ADAPTIVE PARALLELIZATION AND OPTIMIZATION FOR THE JAMAICA CHIP MULTI-PROCESSOR ARCHITECTURE

A thesis submitted to the University of Manchester
for the purpose of a PhD
in the Faculty of Science and Engineering

2007

By
Jisheng Zhao
Department of Computer Science

Contents

List of Tables

List of Figures

Abstract

Chip Multi-Processor (CMP) systems are now very popular. This trend to have multi-core and multi-threading makes the system increasingly difficult to target. Also, the lack of runtime information stretches the compiler's abilities to make accurate performance predictions. So how can sequential applications benefit from the ubiquitous CMP? A good choice is a dynamic execution environment that automatically parallelizes programs and adaptively optimizes the code at runtime.

This work investigates how adaptive parallelization and optimization, directed by hardware feedback, improves runtime performance of sequential applications. A Java Virtual Machine based, fully-runtime, parallelization and optimization system is built and evaluated on top of a particular CMP architecture, the JAMAICA CMP, which provides fine-grain parallelism support. This runtime system performs loop-level parallelization for both the normal CMP system and the CMP system with thread level speculation (TLS) support. The developed adaptive optimizations are performed by an online tuning system which tunes parallelized loops adaptively, driven by runtime feedback. These adaptive optimizations concentrate on improving the load balance and data locality for the normal CMP system, and finding the best decomposition to reduce the runtime overhead for the CMP with TLS support.

The evaluation is based on a cycle-level simulation system, which can be easily configured as different hardware configurations. Experiments show that this purely runtime adaptive system is capable of parallelizing and tuning standard

benchmarks and achieving performance improvements as much as 12.5% compared with the scheme used by static compilation with parallelization. By evaluating this system with various hardware configurations, good scalability is demonstrated which means that the applications can be well adapted to different hardware configurations and achieve good performance.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

Acknowledgements

I would like to thank the many people who have provided me the support and encouragement to complete this thesis and my Ph.D. I would like to first thank my supervisor, Dr. Chris Kirkham, who taught me how to work as an independent researcher and has been a steady source of wise discussions, cordial encouragement and support. It has been a true pleasure to work with him.

Thanks to Prof Ian Watson who heads this wonderful research group which I have been in and learned so much. My thanks go to Dr. Ian Rogers and Andrew Dinn, who provided substantial technical support for the compiler and virtual machine system, many valuable discussion, and enhanced my knowledge for building the complex software/hardware system. Matthew Horsnell, who worked on hardware simulation and cooperated with me for investigating the optimizations. Here I also thank to all of the JAMAICA group members for their friendship.

Most of all I thank my grandparents and parents for their love, sacrifice and devotion. They have provided the foundation for all that I have accomplished and all that I ever will.

Chapter 1

Introduction

Chip Multi-Processor (CMP) [36, 53, 59, 48, 65, 83] architectures provide extra resources that can be exploited by threaded applications. As the number of processing cores and threads increases, there is a trend toward fine-grain parallelism [3, 55]. Managed languages based on high level language virtual machines (e.g. Java [61] and C# [27]) which employ dynamic compilation and optimization have been widely used in the software industry. Is it feasible to build a new runtime system which can perform runtime parallelization and optimization on a CMP architecture? Does this approach make sense and compare to the programming models used in scientific computation?

The research work presented in this thesis will answer these questions by developing and evaluating a Java virtual machine based on runtime parallelization and optimization system. This work then goes on to look at adaptively configuring and improving this runtime environment achieving performance beyond what is possible with parallelization alone.

1.1 Motivation

1.1.1 New Benefits and Challenges in CMP Architecture

Traditional large-scale and small-scale multiprocessor systems have been able to obtain speedups with large scientific programs that contain a large amount

of parallelism. This parallelism can be found and exploited by parallel compilers [91, 13, 64, 2, 5, 75]. The development of new CMP architectures with lightweight threading provides the ability to extract small parallel tasks (i.e. parallel tasks which have about 30 to 100 instructions). By integrating multiple processor cores into one die, CMP architectures make the interprocessor communication process quicker and more efficient, so running even 30-100 instructions in parallel will obtain a speedup.

But the ability to extract fine-grain parallelism using much smaller threads can also trigger new problems for the efficiency of parallelism. Compared with parallel tasks on a multiprocessor system that contain more than 1,000 instruction cycles, the performance of the CMP's small task is more easily affected by load imbalance and memory delay. For example, given a fine-grain task whose execution cycle count is 100 ideally, if there is one more cache miss in execution and the memory delay between L1 cache and L2 cache is 16 cycles, then the performance will decrease by 13.7%. So the change of hardware infrastructure needs more precise compiler analysis to improve hardware utilization.

Thread Level Speculation (TLS) [38, 80, 78, 6, 25, 34, 54] is another trend in CMP development. It splits the sequential program into parallel threads and executes them speculatively. When a true dependence violation is detected, the hardware must ensure that the "later" thread in the sequence executes with the proper data by dynamically discarding threads that have speculatively executed with the wrong data. The major challenge of TLS is how to decompose the program efficiently, thus reducing the number of collapsed threads and improving the parallelism.

1.1.2 Lack of Runtime Information

Optimizing compilers are limited by the information that is available to them at compile-time. Optimizations implemented in a compiler are applied using models of program and machine behaviour that include many simplifying assumptions due to the complexity of modern computing systems and the lack of accurate runtime information. There are two types of the runtime information: the applications' input data sets, and information about the target machine configuration. Most applications are written so that they can be applied to different sizes of

input data set, and so, at compile-time, the exact input data set information is not available. The target machine's configuration can also be varied, e.g. different sizes of cache and memory, different speeds of processor.

Optimizing compilers are forced to evaluate the effectiveness of optimizations statically, most often without any knowledge of the input data set and hardware configuration. The lack of runtime information stresses the compiler's abilities to make accurate performance predictions. For example, it is difficult to give at compile-time a suitable tile size for a parallelized loop which works on a CMP, since the compiler needs to consider : the size of input data, the size of the loop (or the number of iterations), the size of L1 cache for the processors, and the number of processors that are available for the parallel loop.

1.1.3 Iterative Compilation and Runtime Optimization

Using runtime information to exploit more optimizations is a good choice for improving applications' performance. By employing an efficient runtime profiling mechanism, the optimizer can get enough information to decide how to optimize the program (e.g. code specialization, tile size selection). Some runtime optimization frameworks have been built for optimizing *Dynamic Binary Translation* (DBT) [77] and a linear algebra library [90].

The basic issues for runtime optimization are:

- How to construct a search space, i.e. the multi-version code, configuration parameters, and the optimizations used for recompilation.
- How to search for an optimum? This depends on the selection of the search algorithm and the size of the search space.

Iterative compilation uses runtime feedback to improve the optimization iteratively. Recent progress in iterative compilation [50, 51] has shown that this approach can achieve high speedups, outperforming static techniques. Given a large search space, an acceptable performance (i.e. 90% of the optimal) can be found in limited iterations by employing a simple search algorithm (i.e. hill-climbing [66]). The shortcoming of this methodology is the overhead for recompilation

(i.e. building multiple target programs with different optimizing parameters). So while using a runtime search mechanism can get potential benefit, the major problem is how to handle the overhead.

There are three major types of runtime program optimization: multi-version code selection, runtime parameterization, and runtime code generation.

- *Multi-version code selection*: multiple versions of a code section are generated at compile-time, and at runtime one of these versions is selected based on the monitored performance.
- *Runtime parameterization*: the code is generated at compile-time, but it is configurable, so the runtime tuning system can reset the configuration (e.g. reset the tile size for blocking).
- *Runtime code generation*: generating code on-the-fly at runtime can be the most powerful runtime optimization, since it can use various methods to optimize the code. But it also has the highest overhead, because it needs to invoke the compiler at runtime.

The first two methods have small runtime overhead, because they are easy to implement by runtime reconfiguration. The third method needs the assistance of the compiler. To make the runtime system work efficiently, the compilation thread should be dispatched to a separate processor core in the CMP system. The optimizer can also allocate multiple compilation threads to build multiple versions of code, when the runtime system is not busy and there are enough idle processor cores.

1.1.4 High Level Language Virtual Machine

High level language virtual machines provide an efficient facility to perform runtime optimization and code generation. Using runtime feedback information to drive runtime optimization has been widely used in Java virtual machine implementations [44, 1, 81] to improve runtime performance. For example, Sun HotSpot JVM uses runtime profiling to drive the runtime recompilations;

JikesRVM uses the dynamic compiler to plant instrumenting code into the compiled Java method; this code performs runtime profiling and the runtime optimizers can use this statistical data to drive the optimizations adaptively (e.g. adaptive inlining, and adaptive recompilation with more optimization phases).

1.2 Adaptive Runtime Optimization

Adaptive runtime optimization means using runtime feedback to drive optimization adaptively. Figure 1.1 shows a basic model of an adaptive controller. It uses feedback to drive the control process.

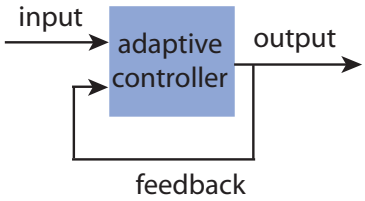


Figure 1.1: A Simple Model of An Adaptive Control System.

To allow optimization to occur in the context of more accurate knowledge of the size of input data set and hardware configuration, exploring runtime optimization (or parallelization) is necessary. This thesis will address whether it is feasible to build a fully-runtime optimization system which can perform parallelism creation and optimization driven adaptively by runtime feedback information.

The runtime system presented in this work is an adaptive runtime optimization system which combines a loop level parallel compiler and an automatic online tuning framework within a Java virtual machine that works on the JAMAICA CMP architecture [93]. By employing the JAMAICA CMP's capability of distributing fine-grain parallelism, the adaptive optimization system can perform online tuning to improve the performance of parallelized code with acceptable overheads.

In this runtime system the input is the program code (i.e. Java bytecode); the

adaptive controller is the runtime executing environment (including the parallelization and optimization); the output and the feedback are the runtime information generated by executing the input program code (e.g. the execution cycle count). The runtime feedback information is used to drive the optimizing process.

To construct this runtime system, several issues need to be considered (details will be discussed in Chapters 4, 5 and 7):

- How to construct an efficient dynamic executing environment on a CMP architecture? This depends on both hardware and software support.
- How to perform runtime parallelization to improve the sequential application's performance on CMP architecture?
- How to utilize runtime information and perform runtime optimization? This will need to consider: how to construct a search space? And how to search for an optimum?

1.3 Related Work

To open the discussion of this research, this section briefly reviews the evolution of and current issues surrounding both runtime optimization and iterative compilation, including the typical frameworks and research work that have applied different methods for runtime evaluation to improve an application's runtime performance [1].

1.3.1 Runtime Optimization Frameworks

ATLAS

ATLAS is a widely used linear algebra library which was built by the *Automatically Tuned Linear Algebra Software* (ATLAS) project [90, 89]. This project applied an automatic tuning mechanism to software packages (e.g. basic linear algebra software library(BLAS)). The software package provides many ways of

[1] The specific points of comparison will occur later in Chapters 5 and 7.

doing the required computational work, and uses empirical timings to choose the best method for a given architecture. ATLAS typically uses code generators to generate multiple code versions, and has sophisticated search scripts to find the best choice.

ATLAS uses a profiling run to perform optimization and find the optimal choices for the basic BLAS routines, then these optimized routines will be used to construct the application. So, ATLAS is not a fully-runtime optimization system, but it is still typical of one type of optimization system.

ADAPT

Voss and Eigenmann [86, 85] established an adaptive optimization framework named *Automatic Decoupled Adaptive Program Transformation* (ADAPT) which performs dynamic optimization on hot spots through empirical search. ADAPT is built in a multiprocessor environment and is fully-runtime. It uses dynamic recompilation to evaluate different optimizations and a domain-specific language to drive the search in the optimization space for a specific optimization (e.g. for loop unrolling, each level of unrolling will be compiled, run and timed, and the fastest version will be kept and used for the hot spot). ADAPT evaluated a series of optimizations which have various parameters (e.g. loop tiling, loop unrolling). The optimizer (i.e. profiler and compiler) used for collecting statistics and recompiling was run on an isolated processor; this mechanism could reduce the recompilation overhead at runtime and the uniprocessor related optimization (e.g. loop tiling/unrolling, useless copying, flag selection) can benefit from this improvement without interference, however optimization for automatic parallelization could be affected by losing one processor which may be required in a multi-threaded or multi-programmed environment.

Jrpm

Jrpm [21] is a Java virtual machine running on the Hydra CMP [37] system which has TLS support. This runtime system performs runtime speculative loop-level parallelism. Its runtime optimization concentrates on how to decompose sequential Java programs efficiently, so they can benefit more from speculative

parallelism. Jrpm uses the compiler to generate code for the profiling run, and employs a hardware based profiling mechanism [22] to help the compiler identify which loops are most suitable for speculative parallelism. Jrpm needs to generate code at least twice to perform the profiling run and the optimized run.

Dynamic Feedback

Diniz and Rinard [28] use a simple version selection mechanism which reacts to runtime inputs and loop parameters. Their dynamic optimization system also generates code to provide dynamic feedback, allowing automated selection of the best synchronization policy for parallel execution. In their scheme, a program has alternating sampling and production phases. In the sampling phase, code variants (multi-version code) generated at compile-time are executed and timed. The phase continues for a user-defined interval. After the interval expires, the version which has the shortest average execution time is selected to be used in the production phase.

DyC

DyC [35] selectively dynamically compiles programs during their execution, utilizing the runtime computed values of variables and data structures to apply optimizations that are based on partial evaluation. The dynamic optimizations are mainly specialization related optimizations. These optimizations are preplanned at compile time in order to reduce their runtime cost. Because of the large suite of optimizations and its low dynamic compilation overhead, DyC achieves good performance improvements on practical programs that are larger and more complex than just kernels.

Dynamo

The HP Dynamo project [11] is a runtime binary translation system. Its goal is to optimize a native executable as it runs. Dynamo begins by interpreting the native executable and then collects and optimizes frequently executed code traces. These traces can extend beyond basic block and subroutine boundaries, creating larger

blocks with simplified control flow. Dynamo operates on the runtime stream of binary instructions and its overhead is directly in the critical path of the application. As it works at binary level, the adaptive optimizations are also performed at the binary level.

1.3.2 Iterative Compilation

Kisuki, O'Boyle and Knijnebury et al. [50, 51, 15] investigated iterative compilation for loop tiling and loop unrolling. Their proposed compilation system achieved high speedups, outperforming static techniques. The system shows that high levels of optimization can be achieved in a limited number of iterations by applying a hill-climb-like search algorithm. In recent progress, *Fursin et al.* [33, 32] explored online empirical searches using scientific benchmarks. To reduce runtime code generation overheads, a set of optimized versions of code were created prior to the execution of a program. These versions were then evaluated at runtime with the best performing version chosen for subsequent execution. They employed predictive phase detection to identify the periods of stable repetitive behaviour of a program and used these phases to improve the evaluation of alternative optimized versions.

Similarly *Lau, Arnold et al.* [57] investigated an online framework for evaluating the effectiveness of optimizations. They present a virtual machine based online system that automatically identifies the optimal parameters for optimizations, and give an example for selecting a method inlining policy by utilizing the framework. By deploying optimizations at the method-level, more runtime noise is present in the system, and they use a large number of iterations to assess the effectiveness of optimizations.

1.4 Contributions

The major contributions of this thesis are building and evaluating an adaptive optimization framework for runtime parallelization and thread level speculation: *Online Tuning Framework* (OTF). This system is built with a high level language

virtual machine (i.e. a Java Virtual Machine) which works on a novel CMP architecture (JAMAICA CMP architecture with lightweight task creation). Table 1.1 shows the features of this research compared with related work.

The detailed list of contributions is:

- A loop-level parallel compiler which can perform automatic parallelization [2] and is embedded in JAMAICA Virtual Machine's optimizing compiler [29]. This compiler implementation demonstrated how to handle runtime parallelization problems in a Java Virtual Machine (e.g. data dependence analysis, long running loop promotion, alias analysis et al.) and utilize the lightweight mechanism to achieve speedup and scalability on different hardware configurations with acceptable runtime overhead.

- An adaptive optimization framework which can evaluate the parallelized code segment (parallelized loops) and perform automatic tuning to improve runtime performance adaptively with low overhead. The performance goes beyond naïve static analysis and gains up to 12.5% in certain input data size and hardware configurations.

- Several runtime optimizations based on the adaptive mechanisms, which can improve the load balance and data locality. These optimizations can be dynamically tuned within the adaptive optimization framework.

- An extended multi-version cache module and instruction set which can support thread level speculation on the JAMAICA architecture. This brings about a novel combination of lightweight threading and speculative execution.

- Analysis of adaptive decomposition for speculative loop-level parallelization. By searching for the best decomposition at runtime, the performance gains up to 12% improvement compared to naïve TLS.

Parts of this research work have already been published [94, 95, 97, 74, 96].

[2] The current implementation can perform *DoAll* parallelization and part of *DoAcross* parallelization for scatter operations.

Optimization Framework	High-Level Optimization	Online Optimization	Runtime Generation	Runtime Sampling	Runtime Reconfig	Iterative Modification	Parallelization Support
OTF	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Dynamic Feedback	Yes	Yes	No	Yes	No	No	No
Iterative Compilation	Yes	No	No	Yes	No	Yes	No
Jrpm	Yes	Yes	Yes	Yes	No	No	Yes
ATLAS	Yes	No	No	Yes	No	Yes	Yes
ADAPT	Yes	Yes	Yes	Yes	No	Yes	Yes
Dynamo	No	Yes	No	Yes	No	Yes	No
DyC	Yes	Yes	Yes	Yes	No	Yes	No

Table 1.1: The Primary Features of JAMAICA OTF Compared with Other Optimization Frameworks.

1.5 Thesis Organization

Chapter 2 introduces the hardware platform: the JAMAICA CMP architecture which is a scalable chip multiprocessor system with a novel token ring mechanism for task distribution. Chapter 3 introduces the software infrastructure: JaVM which is a Java virtual machine running on the JAMAICA CMP. JaVM provides the runtime support for lightweight threads.

Chapter 4 introduces the design and implementation issues of the dynamic loop-level parallel compiler. Chapter 5 gives the detail of the online tuning framework (OTF) which performs adaptive runtime optimizations on the parallelized loops. Several of OTF's adaptive optimizations that improve load balance and data locality are evaluated here.

Chapter 6 introduces the extended hardware and software support for enabling thread level speculation on the JAMAICA CMP architecture. Chapter 7 evaluates the adaptive optimizations which concentrate on how to decompose the program to make the speculative parallelization work efficiently.

Finally, Chapter 8 concludes this work and indicates future work.

Chapter 2

JAMAICA CMP

Java Machine And Integrated Circuit Architecture (JAMAICA) [93] is a CMP architecture with multi-threading [70] processor cores, hardware support for lightweight task distribution, and *Chip Multi-Cluster* (CMC) [43] support.

The major aim of the JAMAICA CMP is to provide a platform upon which dynamic compilation for parallelism can be evaluated. Section 2.1 discuss the basic issues in this architecture. Section 2.2 gives a brief introduction to JAMAICA CMP's simulation environment. The last Section 2.3 summarizes this chapter.

2.1 JAMAICA Architecture

2.1.1 Processor Core

The JAMAICA CMP employs a simple 5-stage single-issue pipeline processor core (shown in figure 2.1)[1], similar to the MIPS R2000 [49], and complicated by the addition of register windows and a token interface unit to support thread distribution.

JAMAICA supports multi-threading processor cores which help to reduce the effect of memory latency and improve overall throughput. Each processor core

[1]F1, F2, M1, M2 cooperate with TLB and caches to handle virtual addresses. F1 and F2 could be treated as one stage F for instruction fetch. M1 and M2 could be treated as one stage M for memory operation.

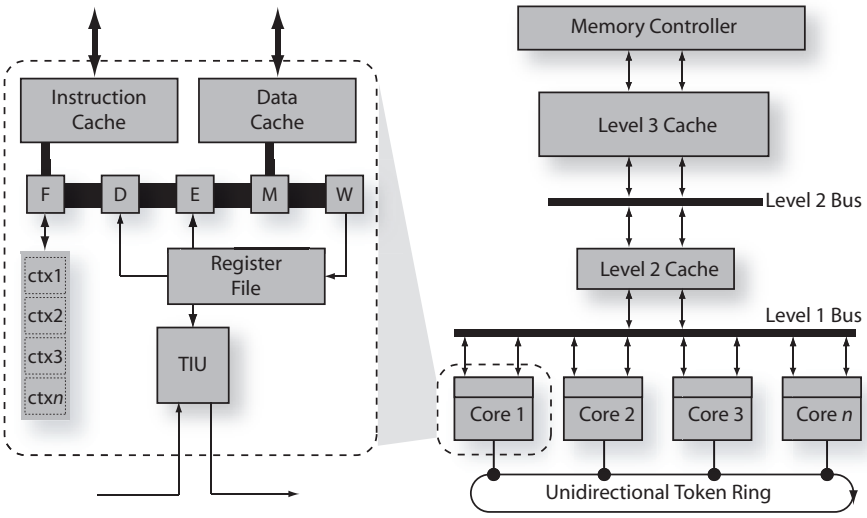


Figure 2.1: JAMAICA CMP Processor Core.

can maintain 1, 2 or 4 hardware thread contexts. Each context maintains its own context specific registers, containing register, interrupt and other thread specific state. Each context shares the core pipeline and level 1 instruction and data caches.

The contexts within a processor core can reside in one of the five states: *runnable*, *stalled*, *waiting*, *empty* and *idle*. The state transformation diagram is shown in Figure 2.2. JAMAICA employs a blocked, switch-on-cache-miss multi-threading policy [84], extended by an additional switch-on-timer policy which triggers a context switch event when a context has been running for 1,000 cycles unhindered by cache misses. A round robin policy is used to rotate the *active* context from the list of *runnable* contexts for scheduling into the pipeline. If no contexts are *runnable* at a context switch event, the core itself becomes idle.

Context switching can help to hide memory latency, by keeping the core busy executing instructions from a runnable context during the memory stall incurred by another context, reducing memory latency and improving overall throughput.

2.1.2 Register Windows

As object oriented languages encourage creating modular code, an application may be composed of lots of small functions that will be executed frequently. In a

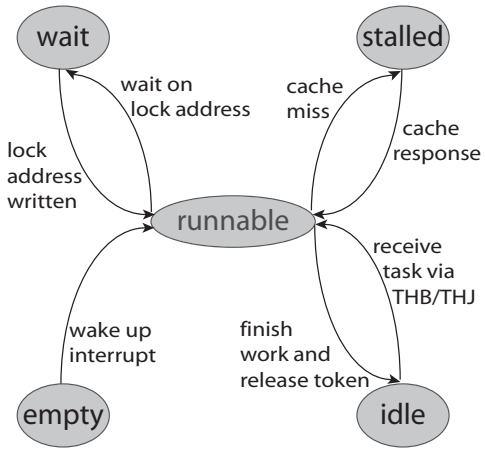


Figure 2.2: Context States Transformation.

processor with a flat register file, calling and returning from functions requires the current active registers (the registers used by caller) to be saved to memory and restored from memory at the callee's return. Such load/store operations are very costly when function calls happen frequently. The JAMAICA CMP architecture employs a register window mechanism to reduce the effects of frequent method calls. A large windowed register file is shared between all of the contexts in a JAMAICA processor core. The hardware supporting the register file implements a register window scheme [87], based on the Multi-Windows proposal [76]. 32 registers are visible to the compiler; these registers are divided into four windows each containing eight 32-bit registers.

- *Global Window*: shared by all contexts on a processor core. (register *%g0 - %g7*)
- *Extra Window*: private per context, statically allocated, non-volatile across methods calls. (register *%x0 - %x7*)
- *Input Window*: private per context, dynamically allocated at each method call, non-volatile across method calls. (register *%i0 - %i7*)
- *Output Window*: private per context, dynamically allocated at each method call, volatile across method calls. (register *%o0 - %o7*)

All the contexts on a processor core share the *Global Window*, which is mapped directly to the bottom eight physical registers. Each context has a private *Extra Window*, mapped into the physical register windows located directly above the *Global Window*. The *Input Window* and *Output Window* are allocated and released dynamically during method calls (shown in Figure 2.3). In JAMIACA the caller's *Output Window* overlaps with callee's *Input Window* and passes six variables [2]. Passing more variables requires spilling into the stack frame.

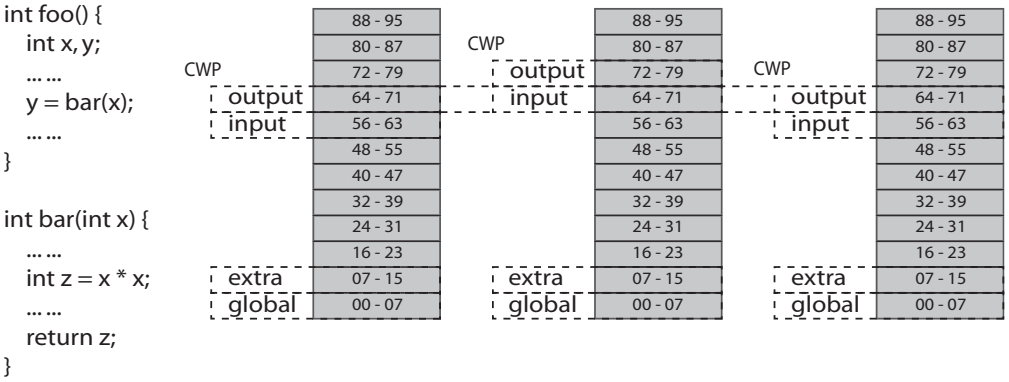


Figure 2.3: Function Call and Return.

