers familiar with the technology, as a modern parallelizing compiler could be expected to perform. Many of the methods used have been explored in the compiler and architecture community, but have not been combined. This combination of existing methods with simple and natural program changes empowered automatic thread extraction to find a high degree of parallelism.

In summary, the contributions of this paper are:

- A description of the necessary compiler and hardware techniques to facilitate automatic parallelization.
- An extension to the existing sequential programming model that allows the programmer to communicate legal alternate program outcomes to the compiler.
- A characterization of changes to each program along with a performance analysis of the resulting effect for the SPEC CINT2000 suite.

Section 2 provides an overview of existing techniques and augmentations to the sequential programming model that can be brought together to parallelize applications. A description of the parallelization and simulation methodology is given in Section 3. Section 4 shows how the framework and extensions from Section 2 allow for the parallelization of the applications in the SPEC CINT2000 suite. Related work in parallelization is presented in Section 5. Section 6 concludes the paper.

2 Framework for Automatic Parallelization

This section describes the framework proposed for automatic parallelization. This framework includes a compiler infrastructure to identify parallelism in sequential code, hardware to efficiently execute the parallelized code, and extensions to a sequential programming model to expose additional parallelism. Using the SPEC CINT2000 benchmark suite as case studies, Section 4 will show how this framework enables the automatic extraction of parallelism.

2.1 Compiler and Hardware Support

To extract parallelism from sequential code, two promising methodologies have been presented in the literature, Thread-Level Speculation (TLS) [13, 30] and Decoupled Software Pipelining (DSWP) [20, 26]. TLS techniques speculatively execute subsequent iterations of a loop before the current iteration finishes, attempting to extract DOALL parallelism. Instead of executing loop iterations in parallel, DSWP partitions each loop iteration into a series of stages, and then executes the stages in parallel, with stages potentially executing code from different iterations of the loop concurrently.

However, for successful parallelization, neither technique is very effective in its original form. For example, with TLS, some dependences must be synchronized, rather than speculated, to avoid excessive misspeculation [30]. Likewise, to maximize parallelism using TLS, cores should not stall waiting to commit a completed speculative iteration, but should be provided with sufficient buffering resources to begin executing subsequent iterations.

Likewise, DSWP must also be extended. In particular, it must support speculation and leverage TLS-like memory subsystems to privatize memory [33]. To leverage additional cores and handle imbalanced pipeline stages, DSWP must replicate stages that contain no loop-carried dependences. This allows DSWP to extract more scalable DOALL parallelism by allowing different iterations to run in parallel on the same static code, similar to TLS.

Both TLS and DSWP require judicious use of speculation to break infrequent or easily predictable dependences inhibiting parallelization. This involves not only alias speculation, but also value speculation [18, 25] and control speculation. The hardware substrate should ensure that misspeculation is avoided if at all possible. For example, silent stores [15] should avoid triggering alias misspeculation, and, similarly, stored values should be eagerly forwarded to later threads to avoid misspeculation [10].

Finally, the compiler should leverage modern transformations and analyses to avoid over-estimating dependences. These techniques range from traditional techniques like reduction expansions [19, 21] to more modern techniques, such as aggressive alias analysis [5], speculative pointer analysis [28], or variable value analysis [22]. Proving two memory operations do not conflict or proving that a variable holds a constant value at a certain program point can be invaluable in unlocking parallelism.

2.2 Compilation Scope

The compiler and hardware support described in the previous section is often sufficient to parallelize portions of an application. Our experience parallelizing the SPEC CINT2000 suite, however, showed that significant parallelism existed at or close to the outermost application loop. Consequently, it is important that the parallelization framework is powerful enough to identify and leverage parallelism that occurs at any loop level in the code.

The ability to find, analyze, and optimize a loop without regard to its position in the code is non-trivial. The scale of the problem at the outer loop level introduces compile-time concerns. Applying transformations (e.g. inserting synchronization) that touch code deeply nested within function calls introduces another degree of complexity. By using whole program optimization [32], procedure boundaries can be removed, giving the compiler the ability to both see and modify code, regardless of location in the program. Additionally, through region formation, the compiler can control the amount of code to analyze and optimize.

2.3 Extending the Sequential Programming Model

Even with the framework described thus far, some applications will defy automatic parallelization. For many of these applications, there is no single required order of execution or even a single correct output. Rather, a multitude of execution orders and outputs are correct, though syntactically or semantically different. This subsection introduces two extensions to a sequential programming model to present this information to the compiler, allowing the extraction of parallelism.

2.3.1 *Y-branch*

Of those applications with many correct outputs, a large subset of these have outputs where some are more preferable than others. When parallelized by hand, the developer must make a choice that trades off parallel performance for optimality of output. Instead, this flexibility should be given to the compiler, as it is often better at targeting the unique features of the machine it is compiling for. To this end, we propose the use of a *Y-branch* in the source code. The semantics of the *Y-branch* is that for all dynamic instances, the *true* path can be taken regardless of the condition of the branch [35]. The compiler is then free to generate code that pursues this path when it is profitable to due so. In particular, this allows the compiler to balance the quality of the output with the parallelism achieved.

```
dict = start_dictionary();
while ((char = read(1)) != EOF) {
  profitable = compress(char, dict)

  @YBRANCH(probability=.00001)
  if (!profitable)
    dict = restart_dictionary(dict);
}
finish_dictionary(dict);
```

(a) *Y-branch*

```
#define CUTOFF 100000
dict = start_dictionary();
int count = 0;
while ((char = read(1)) != EOF) {
  profitable = compress(char, dict)

  if (!profitable) {
    dict = restart_dictionary(dict);
  } else if (count == CUTOFF) {
    dict = restart_dictionary(dict);
    count = 0;
  }
  count++;
}
finish_dictionary(dict);
```

(b) Manual Choice of Parallelism

Figure 1. Motivating Example for *Y-branch*

Figure 1a illustrates a case where the *Y-branch* can be used. The code is a simplified version of a compression algorithm that uses a dictionary. Heuristics are used to restart the dictionary at arbitrary intervals. Rather than inserting code to split the input up into multiple blocks, as is done in Figure 1b, the *Y-branch* communicates to the compiler that it can control when a new dictionary is started, allowing it to choose an appropriate block size. This gives the compiler the ability to break dependences related to the dictionary, and extract multiple threads. A probability argument informs the compiler of the relative importance of compression to performance. In the case of Figure 1a, a probability of .00001 was chosen to indicate the dictionary should not be reset until at least 100000 characters have been compressed. Determination of the proper probability is left to a profiling pass or the programmer. Simple metrics, such as the minimum number of characters, are often sufficient.

