

ABSTRACT

YU, YING. Risky Thread Extraction: An Infrastructure For Aggressive Programmer Assisted Parallelization. (Under the direction of James Tuck.)

With the ubiquity of multicore processors, parallelizing programs to exploit multicores is increasingly important. Automatic parallelization techniques are important as an alternative to the arduous task of manually parallelizing programs. However, automatic techniques are often unable to parallelize programs even when significant parallelism exists. Contrary to human parallelization, compilers are limited by what they can prove, preventing discovery of correct parallelization. The conservative nature of data dependence analysis is a common limitation in parallelization. Speculative parallelization techniques allow the compiler to overcome unknown dependences and parallelize the code, but the success and the extent to which it can be applied is limited because a recovery mechanism is necessary to detect when the parallelization was incorrect. Such recovery mechanisms come with considerable overhead or are limited in how they detect misspeculations. However, if the code is truly parallel, this limitation serves no value and simply prevents parallelization. Instead, if we make risky changes to the program and allow a human to decide when it is correct, we may be able to achieve better parallelization that always works.

Using a genetic algorithm, we have implemented a compiler that arrives at a better parallelization compared to baseline by evolving set of data dependence relationships that can be successfully ignored and proving that the resulting parallelization offers high performance and correct execution. Our genetic algorithm works by ignoring may-alias dependence edges on critical hard-to-parallelize loops to find hidden parallelism. To make our genetic algorithm converge quickly and so that a programmer can understand which dependences it speculated upon, we developed a novel chromosome representation that groups relaxed dependence edges in multiple classes based on semantic level information. To test the correctness of our parallelization techniques, we compare the output of the program with the known correct outputs. Once the final selected parallelization solution is found, the relaxed dependence relationships are provided to the programmer for examination and approval. It is up to the programmer to decide if the result is semantically valid.

We implemented our risky thread extraction on top of the Decoupled Software Pipelining parallelization algorithm. Furthermore, in order to extract as much parallelization as possible, we implemented two new passes: communication optimization and stage replication. Our evaluation of the risky parallelization infrastructure proves that it is able to exploit more parallelism that is limited by base DSWP framework. On the test set of SPECint benchmarks, our infrastructure achieves an average 23% over sequential execution and 14% over our base framework of

DSWP. In the best case, we achieve a 50% improvement over our base DSWP implementation.

© Copyright 2012 by Ying Yu

All Rights Reserved

Risky Thread Extraction: An Infrastructure For Aggressive Programmer Assisted
Parallelization

by
Ying Yu

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Engineering

Raleigh, North Carolina

2012

APPROVED BY:

Eric Rotenberg

Huiyang Zhou

James Tuck
Chair of Advisory Committee

DEDICATION

To my family and my friends and whoever helped. This thesis is a way to express my gratitude about the help given to me in the past and my confidence about future life as a return.

BIOGRAPHY

Ying Yu was born on July 29th, 1985 in the city of Chang Chun, China. She stayed in Changchun until her college in Dalian University of Technology. Her childhood is an interesting and exciting experience. She was able to develop her interested in painting and reading and enjoyed herself from doing that. After high school, she went to Dalian University of Technology (DUT), Software School to have her college life as a student in Computer Science area. Dalian is a beautiful and fantastic city, and DUT is also a famous university that provides high quality education and good opportunities. Ying had a memorable experience there and then after graduation she studied and worked in the University of Science and Technology of China, one of the best research universities in China for a year before her master program in America. In August 2010, Ying joined Department of Electrical and Computer Engineering to work on her master degree of Computer Engineering in North Carolina State University in Raleigh, NC. She conducted thesis research under the supervision of Dr. James Tuck and focused on compiler optimization.

ACKNOWLEDGEMENTS

It is a hard and rewarding study during these two years in NC State. I could not have a better experience without the help of many persons. First of all, I want to express my appreciation and respect to my adviser Dr. James Tuck, who have given me continuous direction and advice towards both my research and life. I also would like to thank all the instructors from whom I learned a tremendous amount of fascinating knowledge in my area. Those will serve me well in my future career in industry after I graduate. Last but not least, I want to thank my parents for their support, advice and consideration during these two years, which makes my life and choice become very easy. I want to thank my boyfriend Deng Zhou who is a master student in University of Science and Technology of China. He is always there to share my happiness and sadness, and gives me coverage to face any difficulties in my life. I am glad he will finish his study in China and chooses to come to America to continue his PHD study.

TABLE OF CONTENTS

List of Tables	vi
List of Figures	vii
Chapter 1 Introduction	1
Chapter 2 Background	4
2.1 DSWP	4
2.2 Genetic Algorithm	8
Chapter 3 Risky Thread Extraction	10
3.1 Overall Approach	10
3.2 Genetic Algorithm	11
3.2.1 Chromosome	11
3.2.2 Mutation and Recombination	12
3.2.3 Termination Criteria	13
3.3 Testing	13
Chapter 4 Decoupled Software Pipelining Optimizations	15
4.1 Communication Optimizations	15
4.1.1 Replicating Code to Eliminate Unnecessary Communication	15
4.1.2 Optimization for Software Queues	17
4.1.3 Computational Bottlenecks through Stage Replication	21
Chapter 5 Evaluation	24
5.1 Set up	24
5.2 Risky Thread Extraction	25
5.3 Optimizations	30
5.3.1 Communication Optimization	30
5.3.2 Software Queue	33
5.3.3 Stage Replication	34
5.4 Discussion	35
Chapter 6 Related Work	36
Chapter 7 Conclusion	38
References	39

LIST OF TABLES

Table 5.1	Architecture simulation details	24
Table 5.2	Details of genetic run on <i>sesc.</i>	26
Table 5.3	False memory dependence edges in the loop.	28

LIST OF FIGURES

Figure 2.1	PDG for loopExample	5
Figure 2.2	Task partition for loopExample	6
Figure 2.3	Genetic process.	9
Figure 2.4	Crossover and Mutation.	9
Figure 3.1	Risky Thread Extraction System.	10
Figure 3.2	Recombination.	13
Figure 3.3	RTE algorithm flow.	14
Figure 4.1	Loop chain replicate from thread 0 to thread 1.	17
Figure 4.2	Pipeline execution with the second stage replicated.	21
Figure 5.1	Benchmark simulation results of risky thread extraction.	25
Figure 5.2	Speedup for each loops in mcf06.	26
Figure 5.3	Performance improvement with different edges removed in mcf06.	27
Figure 5.4	Loop 4 in mcf2006.	27
Figure 5.5	Loop 5 in mcf2006.	28
Figure 5.6	Speedup for each loops in clustalw.	29
Figure 5.7	Performance improvement with different edges removed in clustalw.	30
Figure 5.8	Results of loop chain replication on x86 machine.	31
Figure 5.9	Results of loop chain replication on sesc machine.	31
Figure 5.10	Communication reduction for each benchmarks.	32
Figure 5.11	Comparison of sesc simulation for gzip,vpr and gap.	32
Figure 5.12	Execution results of software queue optimization.	33
Figure 5.13	Results of stage replication for bzip2.	34
Figure 5.14	Stage replication pattern for bzip2 with 4 threads.	34
Figure 5.15	Results of stage replication for vpr.	35

Chapter 1

Introduction

With multicore processors infiltrating every aspect of the modern computing landscape, program parallelization is more important than ever to fully exploit the capabilities of these new processors. While most programs are parallelized by hand explicitly using parallel programming languages [5] or libraries [16], such explicit parallelization technology requires programmers to identify the parallelism and verify the correctness and performance of parallelism by themselves. These requirements introduce significant difficulty and cost in parallelizing and debugging the code. If compilers can automatically parallelize code, it will greatly reduce the burden on the programmer and allow cost effective conversion of serial programs to multicores.

Automatic parallelization in compilers has been well studied. It has been used successfully at parallelizing DOALL loops[11, 15, 4]. However, irregular integer codes and codes with complex recursive data structures often have loop carried dependences which cannot easily be handled using a DOALL approach. These loops can be supported by Decoupled Software Pipelining (DSWP) [18]. Rather than spreading whole iterations across different cores, DSWP parallelizes a single iteration across cores by pipelining its execution. It must first determine which pieces of code are independent, and then it can schedule them in different pipeline stages such that communication between stages occurs in a directed, acyclic flow. Even though DSWP is able to parallelize a wide variety of loops, the degree of parallelization is often limited by the conservative capabilities of data dependence analysis and by the depth of the pipeline it can create.

One way to overcome the conservative nature of data dependence analysis is through speculation. Speculation can be applied to any parallelization technique by ignoring data dependences which cannot be proven, either as a dependence or not, at compile time and parallelizing the loop anyway. However, a runtime mechanism must observe the access patterns made by the speculative code and ensure they are consistent with the original sequential program. If a data dependence speculated not to occur does indeed occur, then the runtime mechanism must re-

cover from speculation and re-execute the loop in such a way as to provide correct execution. These techniques are getting better all the time, but do need expensive runtime mechanisms in either hardware or software to work well. Unless the overheads are kept low through hardware mechanisms or parallelism is abundant and recovery costs are kept to a minimum, they will not be profitable. Also, a more subtle point exists. Since all speculative changes must be recoverable, the cumulative effects of optimization based on speculative assumptions must not interfere with the runtime recovery mechanism. Hence, optimizations applied speculatively tend not to be as aggressive as their nonspeculative counterparts. While speculation is valuable, it inhibits the very process it is meant to enable.

Recently, some researches have proposed adding annotations or hints to their programs and to facilitate automatic parallelization on pointer-intensive C codes [27, 29]. While programmer annotations ease the parallelization process, these compilers still require careful examination and guidance from programmers in order to parallelize the codes.

Both speculative parallelization and programmer annotations are valuable, but we believe a better formulation is possible when the two are integrated in the right way. We want speculation to enable aggressive, unrestricted parallelization. The compiler can attempt multiple parallelization trials and select the best ones that work correctly on a software testing framework. Then, the programmer can identify if the parallelizations are correct and trustworthy by examining a few important relationships that the compiler chose to speculate upon rather than requiring identification of critical relationships from the programmer upfront. This enables aggressive parallelization without constraint and reduces the work required of the programmer. To support speculative parallelization with multiple trials, we adopted a genetic algorithm [21] which systematically searches for the best ways to parallelize a loop using Decoupled Software Pipelining. We form genes out of dependences on which we wish to speculate. Then, we recombine and mutate genes that are passed as input to our compiler so that it makes different assumptions during each parallelization trial. A gene that results in a successful execution that passes the test suite and provides high performance is kept in the next generation. Otherwise, they are kept or discarded at a certain probabilistic rate to ensure a rich gene pool. To ensure that we find parallelization that a programmer is likely able to validate, we have developed novel ways of classifying dependences within our gene.