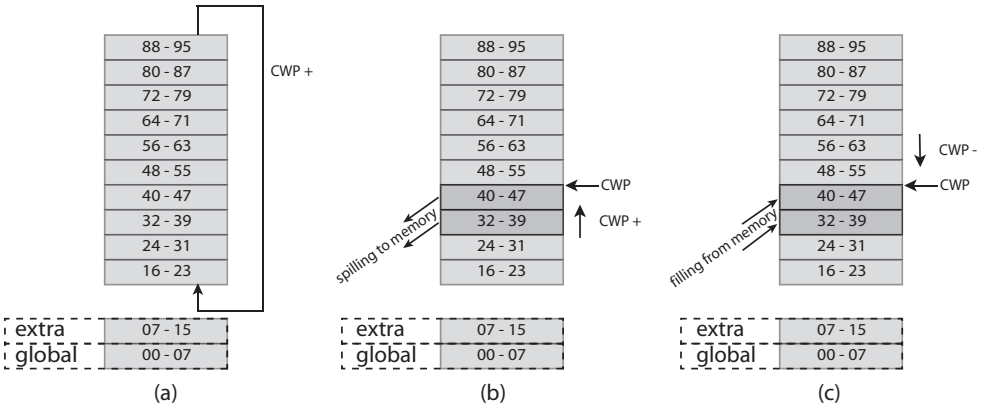


Figure 2.4: Spilling and Filling using Register Windows.

In this register window scheme, a large register file is divided into windows each containing a number of registers, the register fields in instructions are interpreted as offsets from a *Current Window Pointer* (CWP). When the application's call

[2]Two registers are explicitly used to pass the return PC and stack pointer.

depth exceeds the number of windows available in the physical register file, the system still needs to spill and fill the registers from memory. When the incrementing of CWP overflows the top of physical registers and the CWP circles around to the bottom of the physical registers (shown in Figure 2.4 (a)), the bottom window is spilled to memory (shown in Figure 2.4 (b)). When the CWP unwinds again, the spilled window will be missing and this window must be filled from memory in order to complete a return (shown in Figure 2.4 (c)).

2.1.3 Token Ring and Task Distribution

A novel feature of the JAMAICA CMP is explicit hardware support for lightweight task distribution. This hardware support consists of a ring bus connecting all of the processing cores (shown in Figure 2.5). The ring allows *active* processor contexts to locate *idle* processor contexts.

The major advantages of this token ring mechanism is that it provides a simple mechanism to: distribute information about idle processing resources; resolve competition to allocate an idle processor context; and schedule a task for execution on the allocated processor context. Any runtime thread can simply grab a token as it passes, granting it exclusive ability to distribute a parallel task to the processor context which released the token (the next chapter discusses how the runtime system supports token-based task distribution). This avoids many of the complexities and costs involved in employing a software scheduler, particularly those associated with memory-based synchronization. The following subsections discusses the implementation details of this mechanism.

Token Release

When an *active* context exits from the bottom of its executing stack, detected in hardware by a return from an *Input Window* that has no predecessor, the context's state changes from *runnable* to *idle* (see Section 2.1.1) and a token is released onto the ring bus where the tokens circulate. The token placed onto the ring simply contains the identity, a unique number referred to as the *Context ID* stored in a hardware context register, of the context now in the *idle* state.

Token Selection

The JAMAICA instruction set provides an additional instruction: *Token Request* (TRQ) instruction which requests a token from the ring bus. To help the *active* thread grab tokens from different processors or clusters selectively, a token distance mechanism is employed here. The token distance is a special flag used to annotate whether the token is from a local processor context or remote processor context in a single CMP. For CMC architecture, the token distance is used to annotate whether the token should come from a local cluster or remote cluster. When the *active* thread sends a token request on the ring bus by using TRQ instruction, it can set the token distance in the operator register to inform the ring which token it prefers.

Task Distribution

To distribute tasks, the JAMAICA instruction set provides two special instructions: *Thread Jump* (THJ) and *Thread Branch* (THB) which distribute a task to the context identified by the *Context ID* stored in the token.

The *active* thread gets a token from the ring bus by using a TRQ instruction and uses THB/THJ [3] to distribute a task to the *idle* context identified by the token got by TRQ. Ten 32-bits values are transferred by THB/THJ instructions: start PC value, the *active* context's *Context ID* and the *active* context's current *Output Window* registers [4]. The start PC value will be set in the *idle* context's PC register, the *active* context's *Context ID* will be set in the corresponding context register and the *Output Window* register will be assigned to the *idle* context's *Input Window* registers. The data transmission is performed on the L1 cache bus directly [5]. After the transformation, the *idle* context is *runnable* and starts to execute at the new PC.

In contrast to some of the traditional task-spawn mechanisms which replicate

[3]The difference between the THB and THJ is the transformed PC value. THJ need to be assigned an address which is PC value. THB need to be assigned a branch offset (integer value), and the transformed PC value is the *active* context's current PC value plus branch offset.

[4]The *Output Window* registers' values are carried by data lines, PC and *Context ID* are carried by address lines.

[5]For a CMC extension, the data transmission needs to pass the level 2 cache bus, and details will be discussed in Section 2.1.4

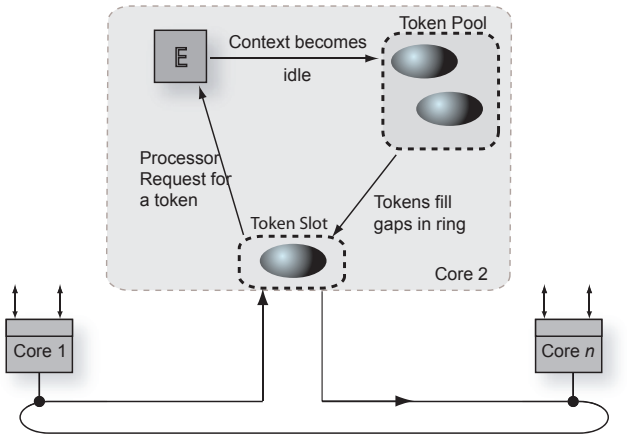


Figure 2.5: The Token Ring Mechanism in the JAMAICA CMP Architecture (refer to Figure 2.1).

all registers and the PC to remote processors, the JAMAICA architecture just replicates the output window registers, the PC and *context ID*s when it ships tasks to another processor context. So the shipped task must have an initialization process to initialize some values so it can start to execute the working code.

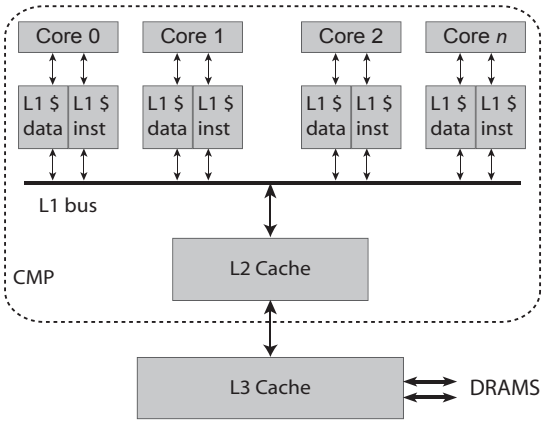


Figure 2.6: Memory Hierarchy on Single CMP.

2.1.4 Memory Hierarchy and Scalability

The JAMAICA architecture supports single CMP and CMC [43]. The single CMP (or "normal" JAMAICA) has private level 1 (L1) instruction and data

caches, connected via a shared memory bus to a single shared level 2 (L2) cache (shown in Figure 2.6). The CMC, combines multiple single CMPs together and connects them via a shared memory bus to a single shared level 3 (L3) cache (shown in Figure 2.7).

Cache Coherence

Between each level of caches, a shared split-transaction bus is used for connection. All caches in the architecture maintain sequential consistency to allow for standard shared memory programming.

The CMP architecture must maintain memory consistency because each processor core has its own L1 cache. On the JAMAICA CMP, memory consistency is maintained through cache coherence protocols implemented across the shared bus that connects the L1 and L2 caches. The cache coherence protocol implemented in the JAMAICA architecture is based on the MOESI protocol [39]. Each processor core can communicate with its L1 data cache privately. If an address is requested that is not currently cached, a request must be made across the shared bus. This request, and the continual snooping by all L1 caches on the bus, updates the state of cache lines loaded from the L2 shared cache.

The bus used to connect L1 and L2 caches is a split transaction arbitrated bus. Only one processor will be given access to the bus that it shares during the arbitration phase. Once bus ownership is granted and the core has been given a transaction id, a processor core's L1 cache can request a cache line from the L2 cache or place a cache line on the data wires to write it back to the L2 cache.

As mentioned in the previous section, the bus is also used for thread distribution between the processor cores, it transfers the register data (the input window registers, PC and SP).

Multi-Cluster Cache Coherence

To increase the ability of the JAMAICA architecture to scale with the addition of more processing cores the single shared bus architecture is replaced by a scalable multi-level cache hierarchy (shown in 2.7). The multi-level hierarchy, by dividing

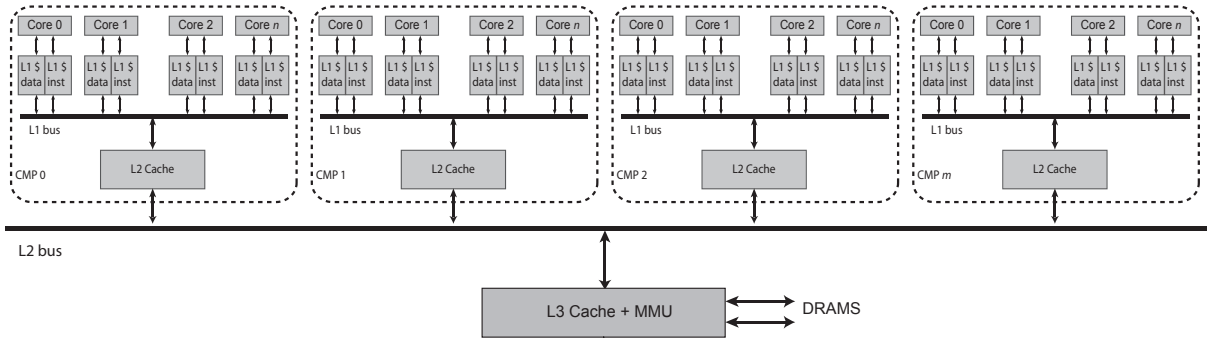


Figure 2.7: Memory Hierarchy on the CMC.

the total number of cores into *clusters* each connected through a hierarchy of memory bus and caches, can allow many more cores to be integrated onto a single chip, whilst maintaining shared memory and limiting the span of each interconnect to reduce the effects of cross-chip wire-delay.

Each intra-cluster network is independently arbitrated and accessed concurrently allowing the cores within each cluster to access the larger cluster-shared cache with less contention. The additional scalability, however, comes at the expense of a more complex coherence protocol that needs to maintain coherence across multiple clusters, and the need to maintain cache inclusion.

2.1.5 Interrupts

The JAMAICA architecture supports a limited number of hardware and software generated interrupts. Interrupts vector a context's execution path to interrupt handler code located at the bottom of the memory image, addressed by the type of interrupt. A software interrupt, SIRQ, can be delivered to any context in the *runnable*, *waiting*, *stalled* or *empty* states. The SIRQ is delivered to a processor context by the shared memory bus in a similar manner to the THB/THJ. Contexts in the *idle* state can only be woken up by using THB/THJ instructions, therefore a SIRQ to an *idle* context is ignored.

Currently, a single software interrupt is used to wakeup all contexts during the system booting process, and a hardware interrupt is used to trap on access to an invalid memory address.

2.1.6 Devices

There is no defined device interface in the JAMAICA architecture, and hence no associated device hardware interrupts. The JAMAICA simulator enables calls to the underlying operating system for I/O operations through a set of defined built-in operations (discussed in the next section).

2.2 JAMAICA Simulation

JAMAICA is a simulated architecture which is currently implemented in a software simulator. The JAMAICA simulation platform, *jamsim*, is a Java simulation platform that has been developed to execute binaries created for the JAMAICA ISA. *jamsim* supports several models of simulation: fast, functional simulations required for system software development as well as cycle-level simulations, which are essential for quantitative evaluation of the architecture.

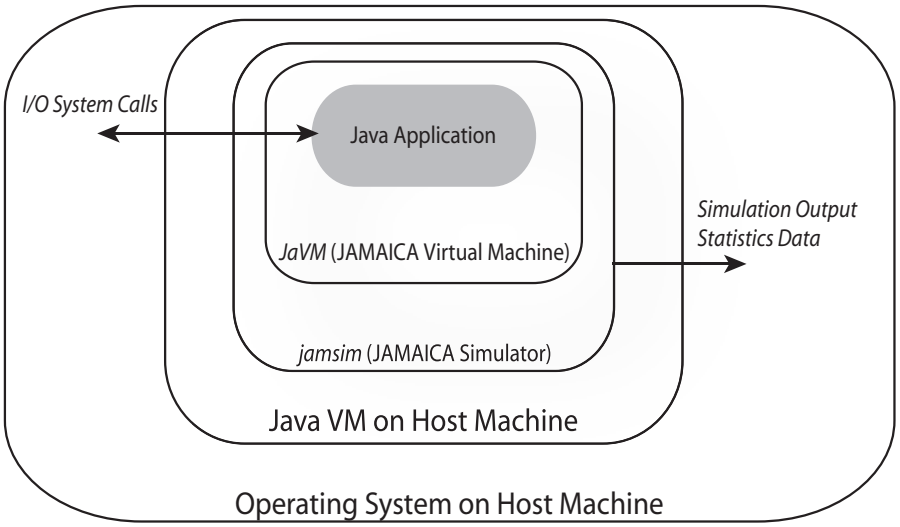


Figure 2.8: JAMAICA Simulator.

jamsim is capable of simulating the processor, the interconnection and the memory hierarchy. Parameterizable components of the simulated architecture include the number of processor cores, the number of contexts per core, the different cache sizes and the memory hierarchy. Where it makes sense the simulation platform can be composed of components at different levels of modelling.

As JAMIACA does not support a device interface (mentioned in Section 2.1.6), complete system simulation is enabled using a special range of built-in subroutines. These subroutines, which attempt a call to small negative memory addresses are trapped during simulation, and the simulator calls out to the underlying operating system through the Java virtual machine in which the simulator is running (shown in Figure 2.8).

2.3 Summary

This chapter introduced the hardware infrastructure on which this thesis is based. Compared to general CMP architectures, the JAMAICA CMP employs a novel token ring mechanism and thread branch instructions which can help software to distribute tasks without a traditional centralized scheduling mechanism. The JAMAICA CMP also supports multiple CMP clusters which can reduce bus transactions by separating tasks.

Chapter 3

JAMAICA Virtual Machine

The JAMAICA virtual machine (JaVM) [29] is an environment allowing the execution of Java programs directly on the JAMAICA architecture without operating system support. This chapter introduces the working mechanism of JaVM, and its interaction with the JAMAICA hardware, especially for the lightweight thread mechanism.

Section 3.1 gives an overview of JaVM, including the basic components and how these components co-operate with each other. Section 3.2 introduces the detail of the JaVM runtime system. Section 3.3 and Section 3.4 give a brief introduction to the dynamic compilation and adaptive optimization systems (discussed in Chapter 4 and 5). Section 3.6 introduces how the JaVM can cooperate with the simulation environment. Section 3.7 summarizes this chapter.

3.1 Overview

The JaVM is the Jikes Research Virtual Machine (RVM) [44, 9] ported to the JAMAICA architecture and running without an underlying operating system. The major modifications are in the following three areas:

- The runtime service system which provides support for thread scheduling, loading and linking code, and garbage collected memory management.

- The dynamic compilation system which comprises two compilers: the baseline compiler, that performs a simple mapping of bytecode to machine code, and a more advanced optimizing compiler.
- The adaptive recompilation system uses runtime profiling information to determine whether to recompile and thereby improve the performance of executing code.

The interaction between the components of the system can be seen in Figure 3.1.

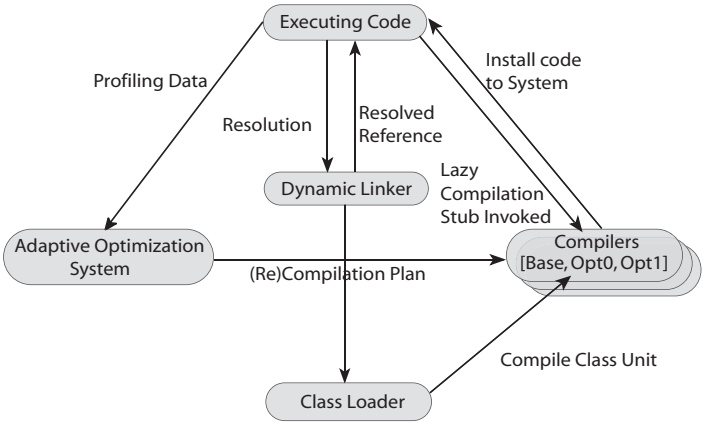


Figure 3.1: The Interaction between JaVM's Adaptive Optimization System and Dynamic Compilation System.

3.2 Runtime System

The JaVM runtime system supports multi-threading from Java threads and fine-grain threads created by the parallelization system. There are two basic elements in the runtime system: *Virtual Processor* and *Java Thread*. The Java threads should be scheduled among the virtual processors, and each virtual processor has its own corresponding physical thread context[1].

[1]In JAMAICA CMP, the basic physical execution unit is a thread context (discussed in Chapter 2), each processor core can contain one or multiple thread contexts.

3.2.1 Virtual Processor and Thread Scheduling

To manage the physical processor contexts, JaVM employs a virtual processor object, VM_Processor, which encapsulates the information related to the processor context. Each processor context has one corresponding VM_Processor object. The basic data structures in VM_Processor are the thread queues which are used to support the thread scheduling (shown in Figure 3.2). Each VM_Processor has one *Active Thread* which is a Java thread executing on the physical context. The active thread should yield after an execution interval, then other Java threads can get the physical context and execute. To perform the thread scheduling mechanism, JaVM employs five types of thread queues, three in each VM_Processor, a single global queue for suspended GC threads, plus a wait queue used to implement heavy-weight locks, object wait and notify calls.

- *Ready Queue* (rq): an un-synchronized queue, which contains the threads that can be activited.
- *Transfer Queue* (tq): a synchronized queue, which contains the threads that are transfered from other VM_Processors to rq.
- *Idle Queue* (iq): contains a *idle thread*, and the idle thread will be transfered to tq when there are no threads in rq and tq.
- *Collector Queue* (cq): contains the GC threads. Each GC thread has its own corresponding virtual processor. The GC threads are activated directly, without needing to pass though rq or tq.
- *Wait Queue* (wq): wait queues are allocated from a global pool and transiently associated with: a VM_Lock which a competing thread currently owns; a synchronized method or a synchronized object guarding a block which is being executed concurrently by a competing thread; an object which is the target of a call to Object.wait(). In the first two cases threads will only insert themselves into a wait queue after busy waiting for a short period. In the third case the waiting thread owns a lock on the target object which it releases as it inserts itself into the associated wait queue. Threads are removed from wait queues and rescheduled when, respectively: the VM_Lock is unlocked; the synchronized method/block is exited or the object being waited on it notified.

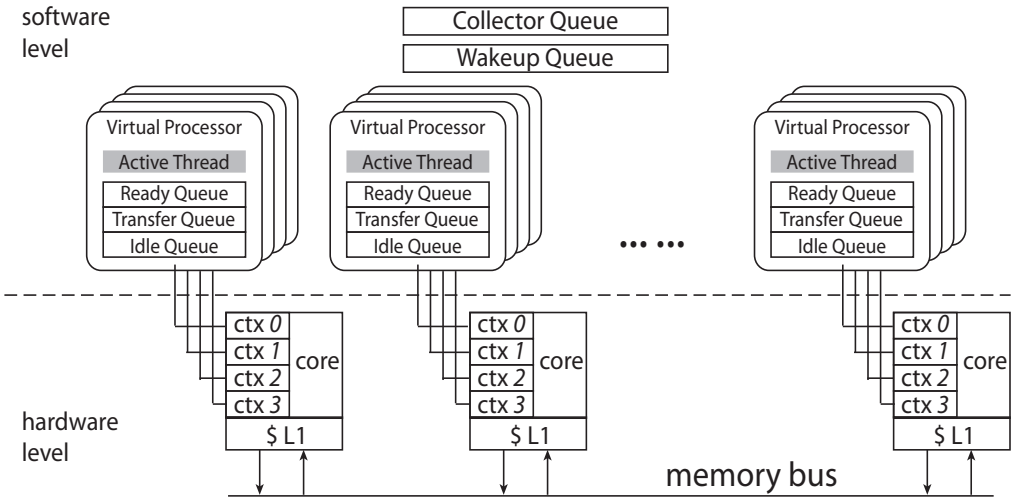


Figure 3.2: The Software/Hardware View of Virtual Processors.

The Java threads are the basic execution units in JaVM at the software level [2]. Each Java thread should get a fair opportunity to execute on the physical context. The general thread scheduling scheme is shown in Figure 3.3 (a). Given a Java thread which is being scheduled or a newly created Java thread, it should be placed in the *tq* at first. Before putting the scheduled thread into the *tq*, the thread scheduler process should check if there is an idle processor context, if so, use the branching mechanism to insert this Java thread into the remote context's *tq* to improve the load balance (details of how the context is restarted and control switched into the queued Java thread are described in the next section). If a Java thread calls to continue running by scheduling the queued thread on an idle processor context using THB/THJ. It will only place itself in *rq* and context switch into another Java thread if it fails to detect an idle processor context. If a Java thread yields into a wait queue to wait on a lock or synchronized method/block then it context switches into another Java thread from *tq*/*rq* or, failing that, into an idle thread obtained from *iq*.

A second scheme is the GC thread scheduling (shown in Figure 3.3 (b)). JaVM implements a *Stop-The-World* GC policy, so each virtual processor should have its own GC thread and all of the GC threads should be activated/deactivated at the same time. JaVM uses a global thread queue *cq* to store all these GC threads. When GC is triggered, the thread which starts GC will dequeue each of the GC

[2]The interrupt handler is just used for system booting (discussed in Section 3.6)

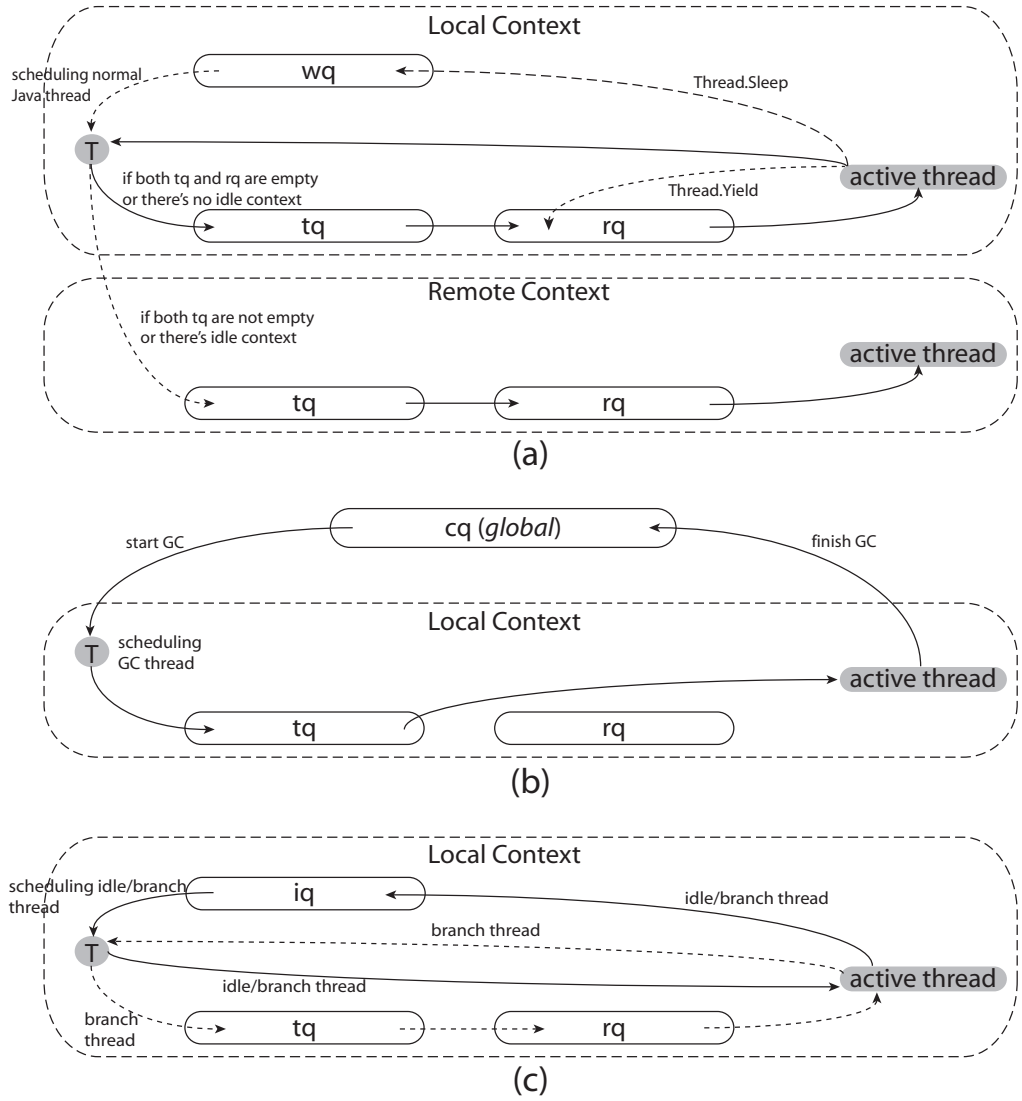


Figure 3.3: The Thread Scheduling Schemes.

```
public static boolean availableWork() {
  VM_Processor myProcessor = VM_Processor.getCurrentProcessor();

  // if the ready queue or transfer queue has something in it
  if (!myProcessor.readyQueue.isEmpty()
      || !myProcessor.transferQueue.isEmpty()) {
    return true;
  }

  // nothing to run
  return false;
}
```

Figure 3.4: The Code for Checking Available Work.

threads from the *cq* and insert it into the *tq* of its associated VM_Processor. The time-slice of the VM_Processor the then decremented to zero to force a thread switch when each VM_Processor's active thread reaches its next yield check point. Idle processor contexts are restarted by polling for tokens and using calls to THB/THJ to branch a call to Thread.yield() to the remote context in order to context switch into the GC thread (details of the branching operation are discussed in next section).

The last scheme is the idle/branch thread scheduling (shown in Figure 3.3 (c)). There are two threads in *iq*: *Idle Thread* (VM_IdleThread) and *Branch Thread* (VM_BranchThread). When there is no Java thread in *tq* or *rq*, no triggered GC event and no Java thread woken up from *wq*, the threads in *iq* can be scheduled. The first scheduled thread is VM_IdleThread; it just checks again the *rq*, *tq* and *wq* to make sure there is no work in the current physical context (see the code listed below).