2.3.2 *Commutative*

Many functions have the property that multiple calls to them are interchangeable even though they maintain internal state. Figure 2 is the code for the random number generator, `Yacm_random`, from `300.twolf`, which contains an internal dependences recurrence on the *seed* variable. Multiple calls to this function will be forced to execute serially due to this dependence. The *Commutative* annotation informs the compiler that the calls to `Yacm_random` can occur in any order.

```
static int seed;

@Commutative
int Yacm_random() {
  int temp = seed / 127773L;
  seed = 16807L * (seed - temp * 127773L)
         - (temp * 2836L);
  if( seed < 0 )
    seed += 2147483647L;
  return seed;
}
```

Figure 2. Motivating Example for *Commutative*

In general, the *Commutative* annotation allows the developer to leverage the notion of a commutative mathematical operator, which can facilitate parallelism by allowing function calls to execute in any order. This annotation is similar to commutativity analysis [27]: both have the goal of facilitating parallelization. However, calls to *Commutative* functions are generally not commutative in the way that commutativity analysis requires. Commutativity analysis looks for sets of operations whose order can be changed, but that result in the same data in the same location. *Commutative* functions can be executed in an order that leads to different values than the sequential version. The differences that result are only relevant inside the function and not in the application as a whole. Finally, the programmer annotates

Commutative based on the definition of a function and not the many call sites it may have, making it easy to apply.

The semantics of the *Commutative* annotation is that, outside of the function, the outputs of the function call are only dependent upon its inputs. This allows the compiler to reorder calls to a *Commutative* function without the internal dependence getting in the way. The *Commutative* function itself executes atomically when called and, inside the function, dependences that are local to the function are respected. This ensures that a well-defined sequence of calls to the *Commutative* function exists.

The *Commutative* annotation can also take an argument which communicates that groups of functions share internal state. When parallelizing, any function in the group must execute atomically with respect to every function in the group. For example, `malloc` and `free` would all use the same argument since they share internal state.

The use of *Commutative* in a speculative execution environment requires additional care. There must always be a well-defined sequential sequence of calls to the *Commutative* function, particularly in the face of rollback of state or versioning of memory. For the purposes of this paper, a well-defined ordering was maintained by ensuring that *Commutative* functions executed in non-transactional memory and that a rollback function existed to undo the effects of calls to the *Commutative* function. For example, the rollback function for `malloc` was `free`.

3 Obtaining and Measuring Parallelism

This work explores how single-threaded applications can be parallelized to leverage tomorrow's many-core processors. Additionally, much of the parallelism discussed in the paper is obtained by parallelizing loops close to the outermost program loop. Unfortunately, parallelizing large fragments of an application and targeting many cores renders traditional simulation techniques impractical. This section will discuss how parallel performance was measured. Subsequent section will describe in detail the techniques necessary to parallelize the applications studied in this work.

3.1 Measuring Parallel Performance

To efficiently measure parallel performance, a combination of native execution and simulation was used. Each single-threaded application was decomposed into a set of tasks. Each task corresponds to a region of a single loop iteration and was statically marked in the application's code. These regions were selected to maximize parallelism and were not constrained to be contiguous regions of the static code, similar to existing algorithms [20]. Once the regions were selected, the code was instrumented to use hardware performance counters to measure the time spent executing each dynamic task, a dynamic instance of the statically marked region. For clarity, *phases* refer to statically selected regions and *tasks* refer dynamic instances of a phase.

The native runtimes were obtained by compiling the application using '-O3' optimizations with the gcc compiler version 3.4. The resulting binary was run on a HP workstation zx2000 with a 900Mhz Intel Itanium 2 processor and 2GB of memory, running CentOS release 4.4. Execution times were obtained using the performance counters of the IPF architecture and the *pfmon 3.0* tool [8].

For each application, an execution plan was devised describing which core(s) would be responsible for executing tasks from a particular phase. The execution plan (described in Section 3.2), along with the task execution times, a task dependence graph, and a simulator was then used to estimate the total execution time of the parallel application. The model assumes that tasks communicate via shared memory and core-to-core communication queues. It further assumes a versioned memory hardware subsystem [33], allowing for privatization of data and memory alias speculation. In this paper, studies were conducted for machines ranging from 1 to 32 cores, and the simulator accurately modeled full and empty conditions on 256 32-entry queues.

Task dependences were obtained via static analysis for register dependences and memory dependences [5]. The parallelizations assume various dependences could be speculated. The speculation was modeled by informing the simulator of the dynamic dependences that actually occurred, which were obtained from a memory profiling pass run prior to simulation. This effectively models serialization (loss of benefit for speculative execution) due to misspeculation, but imposes no additional cost to misspeculation.

Finally the simulator does not model microarchitectural effects, such as bandwidth limitations or cache coherence. However, our results are on par with existing manual parallel results for several applications [7, 25, 31], when the same number of threads, usually 4, are used. This suggests that this methodology at least captures all first-order effects.

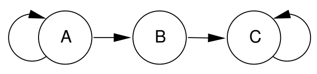
3.2 Parallelization Paradigm

All application loops studied in this paper were decomposed into three phases, with each phase manifesting a different dependence pattern. Ignoring dependences that were speculated, the tasks from the first phase of each application depended only on prior tasks from the first phase. Tasks from the second phase depended on the corresponding task (from the same original loop iteration) from the first phase. Finally, tasks from the third phase depended on the corresponding task from the second phase as well as prior tasks from the third phase. Figure 3b illustrates this static phase dependence graph for a simple code example in Figure 3a.

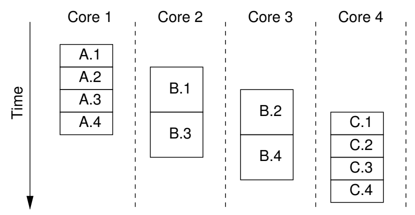
This dependence pattern naturally leads to an execution plan where tasks from the first phase were executed serially on a single core. Tasks from the second phase were then executed in parallel with one another through dynamic assignment to the core with the least amount of work enqueued.

```
  while (condition) {
A:  line = read();
B:  result = work(line);
C:  printf(result);
  }
```

(a) Example Code



(b) Static Stage Dependences



(c) Potential Task Execution

Figure 3. Parallelization Diagram

Finally, like the first phase, tasks from the third phase executed serially on a single core. Figure 3c illustrates this execution for the code example.

This pattern is a generalization of Decoupled Software Pipelining (DSWP) [20, 26], augmenting it with speculation and the ability to run many parallel versions of a single pipeline stage. Note that while a DSWP-style execution plan was used in the experiments in this paper, similar parallelizations and results could be obtained with execution plans that more closely resemble TLS.