In order to extract the full benefits of our approach, we needed to optimize our base DSWP parallelization algorithm. These optimizations included (1) communication reduction, (2) stage replication to use more threads and eliminate key pipeline bottlenecks, and (3) support for software queuing to support evaluation on real (not simulated) hardware. We explain these optimizations in detail in Section 4.

Altogether, the genetic algorithm coupled with our testing framework makes up our Risky Thread Extraction system. We applied our system on a set of SPECint and BioPerf benchmarks,

and we uncovered more parallelism than our original parallelization algorithm could on its own. Our risky parallelization leads to an average 23% improvement over sequential execution and 14% over our base framework of DSWP. To achieve that additional 14% performance improvement, the programmer only needed to validate that 24 dependence edges per benchmark, on average of all the benchmarks evaluated, could be ignored. The best speed up is observed in mcf06 benchmark, we are able to achieve a 50% improvement within 24 hours.

The optimization for communication reduction itself achieves a maximum 7% and an average 2.5% speed up on the SESC simulator and 25% speedup on 4-core x86 processor over DSWP.

We make the following contributions: (1) Brought up with an automatic compiler infrastructure for discovering parallelism in a relaxed dependence environment. (2) Implemented and evaluated several optimizations to reduce the overhead of thread extraction including reduction of communication and maintaining load balance between threads.

This thesis is structured as follows. Section 2 presents the background of DSWP and genetic algorithm, section 3 presents implementation of our risky thread extraction. Section 4 presents several optimizations we made to DSWP. Section 5 is the evaluation part of our compiler technique. Section 6 shows related work and section 7 concludes the thesis.

Chapter 2

Background

2.1 DSWP

Auto parallelization is NP-hard, hence we do not know if the parallelization is optimal or how close to that, thus our goal is always to do better. The baseline parallelization algorithm we use is Decoupled Software Pipelining. DSWP is a fully automatic nonspeculative thread extraction methodology to exploit pipeline parallelism. It achieves auto-parallelization by examining program dependence and extracting independent long-running workload concurrently executed threads.

Here is a brief introduction of the DSWP algorithm, detailed implementation will be shown later. First, a program dependence graph(PDG)[18] is constructed that includes all control, register and memory dependence edges for each loop and DSWP extracts independent working threads based on that. After building the PDG, it finds all the Strongly Connected Components (SCC)[18] in the PDG. A Strongly Connected Component (SCC) is a group of instructions and instructions in this group forms a cycle. DSWP requires acyclic communication between threads, thus instructions in the same SCC should be assigned to the same thread. Acyclic communication is much more effective than cyclic communication because it flows from one direction to the other direction, in our case, it flows from main thread to extracted threads. Thus only the execution of extracted threads dependent on main thread execution, but main thread execution will not be stalled for incoming communications. After all the SCCs are labeled in the graph, the graph will be coalesced such each node in the graph represents one SCC.

Next step is to select SCCs for different threads. The task selection algorithm will take communication cost and load balance problem into consideration, aimed to get a task selection with good runtime performance. Finally, system inserts communication channels between threads for required data communication during execution. There are three kinds of communications: data dependence, control dependence and memory dependence. Data dependence communication

means a value need to be transmitted, control dependence communication means a branch direction is required and memory dependence communication enforces operation ordering constraints.

Such thread extraction process is done during compile time, then when the program executes, system will generate multiple threads and each thread will work on its assigned task. Here is an example of loops that can be parallelized automatically using our thread extraction infrastructure.

Algorithm 1 loopExample

```

loopExample()
A: foreach i from 0 to bound do
B:   index=getIndex(i);
C:   array[index]=updateArray(index);
D:   bound=updateBound();

```

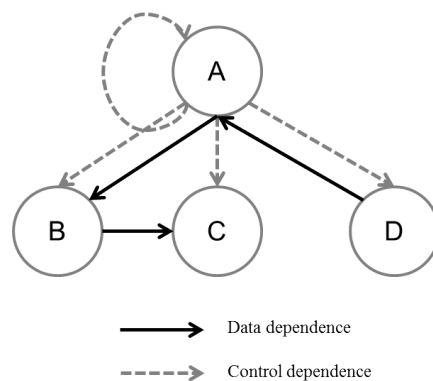


Figure 2.1: PDG for loopExample

loopExample shows a candidate loop for DSWP. It is a *for* loop working on updating array content. There are four parts for the loop, marked A,B,C and D and the dependence graph is shown in figure 2.1. Statement B, C and D are control dependent on the branch direction of A. With regard to dependences, B consumes data *i* from A, and C needs *index* from B, then D updates *bound* variable and passes it to A. One possible partitioning splits the PDG into three stages. The first stage includes A and D because they form a cycle, B is in the second stage and C is in the third stage. Those stages will be executed by different threads.

The implementation of our thread extraction is shown below. As described in function

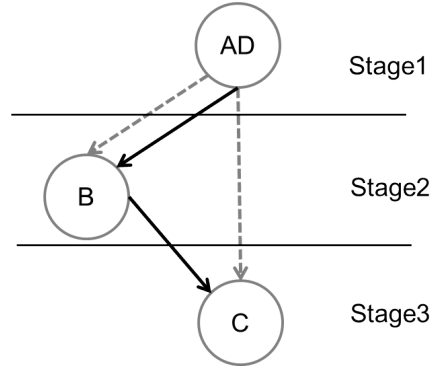


Figure 2.2: Task partition for loopExample

labelStages, it first builds PDG for the program then consolidates PDG to make each node to represent one SCC. Next it passes a list of SCC nodes to function *taskSelection* to select SCC for each thread. Function returns the number of pipeline stages, each thread will work on one stage.

Algorithm 2 labelStages

Input: 1 *f*, the original function

Input: 2 *tid*, thread id

Output: number of pipeline stages

labelStages(*f*,*tid*)

 build PDG *g* for *f*

 consolidate PDG

 construct a list of PDG nodes in topological order

 taskSelection(nodelist)

return number of stages

Task selection is crucial in determining program execution time. Bad selection could slow down the execution. The heuristic computes the estimated weight of each SCC using profile data. The algorithm maintains a list of nodes in the consolidated graph in an order that communication will flow from the header to the end of list, with no backward communication, this order is called topological order. The partition process is shown in function *taskSelection*, the nodes in the list will be selected one by one, and when the total estimated weight of the current task gets close to the overall weight divided by desired number of threads, the algorithm will end this task and start a new task to continue. When all the nodes have been properly assigned to different threads, we can continue to extract tasks from the original loop. At this point, the

Algorithm 3 taskSelection

Input: 1 a list of nodes in topological order

Output: number of partitions

taskSelection(nodelist)

foreach node n in nodelist *do*

if weight of partition P_i *plus* weight of n *lessthan* average weight

 add n to partition P_i

else

 end partition P_i , add n to partition P_{i+1}

endif

endfor

return number of partitions

extracted workload will not result in cyclic communication between main thread and extracted thread. This is guaranteed by assigning nodes one by one from the topological list.

Algorithm 4 extractThreads

Input: 1 f, the original function

Input: 2 S, set of statements that should be included in cloned function

Input: 3 tid, thread id

extractThreads(f,S,tid)

 createClonedFunction(f,S,tid)

 insertCommunicationChannel()

Algorithm 5 createClonedFunction

Input: 1 f, the original function

Input: 2 S, set of statements that should be included in cloned function

Input: 3 tid, thread id

Output: Cloned function

createClonedFunction(f,S,tid)

 f' = clone of f

foreach stmt s \in f *do*

if s not in S *then*

 remove s from f'

end

end

return f'

After task selection, system will collect statements belonged to different stages by traversing the consolidated graph and store them in different sets S . Function *extractThreads* creates the specific cloned function for a thread and then insert communication channels between threads. Input of the function is the original function in SSA form, set of statements collected for this stage and the stage id which is the same as thread id as each thread works on a particular stage. *createClonedFunction* creates a clone of the original loop in SSA form without communication inserted, and then removes statements that not belonged to this stage. *insertCommunicationChannel* inserts producer and consumer channel pairs for all the live in and live out variables. Since f is in SSA form, for any variable u that is defined in main thread and used in the extracted thread, the algorithm will insert a *send(u)* just after the definition of the variable in f , and replace its definition in f' with $u = recv()$.

DSWP system creates cloned function for all the extracted thread at compile time. When the program executed, main thread will pass the cloned function pointer to specific threads so they can execute their tasks. As the program is concurrently executed by multiple threads, the overall execution time should be reduced if the loop is big enough to compensate the overhead of creating threads and communication between threads.

2.2 Genetic Algorithm

A genetic algorithm is a search algorithm that mimics the natural process of evolution. In genetic algorithm, the evolution that is the search for a solution for certain problem usually starts from a population of randomly generated individuals and goes on for several generations until termination criteria is met. The population can be a string of 0s and 1s, or other encodings are also possible. These strings are called individuals and each bit in the string encodes a property of the problem. With the variation of solution strings, different solutions are created and evolve towards a better one. For different problems, different fitness functions are created and used to screen candidate individuals each generation. After evaluation using a fitness function for the solutions created each generation, some individuals are selected to generate the next population by crossover between two individuals or mutation. This evolution process will repeat until some termination criteria is reached. The algorithm could terminate when a certain number of generations has been tested, a satisfactory fitness value is achieved or timeout. Figure 2.3 shows the basic steps of genetic algorithm.

A genetic algorithm commonly has the following steps after solution string and fitness function are defined: initialization, selection, reproduction and termination. In the initialization stage, many individuals are randomly created, and the size of initial population may contain hundreds of solutions, depends on the nature of the problem. After generation of a population, the algorithm determines whether to keep or discard a solution by fitness calculation. The fitness

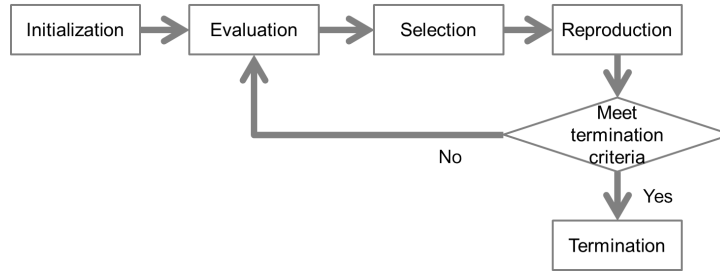


Figure 2.3: Genetic process.