If there is no work, the VM_IdleThread itself yields and sets the VM_BranchThread as the active thread. The VM_BranchThread will trigger the hardware to freeze and release the token (discussed in the next section). When the VM_BranchThread is shipped a task from remote processor context, it will act as a normal Java thread and be transferred through *tq*, *rq* or *wq*.

3.2.2 Runtime Support for Light Weight Threads

A special type of thread, VM_BranchThread (referred to as a branch thread), is used to enable distribution of light weight tasks to idle processor contexts using the THB/THJ instructions. The same thread also enables scheduling of Java threads and GC threads on idle contexts, once again using the THB/THJ instructions.

When a THB/THJ is used to restart execution in an idle processor context it executes code at PC supplied as argument to the THB/THJ call using arguments transferred from the caller's output window to the idle processor context's input window. A prologue associated with the target code ensures that execution uses the branch thread stack and that attempts by the shipped code to reference the active thread or current processor context resolve to the branch thread and the processor context's associated VM_Processor. So, a branch thread provides a 'ready' Java thread context which is capable of executing a code segment with almost no setup overhead other than the costs of locating a token and shipping a target PC/register window across the bus.

Branch threads operate in a similar way to an idle thread but they use the token hardware to suspend an idle processor context and release a token. Branch threads are created at startup, one for each processor context. A branch thread is scheduled in the run queue of each processor context and an interrupt is delivered causing the processor context to enter the branch thread start method. This method evicts all resident register windows and removes all frames in its stack so that it is running with a single stack frame/window. If it finds that there are threads in tq/rq, it yields into the iq (at startup, this only happens on the processor context which is bootstrapping the runtime). If tq/rq are empty then the processor context has no work to do, so the branch thread exits from the processor context's bottom frame and releases an idle token which can be used to re-enter it via a THB/THJ call (see Figure 3.5).

After a branch thread has executed code shipped by a THB/THJ, an epilogue associated with the target code checks the tq/rq for active threads (see Figure 3.4). This may happen because the shipped code scheduled a thread (e.g. as a side effect of unlocking a VM_Lock or, by explicitly as part of Java thread distribution). While there are threads in these queues, the branch thread schedules

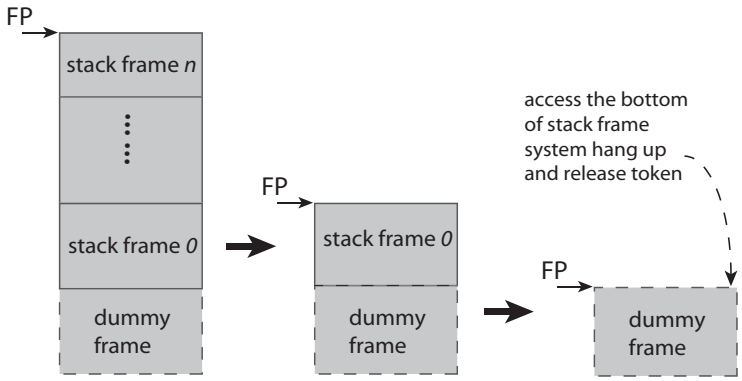


Figure 3.5: The Token Release Process.

them, yielding into *iq*. When it resumes and finds the *tq*/*rq* empty, it exits, releasing another token.

Java's thread scheduling is achieved simply by branching a call to the scheduler method: VM_Scheduler.addToTransferQueue() (see Figure 3.6) with a Java thread as argument. The branch thread calls this code which inserts the shipped Java thread into the restarted processor context's VM_Processor queue. On return, the branch thread yields and runs the Java thread. If the Java thread exits or suspends into, say, a lock queue and no other Java threads are found, the branch thread is rescheduled from the *iq* and releases another token.

```
public static void addToTransferQueue(VM_Thread t) {
  getCurrentProcessor().transferQueue.enqueue(t);
}
```

Figure 3.6: The Code for Branching Thread.

GC thread scheduling is equally as simple. The GC initiator branches a call to VM_Synchronization.yieldToGC(), an empty method with no arguments. After this call returns, the branch thread finds the GC thread in its *tq* and context switches into it, yielding into *iq*.

Note that a branch thread may sometimes suspend in a wait queue when executing branched code, because, say, it needs to wait on a lock. When the thread is rescheduled, it must be returned to the processor context it is tied to rather than

shipping it to an arbitrary idle processor context or scheduling it locally. Branch threads, like GC threads, are tagged with a processor affinity. Any attempt to schedule a thread with an affinity is resolved by inserting the thread into the relevant VM_Processor's *tq*. Note that the processor context cannot be idle when scheduling a branch thread in this way because only the branch thread can freeze it. So, the branch thread will at some point be rescheduled on its own processor context.

3.2.3 Explicit Parallelization and Implicit Parallelization

JaVM provides two types of parallelization support: *Explicit Parallelization* and *Implicit Parallelization*.

To support explicit parallelization, JaVM provides a series of magic APIs which can be explicitly called from Java programs to create, distribute and synchronize parallel threads, including the fork/join and barrier synchronization operations. The task will be shipped to a free thread context, if there is a free context identified by a token. If there is no free context (the thread creator failed to grab a token), the task will be executed as a normal method call on the current processor context. Also, the number of input parameters is limited to 6, because of the limitation on the number of output window registers.

Implicit parallelization is performed by automatic parallelization compilation. JaVM's dynamic compilation system exploits loop level automatic parallelization (to be discussed in Chapter 4). The optimizing compiler analyzes a hot method to identify suitable loops and reconstructs the method to split the parallelizable code segment and insert the thread creation code.

3.3 Dynamic Compilation System

Although a prototype interpreter exists, the JaVM is currently built around two compilers: the baseline compiler and the optimizing compiler.

The baseline compiler does simple translation from Java bytecode to JAMAICA

machine code by simulating Java's operand stack. No register allocation is performed. The performance of compiled code is only slightly better than bytecode interpretation. But the speed of compilation is fast.

The optimizing compiler maintains an intermediate form which starts with instructions resembling Java bytecode but ends in the eventual machine code. All the intermediate form instructions operate on operands, typically constant values or register values. In the high-level phases, where instructions are close to Java bytecode, the number of registers is unlimited. Register allocation at the transition from low-level to machine specific intermediate representation removes this property and later optimisation phases are responsible to uphold the allocation. The main phases operate on high, low and machine level intermediate representation - HIR, LIR and MIR respectively (shown in Figure 3.7).

Each level IR code has its corresponding optimization phases.

- The front-end related optimizations and some machine independent optimizations (on-the-fly optimizations) are performed at HIR level. e.g. copy propagation, constant propagation, dead-code elimination.
- The machine independent and data structure dependent optimizations are performed at LIR level. The optimizations may be the same as are done on the HIR, but introduce the issues related to the JikesRVM runtime and object layout.
- The machine dependent optimizations are performed at MIR level. eg. register allocation, instruction scheduling.

The whole architecture of the optimizing compiler is shown in Figure 3.7.

The baseline compiler is responsible for the Just-In-Time (JIT) compilation of methods. That is, when a method is executed for the first time, the baseline compiler generates the code to be executed, and the dynamic linking system then executes the generated code. By performing runtime profiling, the adaptive optimization system (AOS), described in the next section, is responsible for improving the performance of frequently executed sections of code. The AOS will run the optimizing compiler, which can be tuned for what optimizations to run.

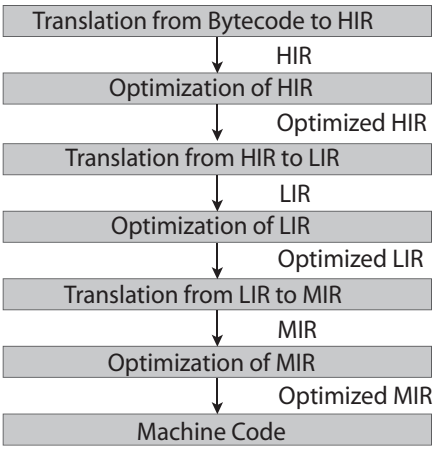


Figure 3.7: Structure of Optimizing Compiler.

To enable thread switching and capture information about the running code, the dynamic compiler (both the baseline and the optimizing compiler) inserts instrumentation code at points such as loop back edges, method entry and method exit. These small code stubs are in charge of checking thread status, and doing runtime profiling.

3.4 Adaptive Optimization System

When the code is executed more than a threshold number of times, a hot method event is generated and queued in the AOS system [9]. The AOS system uses the runtime profiling information to make a decision as to whether the compilation thread should recompile the code with the optimizing compiler (or increase the optimizations used by the optimizing compiler) or whether the associated overhead means the compilation would slow the overall performance of the system. This decision is based on assumptions of how long the method will take to compile (a summary of bytecodes contained in methods is made on class loading), the past execution time for this method and the expected speedup. The adaptive optimizations include: adaptive recompilation with higher optimization level, adaptive inlining and adaptive parallelization, which will be discussed in the next chapter.

The AOS employs a group of associated threads, known as *Listeners*/*Organizers*,

that are stored in adaptive system thread queues and can be awoken by the profiling code. These threads perform runtime analysis of the profiling data to make the decision to select adaptive optimizations. The adaptive recompilation is performed in a separate compilation thread. When AOS identifies a hot method and figures out how to optimize it, it will send a recompilation message to the compilation thread. The recompilation message includes the hot method object and the compilation plan which identifies how to optimize the hot method.

3.5 Phase Detection

JaVM is capable of executing multi-threaded Java program and multiple Java programs. As these Java threads work together and the thread scheduler schedules Java threads to idle processor contexts to maximise the hardware utilization, the number of free processor contexts will vary at runtime. To help the AOS, parallel programs need to be aware how many free processor contexts are available in the current system and decide how to distribute the work load. JaVM employs a simple phase detector which can count the number of free processor contexts in a short interval[3]. This module is embedded in JaVM's thread scheduler.

Figure 3.8 (a) shows the basic mechanism for the phase detection. The JAMAICA CMP employs tokens to identify idle processor contexts (see Section 2.1.3). Each processor has its own token ID which is an integer starting from 1 to the number of processor contexts. The phase detector allocates a global array whose elements correspond to each of the processor contexts' token IDs, and plants interceptor code to the procedures for thread branch and token release (see Section 3.2). In the thread branch procedure, the interceptor sets token flag to 1 which marks the token's corresponding processor context as busy. In the token release procedure, the interceptor resets the token flag to 0.

Shown in Figure 3.8 (b), the two interceptors also count the non-zero token flags to get the number of free processor contexts, and record this number and the current execution cycle count into two sampling arrays. The phase detector calculates statistics based on these values in a configurable interval based on the number

[3]The "Phase" defined here is a runtime stage within which the number of free processor contexts is stable.

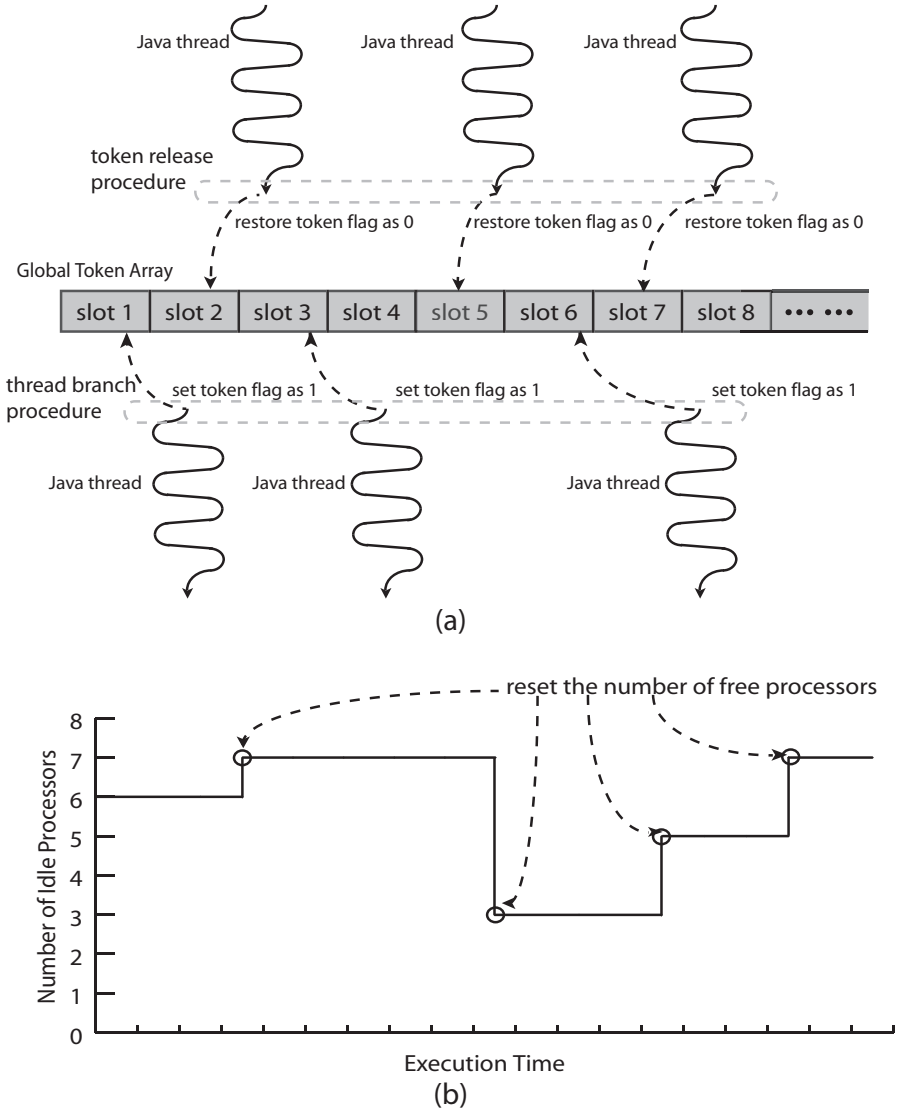


Figure 3.8: The Online Phase Detection Mechanism in OTF.

of recorded operations (15 recording operations are used for an interval in this work). Such statistical data will be used to help the runtime profiler recognize noise and an unstable runtime environment (to be discussed in Chapter 5).

3.6 JAMAICA Boot Procedure

As JAMAICA is a simulated architecture, the JaVM is booted via the *jamsim* simulator (see Section 2.2). The JaVM implements a cold start protocol whereby only a single context, the primordial context, begins the execution of code, all other contexts start in an idle state. Prior to execution, the simulation environment loads an ELF binary, containing a boot procedure, into physical memory, placing code and data segments at addresses specified by the ELF file. The start address is extracted from the ELF file and is used as the initial PC value for the primordial context. The code contained in the boot procedure[4] is responsible for initializing registers and memory, including the initialisation of interrupt vectors and loading any other required code into physical memory. After this initial phase all auxiliary contexts can be woken using a software interrupt, SIRQ. The software interrupt vectors execution to an initial wake-up routine that sets up a minimal stack for each context capable of handling code shipped via the THJ/THB instructions. Upon completion of this phase each context releases a token onto the work distribution ring and switches to the idle state awaiting incoming work.

3.7 Summary

This chapter introduced the software operating platform JaVM. JaVM utilizes the JAMAICA CMP infrastructure to enable a lightweight thread mechanism which makes thread level parallelization more efficient. It also briefly introduced the dynamic compilation system and adaptive optimization system which perform the normal Java JIT compilation and optimization. Finally, this chapter introduced

[4]The booting code is composed of several subroutines which are written in C and JAMAICA assembler.

the booting procedure of JaVM and explained how it can boot on the JAMAICA CMP.

Chapter 4

Parallel Compiler

As introduced in the previous two chapters, the JAMAICA architecture provides an efficient infrastructure for lightweight thread execution. The next step is to exploit this advantage to parallelize Java programs. This chapter discusses the design and implementation of the parallel compiler in JaVM. This dynamic compilation system performs automatic loop-level parallelization (LLP), so it is called the *Loop-Level Parallel Compiler* (LPC) here.

The input program is a normal sequential application (Java bytecode); the LPC analyzes the program, identifies suitable loops that can be parallelized and reconstructs the original program to enable the parallel execution.

In this chapter, Section 4.1 introduces the design issues about the LPC, including the loop analysis, parallel code generation, synchronization, memory allocation, loop distribution and exception handling and parallel code generation. Section 4.2 discusses the cost model for dynamic compilation. Section 4.3 discusses some experimental results. Section 4.4 summarizes this chapter.

4.1 Design Issues

At first, all Java applications running in JaVM are built by the baseline compiler, the AOS captures runtime information by instrumenting the running code at the

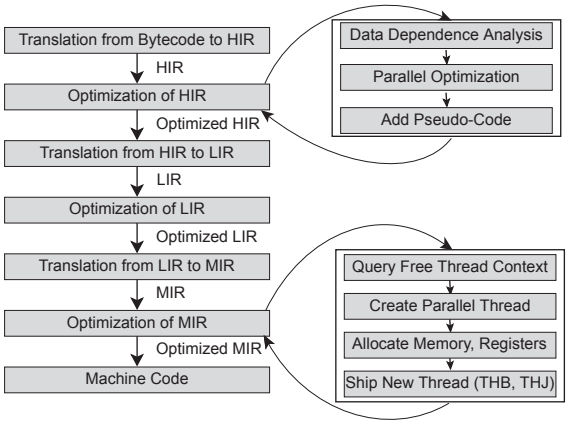


Figure 4.1: Parallel Compiler.

method-level[1]. Once the instrumentation indicates that a given method is *hot* (i.e. the number of times the method is executed is above a threshold), the AOS decides whether to compile it using the optimizing compiler[17] (shown in Figure 3.1). The LPC hijacks this decision, so that any hot method is also considered for parallelization.

To implement the LLP, several issues need to be considered. The following sections will discuss how to perform loop analysis, how to generate thread *Fork* and *Join* operations, how to synchronize the thread and reconstruct the compiled method.

4.1.1 Loop Analysis

Shown in Figure 4.1, the LPC works within two phases of the JaVM optimizing compiler's workflow. Loop analysis and annotation occurs in the high-level optimization phase. In this phase the LPC detects loop structures, analyses the data dependencies within them, creates parallel loops where these dependencies can be maintained, and annotates the loops with high-level pseudo-code. The loop annotation process does pre-order depth-first traversal of the loop tree[2]. If the outer loop is parallelizable, then it is annotated and the inner loop will not be

[1]The code stubs for runtime sampling are inserted at a method's prologue and epilogue and at a loop's backedge (as mentioned in Chapter 3).

[2]The nested loops can be expressed as a tree structure.

checked (see Algorithm 1).

To optimize the loops and make them easier for analysis, several JikesRVM's high-level optimizations are employed by LPC for loop preprocessing, including constant propagation, loop constant code motion, global common sub expression elimination and extended array single static assignment form (array SSA) [14, 52] [3].

LPC can only analyze a method only if this method has been identified by AOS at runtime as being hot. There is a risk that the LPC can not parallelize a loop in a method where a call exists that may cause a dependence violation. A lazy inter-procedural analysis scheme [95] was implemented to enable the early and conditional parallelization of loops, before the full program state is known.

Data Dependence Analysis

In order to determine whether array accesses within the loops are amenable to parallelization, the Banerjee Test [12] is performed (see Algorithm 1). This allows *DoAll* and *DoAcross* [92, 7] loops to be created when presented with loop carried dependencies on arrays with affine indices. The algorithms 1 2 and 3 show the basic mechanism that the LPC uses to check the loop hierarchy and identify the suitable loops that can be parallelized.

The loop checking progress works from the outer-most level to the inner-most level (shown in Algorithm 1). For each loop, the LPC will check if there is loop carried register dependency, or normal memory dependency (shown in Algorithm 2). If the loop carried register operands are simply referred by scalar operations, then the loop still can be parallelized. To check the array dependency, the classic GCD algorithm [26] is employed (shown in Algorithm 3).

Loop Annotation

The LPC employs a series of pseudo instructions to annotate the target loops which are suitable for parallelization. These instructions should annotate the

[3]The SSA form is helpful for building the use-def chain and performing data dependence analysis.

Input: the tree structure of nested loops $Loop\_Tree$
Output: annotated loops that are suitable for parallelization
Implementation:
Step 1: get the root node $t \leftarrow GetRoot(Loop\_Tree)$;
step 2:
$loopType \leftarrow Loop\_Check(t)$;
if $loopType = DoAll$ **then**
  $Annotate(t)$;
else if $loopType = DoAcross$ **then**
  $Annotate(t)$;
else
  $Leaf\_Set \leftarrow GetLeafNodes(t)$;
  foreach *element* $l \in Leaf\_Set$ **do**
    $Loop\_Search(l)$;
  end
end if

Algorithm 1: Loop_Search (Search Parallelizable Loops).

fork/join points of a parallelized loop and its loop constants. Table 4.1.1 lists the instructions and their functions.

Instruction	Function
JAM_FORK	Annotate the fork point of the parallelized loop
JAM_JOIN	Annotate the join point of the parallelized loop
JAM_LOOPCONSTANT	Annotate the short/integer/float type of loop constant
JAM_LOOPCONSTANTLD	Annotate the double/long type of loop constant
JAM_COLLECT	Annotate the loop initial/terminal values

The LPC checks the code in a loop body and identifies the loop constants. All of the loop constants and iteration space values (initial iterator value and terminal iterator value) will be annotated by pseudo functions and replaced by these function result operands. Figure 4.2 shows the annotation process for a simple *DoAll* loop. x and y are loop constants, they are replaced with $\_x$ and $\_y$ which are the result operands for pseudo functions. The iteration space values 0 and n are also replaced by *init* and *term*. The benefit of this annotation approach is that the annotated code still can be optimized by the following compiler optimization passes, until it reaches the thread creation phase at MIR level.

Input: the loop node t
Output: loop type (this is an enumerated value: *DoAll*, *DoAcross*, *Dependence*)
Implementation:
Step 1:
collect the loop carried register operands into LC_Set;
collect class field store operations into CF_Set;
collect the array load/store operations into $Load\_Set/Store\_Set$;
$ADep\_Set \leftarrow \{\}$;
Step 2:
foreach *element* $s \in Store\_Set$ **do**
 foreach *element* $l \in Load\_Set$ **do**
  if $ArrayRef(s) = ArrayRef(l)$ **then**
   $GCD\_Check(s, l, t, ADep\_Set)$;
  end if
 end
end
Step 3:
foreach *element* $lc \in LC\_Set$ **do**
 if lc is not simply referred by a scalar operation **then**
  $isLCDependency \leftarrow true$;
 end if
end
Step 4:
if $CF\_Set \neq \{\}$ **then**
  $isCFDependency \leftarrow true$;
end if
if $ADep\_Set \neq \{\}$ **then**
  return *Dependence*;
end if
if $isLCDependency$ **then**
  return *Dependence*;
end if
if $isCFDependency$ **then**
  return *Dependence*;
end if
if $LC\_Set \neq \{\}$ **then**
  return *DoAcross*;
else
  return *DoAll*;
end if

Algorithm 2: Loop_Check (DOALL/DOACROSS Loop Check).

Input: a load operation l, a store operation s, loop node t, and Dep_Set a set which is used to store the dependence pair
Output: boolean result
Implementation:
Step 1: Analyze the array index for both l and s;
Step 2: Build the dependence equations (coefficient matrix);
Step 3: Apply general GCD test to equations and get a set of general GCD solutions: GS_Set;
if $GS\_Set = \{\}$ **then**
  return $false$;
end if
Step 4:
apply GS_Set to t's loop iteration space to get a set of integer solutions: IS_Set;
if $IS\_Set = \{\}$ **then**
  return $false$;
end if
Use the integer solutions to calculate the dependence distance vector set: DV_Set;
foreach *element* $d \in DV\_Set$ **do**
 $Dep\_Set \leftarrow Dep\_Set \cup (l, s, d)$;
end
return $true$;

Algorithm 3: GCD_Check (Check Array Dependency).

```
// Orignal loop code
double x = ...;
int y = ...;
for (int i = 0; i < n; i ++) {
  A[i] = x * (B[i - 1] + B[i]);
  C[i + 1] = y + 1;
}

// Annotated loop code
double x = ...;
int y = ...;

int init = jam_fork(0, n);
int _y = jam_loopconstant(y);
double _x = jam_loopconstantld(x);
double[] _A = jam_loopconstant(A);
double[] _B = jam_loopconstant(B);
int[] _C = jam_loopconstant(C);
int term = jam_collect(n);
for (int i = init; i < term; i ++) {
  _A[i] = _x * (_B[i - 1] + _B[i]);
  _C[i + 1] = _y + 1;
}
jam_join();
```

Figure 4.2: Example of Loop Annotation.

```
// Orignal code
public void foo(int[] A, int[] B, int c) {
  for (int i = 0; i < n; i ++) {
    A[i] = B[i - 2] + c;
  }
}

// Transferred code by adding alias check
public void par_foo(int[] A, int[] B, int c) {
  if (A == B) {
    for (int i = 0; i < n; i ++) {
      A[i] = B[i - 2] + c;
    }
  } else {
    int init = jam_fork(0, n);
    ... ...
    for (int i = init; i < term; i ++) {
      _A[i] = _B[i - 2] + _c;
    }
    jam_join();
  }
}
```

Figure 4.3: Alias Analysis for Single-Dimensional Array.

Alias Analysis

For data dependence analysis, the compiler needs to know if two array references are the same or different. For example, Figure 4.3 (a) shows a simple loop which contains two array references A and B. If A and B are same, then this loop has dependence distance 2 and is not parallelizable. As both of these two array references are input parameters of the method, it needs complex inter-procedural alias analysis to identify if these two reference are the same or not. Currently, LPC does not support inter-procedural alias analysis, it employs another straightforward way to solve this alias problem: adding a comparison statement which can decide if the execution path should go through the parallelized or non-parallelized version of the loop (shown in Figure 4.3 (b)).

This approach is feasible for single dimensional array references, but not for multi-dimensional arrays, because the multi-dimensional array in Java is defined as multiple sub-arrays. Given a two dimensional array A, its sub-arrays may be allocated

in different regions, and have various lengths (shown in Figure 4.4 (a)). Some research on Java numerical libraries [67] proposed a uniform multi-dimensional array class which has the same memory layout as either C/C++ or Fortran. Similarly the X10 language [20] extends Java with true multi-dimensional array support. To enable real multi-dimensional arrays and to simplify the alias analysis, JaVM uses an annotation policy. If a class or method is annotated with $RealMultiDim$, the LPC will check the code inside the annotated scope. There are two constraints inside the annotated scope.