4 Case Studies

To illustrate the applicability of the framework and augmented sequential programming model, this section describes the manual application of the techniques from Section 2 to the C applications in the SPEC CINT2000 benchmark suite. In particular, for each benchmark, the loop and its parallelization are described. Particularly problematic dependences in each application and the solution to overcoming them are also described.

4.1 Parallelizable by the Framework

Several benchmarks can be parallelized using just an aggressive parallelization framework, including `181.mcf`, `153.perlbmk`, `155.vortex`, and `156.bzip2`, whose speedups on up to 32 threads are shown in Figure 4.

4.1.1 256.bzip2

`256.bzip2` is an application to compress and decompress a file using the Burrows-Wheeler transform and Huffman encoding. This paper focuses only on the compression portion of the benchmark. Each file is compressed, using the `compressStream` function, in independent blocks of the same size. The block size itself is dependent upon the compression level and varies between 100KB (lowest compression) and 900KB (highest compression).

Versions of the bzip2 algorithm that compress independent blocks in parallel have been implemented in parallel-programming paradigms [23]. The parallelization for `256.bzip2` is effectively the same as these hand parallelized versions, but can be extracted automatically using the DSWP parallelization rather than manually with mutexes and locks. The phase A thread reads in each block. Effectively, this means that for each iteration, the *block* variable, which holds the data for each iteration to compress, is privatized by the TLS memory subsystem. Each block is then compressed in parallel by executing the `doReversibleTransformation` and `moveToFrontCodeAndSend` functions in one of several phase B threads. Writes into the output stream are buffered until the position of the writes are known in phase C. The only limitation to performance is the input file's size, which is only a few megabtyes. Combined with a high level of compression, only a few independent blocks exist to compress in parallel.

4.1.2 255.vortex

`255.vortex` is a derivative of a single-user object-oriented database transaction benchmark. The high-level loop in the `BMT_Test` function tests the database with a series of commands. In particular, the inner loop of `BMT_Test` calls the Lookup, Delete, and Create functions in that order. Each call to these functions will lookup, delete, or create a number of database items as specified by an input file. The actual part to lookup, delete, or create is based on a random number(s).

The parallelization for this application executes the iterations of the loops in `BMT_CreateParts` and `BMT_DeleteParts` in parallel. To achieve this, both value speculation and alias speculation are necessary.

Value speculation is needed for the *STATUS* variable. `255.vortex` relies heavily on the usage of this variable that is passed as an argument to almost every function. Each function will appropriately update *STATUS* to indicate NORMAL execution or one of many failure conditions. Most function calls in the benchmark are written as *if func(STATUS)*, where *func* will return false only if *STATUS* is not NORMAL. Speculating a value of *NORMAL* for *STATUS* around the backedge is essential to breaking several loop-carried dependences.

Alias speculation is used to deal with the rare case that an update to the database is dependent on a previous update's modification of the internal representation. Specifically, the internal structure of the database is a B-tree, which is only rarely rebalanced during calls to create and delete. Alias

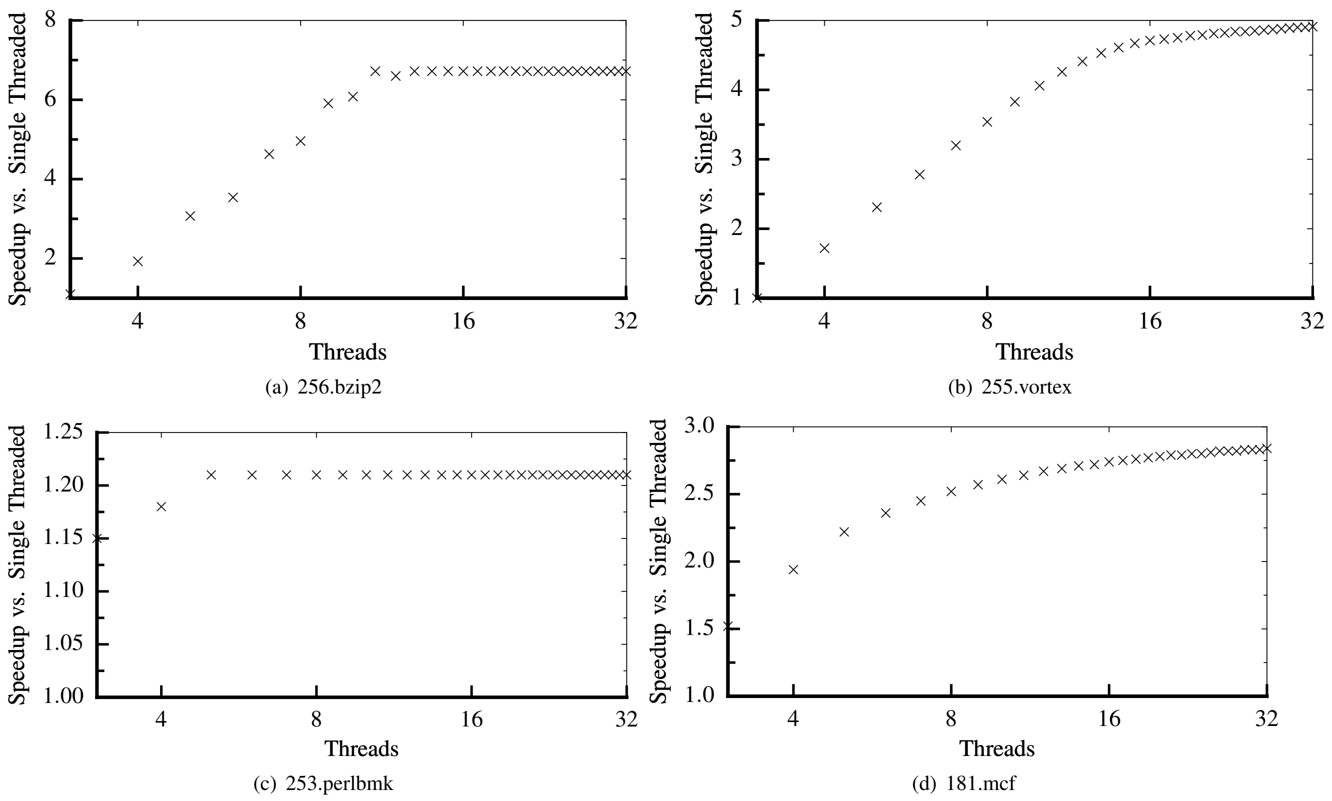


Figure 4. Speedup of multi-threaded (MT) execution over single-threaded (ST) execution.

speculation also removes dependences that occur from the expansion of memory blocks, effectively a `realloc` on the internal memory manager, in calls to `ExpandChunk`. Alias misspeculation on these dependences, though rare, is the limiting factor in the speedup obtained.