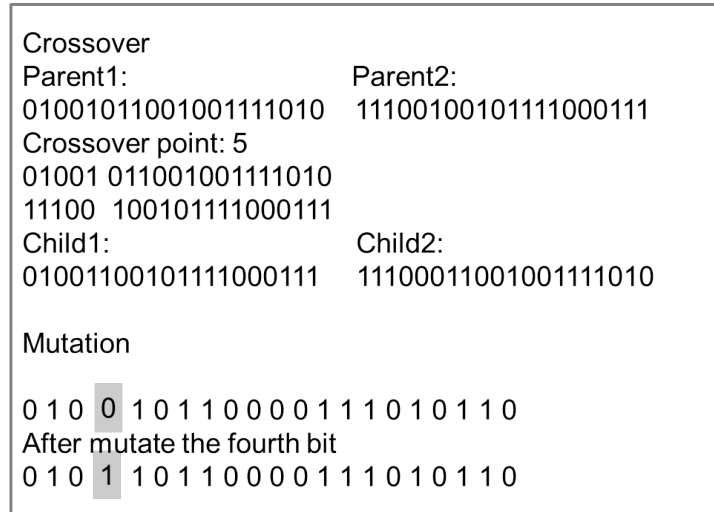


Figure 2.4: Crossover and Mutation.

value of individual solution decides whether it will be selected. Reproduction stage generates next population from the selected population last step. There are two common ways to create new individuals, crossover and mutation. Crossover is done by two parents switch their solution strings at a given crossover point, this will create two new solutions. And mutation is achieved by randomly mutate a bit in the string to a new value, it can be 0 to 1 or 1 to 0. Figure 2.4 shows an example of create child solution using crossover and mutation. By this way, child solution could inherit most of the characteristic of their parents. The genetic process will be repeated until termination condition is met. Termination condition could be a satisfactory performance generated, fixed number of generations tested, computation time out or a combination of the above.

In conclusion, a genetic algorithm is used to search a high dimensional space and usually takes several steps. 1. Initialization of a population. 2. Evaluation of fitness for each individual in that population. 3. Selection of the good individuals from this generation. 4. Production of next generation using selected individuals. 5. Go to step 2 and repeat until termination.

Chapter 3

Risky Thread Extraction

3.1 Overall Approach

The existence of many alias dependences restricts parallelization of the PDG, thus they are a key limitation of DSWP. The compiler is conservative in proving which edge is a must alias edge, so in reality, some of them must be kept but others could be removed. We want to remove those that can be removed and exploit better parallelism based on a relaxed dependence graph. To find a relaxed graph, we designed a risky thread extraction infrastructure using a genetic algorithm to generate several populations that are hundreds of parallelization solutions for the program, each population each solution corresponds to a relaxed dependence graph and some other features for program parallelization. Those solutions will be tested for correctness and performance by the simulator and the best solution of the ones considered will be selected at the end.

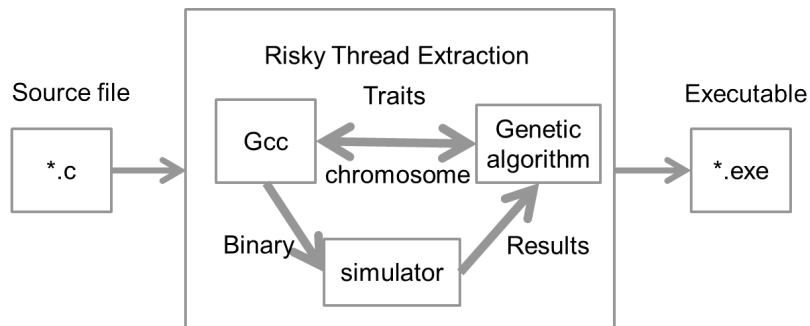


Figure 3.1: Risky Thread Extraction System.

RTE System is shown in figure 3.1. There are three key components in our system, compiler, genetic algorithm and the simulator. Compiler generates binaries using parallelization solutions

provided by genetic algorithm and the simulation results will act as feedback used by genetic algorithm for population selection and reproduction. The three of them will work together for testing and selecting a better parallelization solution. Input to the system is the program and output is the selected binary which achieves best speedup among all the solution tested.

3.2 Genetic Algorithm

The reason to use genetic algorithm is for a search algorithm, it can easily create lots of different solutions, we can seed the solutions in the area that a better solution is more likely to be found, also tune the reproduction method to fit property of the problem. The heritable nature of genetic algorithm enables us to utilize results from previous test to generate better solutions generation by generation. We implement our genetic algorithm as a script and use it to drive the whole thread extraction process. Function *geneticAlgorithm* illustrates the genetic steps for RTE system, input of the function are traits collected from program. Step 1 and 2 create chromosome based on traits and generate initial population. Step 3 and 4 call compiler to create and test binary according to every chromosome, and step 5 collects simulated results, step 6 select and produce next population. This process repeats from step 3 to step 6 until termination criteria is met.

Algorithm 6 geneticAlgorithm

```

Input: traits of program geneticAlgorithm(traits)
  create chromosome for the whole program using traits
  create initial population
  do
    compiler apply each chromosome to generate binary
    run simulation
    collect simulation results for this generation
    reproduction:crossover and mutation
  while(!termination criteria is met)

```

3.2.1 Chromosome

In RTE system, a chromosome represents whether to enable or disable some features of the loop, such as whether to relax an edge in PDG, or to enable certain optimization for this loop. A feature of a loop is called a trait. When genetic algorithm process the traits information of loops, it collects all such information such as loop id, number of edges, edge number, and creates chromosome string accordingly. For example, the first bit of chromosome indicates whether to

parallelize the loop or not, the second bit represents enabling or disabling one optimization, the third bit represents keep or remove a certain edge in PDG, etc. When it writes the chromosome to file, it does not write a string of 0 and 1, but rather it specifies the bit information and value of that bit in chromosome. Thus, when our compiler reads this chromosome information from file, it will learn what each bit means and act accordingly. Function *applyChromosome* reads a chromosome from file that was generated by the genetic algorithm script, and according to edge number and edge gene in the chromosome, it chooses to keep or remove some edges in PDG, and sets or unsets several optimization flags.

Algorithm 7 applyChromosome

```

applyChromosome()
  read chromosome from file
  rebuild PDG by removing edges specified by chromosome
  setup optimization flags according to chromosome
  begin thread extraction with new PDG
  end

```

For edge gene in the chromosome, we use edge classes instead of individual edges to be genes in chromosome, thus if one gene in the chromosome is 0, then a group of edges belonged to that edge class is removed instead of only one edge got removed. This methodology makes sense because if the compiler removes the memory dependence of two variables, that means they not alias, then it should remove all the memory dependence edges pointed to the two variables, that is remove all edges belongs to that edge class. The way to build a edge class is by gathering edges that represent the memory dependences of the two given variables.

3.2.2 Mutation and Recombination

The reproduction methodology we use is mutation and recombination of parents chromosome(crossover). After simulation, the result will be read to the genetic script. We use these results to select solutions and generate next population of solutions using crossover or mutation. Solutions with a correct simulation result will be selected as parents during reproduction process. Crossover is done by parents switching their chromosome at certain crossover points to create two new chromosome, and mutation is done by changing the bit of the chromosome from 0 to 1 or from 1 to 0. Our chromosome usually includes genes from several loops in the benchmark, so crossover should only act on the genes belonged to each loop individually rather than on the whole chromosome. Figure 3.2 illustrate this process. The chromosome shown in the figure has genes for two loops, and to differentiate them, genes for loop 1 is marked with

grey colour, genes for loop 2 is marked with black colour. Crossover points are carefully selected for not letting genes from different loops polluting each other. As shown in the figure, several crossover points are selected at the boundary of each loop genes and in the middle of loop genes. Generally, crossover rate should be high, we set the rate to be 95%, but on the other side, mutation rate should be very low, we set it to 1% in the experiment.

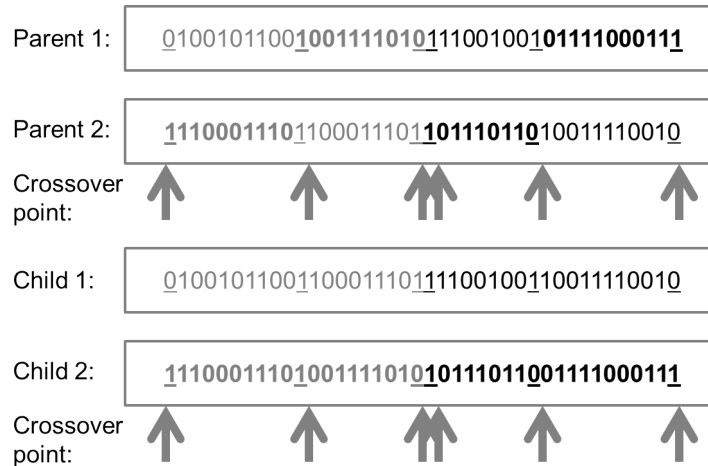


Figure 3.2: Recombination.

3.2.3 Termination Criteria

We terminate the genetic process when a certain number of generation is simulated. We set the initial population size to be 100 to make sure there is enough different solutions to start with as we want to test as many different solutions as possible in a given amount of time. If the initial population is too small, number of different solutions in the following generations could be small, but if the initial size is too big, we end up with spending too much time on one generation thus could not fully utilize of the heuristic nature of genetic algorithm as less generations being tested. The genetic process terminates on 10th generation or time out. Then the best result among all the simulation is selected.

3.3 Testing

We select the solution with correct simulation result and the least execution time to be the final solution. This solution depends on certain input, to further verify the correctness on any given input, we could collect the removed memory dependence in terms of variables related. Those

variables can be presented to the programmer and thus they could verify those dependences using their knowledge of the program.

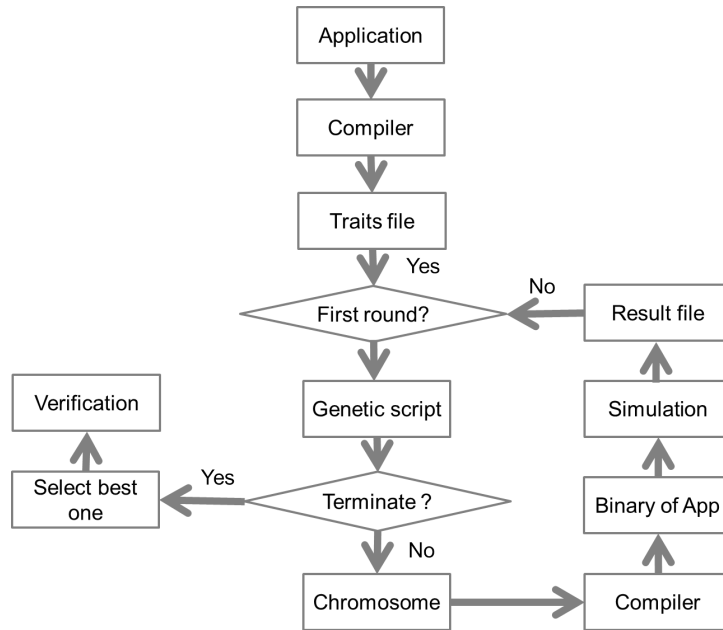


Figure 3.3: RTE algorithm flow.