- The memory for the creation of multi-dimensional array objects is allocated in a linear memory space (shown in Figure 4.4 (b)), so the memory layout is the same as C/C++.
- The assignment operation for assigning an array object to another object array is prohibited [4]

To ensure the GC operation will not destroy the memory layout, JaVM uses the $MarkSweep$ [47] GC policy which maintains the original memory address for each object.

Figure 4.5 shows the loop parallelization within an annotated method scope. If the LPC can not guarantee that the array A is a real multi-dimensional array (i.e. there is not any interference between array A and B), it needs to generate the code for comparing each of A's level 1 array elements $A[i]$ with B to make sure there is no interference, and this will result in unacceptable overhead at runtime. The benefit of using the annotation policy is a simple programming model, the application programmers does not need to be aware about which array should be real multi-dimensional and which not. The disadvantage is the definition of the annotation scope needs to be handled carefully. The annotation is a mechanism for protecting the real multi-dimensional array, so the annotation scope should cover all of the methods or classes that may assign array objects to the multi-dimensional array.

[4]Currently, if LPC identifies an assignment operation which assigns an array reference to an element in an object array, it will throw a runtime exception.

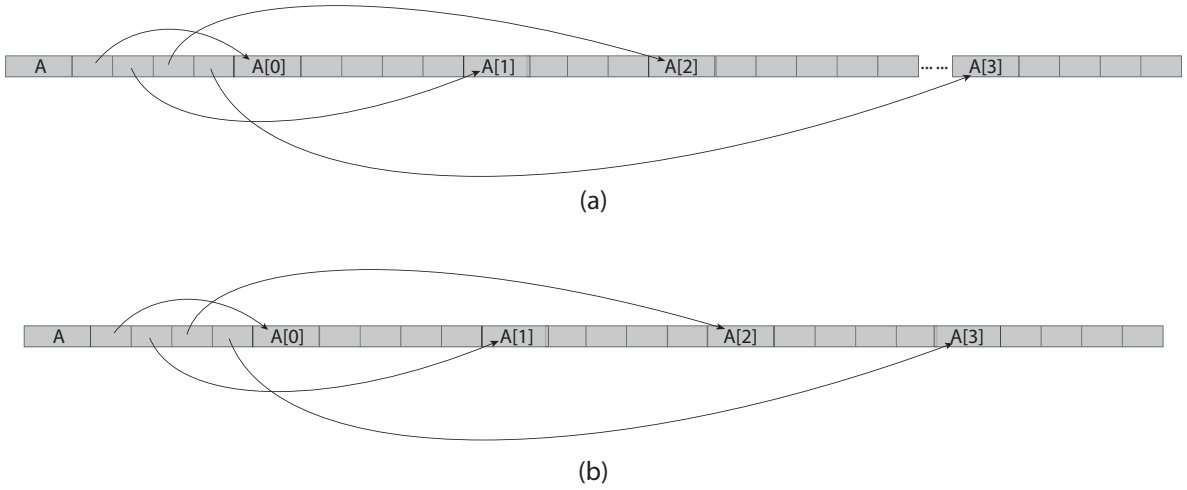


Figure 4.4: The Java Multi-Dimensional Array and Real Multi-Dimensional Array.

```
// Orignal code
public void foo(int[][] A, int[] B, int c) throws RealMultiDim {
  for (int i = 0; i < n; i ++) {
    A[i][i] = B[i - 2] + c;
  }
}

// Transferred code by adding alias check
public void par_foo(int[][] A, int[] B, int c) throws RealMultiDim {
  if (A[0] > B || A[A.length - 1] < B) {
    for (int i = 0; i < n; i ++) {
      A[i][i] = B[i - 2] + c;
    }
  } else {
    int init = jam_fork(0, n);
    ... ...
    for (int i = init; i < term; i ++) {
      _A[i][i] = _B[i - 2] + _c;
    }
    jam_join();
  }
}
```

Figure 4.5: Alias Analysis for Multi-Dimensional Array.

4.1.2 Parallel Code Generation

Parallel code generation occurs in the machine-level optimization phase. It works before the register allocation phase, so it can still use logical registers. In this phase the previously inserted pseudo-code is replaced by machine specific code, enabling the code to fork new threads on idle processor contexts, as well as applying different adaptively optimizing distribution policies. The parallelized loop will be reorganized, and the loop body acts as the main body of a parallel thread.

Three pieces of code are generated in this phase: parallel thread creation (fork point), the parallel thread and a barrier synchronization (join point).

Parallel Thread Creation

The creation of parallel thread has three steps:

1. Load runtime parameters from AOS database (to be discussed in Chapter 5), for example the chunk size [5] and tile size.
2. Allocate a memory region on stack, and store the loop constants into this region.
3. Enter loop distributor to create parallel threads.

The loop distributor encapsulates the distribution policy for parallel threads; the distribution policies will be discussed in Section 4.1.4. Here is a basic work flow for a simple fixed-number based distribution:

- Step 1: Use TRQ instruction to request token on the ring. If a token is received, goto step 2; if not, goto step 6.
- Step 2: Set the synchronize state (the detail will be discussed in Section 4.1.3).

[5] The term *chunk* is used to mean a contiguous sequence of loop iterations.

- Step 3: Set output window registers: including chunk or tile size parameters, and synchronization address[6].
- Step 4: Use THJ or THB to ship the task to the processor context indicated by the token ID.
- Step 5: Recalculate the initial iterator value and terminal iterator value. Check if the loop should terminate, if so, goto join point; if not, goto step 1.
- Step 6: Execute the loop chunk in the local processor context. Recalculate the initial iterator value. Check if the loop should terminate, if so, goto join point; if not, goto step 1.

Parallel Thread

The parallel thread is a simple lightweight thread which encapsulates the parallelized loop body. The parallel thread is composed of three parts: prologue, thread body, and epilogue.

The thread prologue does such tasks as:

1. Reset and recalculate the frame pointer register %x4.
2. Reset the Stack Pointer register %o6 and save the previous Frame Pointer and current method ID on the stack.
3. Load input parameters from input window registers and loop constants from memory.
4. Initialize the constant iteration value that will be used in the thread body.

The thread body is a copy of the parallelized loop. The replicated loop's initial iterator value, terminal iterator value and stride value can be reconfigured by the values set in the thread prologue.

The thread epilogue does such tasks as:

[6]For different distribution policies, the output window registers will be assigned different values. For example, for chunk based distribution, the registers %o1, %o2 are used to store the initial and terminal values for a chunk; for tile based distribution, the registers %o1, %o2, %o3, %o4 are used to store two initial iterator values and two terminal iterator values.

1. Store the scalar values into the thread local storage, if the parallelized loop has scalar operations.

2. Reset the barrier synchronization field.

3. Restore the frame pointer and stack pointer registers.

4.1.3 Barrier Synchronization

In JaVM, a simple barrier synchronization mechanism is employed to perform the parallel thread joining operation (shown in Figure 4.6). Each VM_Thread object has an integer array which is used to annotate the synchronization state for shipped tasks (parallel threads). Each array element is a synchronization flag which corresponds to a processor context and indexed by the token ID [7], so if there are 8 processor contexts in the system, the length of synchronization array is 8. The main thread and the branch threads that are created by the main thread for parallel computing use these synchronization flags to perform barrier synchronization.

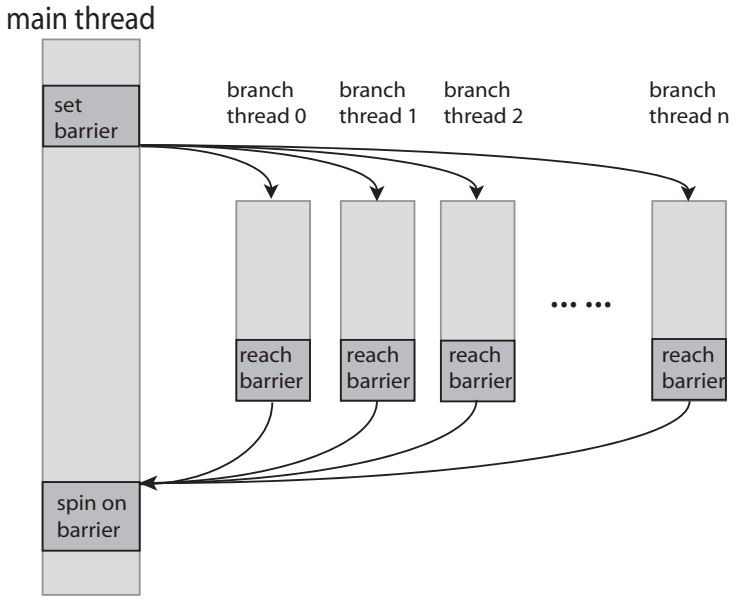


Figure 4.6: Branch Thread Synchronization.

[7]When the JaVM is booted, each processor context is assigned a context ID. This ID will be the token ID, when the processor context produces a token on the ring.

To help the branch threads and main thread that creates branch threads share the synchronization flags, a thread ID mechanism is employed by JaVM. JaVM uses a *Global Thread Array* (GTA) to store all of the VM_Thread objects that are available in the system. When a Java thread is created, it will allocate a free slot in the GTA and register itself on it. Every Java thread object has a corresponding VM_Thread object in JaVM. A global thread ID which is actually the index in the GTA is issued to the Java thread at the same time. When a Java thread will exit, it needs to deregister itself and free the slot in the GTA. The ID got from thread registration should be set into a hardware context register named *ThreadID Context Register* in the processor context on which the Java thread is activated every time. The value of the *ThreadID context register* can be transferred to another processor context by THB/THJ instructions (mentioned in Chapter 2). The transferred value is stored in another context register: *SrcID Context Register*. So the branch threads can use this *SrcID* to get its creator thread object in the GTA, and use the current processor context's token ID to index the synchronization flag in the creator thread's synchronization array.

When a Java thread creates a branch thread on an idle processor context, it sets the branch thread's corresponding synchronization flag field to 1. The synchronization flag is one of the elements in the synchronization array and indexed by the token ID which is produced by the idle processor context and grabbed by the TRQ instruction. The branch thread should set the synchronization flag back to 0 when it finishes the shipped job. It uses the SrcID to get its creator thread object's synchronization array and uses its processor ID (token ID) to identify the synchronization flag.

To perform the joining operation, the main thread should block on its synchronization state array and check each array element, until all of the elements in the synchronization state array are 0 (i.e. all of the created branch threads have finished).

4.1.4 Loop Distribution Policies

To distribute the loop iterations to parallel processor contexts, five different distribution policies are implemented within the LPC.

Fixed Number Based Distribution (FBD)

In FBD, if an idle context is available, a fixed number of loop iterations are executed by it. The number of iterations is configured by the adaptive system.

Chunk Based Distribution (CHBD)

In chunk distribution [60], N loop iterations are divided into $\frac{N}{P}$ rounds (P is the number of processors). Each round consists of consecutive iterations and is assigned to one processor context.

Tile Based Distribution (TBD)

This is an improvement on CHBD. For a 2 level perfectly nested loop, N loop iterations are divided into P tiles and the tiles will be assigned to P different processors. Currently, the LPC only supports 2-dimensional tiles.

Cyclic Based Distribution (CYBD)

Instead of assigning to a processor a consecutive chunk of loop iterations, the iteration are assigned to different processors in a cyclic fashion [60] which means that iteration i is assigned to processor $i \ mod \ P$.

Cyclic Recursive Distribution (CRD)

In CRD, if an idle context is available the total iterations are divided between the parent thread and a created child thread. The parent and child thread then recursively try to divide the work in half again. A disadvantage of this scheme is that it works best when the number of free processor contexts is a power of two.

Dynamic Assignment Distribution (DAD)

DAD policy employs a dynamic work assignment mechanism to distribute loop iteration chunks to parallel threads working on different processor contexts. Algorithms 4 and 5 show the work mechanism for this distribution policy.

Input: loop iteration chunk size c, number of loop iterations N
Implementation:
Step 1: $ParallelThread\_Set \leftarrow \{\}$;
Step 2:
while $N > 0$ **do**
 $t \leftarrow Token\_Request()$;
 if t is a token ID **then**
  create a new parallel thread p;
  $ParallelThread\_Set \leftarrow ParallelThread\_Set \cup p$;
  assign c loop iterations to p;
  $N \leftarrow N - c$;
 end if
 foreach $parallelthreadp \in ParallelThread\_Set$ **do**
  if N = 0 **then**
   goto Step 3;
  end if
  if p is waiting **then**
   assign c loop iterations to p;
   $N \leftarrow N - c$;
  end if
 end
 if $N > 0$ **then**
  execute c loop iteration;
  $N \leftarrow N - c$;
 end if
end
Step 3: set the *stop* flag for all of the threads in $ParallelThread\_Set$ and synchronize them

Algorithm 4: Dynamic Assignment Distribution.

This loop distribution policy is a greedy policy which tries to get as many processor contexts as possible. It is suitable for improving the parallelized loop's performance in an unstable runtime environment where several concurrent Java threads are activated/deactivated frequently and the parallelized code can not get enough computing resource for parallel execution. The drawback of this policy is the communication overhead generated by the producer/consumer execution

Input: loop iteration chunk: $(init, term)$
Implementation:
Step 1: initialize the input parameters;
Step 2:
check the operation flag;
if flag = *working* **then**
  execute the loop iteration chunk defined by $(init, term)$;
  change the main thread's state to *waiting*;
  goto step 2
end if
if flag = *waiting* **then**
  goto step 2
end if
Step 3: set synchronization flag and thread terminates

Algorithm 5: Parallel Thread's Work Mechanism in DAD.

mode.

4.1.5 Memory Allocation

To perform loop-level parallelization, the loop constants should be passed from the main thread to the parallel threads so the split loop body can be initialized correctly. As introduced in chapter 2, the task shipping mechanism (THB and THJ) can only carry the output window registers, PC and one context register to the target context. The limited number of input window registers may not satisfy the requirement of transferring more loop constants, so a shared memory mechanism needs to be applied here to carry more data.

JaVM allocates a spill region on the main thread's stack frame by reshaping the stack frame layout. The size of the region is calculated by the LPC, when it analyzes the parallel loops. The pointer to the allocated memory is transferred to the parallel thread by one of the output window registers (%o0). Then the parallel thread can load the data from the memory indicated by its input window register (%i0) (shown in Figure 4.7). The data that will be different in different parallel threads and the frequently used data (e.g. loop iterator values, chunk size, tile size) still needs to be carried by the input window registers, as it is processor context specific and more efficient. The advantage of this solution is that the memory allocation is performed at compile time by calculating the memory size

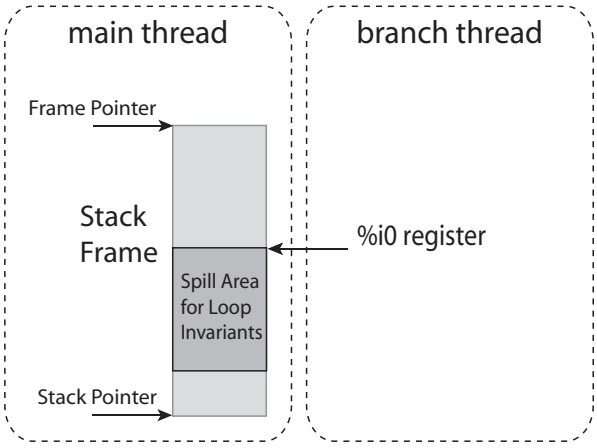


Figure 4.7: Memory Allocation.

and reshaping stack frame layout. There is no need for any additional memory management, so it has zero cost at runtime.

4.1.6 Code Layout

As mentioned in chapter 2, the two task shipping instructions (THB and THJ) have different definitions for the PC values that the branch thread will start to execute. THJ uses the PC value assigned in its input register. THB uses the current PC value plus the offset assigned in its input register. To support these two task shipping mechanisms, LPC provides two code layouts for parallelized programs: method based parallel thread (MBPT) for THJ based parallelization and branch offset based parallel thread (BBPT) for THB based parallelization.

Figure 4.8 shows the reconstructed method by BBPT. Figure 4.9 shows the reconstructed method by MBPT. Here we use the simple FBD loop distribution policy for both of these two examples.

For BBPT, the loop distribution code and the parallel thread code are placed in the same method as their creator. Shown in Figure 4.8, the code which belongs to the parallel thread (replicated loop body, prologue block and epilogue block) are located at the end of the method (actually their position is located between the method's return block and the exit block in the control flow graph) and do

not have any edges connected to the original method's blocks. So these pieces of code will not take part in the register allocation in the original method. Or we can say that all the physical registers are free when the register allocator begins to allocate registers for the branch thread's code, because the original method has reached its RET instruction.

For MBPT, the loop distributor process and the parallel thread code are placed in an implicit method object which means that only the LPC and adaptive system can generate and manipulate it. These methods do not have any return value. The methods for the loop distributor process can be reused by any parallelized loop (shown in Figure 4.9.). These methods can also work as a parallel thread, so one parallelized loop may have multiple loop distributors (thread creators) working in parallel.

The advantage of BBPT is that all the generated code is still located in the same method code space. This saves compilation resource, because the compiler need not create a new compilation process (e.g. register allocation) to handle the newly created sub-method in MBPT. The disadvantage of BBPT is the difficulty in generating multi-version code. For example, to switch between different loop schedulers, MBPT can generate multiple loop scheduler methods. BBPT has to generate the multi-version codes in the same method space, and the control flow graph will be complex. The drawback is that a complex control flow graph will make the operands' liveness analysis more complex and the register allocator has to allocate more spill area for those operands which can not be assigned to physical registers; and this thereby decreases the performance of the generated code.

4.1.7 Handling Exceptions in Parallelized Loop

The Java language uses an exception mechanism to maintain type safe memory accesses, and this will generate some performance problems with parallel loops.

At first, there is some checking code in the loop body that will be parallelized, eg. BOUND_CHECK (array bound check), NULL_CHECK (null reference check), CHECKCAST (check the class type cast). If there are any checking violations, the execution sequence will be branch to the exception handler code that does

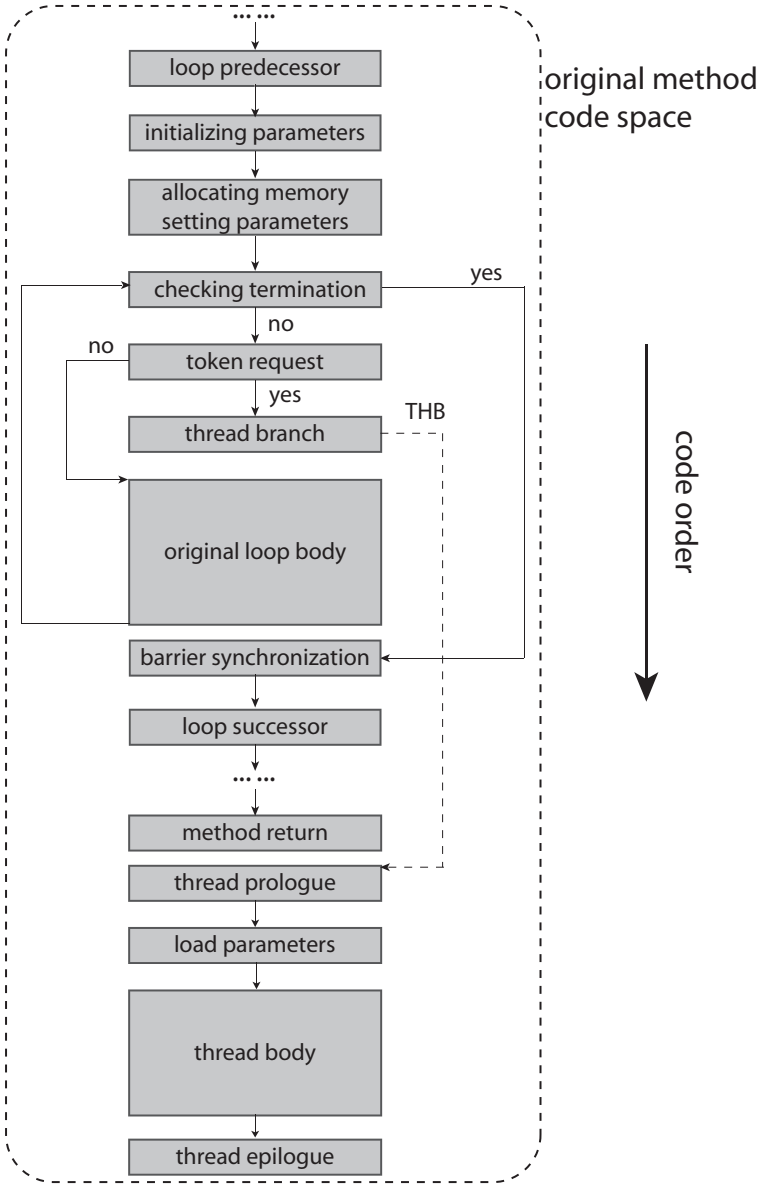


Figure 4.8: Loop Reconstructed by BBPT.

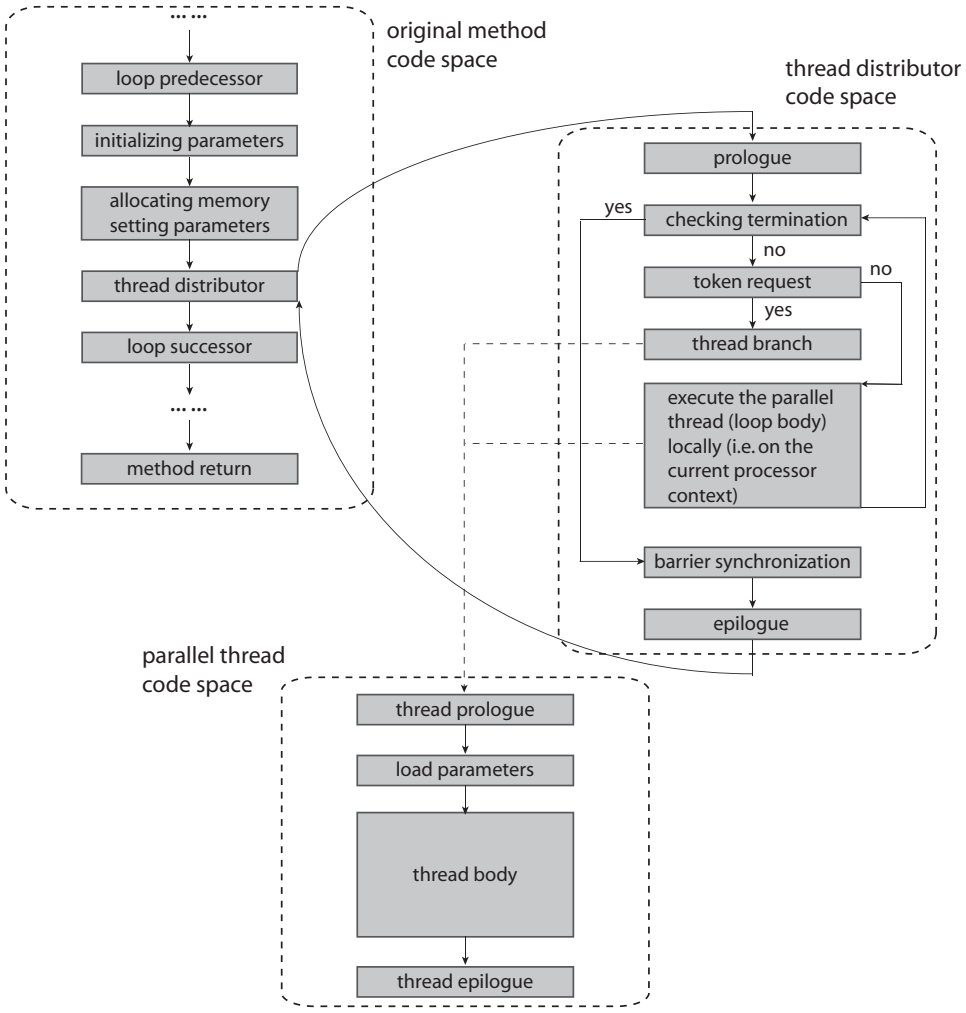


Figure 4.9: Loop Reconstructed by MBPT.

not belong to the loop body. This will increase the complexity of thread synchronization. As shown in Figure 4.6, the main thread uses a barrier to synchronize with all its branch threads. Currently, the branch threads perform thread synchronization at the end of their execution. If one of the branch threads is diverted to the exception handler code, it can not set the synchronization status, and the main thread must be blocked on its barrier.

In Java semantics, if one loop iteration throws an exception at runtime, the following iterations should not be executed anyway. But in a parallelization scenario, several loop iteration are executed simultaneously, so one thread needs to cancel other threads that are running on other processor contexts. The cancellation process is like rolling back a database transaction, the executing thread must be stopped and the memory that had been modified must be restored. It is difficult or quite expensive to implement in software.

The current solution to this problem is array bound check and null check elimination. All of the array references that are used in a loop body are checked, and if the array bound check and null check are redundant (the array length is longer than the loop requirement, and the array reference is not null), then such check operations can be eliminated from the loop body and we can get an execution segment guaranteed not to have these runtime exceptions. This is known as loop versioning, which we contributed back into the JikesRVM in release 2.4.3. This approach still has shortcomings:

1. It can only handle the bound check and null check, there are still some check operations that could generate runtime exceptions.