4.1.3 253.perlbmk

`253.perlbmk` is an interpreter for the Perl language. The loop in the `Perl_runops_standard` function executes a sequence of operations, including the high-level operations derived from the input file. Parallelization of this loop speculatively executes statements from this input in parallel, taking advantage of data-independent Perl statements in the source file to achieve speedup.

Unfortunately, it is not easy to find these statements in the code. In particular, the `Perl_runops_standard` loop simply executes the current operation. This execution returns the next operation to execute or *null* to indicate that the loop should exit. Perl source-level statements are composed of operations demarcated by NEXTSTATE operations. Many dependences exist among these sequences of operations that are not directly related to the input. In particular, operations are evaluated using a virtual stack machine, which leads to many memory dependences.

Fortunately, when a NEXTSTATE operation is executed, it is likely that many global variables in this virtual machine will have the same values as they did just after the previous NEXTSTATE operation. In particular, value profiling reveals that the *PL_stack_sp* and *PL_temp_ixs* variables will often have the same value every time a NEXTSTATE operation finishes. A compiler can introduce a loop to precompute the next NEXTSTATE operation by speculatively chasing the *next_op*. Combined with value speculation to break the dependences by asserting and checking that the *PL_stack_sp* and *PL_temp_ixs* variables have this property around the backedge, series of operations representing statements can execute in parallel. The parallelization is limited by misspeculation that occurs because the input statements are truly data dependent.

4.1.4 181.mcf

`181.mcf` is an application that solves the single-depot vehicle scheduling problem in public mass transportation, essentially a combinatorial optimization problem solved using a network simplex algorithm. The high-level loop occurs in `global_opt` which calls both the `price_out_impl` and `primal_net_simplex` functions.

Parallelizing `181.mcf` is non-trivial and requires parallelizing several loops. `primal_net_simplex` takes approximately 65-75% of the execution time and can first be parallelized by allowing refresh_potential to execute in parallel with the rest of primal_net_simplex. This is accomplished by speculating that refresh_potential will not change the actual potential of any node in the tree, which is almost always the case [33]. The performance of the remaining primal_net_simplex can also be improved by a parallelization of the two loops in primal_bea_mpp, as performed in prior work [25].

The remaining 25-35% of the runtime occurs in price_out_impl, which consists of an outer and inner loop. With alias speculation, the outerloop can be parallelized, though it is important that the *arcout->head->firstout->head->mark* update be placed in phase A to avoid almost constant misspeculation. Additionally, the inner loop can also be parallelized, as in previous work [25].

After all these loops are optimized, performance is limited mostly by misspeculation in price_out_impl and the lack of scalable parallelism in the primal_net_simplex parallelization.

4.2 Specifying Alternate Outcomes

Though existing techniques can parallelize the applications in Section 4.1, many are still left unparallelized by this framework. In particular, the dependences that prevent parallelization or lead to excessive misspeculation are often a result of artificial constraints imposed by the sequential programming model. This section shows how the *Commutative* extension to the sequential programming model discussed in Section 2.3 allows the framework to extract parallelism. Figure 5 shows the speedups for the benchmarks described as the number of threads is increased.

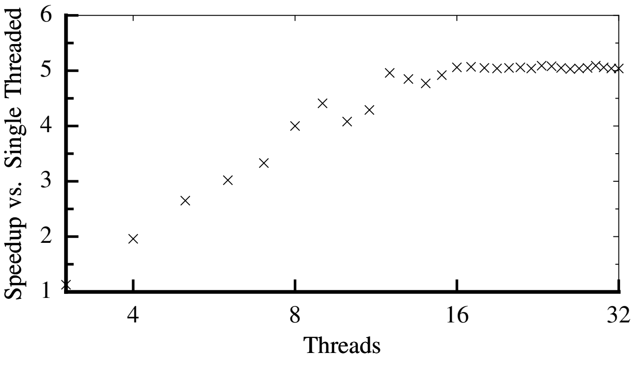
4.2.1 176.gcc

176.gcc is an application that compiles C programs down to MIPS assembly and can process only one C file per run. The outermost loop is the parse loop, which is generated from a pair of lex and yacc files. When the last token of a function is read in, the grammar action calls finish_function. Eventually, this function will call rest_of_compilation, which optimizes the function in a single pass of optimizations, though some optimizations are applied multiple times, and then prints the function to the assembly file. The optimization sequence dominates the runtime of the benchmark, accounting for 80-90% of the runtime, which is not surprising as reading and printing are roughly linear in the number of lines of code, while many compiler analysis and optimizations are $O(n^2)$ or worse. Since no interprocedural analysis or optimization is applied in the optimization sequence, the sequence can run in parallel on each function, though several dependences must be dealt with first.

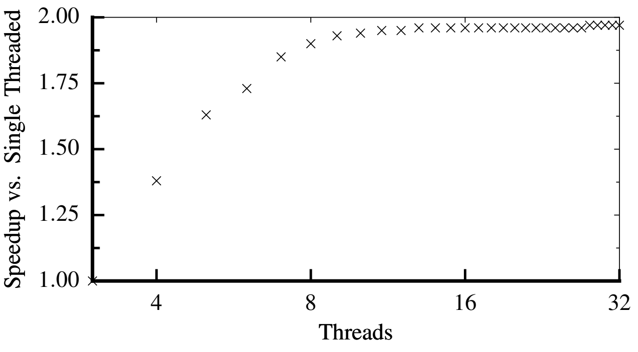
Alias speculation can be used to remove dependences that occur in the global symbol table, implemented as a hash table. The problematic dependence occurs because the table is updated with local symbols just before they are printed. Unfortunately, a large amount of misspeculation results, which, combined with a small number of functions to optimize, severely limits performance. Instead, the symbol table lookup and insert function is annotated as *Commutative*, which prevents the symbol table from causing misspeculation. Additionally, *Commutative* is also used to mark the memory allocation functions for the *permanent_obstack*. There are several other obstacks, which are essentially pointers into memory pools, that are value predicted to have the same value after phase B as they did before.

To avoid other unnecessary misspeculation, several bit flags were expanded to take entire bytes. In particular, this avoids misspeculation that occurs due to a read of the *common.public_flag* variable in the IR structure that appears to be fed by an update of the *common.static_flag* because they are contained in the same byte. If compiler analysis can determine that these loads and stores do not overlap based on field, the fields can be split into multiple locations. If this is not possible, then special loads and stores [14] can be used to communicate this to the alias conflict detection system.