Figure 3.3 shows the RTE algorithm flow. There are three components in the system: compiler, genetic script and simulator. The flow starts from initial compilation of applications, compiler will collect program information we called traits through this compilation. We implement genetic algorithm as a script. On initialization, it will read traits information of the loop and create initial population. Each solution of the initial population is read by compiler and a binary is created accordingly. Those binaries will be simulated by the simulator and results are collected and read by the genetic script. On the next generation, a new population is created as child population of selected parents from previous round, and then the flows repeated as shown in the right hand side of the chart. Genetic process terminates on completion of a certain number of generations and we select the best solution and provide dependences being relaxed for validation.

Chapter 4

Decoupled Software Pipelining Optimizations

We did some optimizations to improve our base DSWP infrastructure, including communication reduction between threads, and optimization of system load balance.

4.1 Communication Optimizations

4.1.1 Replicating Code to Eliminate Unnecessary Communication

The first optimization is the reduction of control dependence communication between main thread and extracted threads. If certain workload in the extracted thread is control dependent on a branch, then system will insert a channel in the code to pass the branch direction or data used to calculate the direction to extracted threads. If the branch condition is the loop exit condition, this communication will happen each iteration which brings big communication overhead. The optimization avoids those communications in this way: it detects if the loop is a scalar evolution[3] loop, and if it is, a chain of statements including initialization of induction variable, the phi statement, and the predicate statement will be replicated to multiple stages so branch direction will not need to be communicated every iteration because each thread now has its own loop chain.

Here is an example of one case that we choose to replicate the loop chain. Function *oldCase* has a *for* loop and the exit condition is $i > k$, and the main thread will communicate i to the extracted thread each iteration as shown in the algorithm. This is the case we could optimize.

In this example, the optimized system will detect the predicate statement $i < k$ and then analyze i and k respectively. It finds i is an induction variable and k is a live in variable initialized outside the loop, then it will replicate the loop chain including initialization statement of i , phi

Algorithm 8 *oldCase*

```
k = computation results
insertCommunication(q,k);
foreach i from 0 to k-1
    insertCommunication(q,i);
    .....//some computation
end
```

Algorithm 9 *newCase*

```
k = computation results
insertCommunication(q,k);
foreach i from 0 to k-1
    .....//some computation
end
```

statement, $i++$ and communicate the value of k just once before loop executes, instead of communicating i in statement $i < k$ for k times. The optimized code is shown in function *newCase*, for both main thread and the extracted thread, the number of communication is reduced to one. The optimization can also detect nested loop and replicate the nested loop chains, which saves even more communication. Figure 4.1 shows the execution comparison between the problem and the solution we provided. In the left hand side, main thread communicates predicate to extracted thread every iteration, but on the right hand side, with the loop chain replicated to extracted thread, main thread only communicates some live in variables once before the loop starts, and as the other thread has the loop chain itself, it does not need to get predicate statement from main thread every iteration.

The algorithm of this optimization is shown below. Function *isScev* is a recursive function, it checks whether all the variables involved in the loop chain are either simple induction variables or just copy in variables that will not be modified inside the loop, function returns true or false to represent whether to replicate the loop chain. The algorithm first checks each argument of the predicate statement and if the argument is simple iv, then it records its definition statement in statement list, and pass this statement as an argument to *isScev* and do the checks again. The function returns true if it finds an argument is already being tested. Function *test_simple_iv(op)* use scalar evolution to test the variable. Function *loopChainReplication* passes the predicate node to *testScev* and if the loop chain can be replicated, the returned statement list will contain all the statements that need to be replicated, otherwise if the loop can not be replicated, *testScev* will return *NULL*. We replicate these statements by marking the nodes to be in multiple stages.

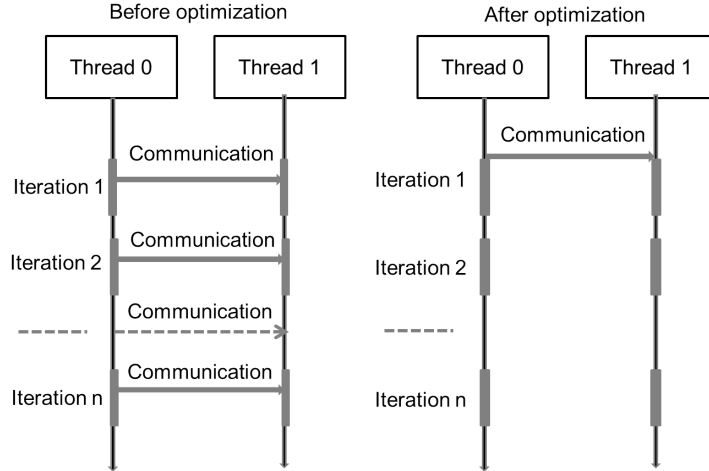


Figure 4.1: Loop chain replicate from thread 0 to thread 1.

Algorithm 10 isScev

Input: 1 statement list which contains all the statements to be replicated

Input: 2 statement to test

Output: true/false(whether the loop chain can be replicated)

isScev(stmtlist,stmt)

```

    foreach operand op of stmt do
        if test_simple_iv(op) is true then
            if this operand already being tested then
                return true means the loop chain can be replicated
            else
                store the definition statement of this operand in statement list
                get definition statement def_stmt of this operand
                call isScev(stmtlist,def_stmt) to test definition statement of this operand
        else
            return false;
    end

```

4.1.2 Optimization for Software Queues

Queues are used to facilitate communications between threads in our system, but current implementation of software queue is too expensive to support this fine-grained parallelism. The motivation of this optimization is to reduce benchmark execution time by providing a faster software queue on x86 machine. On Sesc simulator, we could use hardware queue which only takes one cycle for each enqueue or dequeue, but simulator usually takes longer to finish one simulation. We want to use native execution to quickly get results while do not want to afford the overhead of software queue. Native simulation on x86 machines takes less time than Sesc

Algorithm 11 testScev

Input: 1 predicate node n

Output: statement list that contains all statements to be replicated

```
testScev(n)
  get the simple statement of node n
  If isScev(stmtList,stmt) returns true then
    return stmtList;
  else
    return NULL;
  end
```

Algorithm 12 loopChainReplication

Input: 1 PDG g of the loop

Input: 2 node in PDG

loopChainReplication(node n)

```
If n is predicate node then
  stmtList = testScev(n);
  Label nodes in stmtList to be in multiple stages
end
```

simulation but the bottleneck is overhead of communication using software queue. One possible overhead is when using software queue, each enqueue and dequeue operation will check if the queue is out of boundary and if there are enough space or data in the queue before producing/consuming to/from the queue. This check could slow execution down significantly. The optimization of software queue aimed to reduce the number of these checks. The way to do it is to remove the check for each queue operation but insert such check once for the communication of entire iteration in the loop header. With the optimization, producer and consumer will only stuck in the loop header to wait until the queue is ready, but it will not stall on any enqueue or dequeue in the execution of loop body.

Algorithm 13 flushChannel

Input: channel q

Input: flag to indicate if this a producer flush or consumer flush

flushChannel(q,enqueue)

```
if enqueue is true then
  q.write = q.nextWrite;
else
  q.read = q.nextRead;
```

Algorithm 14 oldEnqueue

Input: channel q

Input: data to enqueue

oldEnqueue(q,data)

if distance between nextWrite and read pointer is less than size of data *then*
 wait until there is enough space to produce
if nextWrite pointer going to access beyond queue boundary *then*
 reset nextWrite pointer to the beginning of the q
else get nextWrite pointer
write data to queue
increment nextWrite pointer

Algorithm 15 oldDequeue

Input: channel q

Output: data

oldDequeue(q)

if distance between nextRead and write pointer is less than size of data *then*
 wait until there is enough data to consume
if nextRead pointer plus size of data will go beyond the end of q *then*
 reset nextRead pointer to the beginning of the q
else get nextRead pointer
read Data from queue
increment nextRead pointer

The structure of the queue has five important variables, read pointer and write pointer are shared by both consumer and producer pointing to read and write position of the queue, nextRead is private pointer for consumer, nextWrite is private pointer for producer, and buffer represents the queue used for communication. When producer or consumer operates the queue, they will use and update nextRead or nextWrite to locate the position of the queue, and also use write or read pointer to check for the distance between producer and consumer's pointer to make sure if the queue is ready to produce or consume. Read and write pointer will only be updated using nextRead and nextWrite when producer or consumer flush their channel each iteration, so the other one could know the updated producer or consumer position of the queue. Function *flushChannel* updates read and write pointer.

Function *oldEnqueue* and *oldDequeue* shows the previous implementation of the queue, when there's a read or write, the first thing to do is to check if the queue is ready to access and the boundary of queue in order to not access unrelated memory thus generate errors. But such check is expensive as memory access takes a few cycles to finish.

The optimized implementation of software queue is shown as follows: for all the communication inserted, *insertSupport* calculates the total size of communication each iteration

Algorithm 16 insertSupport

Input: set of channels inserted

insertSupport(channelsSet)

foreach channel in channelsSet *do*

Calculate enqueue or dequeue size of this iteration

Insert producer or consumer check queue build in function

end

Algorithm 17 producerCheckQueue

Input: channel q

Input: total enqueue size of this iteration

producerCheckQueue(q, enqueueSize)

do *if* empty space in queue is greater than enqueue size *then* *break* *while*(1) *if* producer going to write outside queue boundary *then* reset producer channel

Algorithm 18 consumerCheckQueue

Input: channel q

Input: total dequeue size of this iteration

consumerCheckQueue(q, dequeueSize)

do *if* there is enough data for consume of this iteration *then* *break* *while*(1) *if* consumer going to read outside queue boundary *then* reset consumer channel

Algorithm 19 noWaitEnqueue

Input: channel q

Input: data to enqueue

noWaitEnqueue(q,data)

get nextWrite pointer

write data to queue

 increment nextWrite pointer

and then inserts producer and consumer check at loop header and thus this check is executed each iteration before execution of other works in loop body. *producerCheckQueue* and *consumerCheckQueue* checks if the queue is ready for the communication of entire iteration, so enqueue and dequeue in the loop will use *noWaitEnqueue* and *noWaitDequeue* which just operate on the queue without any checks.