2. This additional optimization phase will be added to the optimizing compiler and increases the cost of compilation.

4.1.8 Long-Running Loop Promotion

The runtime parallelization activations may contain long-running loops. When the JaVM is executing a baseline compiled long-running loop, the LPC may have finished the loop parallelization. To improve the runtime performance, the LPC needs to perform *On Stack Replacement* (OSR) [19, 42] to install the parallelized

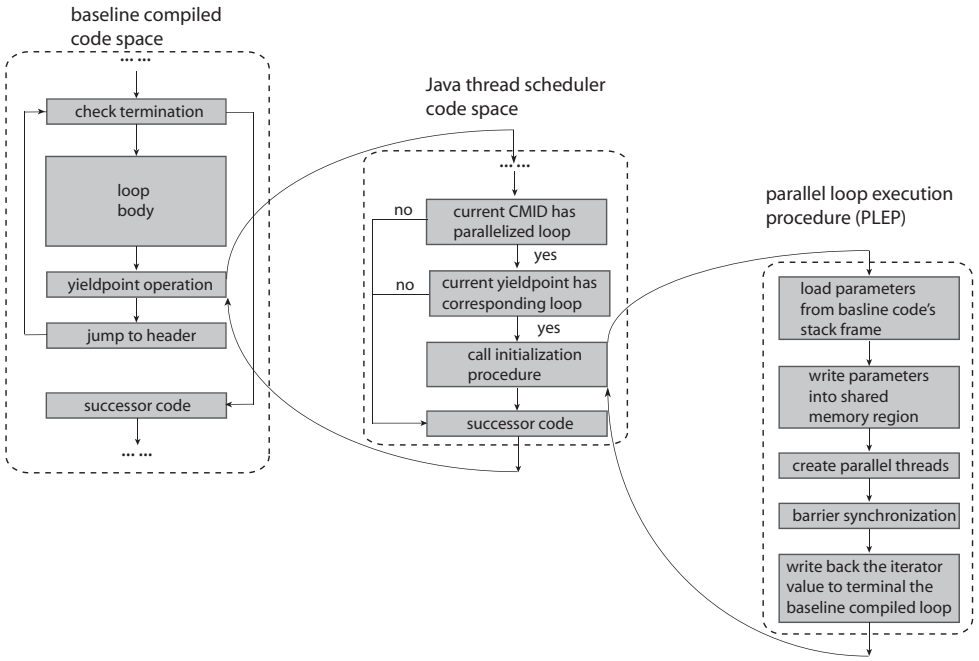


Figure 4.10: The Long Running Loop Promotion.

loop. JikesRVM has implemented OSR support [31], but it works at method level and needs a complex process to handle dynamic class loading. LPC employs a simple policy which can redirect the execution path from baseline compiled loop code to parallelized loop code. This method is based on the MBPT code layout, and the only overhead is generating a *Parallel Loop Execution Procedure* (PLEP) which executes the parallelized loop.

Figure 4.10 demonstrates the workflow of the loop promotion. JaVM inserts yield points into the loop back-edge to perform the thread yield, which collects statistical data for AOS, and triggers garbage collection. Here the loop promotion code is added in the yieldpoint operation. Every time the yield operation is executed, the loop promotion code will check if the current method has a parallelized loop and if the current yield point corresponds to a parallelized loop. If so, the execution path will be redirected to the PLEP. The PLEP goes through the baseline compiled code's stack, picks up the loop constants and writes these values into a shared memory region; then it calls the loop scheduler to create the parallel threads. When the execution of the parallelized loop finishes, the PLEP resets the iteration value on the baseline code's stack, so the baseline loop can finish.

4.1.9 Parallel Compilation

Some of the adaptive runtime optimizations (discussed in the next Chapter) need to consider multiple versions of code, but generating multiple versions of code increases the compilation overhead significantly. As these multiple versions of code are based on the same method, the LPC should be able to build these code versions in parallel by applying different code generation policies to the same IR code. The CMP architecture provides the capacity of parallelization, the LPC's main compilation thread can spawn several sub-compilation threads (depending on how many free processor contexts are available) which use different code generation policies. Shown in Figure 4.11, the creation of sub compilation threads is invoked just before register allocation (in the same place as parallel thread generation). The main compilation thread replicates several copies of MIR code, and the sub compilation thread will use this code to generate machine code with different optimization policies (e.g. unrolling the loop with different unrolling factors).

The replicated MIR code is preserved in a temporary storage (AOS database). When the AOS needs to try more optimizations, it can spawn a compilation thread and reuse the preserved MIR code. The shortcoming for this is more runtime storage is needed for storing this additional information.

4.2 The Cost Model for Dynamic Parallelization

The previous sections have introduced the design issues for LPC. Before evaluating the LPC, this section discusses the cost model for a parallelizable code segment. The cost model is expressed by the equations listed below:

(1.)

$$T_p = T_{par} * N_{par} + T_{unpar}$$

(2.)

$$T_{unpar} = N_{unpar} * T_{baseline} + T_{bc}$$

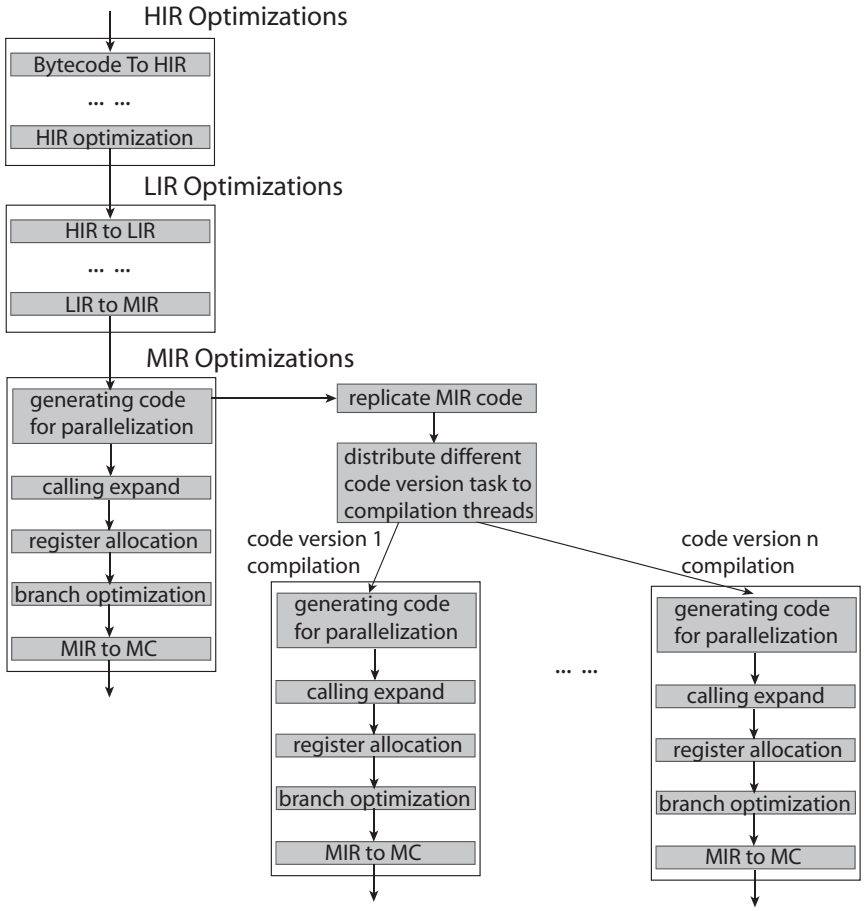


Figure 4.11: Parallel Compilation for Multi-version Code Generation.

(3.)

$$N_{unpar} = (T_{dc} + T_{sample})/T_{baseline}$$

T_{par} : the execution time used for parallelized code segment.

T_p : the total execution time used for executing a parallelizable code segment.

N_{par} : the number of times the parallelized code segment is invoked.

T_{unpar} : the execution time used for unparallelized code segment.

N_{unpar} : the number of times the unparallelized code segment is invoked.

T_{dc} : the execution time used for dynamic compilation (parallelization).

T_{bc} : the execution time used for baseline compilation.

T_{sample} : the execution time for identifying the hot method.

$T_{baseline}$: the execution time for executing the code segment compiled by baseline compiler.

Equation 1 expresses that the whole runtime of a parallelizable code segment is composed of two parts: parallel execution and non-parallel execution. Equation 2 shows that a major part of the non-parallel execution is the baseline code execution. In the current LPC design, the parallelization related compilation components are set at opt level 0 which is higher than baseline compilation. When the baseline compiled code is identified as hot by AOS and recompiled, LPC will parallelize it.

JaVM's recompilation is performed in a separate compilation thread, so the runtime recompilation works in parallel with the program execution in multi-processor environments [8]. So equation 3 defines that N_{unpar} depends on T_{dc} and T_{sample}. When the LPC has parallelized the recompiled code and installed it, the new invocation of the code can work in parallel.

To improve the runtime performance, the LPC needs to reduce T_{unpar}, T_{par} or increase N_{par}. The baseline compiler has been simple enough, because it just map Java bytecode to machine code directly. So the $T_{baseline}$ can not be optimized. AOS needs enough sampling to identify the hot code segment, so T_{sample} can not

[8]This was a contribution back to the JikesRVM release 2.4.6.

be reduced. Thus T_{dc}, N_{par} and T_{par} are the three major determinants of the runtime performance.

LPC has employed the smallest set of compiler phases to perform runtime parallelization efficiently. To perform more optimization on T_{dc}, the dynamic compilation need to be well tuned, but this is not in scope of this work. Increasing N_{par} can get more benefit from parallelization and tradeoff the cost of T_{dc}. Those applications which have a long runtime, they will get more benefit from this runtime parallelization mechanism. Optimizing the parallelized code to reduce T_{par} is the major optimization for improving runtime performance; this issue will be discussed in the next chapter.

4.3 Evaluation and Discussion

4.3.1 Experimental Setup

The experiments are performed on the JAMAICA simulator (described in Chapter 2). Different numbers of processor contexts are evaluated in this chapter. To demonstrate the speedup of parallelization, we did not evaluate the multi-context configuration which can benefit from reducing memory delay but is not good for parallel tasks.

To evaluate the efficiency of the LPC, 11 benchmark programs are selected from SpecJVM, jByteMark[4] and JavaGrande suite [16]. These benchmarks include integer and floating point applications (shown Table 4.1). $JSwim$ and JLB are two independent benchmarks; $JSwim$ is the Java version of 171.$swim$ test in SpecCPU2000 [40] and JLB is a Java implementation of Lattice Boltzmann simulation [23].

4.3.2 Performance Evaluation

Figure 4.12 (a) presents the performance of the JaVM running with a varying number of processors (p) (1 thread context per processor) on the JAMAICA simulator. The distribution policy used here is CHBD. The benchmarks were chosen as they have a mixture of both parallelizable and non-parallelizable code regions.

Benchmark	Description	Type	Loop Count	No. of Parallelized Loops *
IDEATest	IDEA encryption and decryption	Integer	3	2
EMFloat	Emulated floating point set	Integer	3	2
Compress	Compression	Integer	26	2
Linpack	Java Linpack	Float	5	2
Fourier	Fourier coefficients	Float	3	1
NeuralNet	Multi-level Neural Network training	Float	12	9
Euler	Fluid dynamics	Float	12	8
JSwim	Java Swim simulation	Float	6	6
JLB	Java Lattice Boltzmann simulation	Float	2	1
Moldyn	Molecular dynamics	Float	6	3
MPEGAudio	Audio decoder	Float	35	4

* the inner loops are not counted if the outer loops get parallelized

Table 4.1: The Benchmarks Selected for Evaluating LPC.

As LPC has parallelized small sections of code within the larger benchmarks, code that remains serial will mean that performance does not scale with the number of processors. The results show the benefit of the system is a 2% to 300% speed up, depending on the benchmark and the number of processors available. For *Compress*, the speedup is near to zero, because only 2 small loops are parallelized and there is no big effect on the whole runtime performance. For *JSwim* and *JLB*, the parallelized hot loops are the major part of the computation, so the benefit is significant.

Figure 4.12 (b) shows the speedup got from the same run, but includes the cost of dynamic compilation. By counting the cost of compilation, the adaptive parallelization introduces more runtime overhead, which is why *Compress*'s speedup becomes negative.

As introduced in earlier sections, a hot method captured by AOS will be recompiled with a higher optimization phase by JaVM's dynamic compilation system. The optimizations have three levels: *opt*0, *opt*1, *opt*2. Usually, a hot method could be recompiled by *opt*0, and some of them can be recompiled by *opt*1[9] The *opt*0 and *opt*1 levels contain simple compiler optimizations which do not need SSA support. To find the parallelizable loops as early as possible, the LPC works

[9]Current experience is that a hot method will not be recompiled by *opt*2, unless there are a lot of iterations.

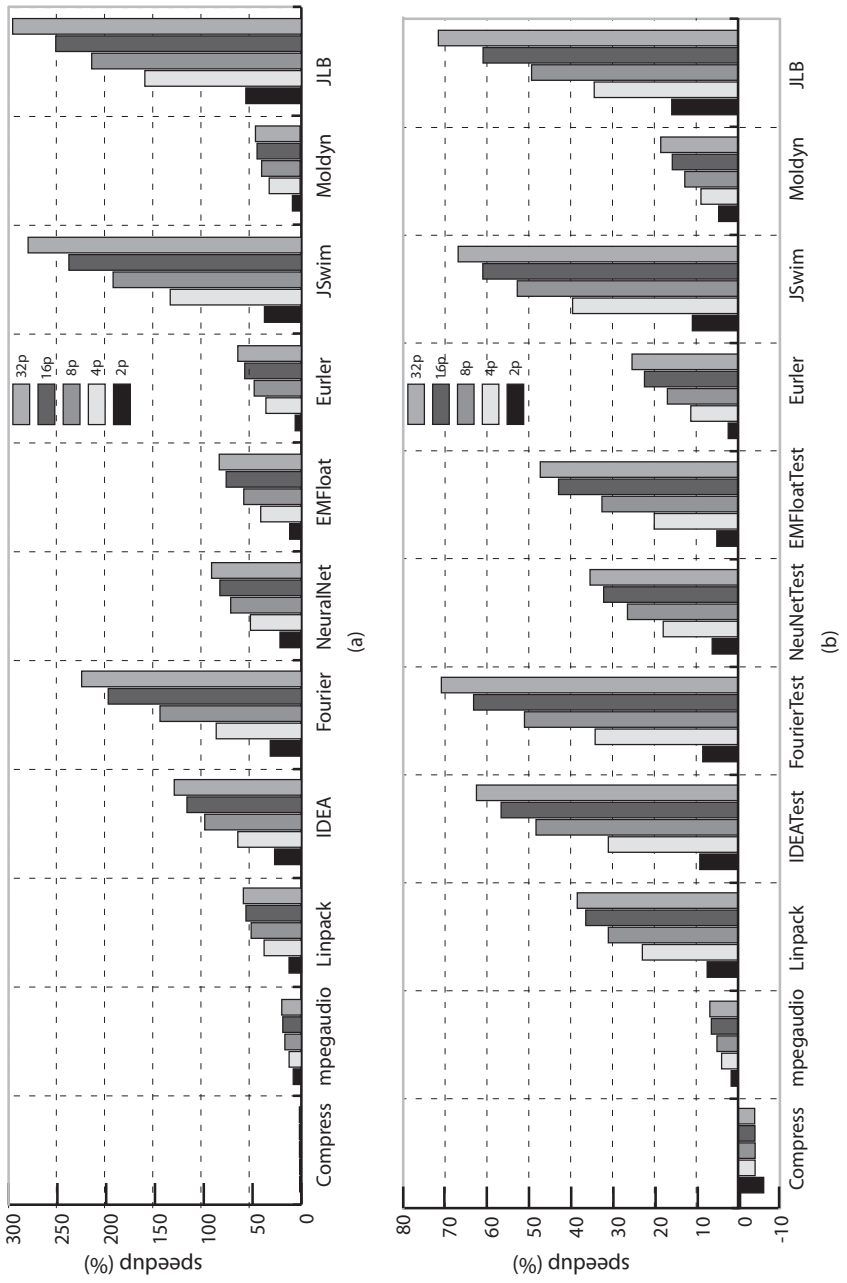


Figure 4.12: Performance of Parallelization on 11 Benchmarks.

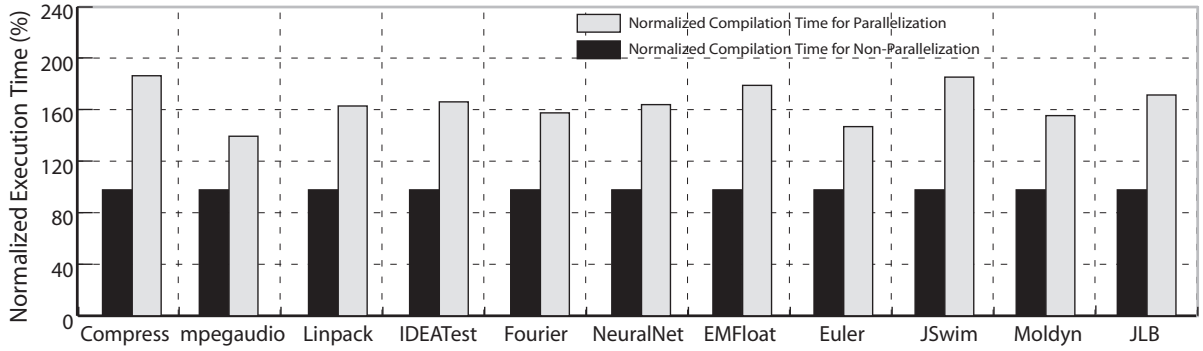


Figure 4.13: The Comparison of The Cost of Parallelization.

at *opt*0. As discussed in Section 4.1.1, the LPC needs additional compilation phase support (e.g. extended array SSA, code motion and constant propagation) to annotate suitable loops for parallelization, so these compilation phases need complex compiler phases (SSA support) and generate additional runtime overhead. So the value of T_{dc} is increased. Another risk is that the LPC may not find a parallelizable loop in the hot method, and then it just applies some expensive compiler phases on the low optimization level. That is why the speedup in Figure 4.12 (b) is much lower than int Figure 4.12 (a).

Figure 4.13 shows the comparison of the cost of dynamic compilation between parallelization and normal non-parallelization. Normally the overhead of parallelization is about 150% to 180% of non-parallelization. Different benchmark programs have different costs of recompilation depending on the sizes of hot methods. For *Euler* and $Moldyn$, the size of hot methods are big and the compilation time is longer (e.g. $164,830,835$ cycles for one hot method). And JLB has a smaller parallelizable loop which costs about $23,289,429$ cycles for compilation, so it is easy to tradeoff this overhead by parallelization.

By studying the experimental results from the simulated multi-processor environment, the runtime parallelization works well on those floating point applications which are easier to parallelize. But the benefit gained for general purpose programs is still limited (the worst example is *Compress*). There are two major problems:

- It is difficult for a compiler to identify a region for parallelization even in

a static compiler. By examining a lot of loops but failing to parallelize them, the LPC generates more overhead and gets less benefit. If the LPC can identify more parallelized loops, the program's performance can be improved, even with more compilation overhead.

- Usually the problem size of general purpose programs is limited. To trade-off the compilation overhead, the program needs to run for long enough. With the current evaluation, if *Compress* could run for more iterations, the speedup shown in Figure 4.12 (b) would be positive.

4.4 Summary

This chapter introduced a parallel compiler (LPC) which can perform LLP dynamically at runtime. By evaluating both general purpose applications and scientific computing applications, the LPC shows different speedups for different programs. Most scientific computing applications get good improvement for their runtime performance. For general purpose applications, some of them could get speedup and therefore the benefits are application specific.

Chapter 5

Adaptive Runtime Optimization

The previous chapter introduced how the JaVM's dynamic compilation system can be extended to perform parallelization at runtime. This chapter addresses whether a runtime optimization system can improve the parallel programs' performance. An *Online Tuning Framework* (OTF) is implemented to perform the runtime optimizations. The OTF is built on top of the JaVM and targets Java programs. It uses LPC to parallelize loop-based programs and empirically searches for suitable optimizations (e.g. optimal loop tile size, optimal code version). The search relies on collecting runtime performance data and is evaluated for the JAMAICA CMP.

Section 5.1 will introduce the basic motivation of the adaptive runtime optimization, the scope of the optimizations and the basic mechanism for performing these optimizations. Section 5.2 introduces JaVM's OTF which is the infrastructure for performing runtime optimization. Section 5.3 gives a list of the adaptive optimizations which are evaluated within the OTF. Section 5.4 discusses the issues of runtime searching. Section 5.5 shows and discusses the evaluation results. Section 5.6 summarizes this chapter.

5.1 Adaptive Runtime Optimization

5.1.1 Motivation for Runtime Optimization

As mentioned in Chapter 1, the power and reach of static analysis is diminishing for modern software. Static compiler optimizations are limited by the accuracy of their predictions of runtime program behaviour. Using profiling information improves the predictions but still falls short for programs whose behaviour changes dynamically (e.g. a different input data set can result in different program behaviour) and different hardware environments (e.g. different cache size, different number of processor contexts). Due to the limited amount of information that is available to a static compiler, shifting optimizations to runtime can solve these problems by getting more efficient runtime information to drive the optimizations.

Adaptive runtime optimization systems have the advantage of being able to observe the real behaviour of an executing application and enable programs to respond to runtime feedback information and change their behaviour to adapt to the runtime environment, whereas static compilers rely on predictions for that behaviour. By performing runtime empirical searching within a dynamic optimization system with adaptive runtime optimization, ongoing runtime profiling can help the searching to progress and identify the effect of different optimizations (or configuration parameters), and select the most efficient one as the runtime optimum. So an efficient runtime optimization mechanism is necessary for improving the parallel execution of a program on many different configurations of a CMP architecture.

5.1.2 Optimizing Approaches

To optimize the parallel program's runtime performance, there are three major issues that need to be considered here:

Load Balance

Load balancing is important to parallel programs for performance reasons. To maximise the utilization of processor resource, the runtime system needs to distribute the tasks to the processor contexts evenly. There are two methods to achieve this aim:

- Equally partition the work each task receives: for loop iterations where the work done in each iteration is similar (e.g. array or matrix operations where each task performs similar work), evenly distribute the iterations across the tasks.
- Use dynamic work assignment: when the amount of work each task will perform is intentionally variable, or is difficult to predict (e.g. the loop iteration has a conditional branch, or an inner loop whose number of iterations is variable), it is helpful to use a simple runtime scheduler to assign work to parallel threads.

Data Locality

Caches take advantage of data locality in programs. Data locality is the property that references to the same memory location or adjacent locations are reused within a short period of time. Each processor context has its own data cache or shares a data cache with other contexts (in a chip multi-threading mode), so improving data locality is an efficient way to improve the parallel thread's performance.

Code Optimization

Improving the efficiency of the generated code can also improve the parallel program's performance. As some of the code optimizations can not achieve an optimum easily by static analysis, researchers have built a runtime tuning mechanisms into their runtime adaptive optimization frameworks [86, 85, 28] to optimize the sequential program at runtime, including runtime specialization, compilation flag

selection and loop unrolling. In this thesis, loop unrolling combined with instruction scheduling is investigated for utilizing the RISC processor core more efficiently (to be discussed in Section 5.3.3).

5.1.3 Online Tuning

The basic mechanism of adaptive runtime optimization uses runtime feedback to drive the search for the best optimizations. This is similar to the *Automated Empirical Optimization of Software* (AEOS) [90] which uses empirical timings to choose the best solution for a given architecture among several ways of doing the same operation. To perform the runtime optimizations, the runtime system needs to be able to do online tuning, evaluate different optimizations at runtime and reset the program to use the selected optimization. There are four issues that need to be considered here:

- *Isolating the performance critical regions.* The performance critical regions are the target of online tuning. As loop-level parallelization is exploited in this work, the parallelized loops are the performance critical regions that will be tuned.
- *The method of adapting code to different environments.* To adapt code efficiently, the runtime reconfiguration scheme and version selection are employed. The detail will be introduced in Section 5.2.2.
- *An efficient and precise method for profiling code at runtime.* The execution cycle count is employed for evaluating different configurations and versions of code. A low cost profiler is implemented to profile data at runtime. The detail will be discussed in Section 5.2.1.
- *An appropriate policy for searching for the most optimal available implementation.* As the search is performed at runtime, the overhead for runtime evaluation should be as low as possible. Two issues are important for reducing runtime overhead: the search space and the search algorithm. The search spaces evaluated in this work are all 1-dimensional or 2-dimensional linear spaces. The search algorithm is based on hill-climbing. By simplifying the search space and search algorithm, the search process can achieve an optimum in a limited number of steps.

5.2 Online Tuning Framework

To perform adaptive runtime optimization and improve the runtime performance of Java applications (especially for parallelized loops), an *Online Tuning Framework* (OTF) is implemented in JaVM.

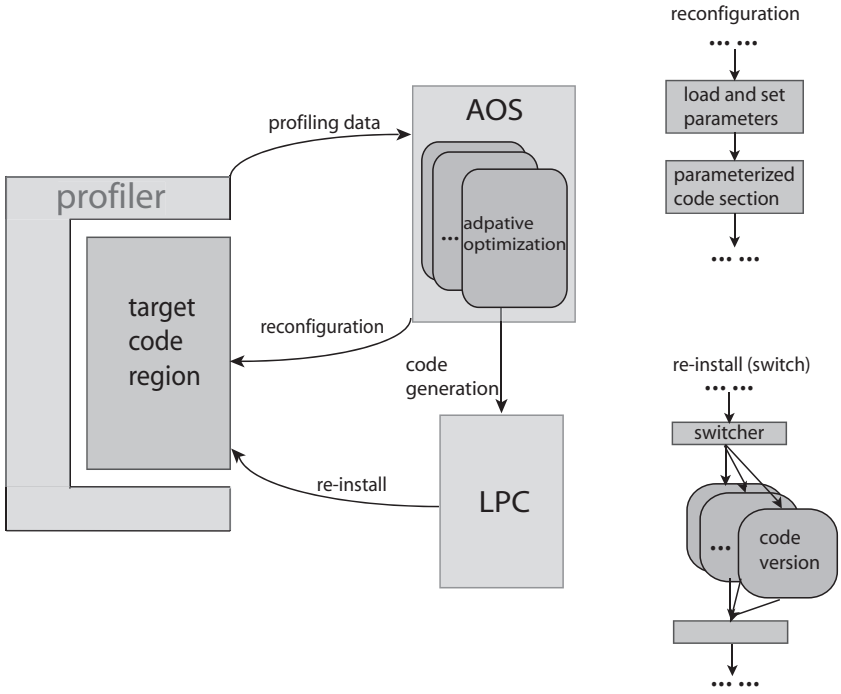


Figure 5.1: Logical View of the OTF.

5.2.1 Logical Structure

The OTF is embedded within the adaptive optimization system (AOS) of JaVM. Its implementation is tightly coupled with the loop parallelizing compiler (LPC) (introduced in Chapter 4) and the AOS, so it is difficult to isolate the OTF components clearly. Logically, the OTF can be divided into three parts: *Compiler Support*, *Runtime Profiler*, and *Adaptive Optimizations* (shown in Figure 5.1).