Another problematic dependence exists on the global counter for labels, called *label_num*. This counter is up-



(a) 176.gcc



(b) 254.gap

Figure 5. Speedup over ST Execution.

dated as labels are created for each function, and labels are created throughout the processing of a function, while reading in, optimizing, and even when printing. Besides being printed, *label_num* is also used to malloc structures with size equal to the number of labels in a function and to test if a label is in the current function or a parent function when nested functions are used. This dependence is effectively impossible to speculate away.

Fortunately, from a programmer standpoint, it is legal to make *label_num* a two dimensional structure that is (function, number) instead of just (number). At the start of a function, the *label_num* variable is updated to point to the new function, and the number is reset to 0. This breaks the *label_num* dependence across functions, allowing parallelization. However, the output of the program is different in that the actual strings used for local labels have changed. Since the actual string used for these labels are irrelevant, so long as it is unique, the output is still semantically, though not syntactically, equivalent.

4.2.2 254.gap

`254.gap` is an interpreter for a computational discrete algebra programming language. The high-level loop in `main` follows the standard `Read`, `Evaluate`, `Print` structure of most interpreters. While there are many loops that account for small portions of the runtime, obtaining significant speedup requires that the parallelization attempt to run statements of the input program in parallel. Since statements can be dependent upon one another, alias speculation is used to obtain the parallelism. In particular, alias speculation must break the dependence on the *Last* variable, which stores the result of the last statement. This parallelization essentially speculates that the statements in the input file are data independent. For the parallelization to be extracted, the memory allocator that the application uses must be marked as *Commutative* .

For the input sets of `254.gap`, this parallelization obtains a speedup of almost 2x before misspeculation becomes a factor. Besides alias misspeculation due to true data dependences in the input statements, misspeculation also occurs because `254.gap` performs its own garbage collection. Not surprisingly, the copy garbage collection causes a large amount of the misspeculation because it touches all "memory", moving around objects to compact the space used. The use of a mark-and-sweep garbage collector would likely reduce the amount of misspeculation.

4.3 Improving Parallelizations

Annotations can be used not only to enable parallelizations, but to improve existing parallelizations. In particular, this fact can be leveraged to reduce the misspeculation of existing parallelizations, achieving the speedups shown in Figure 6.

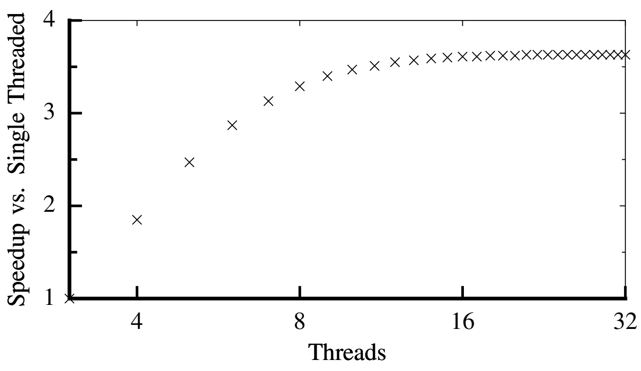
4.3.1 186.crafty

`186.crafty` is an application that plays chess. The high-level loop reads in a chess board and a search depth n. For each iteration of this loop, the `Iterate` function is called, which executes repeated searches from a depth of 1 to n. To perform this, `Iterate` calls the `SearchRoot` function, which calls the recursive `Search` function to perform an alpha-beta search. For each level of the search, several moves are computed, each of which is recursively searched and evaluated to determine the most profitable path. The most profitable move is stored and its alpha value returned.

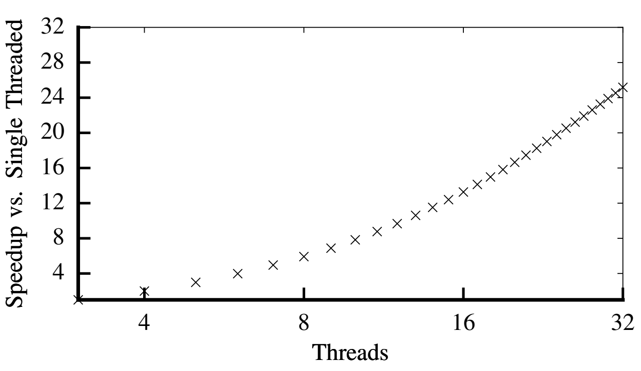
The most obvious parallelization is to search each of the root moves in `SearchRoot` independently, similar to the way the application has been parallelized by hand [6]. This parallelization requires that the *search* variable, which contains many fields related to the current search, be value predicted to be the same after each iteration as it was at the beginning of the iteration. This is always true, but is very hard for the compiler to predict as it requires understanding that the `UnMakeMove` function undoes the effects of `MakeMove` function. Additionally, control speculation is needed to prevent the presence of certain cutoff metrics related to timing and number of nodes searched from preventing parallelization. In particular, the *next_time_check* variable branch in `Search` must be speculated not taken.

Additional dependences manifest themselves on the many caches (ex. *pawn_hash_table* and *trans_ref*) that are used to prune the search space and improve single-threaded performance. Unfortunately, these caches prevent parallelism, as the dependence from the store into the cache to a load from the cache is hard to predict. Alias speculation can break these dependences, but the sheer amount of misspeculation limits performance. Instead, we rely on the programmer to mark each cache lookup function as *Commutative*, which removes dependences on the cache. The resulting speedup barely breaks 2x on 32 threads because the amount of time it takes to search a particular move is highly variable due to the aggressive pruning performed during the search.

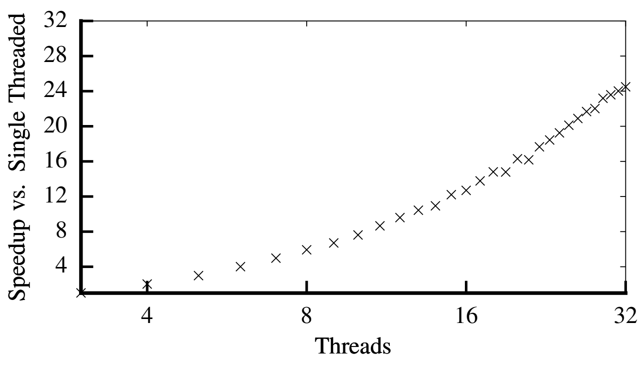
To obtain more performance, the `Search` function is also parallelized. `Search` has a behavior which is almost the same as `SearchRoot`, with the addition of more conditions to prune the search space. Additionally, `Search` is recursive, a problem that has hindered previous work in parallelization [25]. However, the `Search` function is small and the recursion can be "unrolled" by repeatedly specializing the function to a particular depth in the recursion. This was performed on `Search` to unroll the recursion one level, effectively parallelizing both the `SearchRoot` loop and the loop in the first call to `Search`. With this parallelization, the performance obtained scales with the number of threads.