Algorithm 20 noWaitDequeue

Input: channel q

Output: data

noWaitDequeue(q)

get nextRead pointer

increment nextRead pointer

 read Data from queue

4.1.3 Computational Bottlenecks through Stage Replication

In task selection algorithm, nodes will be added to one partition before the estimated weight is greater than the average weight. This algorithm could not maintain load balance in certain case when one partition is much greater than others, for example, if the last node added to one partition is heavily weighted, the result partition will be imbalanced. Imbalanced workload between threads may result in more stalls at runtime and thus slow down execution. The optimization is performed after task selection. If the assignment is imbalanced, then the bigger task can be replicated and executed by more than one thread in the system, and each thread of this task will work on different iterations of the loop with different data set. Figure 4.2 shows the computation pattern with the second stage replicated. The implementation of this

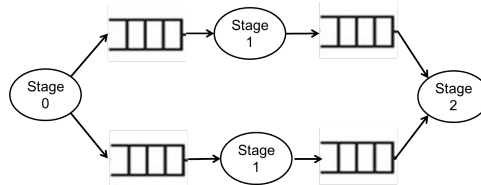


Figure 4.2: Pipeline execution with the second stage replicated.

optimization includes several parts. First part is to mark all SCCs that could be replicated in the PDG. If there is no loop carried dependence in one SCC, then it is a candidate for replication. Second, task selection algorithm should try to group nodes that can be replicated to be in one task, so the weight of that task will become bigger. After task selection, as shown in function *findStageToReplicate*, if the weight of one task is much bigger than other tasks, then system creates more than one cloned function to execute the heavy task. Finally, it inserts code to help with selection of communication channels. As shown in figure 4.2, original there are three stages, but as the second stage is much bigger so it is replicated and executed by two threads. Stage 0 need to communicate with stage 1 and stage 1 will communicate with stage 2. After stage replication, the first replicated thread works on even iterations and the second works on

odd iterations. Stage 0 and stage 2 will communicate alternately with the two threads of stage 1. *insertCounter* function inserts a variable counter to control the changes of channel number each iteration. Counter iterates between 0 and number of replicated threads, and whenever a thread need to communicate with the replicated threads, it needs to use a counter to change the channel number to communicate with the correct thread each iteration.

Algorithm 21 findStageToReplicate

Input: PDG g

findStageToReplicate(g)

 construct a list of PDG nodes in topological order

 taskSelection(nodelist)

foreach task *do*

if estimated weight of this task greater than max weight of all other task *then*

 mark this stage to replicate

end

end

Algorithm 22 insertCounter

insertCounter()

 insert a counter to each cloned function

 channel number = channel number + counter

 increment counter

 counter = counter modulo number of replicated stages

Another important change is the modification of loop exit condition for replicated threads. As each replicated thread executes parts of the whole iteration, so they should have a way to break out of the loop when expected number of iteration is finished. Compiler will transform a loop to *dowhile* loop in gimple representation. In this optimization, if there are two replicated threads, the first replicated thread will execute the first iteration and the second thread will execute the second iteration. The first iteration could be executed without checking exit condition as it's a *dowhile* loop, but the second thread must wait and check the exit condition in order to execute the second iteration, so we need to insert an extra condition check in the loop preheader of the second replicated thread. And in the first iteration of main thread, it should send the exit condition to the second thread to tell it whether to execute the second iteration. When main thread exits the loop, it still need to send extra branch direction to the threads

who are waiting for the condition to break out of the loop. *doWhileLoop_mainThread* and *doWhileLoop_otherThread* illustrate the above analysis.

Algorithm 23 *doWhileLoop_mainThread*

```
doWhile()  
  do  
    some computation here  
    produce(break condition)  
  while(condition)  
    produce(break condition) to the rest replicated threads
```

Algorithm 24 *doWhileLoop_extractThread*

```
doWhile()  
  if(condition is false)  
    goto loop exit  
  do  
    some computation here  
    consume(break condition)  
  while(condition)
```

Chapter 5

Evaluation

5.1 Set up

Table 5.1: Architecture simulation details

2 core CMP	
<hr/>	
Frequency 3.2 GHz	
Fetch width 6	Private L1 Data Cache:
Issue width 3	Size 64KB, assoc 4
Retire width 4	Line 64B, latency 1 cycle
ROB 126	L1 instruction Cache:
Branch predictor: hybrid	Size 64KB, assoc 2
Mispred. Penalty 14 cycles	Line 64, latency 2 cycles
BTB 2K, 2-way	L2 Cache:
I window 68	Size 2MB, assoc 4
LS/SD queue 48/48	Line 64, latency 12 cycles
HW Queue Size 4KB	Memory Latency 300 cycles
Latency 1 cycle	

We evaluated our risky system on Sesc simulator[20],an execution-driven simulator. Table 1 shows the details of the simulated architecture. For this evaluation, Sesc simulation will use hardware queue for communication between threads. Every enqueue and dequeue takes one cycle and the size of hardware queue is 4 KB. We use benchmarks in SpecInt2000, SpecInt2006 and Clustalw in BioPerf in the evaluation and use GCC 4.5 to cross compile mips binary.

”Simulation marks” are inserted into benchmarks and the evaluation simulates certain number of markers that are about millions of instructions. Our thread extraction system is implemented as a plugin for GCC.

5.2 Risky Thread Extraction

Figure 5.1 shows the simulation results for different benchmarks. Sesc simulator is chosen because we can simulate hardware queue which will not bring much communication overhead between threads, and thus we can expect better speed up by parallelizing loops with a good amount of workload and big trip counter. There are three bars for each benchmark, the left one represents speed up for sequential execution, the middle bar shows speed up of parallelization using our baseline thread extraction and the right bar shows speed up using risky system. For all the benchmarks, clustalw and mcf06 show the best speed up over baseline system and sequential execution. We achieve an average 23% improvement over sequential execution and 14% improvement over DSWP. Table 5.2 shows simulation details for each benchmark, including length of chromosome, number of different solutions tested, and total simulation time. We focus on heavily executed loops when using risky system, as the execution coverage of those loops is significant, each parallelization solution is more likely to make a big difference compared to sequential execution. In the experiment, the initial population is 100, and for each generation,

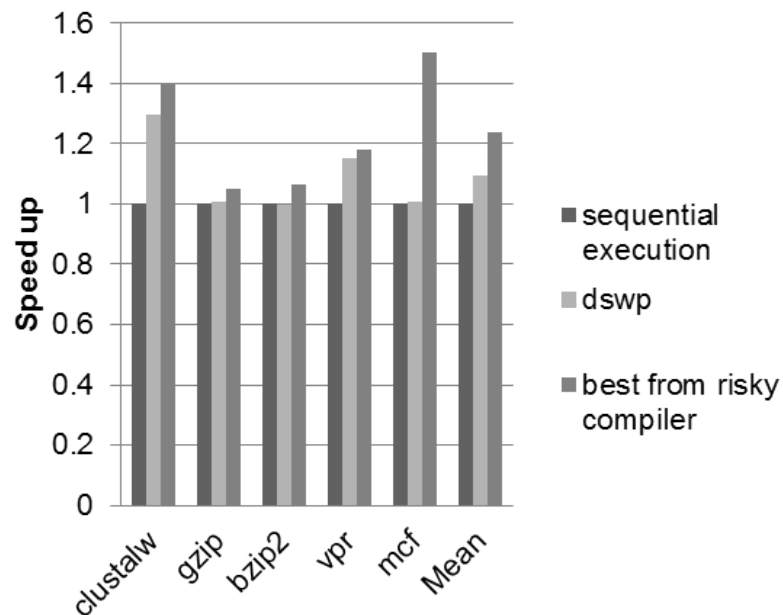


Figure 5.1: Benchmark simulation results of risky thread extraction.

Table 5.2: Details of genetic run on sesc.

benchmark	chromosome length	solutions tested(hours spent)	dependency removed
gzip	301	406(135)	12
vpr	177	204(170)	32
bzip2	401	1000(66)	27
mcf2006	35	1000(250)	20
clustalw	62	1000(8)	38

we select individuals with correct simulation results. If the number of such individual is less than half of the population size of that generation, the correct individuals from the previous generation will be added to this generation as parents to create next population.

From all the benchmarks simulated, mcf2006 got best speed up(1.50) compared to sequential execution within 24 hours of running genetic algorithm, figure 5.2 shows speed up for each loop in mcf06 being parallelized and figure 5.3 shows performance improvement with edges removed from chromosome. In figure 5.2, loop 4 with 8 edges removed achieves best speedup, and the rest loops only contribute little speedup for the overall improvement. In figure 5.3, with no edges removed, there is 1.5% improvement, with the first 8 edges removed, mcf achieves 48%

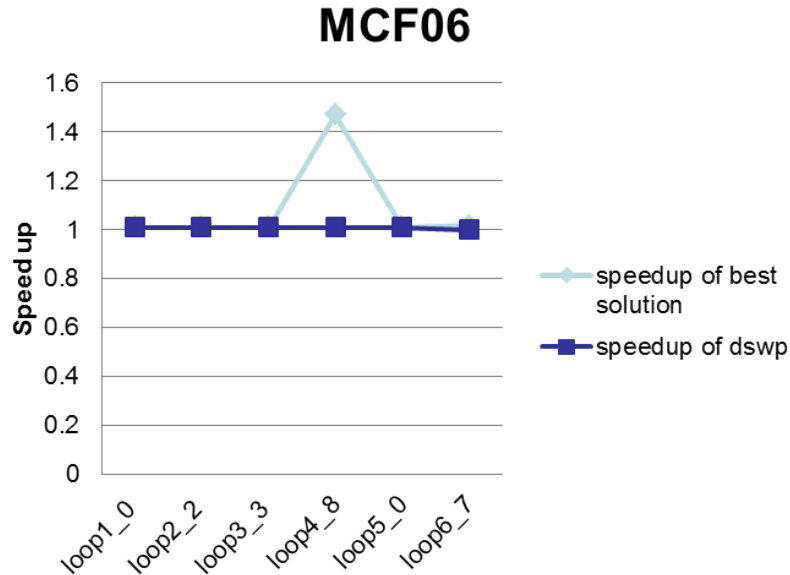


Figure 5.2: Speedup for each loops in mcf06.