Compiler Support

OTF is a compiler-enabled framework which needs compiler support to perform runtime profiling and select optimizations that can be applied to parallelized loops. The Compiler Support Component (CSC) is the LPC, it invokes other OTF components by inserting a code stub which can interact with the AOS, e.g. the code for runtime profiling. When the LPC generates the parallel versions of code for a hot method's loop, it also generates the special profiler code (discussed in Section 5.2.2). As mentioned in Chapter 4, the LPC can generate multiple versions of parallel code which correspond to different loop distribution policies, and unrolling factors; these versions will be evaluated by OTF at runtime and one of the best versions will be selected.

Runtime Profiler

The OTF uses runtime profiling to evaluate the performance of selected adaptive optimizations and drive the search process. The runtime profiler is not an isolated component, it uses embedded code stubs that are inserted into the compiled code by the LPC.

Adaptive Optimizations

The adaptive optimizations work in JaVM's adaptive optimization system (AOS) and perform online tuning for different types of optimization. The search process is driven by the profiling data from the runtime profiler. To evaluate different optimizations (i.e. different compiler optimization parameters), the adaptive optimization system uses runtime reconfiguration or code generation to reset the target code or to install new versions of the code.

5.2.2 Basic Infrastructure

To support adaptive runtime optimization, OTF employs the following basic infrastructure:

Adaptive Optimization System Database

To perform runtime searching and tuning, some data structures are needed for storing runtime information. An *Adaptive Optimization System Database* (AOSD) performs this role, it stores runtime information for parallelized loops and the OTF, and helps them interact with each other. As the loop is the basic unit that is optimized by the OTF, each loop has its own entry in the AOSD and the entry is indexed by a global ID.

A data entry has the fields:

- *Loop ID*: this is a global ID in the scope of JaVM; the compiler and OTF will use this ID to identify the information for a particular loop.
- *Configuration Parameters*: this data will be used to reconfigure the parallel loop's runtime behaviour (e.g. the chunk/tile size of parallel threads).
- *Switcher Collection*: this field is an array which stores instruction offsets in a compiled method (the switcher mechanism will be introduced in the next section).
- *Multiple Version Code Collection*: this field is an array which stores a series of pointers, each pointer pointing to one version of the code (e.g. the unrolled loops with different unrolling factors or the different loop schedulers).
- *Current Version Pointer*: this field is used to maintain the pointer to the current version of code, e.g. the code corresponding to a special unrolling factor or loop distribution policy.
- *Temporary Storage for Runtime Profiling*: this data structure provides temporary storage for evaluating different runtime configurations and code versions, and the search process can use this data to find a runtime optimum.

Version Selection Mechanism

To perform runtime version selection, a runtime mechanism (shown in Figure 5.2 (a)) is needed to redirect the execution path from one code segment to another

code segment. OTF provides two approaches for that: branch code switching and method level switching.

Branch code switching uses a branch instruction to jump between different code segments by applying different branch offsets. Figure 5.2 (b) gives an example. The AOS needs to plant an instruction to choose between two execution paths: to runtime profiling code, or to the parallelized loop directly. To change the execution path, the AOS just needs to change the offset in the branch instruction, so in this example the AOS changes the offset from 0x8 to 0x4. The AOSD records the position (the offset in the code array object) of the branch instruction in the switcher collection field, so AOS can rewrite the instruction for the compiled method.

Method level switching reinstalls a method pointer (the address of the method's entry instruction) for optimized method in the Java Table of Contents (JTOC). The JTOC is a shared data table which is used to store compiled method pointers, each time the program wants to call a method, it needs to load the method's pointer from JTOC first. To change the execution path from one compiled method to another, AOS can just simply change the method pointer in the corresponding JTOC slot (shown in Figure 5.2 (c)). One point which must be emphasised here is that all of these candidate methods must have the same interface (i.e. the different methods have same number and order of the input/output parameters), otherwise runtime switching will be harmful by assigning the wrong registers or destroying stack layout.

Runtime Reconfiguration

Runtime reconfiguration is simpler than switching, because it does not need to change the execution path, it just changes the configuration parameters. In runtime reconfiguration, the application and AOS use the AOSD to interact with each other. The AOS sets new parameters into the AOSD, and the application loads these parameters at runtime. For example, the parallelized loop needs to know how many loop iterations should be executed in each parallel thread, so it loads the loop chunk size (number of loop iterations) every time before it creates parallel threads.

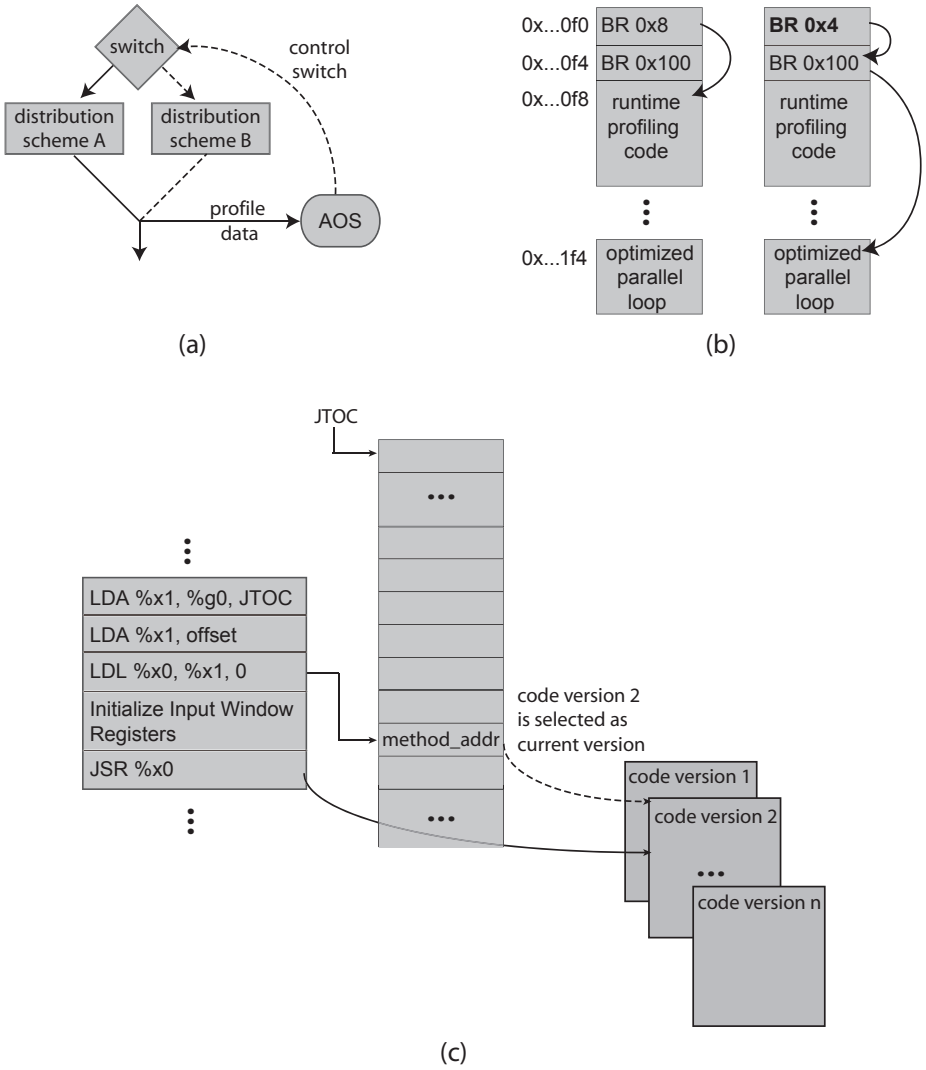


Figure 5.2: Switching Mechanism for OTF.

Adaptive Recompilation

Adaptive recompilation has been employed in JaVM's basic AOS. The profiler helps the runtime system to identify a hot method and recompile it. The LPC will be invoked to try to parallelize the loops in the hot method.

To improve the performance of parallelized loops, adaptive recompilation is still needed to optimize the parallel threads' code which corresponds to the parallelized loops' bodies. As mentioned in Chapter 4, an MIR-level recompilation mechanism is employed here to perform backend optimizations on machine code level.

Runtime Profiling

To evaluate the performance of the selected adaptive optimizations by empirical timing, the OTF needs to be able to calculate each optimization's execution cycles. This is achieved by inserting two additional code stubs, one at the start and one at the end of the parallelized loop being profiled. The first stub extracts from the architecture the cycle count[1] prior to the loop's execution, and the second stub extracts the cycle count after the loop has executed. The second stub is also responsible for reporting the total execution cycle, and the number of loop iterations, back to the AOS, and creating a parallel thread: the AOS thread (shown in Figure 5.3).

The OTF is then able to the do following work in the AOS thread (more detail will be given in Section 5.2.3):

- Calculate the execution time per iteration of each invocation of the loop.
- Make a decision about the comparative performance with other invocations of the loop under different versions of code and configuration parameters by applying the search algorithm.
- If an optimum is found for a given optimization, the AOS stops profiling and, having assessed all optimizations, the AOS switches off the runtime profiler and any subsequent executions of the loop are run using the best optimization found.

[1] Although this mechanism is machine specific, suitable instructions exist in the main architectures: RDTSC (x86), mftb (PPC), TICK register (SPARC)

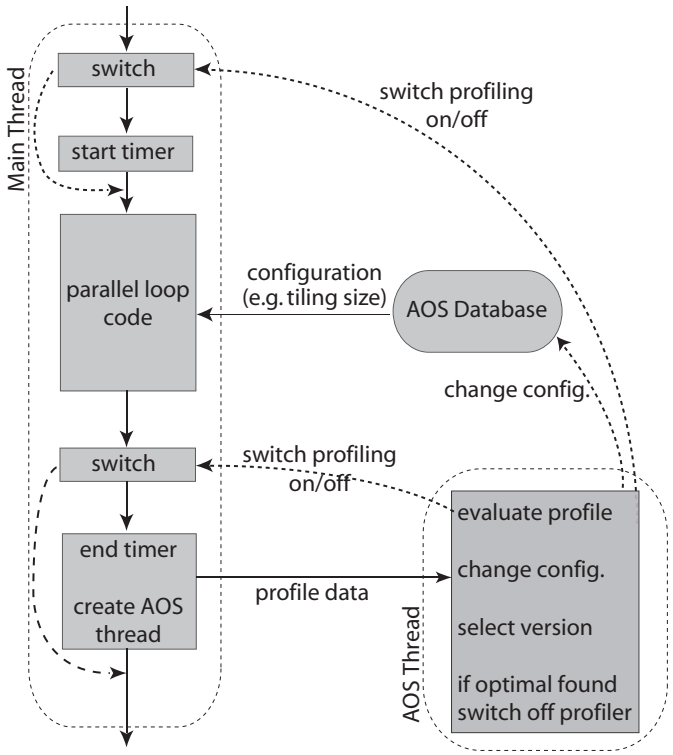


Figure 5.3: Runtime profiling.

All of the evaluation and search processing work is done in the AOS thread. Normally, the application will not use all the processor contexts available for in its execution, so the AOS thread is created on demand and does its job on another free processor context and does not affect the main application thread's work.

The precision of the execution time metric is a major factor in getting good results from the optimizations presented, and there are two issues that effect the precision. The first is that not all loops are of static length or duration, it is possible that both the number of iterations and the loop's content will vary per invocation. The second issue is that the execution timings are affected by system noise, for example cold caches and other unrelated thread activity which changes the number of free processors.

To overcome these issues, the execution time for a given optimization on a parallelized loop is calculated as an arithmetic mean of the cycles per iteration for three or four invocations of that loop. Loops that exhibit large profile deviations,

defined as having a *coefficient of variation*[2] (CV) greater than a configurable threshold[3], are deemed unstable, the AOS thread should check the execution phase (discussed below) and decide if it should keep searching, restart the search or switch to a dynamic_assignment model (see Section 4.1.4).

Phase Detection Mechanism

As introduced in Chapter 3, the runtime phase detector will calculate statistical data based on these profiling values in a configurable interval, which is a fixed number of recording operations (15 is used for this work). In one interval, the statistical data includes:

- The execution cycles for each runtime phase: E_0, E_1, ... E_n (0, 1, ... n are the indexes in the sample array).
- The total execution cycles for all runtime phases: E.
- The execution cycles for the longest runtime phase: E_L (L is an integer which is used to index the sample array).
- The arithmetic mean of all the runtime phases: m.

The aim of getting such statistical data is to help the runtime profiler handle two problems: noise from other runtime threads and changes in the runtime environment.

By observing various applications' runtime behaviour, three typical phase scenarios are derived (shown in Figure 5.4). Figure 5.4 (a) is *stablephase*: the number of free processors did not change during execution of parallelized loop. The scenario is stable for runtime searching, because the parallelized loops' execution cycles can be measured precisely. The second scenario is *stable/noise phase* (shown in Figure 5.4 (b)) which means that a stable phase is interrupted by several noise threads[4]. The noise threads have a low probability of affecting the parallelized loops' runtime performance, so this type of scenario is also suitable for runtime

[2]Coefficient of variation (CV) is the ratio of the standard deviation to the arithmetic mean.
[3]The threshold for CV is configurable; it is set at 0.1 for this work
[4]A noise thread is any thread whose execution is less than 3000 cycles during activation.

searching. The last scenario is *unstable phase* (shown in Figure 5.4 (c)): the number of free processors changes during the period of parallelized loops' execution, and each change also has a long period.

The runtime profiler should eliminate the effect of runtime noise threads whose execution time is less than 3000 instruction cycles. Shown in Figure 5.4 (b), there are five runtime phases (a, b, c, d, and e) in one run of parallelized loop. Phases b and d are very short and these phases have a low probability of affecting parallel thread creation.

The runtime profiler should also detect whether the runtime environment is unstable for the current parallelized loop which means the runtime phases change too frequently to perform runtime searching. Shown in Figure 5.4 (c), there are three runtime phases (a, b and c) in one run of the parallelized loop. These three phases are not generated by noise threads, they can affect parallel thread creation and so affect the profiling results for the current loop.

By applying Algorithm 6, the runtime profiler can figure out which scenario (shown in Figure 5.4 (a), (b) and (c)) corresponds to the current execution. This algorithm checks if the longest runtime phase is less than 20 or 30 percentage of the whole execution time (30 is selected in the current implementation) in one statistical interval. If so, this means that the execution time for these runtime phases are distributed evenly, and the scenario is (c). If not, the scenario could be (a) or (b). The algorithm checks the longest phase's two neighbours. If one of them has very short execution time and its neighbour has execution time which is equal or larger than the mean value m and it also has the same number of free processors as the longest phase, this scenario should be (b) which means a long phase is being interrupted by several short phases. Otherwise the scenario should be (a) which is a normal phase distribution.

Algorithm 7 helps the runtime profiler decide if it should keep searching or switch to DAD mode (discussed in Section 4.1.4). When the runtime is in an unstable scenario, the thread creator may not allocate enough processor contexts for parallel threads, the DAD mode keeps trying to grab more idle processor contexts and this greedy task allocation policy could help the parallelized program get more computing resource and improve performance.

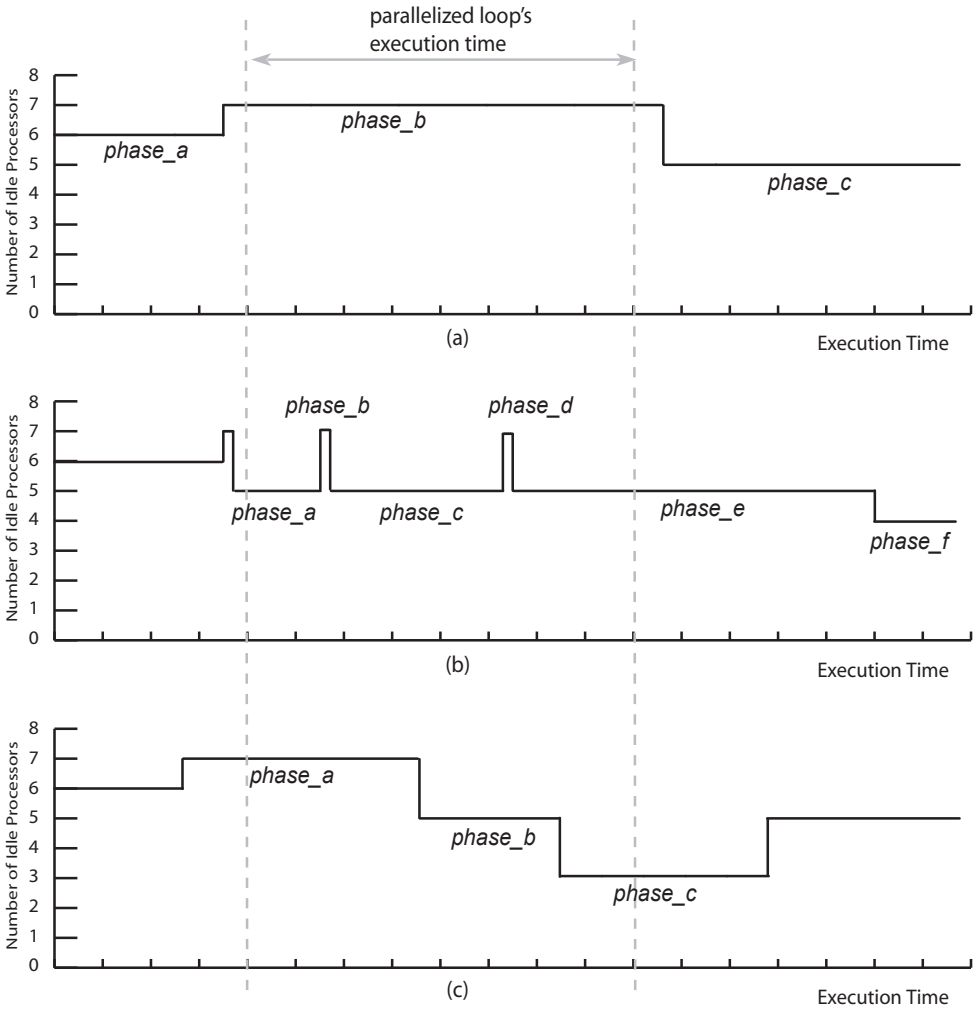


Figure 5.4: The Online Phase Detection Mechanism in OTF.

Input: the statistical data from phase detector (E_0, E_1, ... E_n, E, E_L, m)
Output: current scenario
Implementation:
$r \leftarrow \frac{E_L}{E}$;
if $r \leq 0.3$ **then**
  return $ScenarioC$;
end if
if $E_{L-1} \leq \frac{m}{10}$ and $m \leq E_{L-2}$ **then**
  return $ScenarioB$;
else if $E_{L+1} \leq \frac{m}{10}$ and $m \leq E_{L+2}$ **then**
  return $ScenarioB$;
else
  return $ScenarioA$;
end if
Algorithm 6: scenario_detection (Scenario Detection Mechanism).

Input: the statistic data from phase detector (E_0, E_1, ... E_n, E, E_L, m), the average execution time for parallelized loop E_p
Output: boolean result
Implementation:
$s \leftarrow scenario\_detection(E_0, E_1, ...E_n, E, E_L, m)$;
if $s = ScenarioA$ **then**
  return $E_p \leq E_L$
end if
if $s = ScenarioB$ **then**
  return TRUE;
end if
if $s = ScenarioC$ **then**
  return $E_p \leq m$
end if
Algorithm 7: phase_checking (Phase Checking Mechanism).

5.2.3 Working Mechanism

The basic work flow for runtime empirical searching is composed of five steps:

1. *Search Code Generation:* AOS identifies a hot method by the normal JaVM profiling mechanism, and recompiles it applying more optimization phases. LPC intercepts this procedure and tries to find suitable loops (*DoAll* loops or *DoAcross* loops with simple scalar operations) and parallelizes them. To enable the runtime empirical search on a loop that can be parallelized, LPC will insert the code stubs (introduced in Section 5.2.2) which do empirical timing and invoke the AOS thread, and will initialize a data entry in the AOSD. The adaptive optimization that will be assigned for the loop can be configured by hand, or selected automatically by LPC. LPC should assign different types of adaptive optimization to different types of loop. For example, a 2-Dimensional nested loop can be assigned tile based adaptive optimization.

2. *Code Installation and Starting the Search:* LPC installs the recompiled method which contains the parallelized loop. And the AOS will start runtime empirical searching on the parallelized loop, when this newly installed method is called.

3. *Runtime Empirical Searching:* For each loop running, the installed profiling code stubs report execution cycles and number of iterations to the AOS thread. The AOS thread calculates the *execution cycles per iteration* (ECI) and stores it into a *Sample Window* which is composed of 3 or 4 ECI values [5]. When a sample window is completely full, the AOS thread should calculate the CV, to check whether the samples are stable.

   If the CV does not exceed the threshold, then the mean of the samples is set to the evaluation result for the current configuration and recorded in the AOSD. The AOS can then evaluate the next configuration generated by the search algorithm.

   If the CV exceeds the threshold, the AOS should check whether this is affected by runtime noise or an unstable runtime environment by applying

[5]The number of elements in a sample window is configurable. Normally, 3 or 4 are selected for this work.

Algorithm 7 (described in Section 5.2.2). If it is affected by an unstable runtime environment, the AOS thread should switch off the search process and switch the loop to the dynamic_assignment parallelization model (discussed in Chapter 4). If it is just affected by runtime noise, the AOS thread can shift the sample window to the next run and evaluate the CV again.

4. *Stopping the Search and Applying the Optimal Solution:* When an optimal configuration or code version is selected, the AOS thread switches off the profiler. In switching off the profiler, the code stub that previously invoked runtime profiling is modified, so that future execution of the code no longer needs to execute any code inside the profiling phase.

5. *Restarting the Search:* AOS needs to search for different optimal configurations or code versions for different problem sizes [6] and different numbers of free processors. As these two aspects can be changed at runtime, so AOS needs to restart the search process when it detects the change and there is no optimal search result for the changed environment. Details will be discussed in Section 5.4.

5.3 Adaptive Optimizations

The AOS inserts one or more optimizations deemed to be appropriate for optimizing a given loop, identified by the LPC, into the code. The adaptive optimizations are located by the LPC to place the optimizations around the identified parallel loops. Presently the AOS supports three major types of adaptive optimization for parallelizable code (see below). These optimizations vary either the number of loop iterations inside a chunk or tile, the loop unrolling factors, or the manner in which the chunks or tiles are distributed. By varying these factors the OTF is able to find strategies that best balance the costs associated with threading, the cache performance, and the system load.

[6] In this thesis, the application's problem size means the size of the input data set.

5.3.1 Adaptive Chunk Division (ACD)

For a given loop the total number of iterations is divided into chunks. Each chunk is then distributed through the creation of a parallel thread. The parallel threads can be run on any available processor context. Ideally, given an n iteration loop on a CMP with P_n processor contexts, the best chunk size for division is $\frac{n}{P_n}$. But there are several issues that affect the division, including the overhead of thread creation, data locality and if the loop body which contains an irregular workload.

In all the optimizations if a thread cannot be invoked on a remote processor context, the main thread (i.e. the generator thread) must perform its work before continuing to distribute subsequent threads. This optimization uses a hill-climb-like algorithm (see Algorithm 9) to adaptively divide the total number of iterations in a loop into chunks and search for an optimal divisor. The search starts to test if the chunk size which is smaller than the ideal size is suitable (i.e. larger divisor), because the small granularity makes the workload to be distributed more evenly on parallel processor contexts and also can improve the data locality for some memory intensive programs (e.g. $DGEMM$). If the search process can not benefit from the small chunk size, it will come back to evaluate a larger chunk size (i.e. smaller divisor) by the small steps (i.e. the divisor incremented by $\frac{1}{P_n \times 2}$, see Algorithm 8) to tradeoff the overhead for thread creation. The search space employed here is a 1-dimensional linear space in the range: $1 \leq \text{optimal} \leq P_n \times k$ (k is a positive integer).

5.3.2 Adaptive Tile Division (ATD)

As loops can be tiled to take advantage of data reuse [92, 45, 18], selecting a suitable tile size is a common technique for improving performance. This optimization is applied when a perfectly nested loop is identified by the LPC. The 2-dimensional loop traversal of the iteration space is divided into tiles which are then distributed by the creation of parallel threads. Each tile has a corresponding divisor pair. Given a divisor pair (D_i, D_j), D_i is the divisor corresponding to the outer loop iterator and D_j corresponds to the inner loop iterator.

Adaptive tiling, again using a hill-climbing algorithm, starts from an initial divisor pair (D_{i0}, D_{j0}). The initial divisor pair is calculated such that $D_{i0} \times D_{j0} = P_n$,

Input: P_n, number of processor contexts; G, a threshold, which is used to decide when to stop the search
Output: optimal divisor
Implementation:
Step 1:
$D_m \leftarrow P_n$; $n \leftarrow \frac{1}{2}$; $D_l \leftarrow D_m$;
$E_m \leftarrow$ execution cycles per iteration for loop using D_m; $E_l \leftarrow E_m$;
Step 2:
while $n > G$ **do**
 $E_r \leftarrow$ execution cycles per iteration for loop using D_r;
 if $E_r \geq E_m$ **then**
 $E_l \leftarrow E_m$; $D_l \leftarrow D_m$; $E_m \leftarrow E_r$; $D_m \leftarrow D_r$; $D_r \leftarrow D_m + n$;
 else if $E_r > E_m$ and $E_r < E_l$ **then**
 $D_l \leftarrow D_m$; $E_l \leftarrow E_m$; $D_r \leftarrow D_m + \frac{n}{2}$; $n \leftarrow \frac{n}{2}$;
 else if $E_r > E_m$ and $E_r > E_l$ and $D_m \neq D_l$ **then**
 $D_r \leftarrow D_m$; $E_r \leftarrow E_m$; $D_l \leftarrow D_m - \frac{n}{2}$;
 goto Step 3;
 else if $E_r > E_m$ and $D_m = D_l$ **then**
 $D_r \leftarrow D_m + \frac{n}{2}$; $n \leftarrow \frac{n}{2}$;
 end if
end
Step 3:
while $n > G$ **do**
 $E_l \leftarrow$ execution cycles per iteration for loop using D_l;
 if $E_l \leq E_m$ **then**
 $E_m \leftarrow E_l$; $D_m \leftarrow D_l$; $D_r \leftarrow D_m + \frac{n}{2}$; $n \leftarrow \frac{n}{2}$;
 goto Step 2;
 else if $E_l > E_m$ **then**
 $D_l \leftarrow D_m - \frac{n}{2}$; $n \leftarrow \frac{n}{2}$;
 end if
end
Step 4: return D_m;

Algorithm 8: acd (Adaptive Chunk Division).