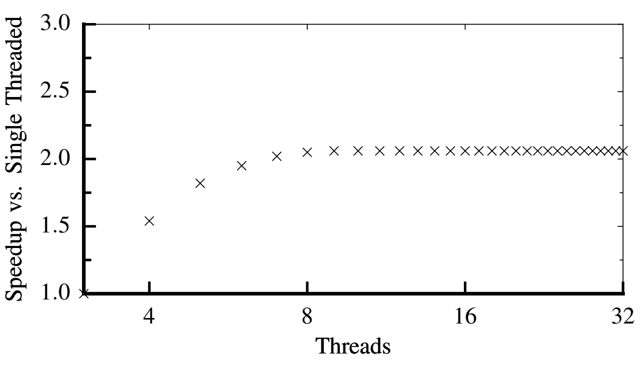
(a) 175.vpr



(b) 186.crafty



(c) 197.parser



(d) 300.twolf

Figure 6. Speedup over ST Execution.

4.3.2 197.parser

`197.parser` is an application that parses a series of sentences, analyzing them to see if they are grammatically correct. The loop in the `batch_process` function is the outermost loop, of which the `parse` function call dominates the runtime of each iteration. As each sentence is grammatically independent of every other sentence, parsing can occur in parallel for each sentence.

While previous manual parallelization [7] has used alias speculation to break dependences between iterations, it is not necessary so long as certain dependences are synchronized. In particular, a sentence may be a command for the parser rather than a sentence to parse, turning on or off echo mode, for example. However, speculation is not required for this application if these operations are placed into the phase A thread. The loss in parallelization is limited, as a majority of the time is taken up by the `parse` call.

To achieve a parallelization that parses each sentence in parallel, dependences arising from the memory allocator must be removed. Upon startup, `197.parser` allocates 60MB of memory, regardless of input, which it then manages internally. To avoid dependences from the memory allocator interfering with parallelization, it is marked with *Commutative* annotation, as in `254.gap`. The memory manager does not perform garbage collection, thus the parallelism achieved scales well with the number of cores, limited only by the time it takes to parse the longest sentence.

4.3.3 300.twolf

`300.twolf` is an application that performs place and route simulation. The loop that comprises most of the execution time is in the `uloop` function, which contains many calls to `ucxx2`. The `ucxx2` function takes up approximately 75% of the execution time. Previous work has executed portions of the `ucxx2` code in parallel via speculative pipelining [25]. This paper instead parallelizes calls to `ucxx2` by parallelizing iterations of the loop in `uloop`.

Predicting which iterations can execute in parallel *a priori* is hard, so value and alias speculation are used to achieve the parallelization. However, misspeculation greatly limits the amount of parallelism extracted [25]. This misspeculation comes from two sources, misprediction of the number of calls to the pseudo-random number generator and memory alias violation on the block and network structures.

The value misspeculation for the random-number generator occurs because of the variable number of calls to the random number generator. In previous work [25], this dependence has been broken by manually speculating the number of calls to the generator and predicting the next iteration's seed. However, it seems counterintuitive for parallelism to be limited by the generation of random numbers. To avoid the misspeculation and allow the compiler to see the parallelism, this paper proposes that the random number generator be marked as *Commutative* by the programmer. For the pseudo-random number generator, this allows the calls to the generator to occur in any order, and, in particular, breaks the dependence that the generator has across iterations on the *randVarS* variable. Though output changes as a result of this, the benchmark still runs as intended.

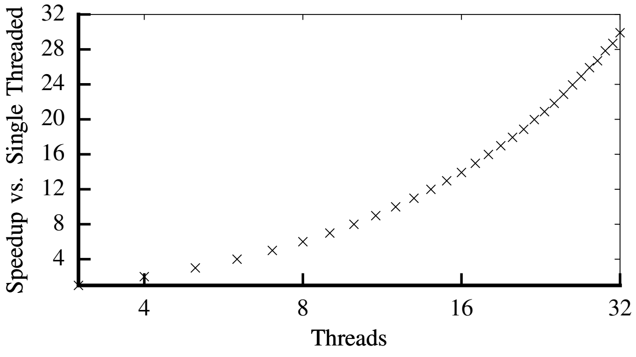
4.3.4 175.vpr

`175.vpr` is an application that performs FPGA place and route calculations. As in previous work [25], this paper focuses on the placement portion of the algorithm, which is distinct from the routing portion. Placement consists of repeated calls to `try_swap` in the `try_place` function. `try_swap`, as its name implies, attempts to switch a block to a random position, also swapping the block at that position if one is already there. A pseudo-random number generator is used to choose a block and an (x, y) position to move the block. Like `300.twolf`, if the coordinates chosen are the same as the block's *x* and *y* value, then a new random coordinate pair is generated until they are distinct. The block's coordinates are then updated and the cost of updating the connecting networks is calculated. If the cost is below a certain threshold, the swap is kept; otherwise, the block's coordinates are reverted to their previous values.

As in previous work [25], the calls to `try_swap` can often be speculatively executed in parallel. Predicting dependences among these iterations *a priori* is hard, so value and alias speculation are used to achieve the parallelization. However, the amount of parallelism extracted is limited by misspeculation, particularly in early iterations of the `try_place` loop. This misspeculation comes from two sources, misprediction of the number of calls to the pseudo-random number generator and memory alias violation on the block coordinates and network structures. As in `300.twolf`, the random number generator is marked as *Commutative*, to avoid misspeculating on it. The alias misspeculation on block structures can be reduced by value speculating that the loads of the block coordinates and network structures will not change.

Additionally, good parallel performance requires many threads, as the misspeculation rate of the iterations in `try_place` varies greatly. In the earlier outer loop iterations, the speculation fails more than 80% of the time, while in the later iterations, the speculation succeeds more than 80% of the time. This occurs because later outerloop iterations impose more stringent conditions on acceptable swapping. Because of this, the amount of parallelism obtained is largely dependent upon the number of threads used in later outer loop iterations.

4.4 Varying output with performance



(a) 164.gzip

Figure 7. Speedup over ST Execution.

4.4.1 164.gzip

`164.gzip` is another compression and decompression application, which uses the Lempel-Ziv 1977 algorithm. As with `256.bzip2`, this paper focuses on the compression portion of the benchmark. Each file is compressed in blocks by either the `deflate` or `deflate_fast` function, which have almost the same code. Unlike `256.bzip2`, the choice of when to end compression of the current block and begin a new block is made based on various factors related to the compression achieved on the current block. This dependence makes it impossible to compress blocks in parallel as it is very hard to predict the point at which a new block will begin.