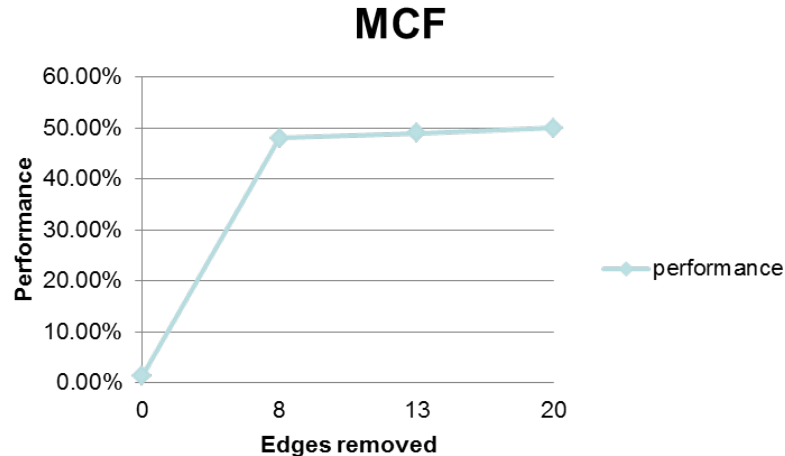


Figure 5.3: Performance improvement with different edges removed in mcf06.

improvement and with all the 15 edges removed, it comes to the pick, 1.5 speed up. As shown in figure 5.2, the speed up mostly comes from parallelization of a heavily executed loop in pbeampp.c after genetic algorithm chooses to relax some edges of that loop. The loop is shown in figure 5.4. This is a 'for' loop and is used to fill the content of an array of structure named perm. From analysing the loop manually, we conclude that the loop body can be divided to two parts. Main thread could work on get the value of arc and red_cost, also pass the predicate to extracted thread. The extracted thread could work on filling the content of the perm array. Before relaxing any edges, there are several false dependence edges that prevent us from parallelizing the loop. Table 5.3 shows some memory dependences in the PDG of this loop.

```

1  for( ; arc < stop_arcs; arc += nr_group )
2  {
3      if( arc->ident > BASIC )
4      {
5          /* red_cost = bea_compute_red_cost( arc ); */
6          red_cost = arc->cost - arc->tail->potential + arc->head->potential;
7          if( bea_is_dual_infeasible( arc, red_cost ) )
8          {
9              basket_size++;
10             perm[basket_size]->a = arc;
11             perm[basket_size]->cost = red_cost;
12             perm[basket_size]->abs_cost = ABS( red_cost );
13         }
14     }
15 }

```

Figure 5.4: Loop 4 in mcf2006.

As shown in the table, the compiler thinks there are memory dependences between variables in

Table 5.3: False memory dependence edges in the loop.

benchmark-loop	memory dependency
Mcf06-primal_bea_mpp	$\text{perm}[\text{next}] \rightarrow \text{cost} \longleftrightarrow \text{arc} \rightarrow \text{ident}$ $\text{perm}[\text{next}] \rightarrow \text{abs_cost} \longleftrightarrow \text{arc} \rightarrow \text{ident}$ $\text{perm}[\text{next}] \rightarrow \text{abs_cost} \longleftrightarrow \text{arc} \rightarrow \text{tail} \rightarrow \text{potential}$ $\text{perm}[\text{next}] \rightarrow \text{cost} \longleftrightarrow \text{arc} \rightarrow \text{tail} \rightarrow \text{potential}$ $\text{perm}[\text{next}] \rightarrow \text{abs_cost} \longleftrightarrow \text{arc} \rightarrow \text{head} \rightarrow \text{potential}$ $\text{perm}[\text{next}] \rightarrow \text{cost} \longleftrightarrow \text{arc} \rightarrow \text{head} \rightarrow \text{potential}$ $\text{perm}[\text{next}] \rightarrow \text{a} \longleftrightarrow \text{arc}$ $\text{perm}[\text{next}] \rightarrow \text{cost} \longleftrightarrow \text{arc} \rightarrow \text{cost}$ $\text{perm}[\text{next}] \rightarrow \text{abs_cost} \longleftrightarrow \text{arc} \rightarrow \text{cost}$

structure *perm* and *arc*, these edges form a cycle in the PDG and thus task selector view the loop as not dividable. But using risky system, we are able to successfully remove those edges and thus being able to parallelize this loop the same way as we analyzed above. The extracted thread will get data input from main thread and work on setting value for the array. To further verify the correctness of removing those may alias edges, programmers need to examine the code to see whether two variables should alias. One simple rule is if one variable is a global variable and its address is never being taken, then it should not alias with any other variables. Another

```

1   for( i = 2, next = 0; i <= B && i <= basket_size; i++ )
2   {
3       arc = perm[i]->a;
4       red_cost = arc->cost - arc->tail->potential + arc->head->potential;
5       if( (red_cost < 0 && arc->ident == AT_LOWER)
6           || (red_cost > 0 && arc->ident == AT_UPPER) )
7       {
8           next++;
9           perm[next]->a = arc;
10          perm[next]->cost = red_cost;
11          perm[next]->abs_cost = ABS(red_cost);
12      }
13  }
```

Figure 5.5: Loop 5 in mcf2006.

loop example of mcf is shown in figure 5.5, in line 3, there is a load of *perm*[*i*], and there is a write to *perm*[*next*] in line 9, that forms a loop carried dependence, thus even with profiler, that dependence still exists, which limits our choice of parallelization. If we analyze the code, it is not difficult to find that *i* is always bigger than *next*, thus *perm*[*i*] and *perm*[*next*] will not alias in this loop, so we could assign the load statement of *perm*[*i*] and write statement of *perm*[*next*]

to different threads. And as there is a data communication of arc between two threads, the extracted thread has to wait until arc is produced, thus `perm[i]` will not be overwritten before read. Using risky compiler, that dependence edge is removed, thus the compiler is being able to assign the task between line 8 and line 11 to a independent working thread. Some memory dependence edges that enforced by profiler may prevent compiler from parallelizing the loop, while with RTE system, more parallelization is exploited with a relaxed dependence graph.

Figure 5.6 and 5.7 shows detailed results of `clustalw` benchmark. In `clustalw`, most performance comes from loop 4 and loop 6. As shown in figure 5.6, if we remove 16 edges of loop 4, there is 18% improvement compared to 13% if we do not remove any edges. And for loop 6, even if we remove some edges, the speed up is the same as base system, that is 16% improvement. In figure 5.7, the base system achieves 29% speed up, which is our start point. When remove 16 edges, we get an extra 6% speed up, and after removing total 38 edges, we finally get a total 40% speed up compared to sequential execution and 10% improvement compared to base thread extraction system.

Other benchmarks also have performance gain, the best simulation result of each benchmark is better than sequential execution and base thread extraction as shown in figure 5.1. One difficulty for genetic algorithm to find a better solution faster is for some loops, the candidate pool of false dependence edges is very big, so it becomes harder for genetic algorithm to find correct edges to remove and generate better solution fast. Using edge class as gene could reduce the chromosome length thus help system to converge faster than using individual edge. And in order to get a better speed up within a fewer amount of time, we only focus on the loops that proved to have good execution time according to profiler, rather than including all the loops

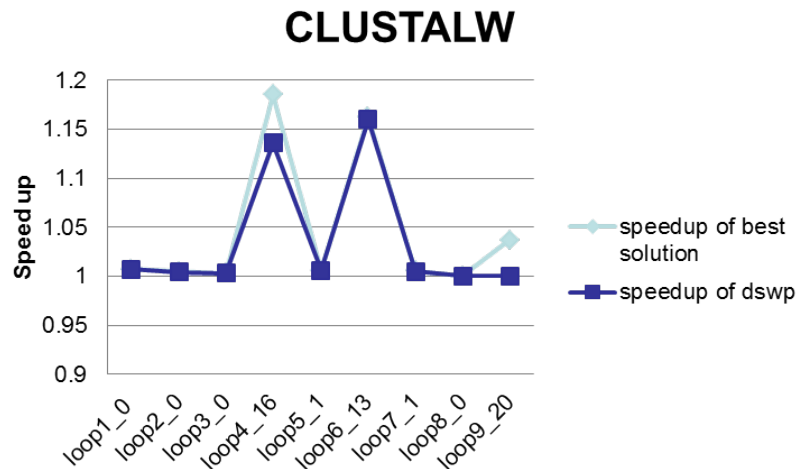


Figure 5.6: Speedup for each loops in `clustalw`.

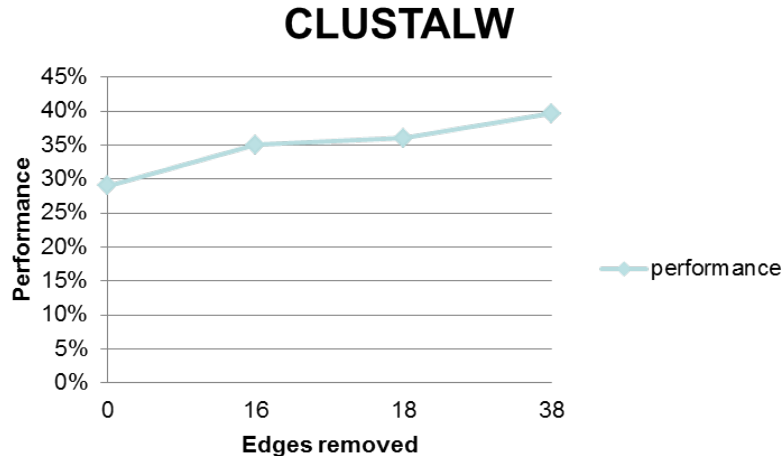


Figure 5.7: Performance improvement with different edges removed in clustalw.

in the chromosome. By this way, we could control the length of chromosome while get better results in a fewer amount of time.

5.3 Optimizations

5.3.1 Communication Optimization

The experiments for loop chain replication are performed on two environments, one is x86 machine running Red Hat 4.1.2, GCC version is 4.5 and the other is Sesc simulator. Figure 5.8 shows the results of simulation on x86 machine with 2 threads execution and figure 5.9 shows the results of simulation on Sesc simulator with 2 threads. The first bar represents base thread extraction without any optimization introduced in this work, the second bar represents results of simulation with the communication optimization enabled. If we compare the results on Sesc simulator and x86 machine, we could see that there are improvements on both experiments, but on x86 machine, speed up is much higher and obvious than simulator. For example, on x86 machine, gzip is observed 1.5 speed up while on Sesc simulator there is only 7% improvement. The reason of this difference between these two simulations is because the cost of communication is different on x86 machine and Sesc simulator. If we simulate hardware queue on Sesc, there is only one cycle cost for queuing operation, but on x86 machine, it usually takes a few cycles to complete one enqueue or dequeue, thus if we reduce the number of communications, native execution will benefit much more than simulations on simulator.