Input: P_n, number of processor contexts; G, a threshold, which is used to decide if stop the search
Output: optimal divisor
Implementation:
Step 1:
$D_m \leftarrow P_n$; $m \leftarrow 2$; $D_m \leftarrow D_m \times m$;
$E_m \leftarrow$ execution cycles per iteration for loop using D_m;
Step 2:
$E_r \leftarrow$ execution cycles per iteration for loop using D_r;
if $E_r \leq E_m$ **then**
  $E_m \leftarrow E_r$; $D_m \leftarrow D_r$; $D_r \leftarrow D_m \times (m+1)$; $m \leftarrow m+1$;
  goto Step 2;
else if $E_r > E_m$ **then**
  return $acd(D_m, G)$;
end if
  Algorithm 9: multi_acd (Multiple Adaptive Chunk Division).

where P_n is the total number of processor contexts. This partition, thereafter referred to as a naïve scheme, simply distributes the tiles evenly among the processor contexts. Algorithm 10 describes how to calculate D_{i0} and D_{j0}.

The searching process increases D_i and D_j iteratively to determine whether smaller tile sizes provide smaller execution times. When no performance improvement is observed, the OTF stops the search. Any divisor pair (D_i, D_j) calculated during iteration is composed such that $D_i \times D_j = k \times P_n$ where k is a positive integer value ($k > 0$) and P_n is the total number of processors. Algorithm 11 presents the search algorithm used to determine the divisor pairs. The search space is a rectangular space which corresponds to the iteration space of a two-level nested loop. Each search step shrinks the area of the tile by half or changes the shape of the tile.

For the specific CMC architecture (introduced in Chapter 2), two tile sizes (for the L1 and L2 cache) are considered. The adaptive search begins by finding an optimal tile size for the L1 cache, which is a subset of the data within the L2 cache. When an optimal L1 tile size is determined, the OTF searches for a L2 cache tile size using the same search algorithm but using different initial divisor pairs. Algorithm 12 describes the combined search mechanism to optimize loop division for a CMC architecture. Consider the example loop shown in Figure 5.5, the search process for the L1 cache tile considers any rectangle which is contained

Input: P_n, number of processor contexts
Output: a initial divisor pair
Implementation:
Step 1: $t \leftarrow \lfloor sqrt(P_n) \rfloor$;
Step 2:
while $(P_n \% t) \neq 0$ **do**
  $t \leftarrow t - 1$
end while
Step 3: return $(\frac{P_n}{t}, t)$;

Algorithm 10: init_tile_rect (Initialize Tile Size for Rectangle Iteration Space).

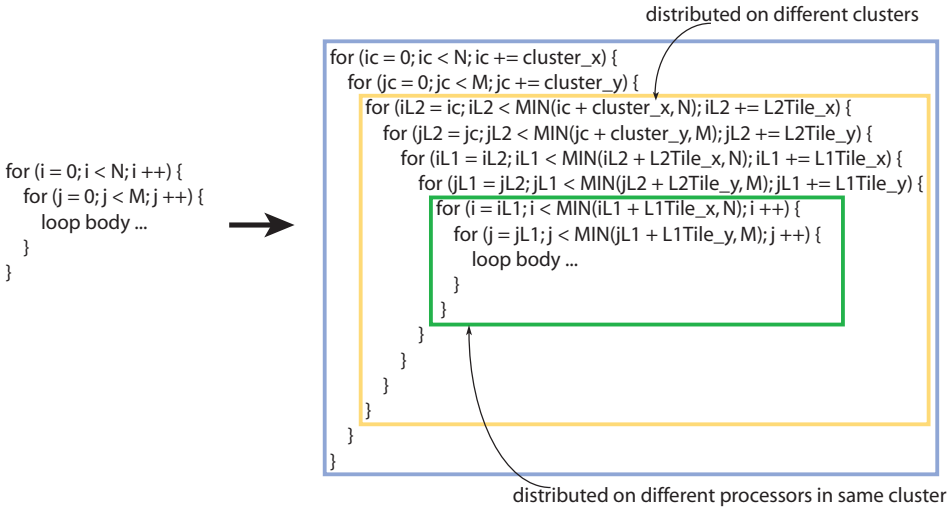


Figure 5.5: Tiling Transformation for Runtime Tuning.

within a rectangle with sides $cluster_x$ and $cluster_y$. The search space for the L2 cache tile is based on the L1 tile size. Given the optimal divisor pair for the L1 tile, (D_x, D_y), the search process for the L2 tile is for any rectangle with sides multiples of $\frac{cluster_x}{D_x}$ and $\frac{cluster_y}{D_y}$, respectively, and contained within the rectangle with sides $cluster_x$ and $cluster_y$.

Another advantage of tile based loop distribution is improving load balance for non-rectangular iteration spaces. Given a triangular iteration space which a two-level nested loop is working on, if a chunk based distribution is applied which is a 1-Dimensional distribution, each parallel task will be assigned a different number of iterations and the load imbalance is obvious. By using a tile based distribution policy, the iteration space is divided into small tiles. This can make the distribution of loop iterations more average, thus improving load balance and

Input: P_n, number of processor contexts
Output: optimal divisor pair
Implementation:
Step 1: $(D_i, D_j) \leftarrow init\_tile\_rect(P_n)$;
Evaluate the runtime performance by initial tile size (D_i, D_j), get execution cycles per iteration E_m
Step 2:
$(D_{il}, D_{jl}) \leftarrow (D_i \times 2, D_j)$;
$(D_{ir}, D_{jr}) \leftarrow (D_i, D_j \times 2)$;
$E_l \leftarrow$ execution cycles per iteration for loop using tile size (D_{il}, D_{jl});
$E_r \leftarrow$ execution cycles per iteration for loop using tile size (D_{ir}, D_{jr});
if $E_m \leq E_l$ and $E_m \leq E_r$ **then**
  goto Step 3;
end if
if $E_r \leq E_l$ **then**
  $(D_i, D_j) \leftarrow (D_{ir}, D_{jr})$; $E_m \leftarrow E_r$;
else
  $(D_i, D_j) \leftarrow (D_{il}, D_{jl})$; $E_m \leftarrow E_l$;
end if
goto Step 2;
Step 3:
$i \leftarrow 2$
Step 4:
$(D_{il}, D_{jl}) \leftarrow (\lfloor \frac{D_i}{i} \rfloor, D_j \times i)$;
$(D_{ir}, D_{jr}) \leftarrow (D_i \times i, \lfloor \frac{D_j}{i} \rfloor)$;
$E_l \leftarrow$ execution cycles per iteration for loop using tile size (D_{il}, D_{jl});
$E_r \leftarrow$ execution cycles per iteration for loop using tile size (D_{ir}, D_{jr});
if $E_m \leq E_l$ and $E_m \leq E_r$ **then**
  return (D_i, D_j)
end if
if $E_r \leq E_l$ **then**
  $(D_i, D_j) \leftarrow (D_{ir}, D_{jr})$; $E_m \leftarrow E_r$;
else
  $(D_i, D_j) \leftarrow (D_{il}, D_{jl})$; $E_m \leftarrow E_l$;
end if
$i \leftarrow i + 1$;
goto Step 4;

Algorithm 11: tile_search_rect (Adaptive Tile Division for Rectangle Iteration Space).

Input: P_n, number of processor contexts; C_n number of clusters.
Output: optimal divisor pairs for L1 and L2 tile
Implementation:
Step1: Search for L1 cache tile size
$L1Tile \leftarrow tile\_search\_rect(P_n)$; // $L1Tile$ is the optimal divisor pair for L1 tile
Step2: Search for L2 cache tile size
$L2Tile \leftarrow tile\_search\_rect(C_n)$; // $L2Tile$ is the optimal divisor pair for L2 tile

Algorithm 12: multi_level_search (Multiple Levels Search).

data locality. Algorithm 14 presents how to search for the optimal tile size for triangular iteration space, the tile size (D_i, D_j) should satisfy: $D_i \times (D_j + 1) = 2 \times P_n \times k$ (k is a positive integer).

Input: P_n, number of processor contexts
Output: a initial divisor pair
Implementation:
Step 1: $t \leftarrow (P_n \times 2)$; $tile\_x \leftarrow P_n$; $tile\_y \leftarrow 1$;
Step 2:
while $tile\_x > tile\_y$ **do**
  $tile\_x \leftarrow \lfloor \frac{t}{tile\_y} \rfloor$; $tile\_y \leftarrow tile\_y + 1$;
end while
Step 3: return $(tile\_x, tile\_y)$

Algorithm 13: init_tile_tria (Initialize Tile Size for Triangular Iteration Space).

5.3.3 Adaptive Version Selection (AVS)

This type of optimization performs runtime selection between different loop scheduling policies or different versions of code. There is no special search algorithm for AVS, because the search space should be small (the number of code versions or loop schedulers is limited). The OTF simply evaluates all code versions and selects the best one.

Loop Distribution Policy Selection

By defining a common interface between the main method which contains the parallel loop, loop scheduler methods and parallel thread body methods, the

Input: P_n, number of processor contexts
Output: optimal divisor pair
Implementation:
Step 1: $t \leftarrow 1$; $(D_{i0}, D_{j0}) \leftarrow init\_tile\_tria(P_n \times t)$; $E_0 \leftarrow$ execution cycles per iteration for loop using tile size (D_{i0}, D_{j0});
Step 2: $t \leftarrow t + 1$; $(D_i, D_j) \leftarrow init\_tile\_tria(P_n \times t)$; $E_t \leftarrow$ execution cycles per iteration for loop using the smaller tile size (D_i, D_j);
if $E_0 \leq E_t$ **then**
  return (D_{i0}, D_{j0})
else
  $(D_{i0}, D_{j0}) \leftarrow (D_i, D_j)$; $E_0 \leftarrow E_t$; goto Step 2;
end if

Algorithm 14: tile_search_tria (Adaptive Tile Division for Triangular Iteration Space).

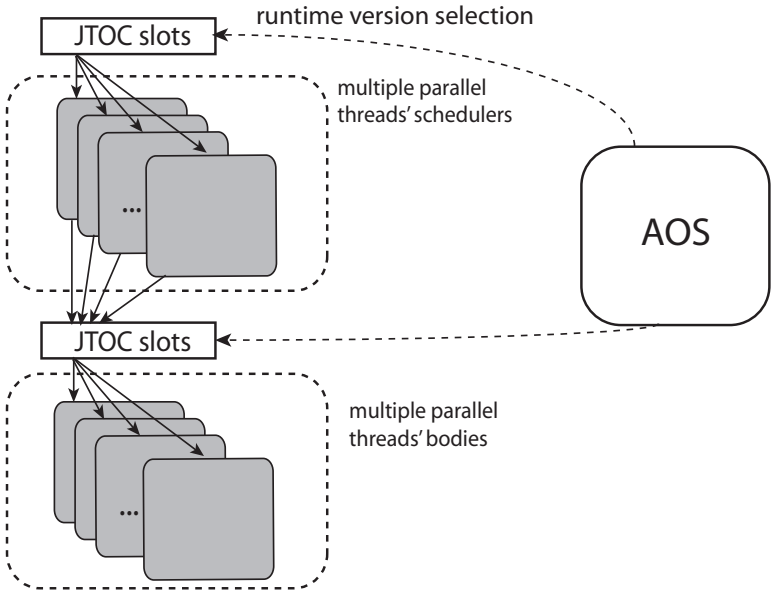


Figure 5.6: Multiple Version of Loop Distributor and Parallel Thread Body.

AOS can select different loop scheduling policies flexibly (shown in Figure 5.6.)

AOS can select between a chunk based scheduler and a tile base scheduler. If the problem size is large, to achieve both load balance and data locality, the tile based loop scheduler should be the better choice. For a small problem size, the chunk based scheduler which has lower overhead for parallel thread creation and is the better solution.

AOS can also select between the chunk/tile based scheduler, and a scheduler using a cyclic recursive distribution (CRD) (introduced in Chapter 4). CRD uses a tree-like distribution policy which reduces thread creation overhead by creating parallel threads recursively. The CRD can also improve load balance for triangular iteration spaces with low overhead. The drawback of CRD, however, is that it increases the number of cache misses when used in a multi-processor environment. This is because a contiguous memory segment will be mapped across different processor cores caches.

Code Version Selection for Code Optimization

Loop unrolling [68] is a widely used compiler optimization, it reduces the overhead of executing an indexed loop and improves the effectiveness of other optimizations. Employing loop unrolling here can improve the effectiveness of instruction scheduling which improves runtime performance when running on a RISC processor core with pipeline support (in the JAMACIA architecture, each processor core has a 5-stage pipeline) [69, 79, 7]. By reducing the data dependency between the instructions, instruction scheduling can generate more overlap between replicated loop bodies and improve the parallel execution on the pipeline. To reduce the data dependency, register renaming is used to rename the local operands in replicated loop bodies. Increasing overlap can improve the efficiency of the pipeline, but also increase the register pressure which will generate more spill operations and decrease the performance. That is why runtime searching is employed here to find a suitable loop unrolling factor.

JaVMs dynamic compiler supports a simple pre-pass local-scheduling mechanism [58], which is a single basic-block based instruction scheduling and uses a list-scheduling algorithm [82]. By the limitation of instruction scheduling support, *Adaptive Loop Unrolling* (ALU) works on the loops which contain just one basic

block. Usually, this type of loop is the innermost loop in loop nests and part of the most heavy computing workload. To find the best loop unrolling factor, the LPC generates different versions of a parallelized loop which are unrolled by different factors that evenly divide the number of innermost iterations, up to a maximum unrolling factor of N (this value is configurable, it is set to 8 in this work). OTF evaluates these versions at runtime to select the best one.

To tradeoff the effect of the runtime code generation for multiple version of code (i.e. for different unrolling factors), LPC performs parallel compilation which can utilize more idle processors to build multiple versions of the code.

5.4 Runtime Searching Issues

As discussed in the previous sections, there are two important issues that affect the result of runtime empirical searching: the problem size and computing resource.

5.4.1 Problem Size

For runtime reconfiguration related adaptive optimizations, namely adaptive chunk/tile size selection, the problem size is an important issue for runtime searching, because the performance of the cache is highly dependent on the problem size and the tile size. The same tile size can give rise to widely varying cache miss rates for similar problem sizes [56]. Although the current solution is based on divisors (the runtime search result is the optimal divisors), the divisors can generate different tile sizes for different problem sizes. One divisor (or one pair of divisors) is still not suitable for all problem sizes.

So we need to restart the search process at runtime when the problem size changes. As discussed in previous sections, the OTF can easily switch on/off the runtime profiler and trigger the search. The key problem is when to restart the search. The current solution is to divide the iteration space into several regions, the size of a region being configurable. Each region has its own optimal configuration, and a flag to notify whether this region has an optimum. Every time the parallelized loop tries to load the divisors and calculate the chunk or tile size, it should check

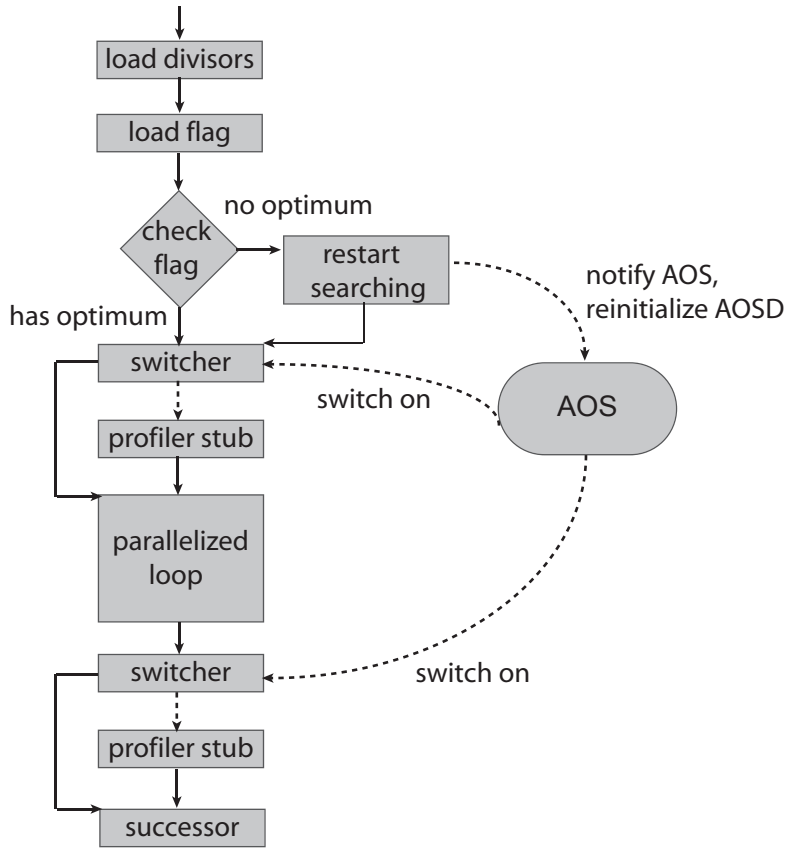


Figure 5.7: The Runtime Restart Mechanism.

the region flag first, to makes sure that this region has an optimal configuration. If there is no optimum in the current region, the program should switch on the runtime profiler (shown in Figure 5.7).

Usually, most test applications have the same problem size for their whole runtime, so the code stub to check for a restart is optional. By evaluating the different schemes, it has been seen that the tile based loop distribution is more easily affected by problem size, because different tile sizes result in different behaviour in data locality.

5.4.2 Computing Resource

The number of available processor contexts (P_n) is most important effect which can affect the parallel program's performance. For different values of P_n, the

optimal chunk/tile size will be different. So the search process should be restarted when the P_n changes.

In JaVM, multiple Java threads work concurrently. These concurrent threads affect the value of P_n. Section 5.2.2 has discussed the runtime phases in a multi-threading environment. The system threads and the application threads are the two major types of thread in the JaVM runtime.

The system threads can be classified into three types:

- *AOS Threads*: performing runtime empirical searching and collecting runtime profiling information. By observing JaVM's behaviour, this type of thread is typically short running. So these threads' execution can be treated as runtime noise for parallelized loops.
- Compilation Thread: performing runtime recompilation (recompiling the hot methods). This is a long running thread, because runtime recompilation with the optimizing compiler is an expensive operation which will go through at least 15 compilation phases.
- GC Thread: performing garbage collection. JaVM employs a *Stop the World* GC policy; there is no other Java thread active when the GC threads are working. As the LPC has excluded the GC triggers when it generates the parallel code and runtime profiler stubs, the GC threads will not affect the parallelization and runtime searching.

These threads can not make the JaVM enter into an unstable phase for measurement, so the system threads will not affect the runtime searching.

The behaviour of the application threads is varies. Current observations of the scientific applications and some of the general purpose applications is that they employ long running threads. So they are stable. Some of the network based applications (e.g. web application, middleware) utilized short term threads which are created frequently for handling user connections, so they are unstable and the phase detector will help AOS decide if it need to switch to DAD mode.

5.5 Evaluation and Discussion

This section shows the experimental results got from the JAMAICA simulator (mentioned in Chapter 2) and discusses the efficiency of the adaptive optimizations. 5 benchmarks selected from Table 4.1, *JavaLinpack*, *LU* decomposition, *Zchol* [7] and three well known BLAS 3 kernels ($DGEM$M, $DSYRK$, $DTRMM$, shown in Figure 5.8) are used for performance evaluation. All of these benchmarks have computation intensive loops that can be parallelized, so they are ideal for demonstrating the optimizations.

5.5.1 Adaptive Chunking

Figure 5.9 shows the OTF searching for an optimal divisor in the *Linpack* benchmark using ACD. The hardware configuration is 8 processors, 1 context per processor, with 16KB L1 cache. By the 9th invocation of the parallelized loop the initial overhead of the runtime profiling code is amortized by the optimized performance, and by 12 invocations a local optimal divisor for this loop has been found (the ACD's threshold used here is $\frac{1}{16}$).

Figure 5.10 presents the results of optimizing the benchmarks using the ACD optimization. The results show the speedup achieved using the adaptively found local-optimal divisor, listed in the table, compared to the naïve divisor which divides the loop iterations evenly between a fixed number of threads, in this case equal to the total number of processor contexts.

For the majority of cases shown in figure 5.10, the optimal divisor is a value less than the naïve divisor [8]. This is due to the nature of the distribution scheme. The processor context responsible for distributing the parallel threads, the generator, is always the last available for processing any of the loop iterations. In the case of a smaller divisor, e.g. 3.75 as opposed to 4, a loop with 100 iterations will be distributed such that the first three distributed threads each contain a chunk of 26 iterations, and the fourth contains only 22. This scheme is therefore able to trade-off the overhead on the generator thread. In the cases where the optimal divisor is larger than the naïve divisor, the optimization overcomes the context contention

[7] Zchol implements the Cholesky decomposition of a positive definite matrix.
[8] The naïve divisor employed here equals the number of processor contexts P_n.

```
for (int i = 0; i < mLength; i ++) {
  for (int j = 0; j < nLength; j ++) {
    double temp = 0.0;
    for (int k = 0; k < nLength; k ++) {
      temp += alpha * matA[i][k] * matB[k][j] + beta * matC[i][j];
    }
    matC[i][j] = temp;
  }
}
```

(a) DGEMM

```
for (int i = 0; i < mLength; i ++) {
  for (int j = i; j < nLength; j ++) {
    double temp = 0.0;
    for (int k = 0; k < nLength; k ++) {
      temp += alpha * matA[i][k] * matB[k][j] + beta * matC[i][j];
    }
    matC[i][j] = matC[j][i] = temp;
  }
}
```

(b) DSYRK

```
for (int i = 0; i < length; i ++) {
  for (int j = 0; j < length; j ++) {
    double temp = 0.0;
    for (int k = i; k < nLength; k ++) {
      temp += matA[i][k] * matB[k][j];
    }
    matC[i][j] = temp;
  }
}
```

(c) DTRMM

Figure 5.8: Level 3 BLAS Kernels.

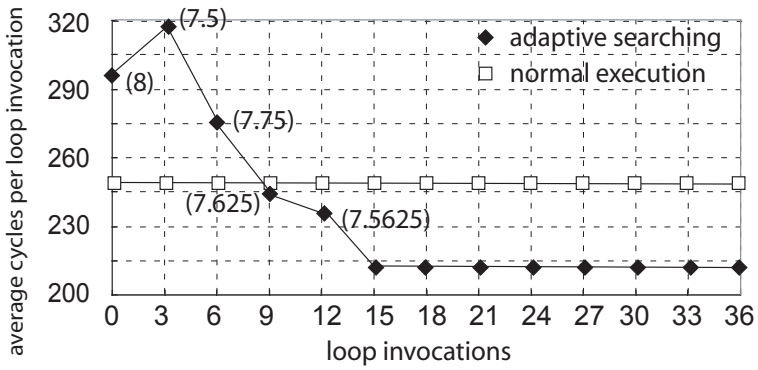


Figure 5.9: Searching Profile Using ACD for The Linpack (100×100) Benchmark.

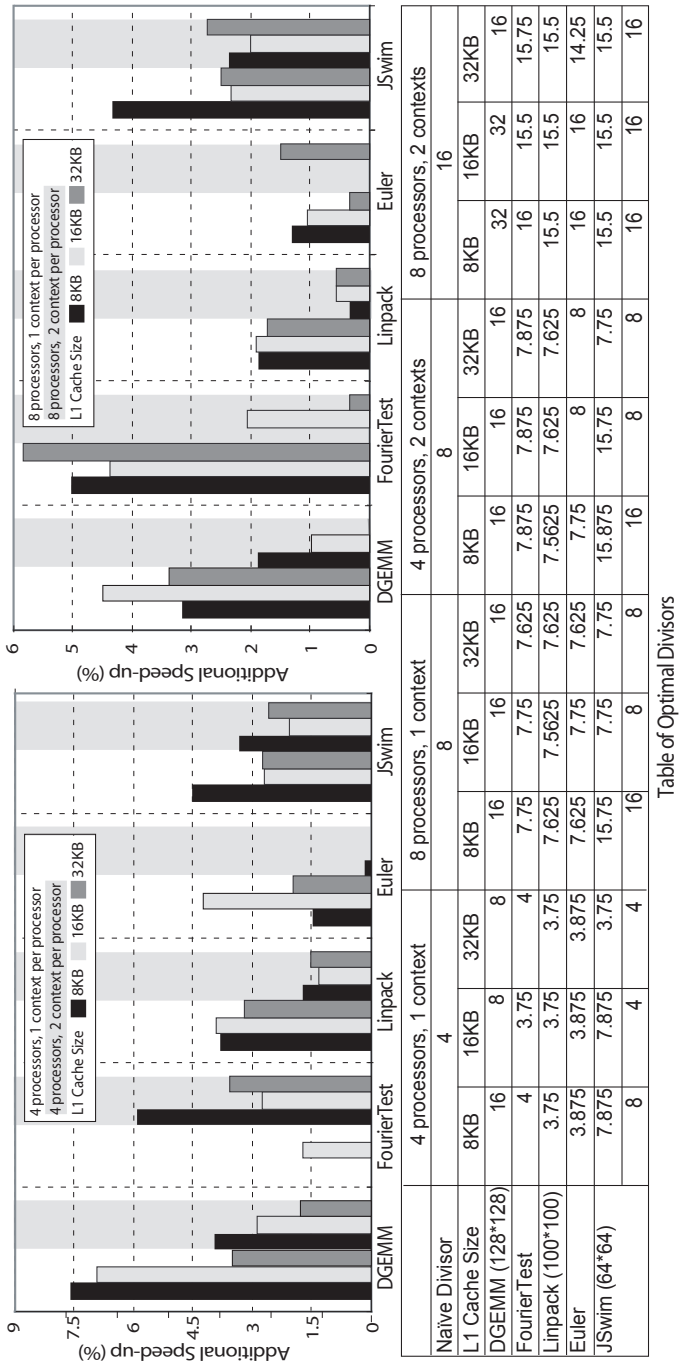
overhead. As mentioned previously, in Section 5.3.1, if the generator thread is not able to distribute a parallel thread to a remote context the work must be done by the generator which prevents subsequent threads being distributed. As the size of the chunk decreases with larger divisors, each parallel thread contains less work, which reduces the amount of serialization caused by context contention. For this reason in some cases a divisor greater than the number of processor contexts performs better.