Manually parallelized versions of the gzip algorithm insert code to ensure that a new block is started at fixed intervals, usually 128KB [24]. To allow the benchmark to be parallelized, a similar change was made to the source code to always start a new block at a fixed interval. As the block size is smaller than bzip's, the parallelism scales to a larger number of threads, as shown in Figure 7.

Unfortunately, this change can cause the application to achieve a lower compression ratio than the single-threaded version. When there are multiple processors, this tradeoff between compression and performance is acceptable. Using the fixed-size blocking, the average compression loss was less than 1%. In general, though, this loss of compression should only occur if parallelization was achieved. This can be achieved through the use of the *Y-branch*, just as in Figure 1a.

4.5 Summary

Table 1 summaries information about the parallelizations performed in this section. For each application, the loop(s) parallelized are given as the function, file, and line num-

Benchmark	Loop	Approx. Exec. Time	Lines Changed (All)	Lines Changed (Model)	Techniques Required
164.gzip	`deflate_fast` (deflate.c:583-655)	30%	26	2	*Y-branch*, TLS Memory, DSWP
	`deflate` (deflate.c:664-762)	70%			
175.vpr	`try_place` (place.c:506-513)	100%	1	1	*Commutative*, Alias, Value, & Control Speculation, TLS Memory, DSWP
176.gcc	`yyparse` (c-parse.c:1396-3380)	95%	18	8	*Commutative*, Alias & Control Speculation, TLS Memory, DSWP
181.mcf	`price_out_impl` (implicit.c:228-273)	25%	0	0	Alias & Control Speculation, TLS Memory, DSWP
	`primal_net_simplex` (psimplex.c:50-138)	75%			Control & Silent Store Speculation, TLS Memory, DSWP
	`primal_bea_mpp` (pbeampp.c:161-172)	4%			Alias Speculation, DSWP, Nested
	`primal_bea_mpp` (pbeampp.c:181-195)	20%			Alias Speculation, DSWP, Nested
186.crafty	`SearchRoot` (searchr.c:52-153)	100%	9	9	*Commutative*, TLS Memory, DSWP, Nested
	`Search` (search.c:218-368)	98%			
197.parser	`batch_process` (main.c:1522-1779)	100%	3	3	*Commutative*, TLS Memory, DSWP
253.perlbmk	`Perl_runops_standard` (run.c:30)	100%	0	0	Alias, Control & Value Speculation, TLS Memory, DSWP
254.gap	`main` (gap.c:191-227)	100%	3	3	*Commutative*, TLS Memory, DSWP, Alias Speculation
255.vortex	`BMT_CreateParts` (bmt01.c:82-252)	20%	0	0	Alias & Value Speculation, TLS Memory, DSWP
	`BMT_DeleteParts` (bmt10.c:371-393)	70%			
256.bzip2	`compressStream` (bzip2.c:2870-2919)	100%	0	0	TLS Memory, DSWP
300.twolf	`uloop` (uloop.c:154-361)	100%	1	1	*Commutative*, Alias & Control Speculation, TLS Memory, DSWP

Table 1. Information about the loop(s) parallelized, the execution time of the loop, the number of lines changed by the programmer, the number of lines changed by the programmer within the augmented sequential model, and the techniques required for the parallelization.

bers it occupies. Additionally, the approximate execution time and the number of lines changed by the programmer, both overall and only using the *Y-branch* and *Commutative* annotations are given. Finally, the techniques required to parallelize each application are described.

Table 2 gives the best speedup achieved for each application and the minimum number of threads for which that speedup occurs. Additionally, the *Moore's Law Speedup* is given, which details the expected performance increase for the number of transistors used. While there are no statistics that directly relate the doubling of cores to performance improvement, historically, the transistors on a chip have doubled every 18 months, while performance has doubled every 3 years. Assuming that all new transistors are used to place new cores on a chip, each doubling of cores must yield approximately 1.4x speedup to maintain existing performance trends. The numbers in this column represent the speedup expected for the minimum number of threads used. The final column gives the ratio of the actual performance improvement to that required to maintain the 1.4x speedup. The overall performance improvement indicates that sufficient parallelism can be extracted to utilize the resources of current and future many-core processors.

5 Related Work in Parallelization

Many research techniques have focused on changes or extensions to existing programming languages to enable easier specification of parallelism [4, 9, 11]. Unfortunately, these approaches do not alleviate all problems of parallel programming. Traditional locking libraries, such as *pthreads*, provide the programmer with the ability to express parallelism, but give little support in achieving correct or effective parallelism. More advanced systems, such as Cilk [9] or OpenMP, provide higher level parallelization primitives, including means to automatically schedule parallelism for performance, but provide little help in achieving correctness. The Cilk *inlet* directive, in particular, is similar to the *Commutative* directive, as it ensures correct execution of code in a non-deterministic fashion. However, *inlet* is meant to serially update state upon return from a spawned function, while *Commutative* is meant to facilitate parallelism by removing serialization.

Alternative techniques, such as memory transactions [4] have been recently proposed to help the programmer ex-

Benchmark	# Threads	Speedup	Moore's Speedup	Ratio
164.gzip	32	29.91	5.38	5.56
175.vpr	15	3.59	3.71	0.97
176.gcc	16	5.06	3.84	1.32
181.mcf	32	2.84	5.38	0.53
186.crafty	32	25.18	5.38	4.68
197.parser	32	24.50	5.38	4.55
253.perlbmk	5	1.21	2.18	0.55
254.gap	10	1.94	3.05	0.64
255.vortex	32	4.92	5.38	0.91
256.bzip2	12	6.72	3.34	2.01
300.twolf	8	2.06	2.74	0.75
GeoMean	17	5.54	3.97	1.39
ArithMean	20	9.81	4.16	2.04

Table 2. The minimum # of threads at which the maximum speedup occurs. Assuming a 1.4x speedup per doubling of cores, the *Moore's Speedup* column gives the speedup needed to maintain existing performance trends. The final column gives the ratio of actual speedup to expected speedup.

press parallelism in an easier manner, but suffer from correctness [12] and performance [36] issues. If the program can be fit into a specific paradigm, specialized languages, such StreamIt [11], can handle the correctness issues and schedule the parallelism for good performance. Unfortunately, many programs and data structures do not fit into these paradigms, or must be rewritten from scratch to take advantage of these languages.

The automatic extraction of scalable parallelism has been achieved on scientific programs through the extraction of DOALL parallelism [2] from loop nests accessing arrays [3, 17, 29]. General-purpose programs have not been so fortunate, leading to many solutions. Beyond the techniques discussed in Section 2.1 [13, 20, 26, 30], there are several existing automatic parallelization techniques that spawn speculative threads along multiple branch paths [34] and procedure calls [1].