Figure 5.10 shows the reduction of communication for each benchmark. The optimization performs well on gzip, vpr, bzip2, because there are big amount of communication reduc-

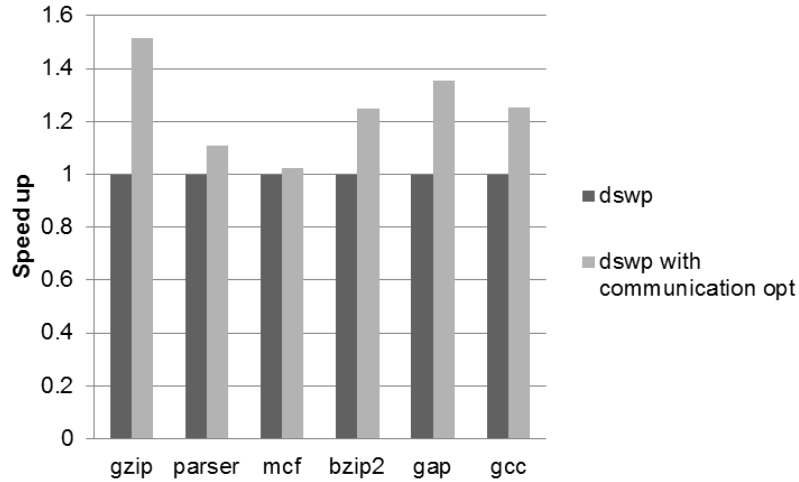


Figure 5.8: Results of loop chain replication on x86 machine.

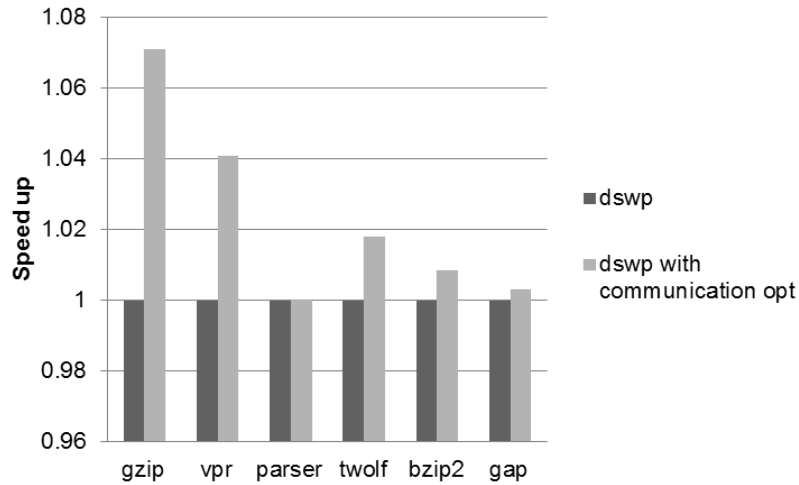


Figure 5.9: Results of loop chain replication on sesc machine.

tion(35% for gzip2, 20% for vpr and 27% for bzip2). For mcf, there is almost no speed up from the optimization due to only 3 loops in mcf is qualified for loop chain replication, and execution time coverage of those loops is less than 1%, same as parser, loops fit the optimization has little execution time, and thus there is only 2% communication reduction, that is why we could not observe speed up from mcf and parser. Gap has a great amount of communication reduction, but speed up is relatively less, that is because the number of communication reduction is not the only issue that could affect the performance. Figure 5.11 shows related results on sesc simulation for gzip, vpr and gap. Main thread execution is the critical path in this multicore execution

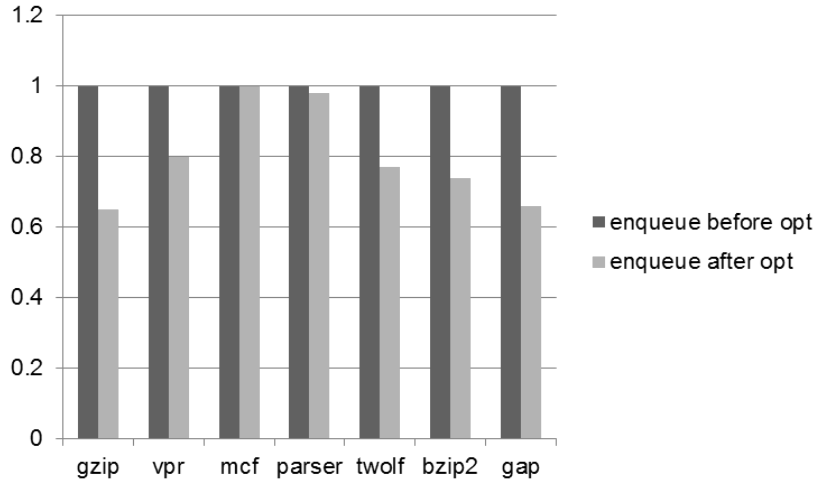


Figure 5.10: Communication reduction for each benchmarks.

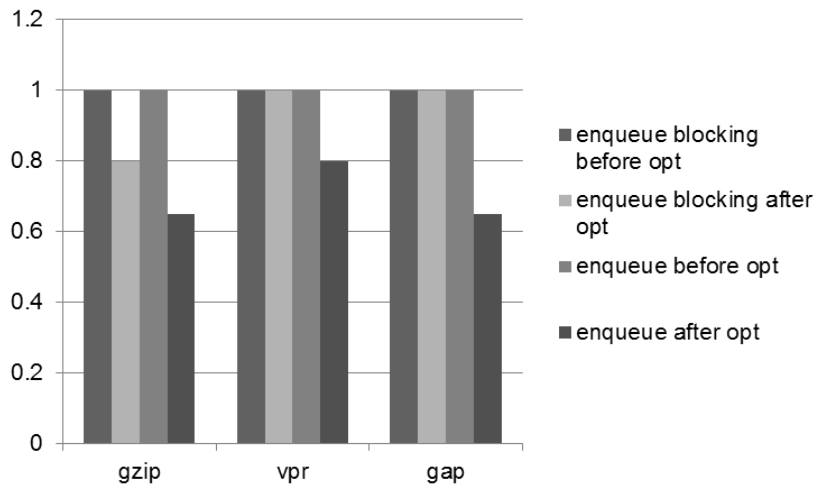


Figure 5.11: Comparison of sesc simulation for gzip, vpr and gap.

as extracted threads need to wait for data from main thread. Gzip has the most significant decrease which is about 20% in the number of enqueue blocking, and also 35% reduction in the number of enqueue as well. So after modification, both main thread and extracted threads execute faster because of less blocking and less communication. Vpr and gap do not have any enqueue blocking before or after modification, so although gap observed the biggest number of enqueue reduction(35%), the performance improvement is still not as good as gzip.

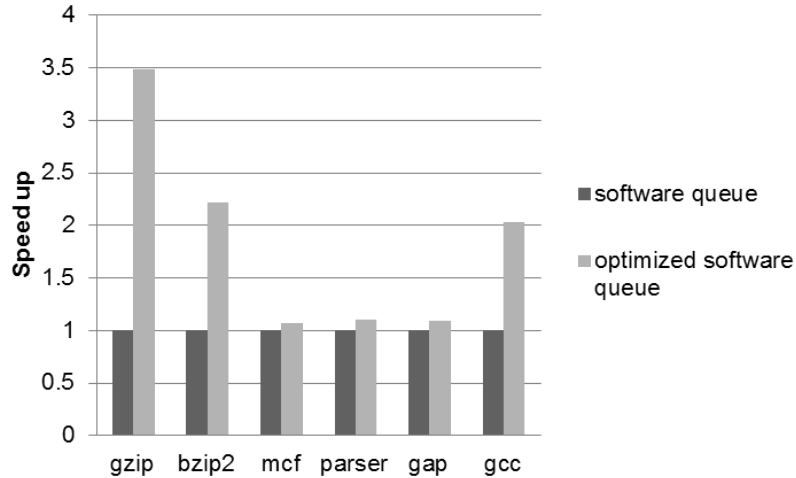


Figure 5.12: Execution results of software queue optimization.

5.3.2 Software Queue

Figure 5.12 shows the results for software queue optimization on x86 machine with two threads. The first bar represents thread extraction with no optimization of software queue, the second bar represents thread extraction with optimized software queue. Gzip, bzip2, gcc have significant speed up because there are several heavily executed loops in gzip with communications each iteration, so the overhead of enqueue and dequeue is very big. With the optimization, the overhead is greatly reduced by checking the queue once each iteration, the rest enqueue and dequeue are just pure queue operation without checking the queue. Parser, mcf and gap have about 10% speed up for enabling software queue optimization, the speed up is not as big as gzip, bzip2 and gcc.

There are several reasons for the variation of speed up for different benchmarks. First, the performance gain for this optimization partially depends on the number of communications each iteration and the coverage of execution time for those loops in the program, so if there are only a few communications for the loop and the loop itself is not heavily executed, this optimization should not be very beneficial. The other reason is that we insert producer and consumer check to each loop header when there are enqueue or dequeue in the loop according to static calculation at compile time, but during runtime, the enqueue and dequeue may not happened, so we introduce extra check each iteration compared to the old version where the check only happens when there is a queuing operation during execution. This happens because some queuing operations are control dependent on certain predicate node, and thus it is not guaranteed to be executed.

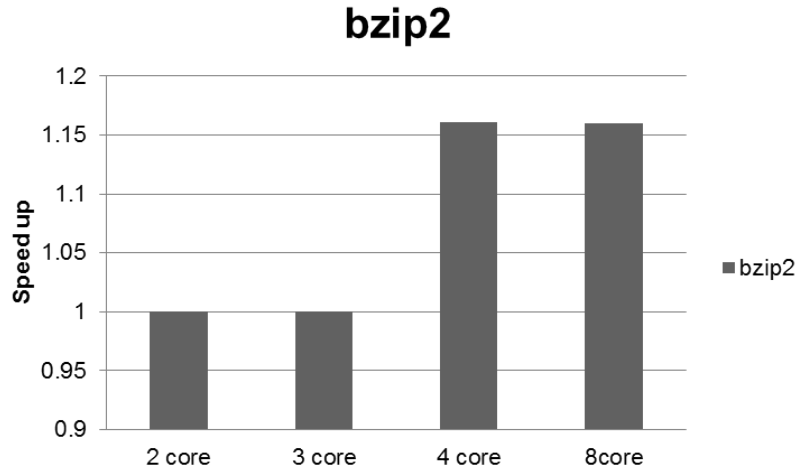


Figure 5.13: Results of stage replication for bzip2.

5.3.3 Stage Replication

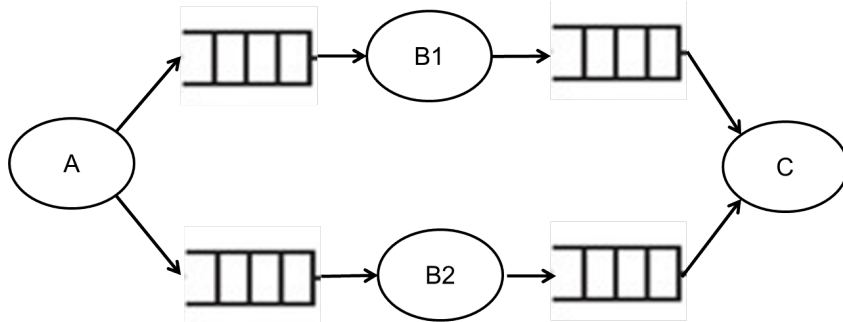


Figure 5.14: Stage replication pattern for bzip2 with 4 threads.

Figure 5.13 shows the result for simulations of stage replication optimization. We did experiments on x86 machine with 2 threads, 3 threads, 4 threads and 8 threads. When we use two or three threads, task selector could not find a task to be replicated, but when the number of threads becomes four, there is one loop in function *moveToFrontCodeAndSend* which will be suitable for replication. The execution pattern of the loop is shown in figure 5.14. The second stage weights more than other stages, so we replicate it to two threads. Each thread will work on different iterations of the second stage. This execution pattern gained 15% speedup compared to 2 core execution, and performance is gained from using an extra thread in the

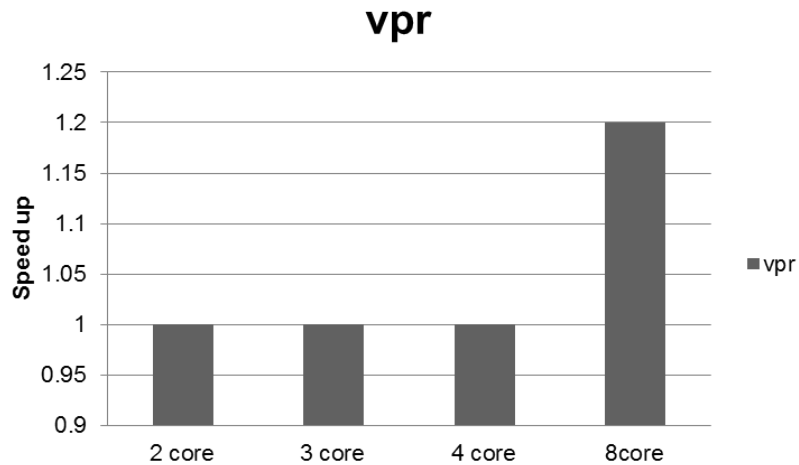


Figure 5.15: Results of stage replication for vpr.

execution while not introducing big overhead. Figure 5.15 shows vpr stage replication results. When enabling 8 threads, it finds a stage to replicate in a loop and achieves a speedup of 20%, the loop that contribute this performance is in function *try_swap*.

5.4 Discussion

From the evaluation above, we have shown the effectiveness of our risky thread extraction framework, we provided good speed up for each optimization applied.

Our parallelization system is transparent to programmers, it does not require the programmer to insert any annotations or directives into the program. Compiler will decide whether to parallelize the loop and how to split task between threads, all these changes are done during compile time.

Chapter 6

Related Work

Various approaches to parallelize sequential programs have been developed, programmers perform different roles in those approaches.

Some approaches rely on programmers to insert some statements that tell the compiler to follow the rules specified by the statements, that what we called as directives. OpenMP pragma is a good example of this approach[5]. Another similar approach uses annotations to inform compiler several properties of the program and variables. The Parallax paper[29] uses annotations to help compiler find more parallelization for loops in a program. Programmers insert annotations to their programs to specify some semantic information of the programs that compiler may miss. One kind of annotations marks kill point of some variables which means after certain point, those variables are no longer used in the program. This could reduce some memory dependent edges in the PDG thus leads to more parallelization. Similarly as the Alter system[27], programmers insert annotations to enable compiler exploit parallelism in loops by reordering iterations or allow stale reads.

Automatic thread extraction has been successfully investigated to avoid human interfere during the parallelization process. Such compiler technique relies on static analysis to identify the parallelization parts of the programs. Those compilers work well on loop parallelism such as DOALL or pipeline parallelism[18, 9, 4, 15]. One limitation of this technique is the parallelization relies on the accuracy of program dependence analysis and alias analysis[2], thus in some cases the performance of those compilers is restricted. [14] utilizes such thread extraction compiler framework to extract memory check meta-functions from main thread and executes them in extracted threads. They use Mudflap[8], a pointer-use checking tool in GCC to evaluate their parallelization strategy, and the overall performance on SPECint 2000 is approved to be good to substitute manual approach.

As opposed to static parallelization approach, dynamic parallelization is performed at runtime[22]. Such profile driven parallelization uses profile inputs to identify parallelism in programs[23, 26].

Speculative parallelization[25, 7, 28, 19]is also a form of parallelization enabled by profile driven tools.

Our framework of parallelization derived from DSWP, we provide several optimizations as well as bring up a solution for the problem of inaccuracy of data dependency analysis and alias analysis. We use genetic algorithm[21] to search for a PDG with less dependence edges and thus our compiler could generate better parallelism. Our scheme does not require any hardware[13] or software[24] support as needed for speculation and also transparent from programmers.

Genetic algorithm has been used to facilitate parallelization process. GAPS compiler[17] uses genetic algorithm to determine the restructuring transformation applied to each statement and its associated iteration space. They focus on finding a better transformation sequence of optimizations such as loop interchange[1], loop distribution[12], loop fusion[6] and statement reordering[10] thus seek better parallelism for programs.

Chapter 7

Conclusion

This work focuses on compiler automatic parallelization, and we use Decoupled Software Pipelining as the baseline parallelization scheme. The conservative alias analysis limits the parallelization explored by DSWP. To solve this, we bring up with a risky thread extraction infrastructure to facilitate the parallelization. Our risky thread extraction system tries to exploit parallelism in a relaxed dependence environment. Using genetic algorithm, we are able to find better parallelization solution for benchmarks evaluated. To ensure the correctness of the parallelization, we compare the results with the known correct results and provide relaxed memory dependence to programmers to let them verify the optimization with their knowledge of the program. To make our compiler works better, we also implement two optimizations, one is the reduction of threads communication generated by the branch direction flag transition, the other optimization balances workload between threads by providing stage replication.

We use SPECint benchmarks to do experiments because those benchmarks show a good amount of independent workload in loops thus are a good fit of our thread extraction methodology. Our evaluation indicated that those optimizations we applied to the base system are effective in providing better speed up and finding more parallelism.

REFERENCES

- [1] J.R. Allen and K. Kennedy. Automatic loop interchange. In *ACM SIGPLAN Notices*, volume 19, pages 233–246. ACM, 1984.
- [2] R. Allen and K. Kennedy. Optimizing compilers for modern architectures. *Recherche*, 67:02, 2001.
- [3] D. Berlin, D. Edelsohn, and S. Pop. High-level loop optimizations for gcc. In *Proceedings of the 2004 GCC Developers Summit*, pages 37–54, 2004.
- [4] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *ACM Sigplan Notices*, volume 21, pages 162–175. ACM, 1986.
- [5] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.
- [6] A. Darte. On the complexity of loop fusion. In *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on*, pages 149–157. IEEE, 1999.
- [7] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *ACM SIGPLAN Notices*, volume 42, pages 223–234. ACM, 2007.
- [8] F.C. Eigler. Mudflap: Pointer use checking for c/c+. In *GCC Developers Summit*, page 57, 2003.
- [9] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S.W. Liao, and E. Bu. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29(12):84–89, 1996.
- [10] W. Kelly and W. Pugh. A framework for unifying reordering transformations. 1998.
- [11] K. Kennedy and J.R. Allen. Optimizing compilers for modern architectures: a dependence-based approach. 2001.
- [12] K. Kennedy and K.S. McKinley. Loop distribution with arbitrary control flow. In *Supercomputing'90. Proceedings of*, pages 407–416. IEEE, 1990.
- [13] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *Computers, IEEE Transactions on*, 48(9):866–880, 1999.
- [14] S. Lee and J. Tuck. Automatic parallelization of fine-grained meta-functions on a chip multiprocessor. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, pages 130–140. IEEE, 2011.
- [15] A.W. Lim and M.S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 201–214. ACM, 1997.

- [16] F. Mueller. A library implementation of posix threads under unix. In *Proceedings of the USENIX Conference*, pages 29–41, 1993.
- [17] A. Nisbet. Gaps: A compiler framework for genetic algorithm (ga) optimised parallelisation. In *High-Performance Computing and Networking*, pages 987–989. Springer, 1998.
- [18] G. Ottoni, R. Rangan, A. Stoler, and D.I. August. Automatic thread extraction with decoupled software pipelining. In *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on*, pages 12–pp. IEEE, 2005.
- [19] E. Raman, R. Rangan, D.I. August, et al. Spice: speculative parallel iteration chunk execution. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 175–184. ACM, 2008.
- [20] J. Renau, B. Fraguera, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. Sesc simulator. 2005; <http://sesc.sourceforge.net>, 2005.
- [21] J.L. Ribeiro Filho, P.C. Treleaven, and C. Alippi. Genetic-algorithm programming environments. *Computer*, 27(6):28–43, 1994.
- [22] S. Rus, M. Pennings, and L. Rauchwerger. Sensitivity analysis for automatic parallelization on multi-cores. In *Proceedings of the 21st annual international conference on Supercomputing*, pages 263–273. ACM, 2007.
- [23] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 356–369. IEEE, 2007.
- [24] C. Tian, M. Feng, and R. Gupta. Supporting speculative parallelization in the presence of dynamic data structures. In *ACM Sigplan Notices*, volume 45, pages 62–73. ACM, 2010.
- [25] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 330–341. IEEE Computer Society, 2008.
- [26] G. Tournavitis, Z. Wang, B. Franke, and M.F.P. O’Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. *ACM Sigplan Notices*, 44(6):177–187, 2009.
- [27] A. Udupa, K. Rajan, and W. Thies. Alter: exploiting breakable dependences for parallelization. *SIGPLAN Notices*, 46(6):480, 2011.
- [28] N. Vachharajani, R. Rangan, E. Raman, M.J. Bridges, G. Ottoni, and D.I. August. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 49–59. IEEE Computer Society, 2007.

- [29] H. Vandierendonck, S. Rul, and K. De Bosschere. The paralax infrastructure: Automatic parallelization with a helping hand. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 389–400. ACM, 2010.