Manual parallelization of applications with the intent of guiding automatic techniques has also been explored by Prabhu et al. [25]. Their manual parallelization of 6 applications, including 3 SPEC CINT2000 benchmarks, is designed to guide future compiler TLS work. The parallelizations described are used as inspiration in the `175.vpr` and `181.mcf` applications. However, this paper is notably different from ours in that it does not consider all SPEC CINT2000 benchmarks, it uses TLS for parallelization, and does not allow the programmer to aid the compiler.

Finally, several techniques in the literature advocate an integrated approach to the extraction of parallelism, combining both manual and automatic parallelization. SUIF Explorer [16] is a tool for programmer specification of parallelism, which is then assisted by tools support to ensure correctness. The Software Behavior-Oriented Parallelization [7] system allows the programmer to specify intended parallelism. If the intended parallelization is incorrect, the worst case is single-threaded performance at the cost of extra execution resources. The parallelization chosen for the `164.gzip` and `197.parser` applications are effectively the same as those in this paper. Concurrent work proposed by Thies et al. [31] also extracts parallelism in conjunction with the programmer. Through profiling of the dynamic behavior of the program, their system extracts pipelined parallelism, and produces parallelizations for `197.parser` and `256.bzip2` that are effectively the same as those described in Section 4. However, their system relies upon the programmer to mark the potentially parallel regions and to often perform several transformations to make the program amenable for parallelization, but that reduce the software-engineering quality of the code.

6 Conclusion

Due to the large number of existing single-threaded applications, automatic parallelization techniques are necessary for performance on tomorrow's processors. Unfortunately, these techniques have yet to extract sufficient parallelism. Parallel programming models offer the ability to extract this performance, but at a high cost.

This paper has shown that, with the proper framework, comprised of existing analysis and optimization techniques along with the proper compilation scope, large amounts of parallelism can be extracted. For those applications that are not parallelized by this framework, simple additions to the standard sequential programming model are proposed to allow the framework to parallelize them. In particular, the SPEC CINT2000 benchmark suite was used as a case study to show that this framework and programming model can be applied to many applications. This allows a software developer to develop in a sequential programming model, but still obtain performance from parallelization.

Acknowledgments

We thank the entire Liberty Research Group for their feedback during this work. Additionally, we thank the anonymous reviewers for their insightful comments. The authors acknowledge the support of the GSRC Focus Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program. This work has been supported by Intel Corporation. Opinions, findings, conclusions, and recommendations expressed throughout this work are not necessarily the views of our sponsors.

References

[1] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *31st International Symposium on Microarchi-*

tecture, 1998.

[2] J. R. Allen and K. Kennedy. *Optimizing compilers for modern architectures: A dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[3] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Lee, T. Lawrence, J. Hoefinger, D. A. Padua, Y. Paek, P. Petersen, W. M. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Restructuring programs for high-speed computers with Polaris. In *ICPP Workshop*, 1996.

[4] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos transactional programming language. In *Conference on Programming Language Design and Implementation*, 2006.

[5] B.-C. Cheng and W. W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Conference on Programming Language Design and Implementation*, 2000.

[6] Crafty. http://www.craftychess.com/.

[7] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *Conference on Programming Language Design and Implementation*, 2007.

[8] S. Eranian. Perfmon: Linux performance monitoring for IA-64. http://www.hpl.hp.com/research/linux/perfmon/, 2003.

[9] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Conference on Programming Language Design and Implementation*, 1998.

[10] M. J. Garzarán, M. Prvulovic, J. M. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas. Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. *ACM Transactions on Architecture Code Optimization*, 2(3):247–279, 2005.

[11] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[12] D. Grossman, J. Manson, and W. Pugh. What do high-level memory models mean for transactions? In *Workshop on Memory System Performance and Correctness*, 2006.

[13] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, 2000.

[14] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Symposium on Principles and Practice of Parallel Programming*, 2006.

[15] K. M. Lepak and M. H. Lipasti. Silent stores for free. In *33rd International Symposium on Microarchitecture*, 2000.

[16] S. Liao, A. Diwan, R. P. Bosch Jr., A. M. Ghuloum, and M. S. Lam. SUIF explorer: An interactive and interprocedural parallelizer. In *Symposium on Principles and Practice of Parallel Programming*, 1999.

[17] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Symposium on Principles of Programming Languages*, 1997.

[18] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *29th International Symposium on Microarchitecture*, pages 226–237, 1996.

[19] S. A. Mahlke, W. Y. Chen, J. C. Gyllenhaal, W. W. Hwu, P. P. Chang, and T. Kiyohara. Compiler code transformations for superscalar-based high-performance systems. In *International Conference on Supercomputing*, 1992.

[20] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *38th International Symposium on Microarchitecture*, 2005.

[21] G. Ottoni, R. Rangan, A. Stoler, M. J. Bridges, and D. I. August. From sequential programs to concurrent threads. *IEEE Computer Architecture Letters*, 4, June 2005.

[22] J. R. C. Patterson. Accurate static branch prediction by value range propagation. In *Conference on Programming Language Design and Implementation*, 1995.

[23] Parallel bzip2 (pbzip2). http://compression.ca/pbzip2/.

[24] Pigz: Parallel gzip. http://zlib.net/pigz15.c.gz.

[25] M. K. Prabhu and K. Olukotun. Exposing speculative thread parallelism in SPEC2000. In *Symposium on Principles and Practice of Parallel Programming*, 2005.

[26] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *International Conference on Parallel Architectures and Compilation Techniques*, 2004.

[27] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. In *Conference on Programming Language Design and Implementation*, 1996.

[28] J. D. Silva and J. G. Steffan. A probabilistic pointer analysis for speculative optimizations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[29] Stanford Compiler Group. SUIF: A parallelizing and optimizing research compiler. Technical Report CSL-TR-94-620, Stanford University, Computer Systems Laboratory, May 1994.

[30] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, 2005.

[31] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *International Symposium on Microarchitecture*, 2007.

[32] S. Triantafyllis, M. J. Bridges, E. Raman, G. Ottoni, and D. I. August. A framework for unrestricted whole-program optimization. In *Conference on Programming Language Design and Implementation*, 2006.

[33] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *International Conference on Parallel Architectures and Compilation Techniques*, 2007.

[34] S. Wallace, B. Calder, and D. M. Tullsen. Threaded multiple path execution. In *International Symposium on Computer Architecture*, 1998.

[35] N. Wang, M. Fertig, and S. J. Patel. Y-branches: When you come to a fork in the road, take it. In *International Conference on Parallel Architectures and Compilation Techniques*, 2003.

[36] C. Zilles and D. Flint. Challenges to providing performance isolation in transactional memories. In *Workshop on Duplicating, Deconstructing, and Debunking*, 2005.