Figure 5.10 shows the search result obtained from ACD. The *DGEMM* (96×96 matrix) and *JSwim* benefit from improving data locality by decreasing the chunk size. In Figure 5.10, *JSwim* has two rows, because there are two types of loops: one is a 2-dimensional nested loop which can benefit from data locality and another is a 1-dimensional loop which benefits from reducing the thread creation overhead.

5.5.2 Adaptive Tiling

Figure 5.11 demonstrates the OTF searching for an optimal divisor during execution of the *DGEMM* benchmark by applying Algorithm 12. The problem size is 256×256 matrix with a hardware configuration of 2 clusters and 4 processors per cluster — notation: 2c/4p [9]. The L1 cache is 8KB and L2 cache is 128KB — notation: 8KB/128KB. Compared with the search process shown in Figure

[9]For a single CMP, we employ a similar notation: 2p/4c means 2 processors and 4 contexts per processor.



	4 processors, 1 context			8 processors, 1 context			4 processors, 2 contexts			8 processors, 2 contexts		
Naïve Divisor	4			8			8			16		
L1 Cache Size	8KB	16KB	32KB	8KB	16KB	32KB	8KB	16KB	32KB	8KB	16KB	32KB
DGEMM (128*128)	16	8	8	16	16	16	16	16	16	32	32	16
FourierTest	4	3.75	4	7.75	7.75	7.625	7.875	7.875	7.875	16	15.5	15.75
Linpack (100*100)	3.75	3.75	3.75	7.625	7.5625	7.625	7.5625	7.625	7.625	15.5	15.5	15.5
Euler	3.875	3.875	3.875	7.625	7.75	7.625	7.75	8	8	16	16	14.25
JSwim (64*64)	7.875	7.875	3.75	15.75	7.75	7.75	15.875	15.75	7.75	15.5	15.5	15.5
	8	4	4	16	8	8	16	8	8	16	16	16

Table of Optimal Divisors

Figure 5.10: Speedup of ACD Compared to naïve Chunk Distribution.

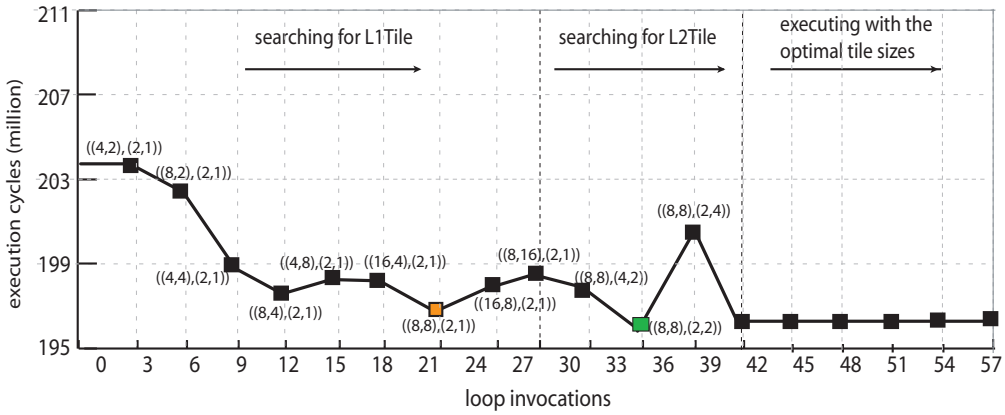
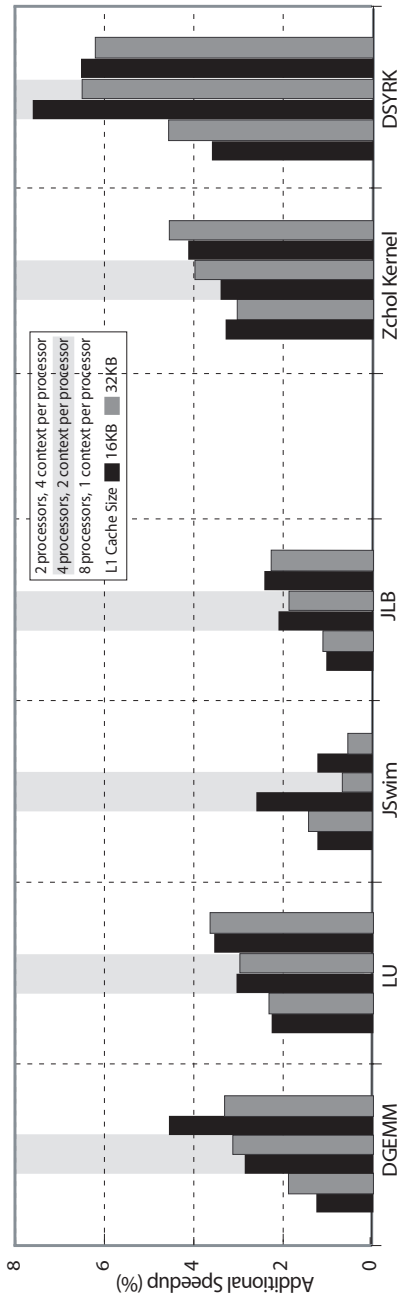


Figure 5.11: The Search Process for DGEMM 256*256.

5.9, this search process is a two-stage search which needs to find tile sizes for level 1 and level 2 caches. The search algorithm starts from the naïve scheme with tile divisors: L1 tile $(4, 2)$, and L2 tile $(2, 1)$. By the 21st invocation of the parallelized loop a locally optimal L1 tile size has been found, and the optimal L2 tile size is found at the 36th invocation. As mentioned in Section 5.2.2, three invocations of the loop are used to assess timing stability. In this experiment the deviation did not exceed the threshold (0.1). The optimal L1 and L2 tile sizes are applied at the 39th invocation finishing the search phase. Note that by the very nature of the hill-climbing algorithms used, the adaptive search finishes after finding local-optimal solutions.

Figure 5.12 shows the result for applying ATD on a single CMP. The hardware configurations are: 2p/4c, 4p/2c, and 8p/1c, and L1 cache sizes: 16KB and 32KB. The L2 caches are all 512KB. For *Zchol* and $DSYRK$ which have triangular iteration spaces, the speedup mainly benefit from improving the load balance. For the LU kernel, there are two rows of divisors corresponding to two groups of problem size. Because this LU kernel uses $DGETRF$ implementation [8] which contains $DGEMM$ operation, for each LU execution, the sizes of the internal $DGEMM$'s input matrix are variable. Here we simply group them into two sets, thus getting two groups of optimal divisors.

As introduced in Chapter 2, chip multi-threading employs a context switching mechanism which can help to hide memory latency. So the single context configurations (i.e. 8p/1c) usually have better speedup than the multiple context



	Loop	Naïve Divisor	2 processors, 4 contexts		4 processors, 2 contexts		8 processors, 1 contexts	
L1 Cache Size			16KB	32KB	16KB	32KB	16KB	32KB
DGEMM (128*128)	Regular	(4.0, 2.0)	(16.0, 8.0)	(8.0, 4.0)	(8.0, 8.0)	(8.0, 4.0)	(8.0, 4.0)	(4.0, 4.0)
LU (1024*1024)	Regular	(4.0, 2.0)	(8.0, 8.0)	(4.0, 4.0)	(8.0, 4.0)	(4.0, 4.0)	(8.0, 4.0)	(4.0, 4.0)
		(4.0, 2.0)	(4.0, 4.0)	(4.0, 2.0)	(4.0, 2.0)	(4.0, 2.0)	(4.0, 2.0)	(4.0, 2.0)
JSwim (400*400)	Regular	(4.0, 2.0)	(8.0, 4.0)	(8.0, 2.0)	(8.0, 2.0)	(8.0, 2.0)	(8.0, 2.0)	(8.0, 2.0)
JLB (128*128)	Regular	(4.0, 2.0)	(8.0, 4.0)	(8.0, 4.0)	(8.0, 4.0)	(8.0, 2.0)	(8.0, 4.0)	(8.0, 2.0)
Zchol (128*128)	Triangular	(4.0, 4.0)	(8.0, 4.0)	(8.0, 4.0)	(8.0, 4.0)	(8.0, 4.0)	(8.0, 4.0)	(8.0, 4.0)
DSYRK (128*128)	Triangular	(4.0, 4.0)	(8.0, 4.0)	(8.0, 4.0)	(8.0, 4.0)	(8.0, 4.0)	(8.0, 4.0)	(8.0, 4.0)

Table of Optimal Divisor Pairs

Figure 5.12: Adaptive Tiling with Different Hardware Configuration on a Single CMP.

configurations, because the hardware can eliminate some effect of a cache miss, so some of the benefits of tiling are hidden.

To evaluate the ACD on CMC, different problem sizes (64×64, 128×128, 256×256 and 352×352 matrix) and different hardware configurations (clusters/processors: 2c/4p and 4c/2p, and L1/L2 cache sizes: 8KB/128KB, 8KB/256KB, 16KB/128KB and 16KB/256KB) are used.

The graph in Figure 5.13 presents the speedup attained using the optimal tile sizes compared with that attained using naïve tile sizes. The naïve tile size is defined as the square root of the number of processor contexts. For example, a system with 16 processors has naïve L1 cache tile divisors $(4, 4)$. The divisors are restricted to integer values, thus in a system with 8 processors, the L1 cache tile could either be $(4, 2)$ or $(2, 4)$ (see Algorithm 10). The naïve scheme is used by static optimizers as it achieves reasonable load balance and data locality.

Figure 5.14 shows the resulting optimal divisors for all of the evaluated benchmarks. The naïve divisors for $DGEMM$ are: 2c/4p and 4c/2p with L1 tile divisor $(4, 2)$ and L2 tile divisor $(2, 1)$; 4c/4p with L1 tile divisor $(4, 4)$ and L2 divisor $(2, 2)$. The speedup for $DGEMM$ is shown in Figure 5.13(a). For small problem sizes (e.g. 64×64 matrix), there is no obvious benefit for those configurations with larger L1 cache sizes when compared to the naïve scheme. For larger problem sizes, however, larger divisors produce performance increases. The optimal L2 tile sizes are related to the number of clusters. For example, the best L2 tile divisors for 64×64 matrix are $(2, 1)$ for the 2c/4p configuration and $(2, 2)$ for the 4c/2p and 4c/4p configurations. By increasing the L2 cache size, the L2 cache tile has less effect and its value is near the naïve configuration. This is why the 256KB L2 cache configurations have less of a speedup than the 128KB L2 cache configurations for the same problem size and L1 cache size.

The $DTRMM$ nested loop is intrinsically load imbalanced, because the number of iterations in the inner most loop (k-loop) depends on the iteration of the i-loop, refer to Figure 5.8 (c). The optimal divisors are shown in Figure 5.14(b). For configurations 2c/4p and 4c/2p, most of the best divisors for the j-loop L1 tile sizes are 8, which is an even distribution of 8 parallel tasks to the 8 processing cores. Similarly, most of the best divisors for the j-loop L1 tile sizes are 16 for 4c/4p. By increasing the problem size, both the L1 and L2 divisors are increased

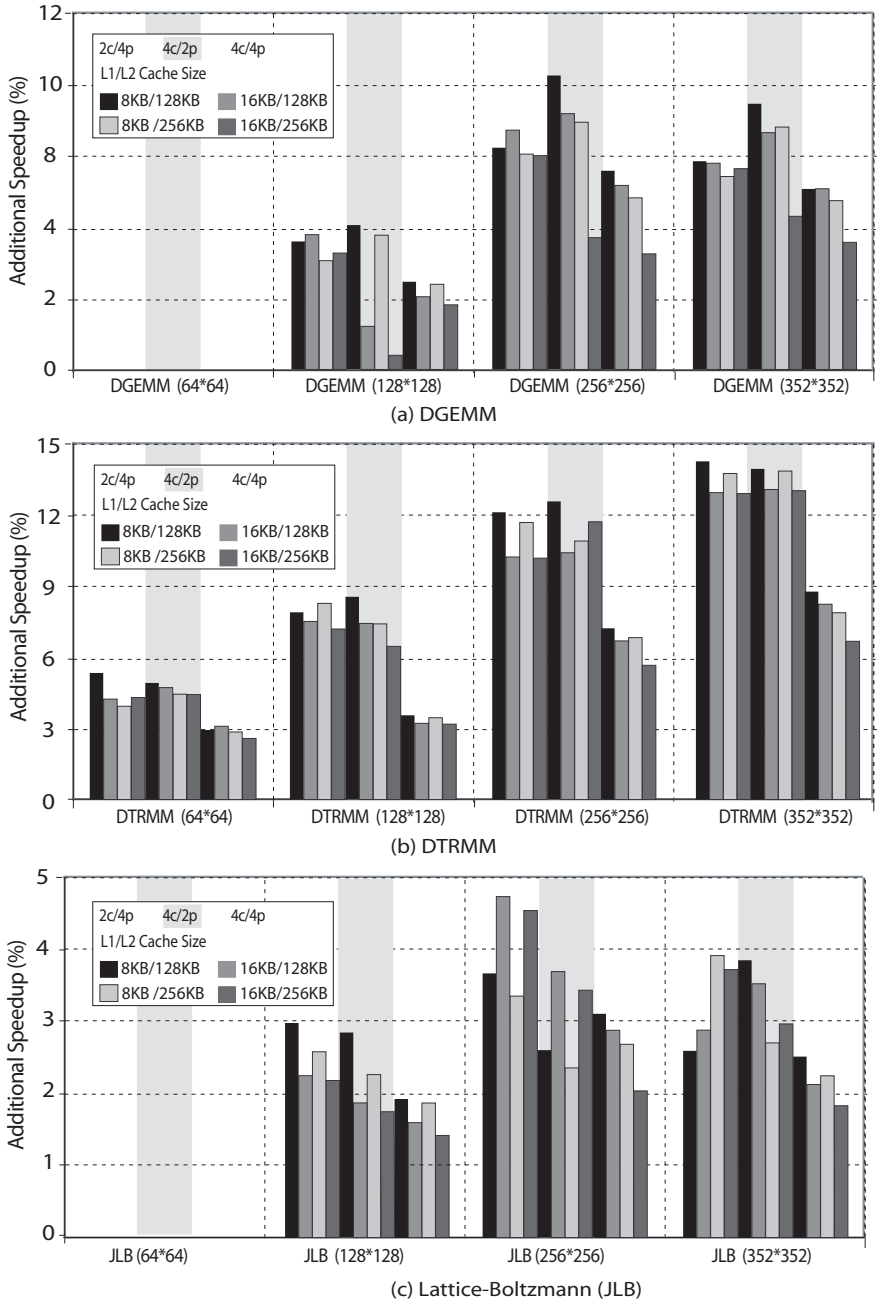


Figure 5.13: The Speedup Compared With a Naïve Tiling Scheme.

	2c/4p				4c/2p				4c/4p			
Naïve Divisor	(4,2) (2,1)				(4,2) (2,2)				(4,4) (2,2)			
L1/L2 Cache Size	8KB, 128KB	16KB, 128KB	8KB, 256KB	16KB, 256KB	8KB, 128KB	16KB, 128KB	8KB, 256KB	16KB, 256KB	8KB, 128KB	16KB, 128KB	8KB, 256KB	16KB, 256KB
64*64	(4,2) (2,1)	(4,2) (2,1)	(4,2) (2,1)	(4,2) (2,1)	(4,2) (2,2)	(4,2) (2,2)	(4,2) (2,2)	(4,2) (2,2)	(4,4) (2,2)	(4,4) (2,2)	(4,4) (2,2)	(4,4) (2,2)
128*128	(4,4) (2,1)	(4,2) (2,1)	(4,4) (2,1)	(4,2) (2,1)	(4,4) (2,2)	(4,2) (2,2)	(4,4) (2,2)	(4,2) (2,2)	(4,4) (2,2)	(4,2) (2,2)	(4,4) (2,2)	(4,2) (2,2)
256*256	(8,8) (2,2)	(4,4) (2,2)	(8,4) (2,2)	(4,4) (2,2)	(8,4) (2,2)	(4,4) (2,2)	(8,4) (2,2)	(4,2) (2,2)	(8,4) (2,2)	(4,4) (2,2)	(8,4) (2,2)	(4,2) (2,2)
352*352	(8,16) (4,4)	(8,16) (4,4)	(8,16) (4,2)	(4,4) (4,2)	(8,16) (4,4)	(8,8) (4,2)	(8,16) (4,2)	(8,4) (4,2)	(8,16) (4,2)	(8,4) (4,2)	(8,16) (4,2)	(4,4) (4,2)

(a) DGEMM

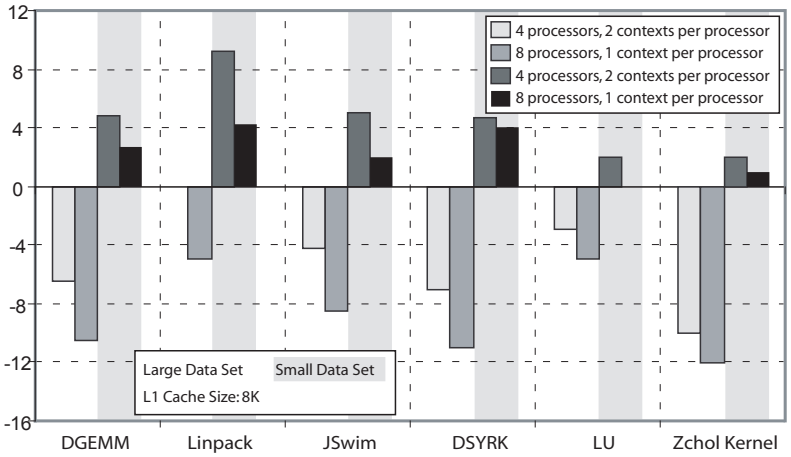
	2c/4p				4c/2p				4c/4p			
Naïve Divisor	(4,2) (2,1)				(4,2) (2,2)				(4,4) (2,2)			
L1/L2 Cache Size	8KB, 128KB	16KB, 128KB	8KB, 256KB	16KB, 256KB	8KB, 128KB	16KB, 128KB	8KB, 256KB	16KB, 256KB	8KB, 128KB	16KB, 128KB	8KB, 256KB	16KB, 256KB
64*64	(1,8) (1,2)	(1,8) (1,2)	(1,8) (1,2)	(1,8) (1,2)	(1,8) (1,4)	(1,8) (1,4)	(1,8) (1,4)	(1,8) (1,4)	(1,16) (1,4)	(1,16) (1,4)	(1,16) (1,4)	(1,16) (1,4)
128*128	(4,8) (1,2)	(2,8) (1,2)	(2,8) (1,2)	(2,8) (1,2)	(2,8) (1,4)	(2,8) (1,4)	(2,8) (1,4)	(2,8) (1,4)	(1,16) (1,4)	(1,16) (1,4)	(1,16) (1,4)	(1,16) (1,4)
256*256	(4,8) (2,2)	(4,8) (2,2)	(4,8) (1,2)	(4,8) (1,2)	(4,8) (1,4)	(4,8) (1,4)	(4,8) (1,4)	(4,8) (1,4)	(2,16) (1,4)	(2,16) (1,4)	(2,16) (1,4)	(1,16) (1,4)
352*352	(8,8) (4,2)	(4,8) (4,2)	(8,8) (2,2)	(4,8) (2,2)	(8,8) (2,4)	(4,8) (2,4)	(8,8) (1,4)	(4,8) (1,4)	(2,16) (2,4)	(2,16) (2,4)	(2,16) (1,4)	(2,16) (1,4)

(b) DTRMM

	2c/4p				4c/2p				4c/4p			
Naïve Divisor	(4,2) (2,1)				(4,2) (2,2)				(4,4) (2,2)			
L1/L2 Cache Size	8KB, 128KB	16KB, 128KB	8KB, 256KB	16KB, 256KB	8KB, 128KB	16KB, 128KB	8KB, 256KB	16KB, 256KB	8KB, 128KB	16KB, 128KB	8KB, 256KB	16KB, 256KB
64*64	(4,4) (2,1)	(4,2) (2,1)	(4,2) (2,1)	(4,2) (2,1)	(4,2) (2,2)	(4,2) (2,2)	(4,2) (2,2)	(4,2) (2,2)	(4,4) (2,2)	(4,4) (2,2)	(4,4) (2,2)	(4,4) (2,2)
128*128	(4,8) (2,2)	(4,8) (2,2)	(4,8) (2,1)	(4,8) (2,1)	(4,8) (2,2)	(2,4) (2,2)	(4,8) (2,2)	(2,4) (2,2)	(4,8) (2,2)	(2,4) (2,2)	(4,8) (2,2)	(2,4) (2,2)
256*256	(4,8) (2,2)	(8,16) (4,2)	(8,8) (4,2)	(8,16) (4,2)	(8,16) (4,2)	(4,8) (4,2)	(8,8) (2,2)	(4,8) (2,2)	(8,8) (2,2)	(4,8) (2,2)	(8,8) (2,2)	(4,8) (2,2)
352*352	(8,8) (4,2)	(8,16) (4,4)	(8,8) (4,4)	(8,16) (4,2)	(8,16) (4,2)	(4,8) (4,2)	(8,16) (2,2)	(4,8) (2,2)	(8,8) (2,2)	(4,8) (2,2)	(8,8) (2,2)	(4,8) (2,2)

(c) Lattice-Boltzmann (LB)

Figure 5.14: Optimal Divisor Pairs for Different Problem Sizes and Hardware Configurations.



Benchmark	small data set	large data set
DGEMM	32 x 32	128 x 128
Linpack	90 x 90	150 x 150
JSwim	32 x 32	128 x 128
DSYRK	64 x 64	128 x 128
LU	64 x 64	128 x 128
Zchol Kernel	64 x 64	128 x 128

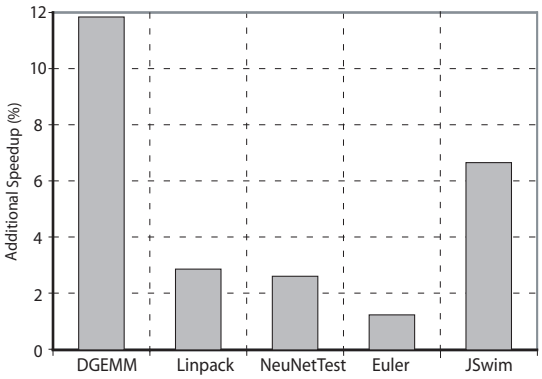
Figure 5.15: Speed-up from CRD on Top of Initial ACD/ATD Gains.

to gain additional benefits from data locality. The speedups, shown in Figure 5.13(b), are more pronounced than for $DGEMM$; however, most of the benefit is attained through better load balancing.

Finally the optimal divisors for JLB, which uses a stencil computation model, are compared to the naïve tile sizes, which are the same as for $DGEMM$. The speedups are shown in Figure 5.13(c). Compared with $DGEMM$, JLB has less cache cross-interference and as a consequence the optimization produces smaller speedups.

5.5.3 Adaptive Version Selection

Figure 5.15 shows the additional performance speedups gained by using CRD with the ACD/ATD optimizations. For each benchmark both a large and small data set were evaluated. Clearly for larger data sets the CRD scheme degrades the performance of the best ACD/ATD using traditional distribution which can improve the data locality; however, for smaller datasets additional performance increases are achieved for most of the benchmarks. CRD gains performance for smaller data sets as it uses a tree-like distribution policy to create parallel threads. This reduces the overhead of thread creation in the initial generator thread. Furthermore, when CRD is used on triangular loops (e.g. $DSYRK$ and $ZcholKernel$), this cyclic distribution leads to less variation in the total amount



Benchmark Name	No. of Unrolled Loops	Loop Unrolling Factor
DGEMM	1	3
Linpack	1	3
NeuNetTest	2	2
	4	3
	2	4
Euler	4	1
	2	4
JSwim	2	1
	3	2
	2	3
	2	4

(b)

Figure 5.16: Speed-up from ALU on Top of ACD Gains.

of work received by each processor context.

Figure 5.16 (a) shows the result of the ALU optimization tested on a 4 processor single threaded configuration. The benchmarks selected here have a simple innermost loop, [10] which is suitable for loop unrolling in our compilation environment. The speedup gained from ALU is on top of normal ACD gains. Figure 5.16 (b) lists the unrolled loops and the loop unrolling factors for each benchmark. Only the parallelized loops can be unrolled in our current ALU implementation.

In *DGEMM*, the innermost loop which computes dot products on vectors is the major computation process, so it gains obvious speedup. In *Euler* and *JSwim*, some loop unrolling factors are 1; this means these loops were not unrolled, because they contain a lot of instructions, and thus generate more register pressure when they are unrolled.

5.6 Summary

This chapter introduced the motivation for runtime adaptive optimization (runtime empirical searching) and the *Online Tuning Framework* which provide the support for runtime adaptive optimization in JaVM. By applying several adaptive optimizations, loop-based parallel programs could achieve better load balance or

[10] As the limitation is described in Section 5.3.3, the simple loop here is the loop which contains no conditional branch operations and the loop body is just one basic block.

data locality, and thus improve their runtime performance.

Chapter 6

Support for Thread Level Speculation

As discussed in previous chapters, dynamic compilation with runtime lightweight thread support is feasible to exploit parallelization especially for fine-grain parallelization. The experimental results show that the performance improvement for general purpose applications is still limited. The major obstacle is that potential data dependence is not statically analysable.

Recent proposals for CMPs advocate *Thread Level Speculation* (TLS), which means splitting sequential execution programs into implicit threads and executing them in parallel on multiple processor cores speculatively. TLS has the advantage that it can simplify the compiler analysis and exploit more potential parallelism [71]. To evaluate adaptive optimization for speculative parallelization, new TLS support is added into the JAMAICA CMP. This chapter will introduce the TLS solution for both hardware and software.

Section 6.1 gives an overview of the basic background of the TLS solution. Section 6.2 introduces the basic architecture of TLS support in JAMAICA and its execution model. Sections 6.3 and 6.4 discuss how to support TLS in JAMAICA by adding components to the hardware and the compiler. Section 6.5 shows some experimental results based on standard Java benchmarks and discusses the issues which affect the performance. Section 6.6 summarizes this chapter.

6.1 TLS Overview

Parallelization within a single application can come from explicitly parallel sections, but it is difficult for programmers to parallelize applications manually. Although compilers are relatively successful at parallelizing numerical applications, dependences that are not statically analysable hinder compilers. To alleviate this problem TLS which supports speculative parallelization exploits the parallelism implicit in an application's sequential instruction stream.

6.1.1 Basic Concepts

The data dependency between the parallel threads would violate the order of memory access in sequential programs, so a data dependency related memory read/write should be *Speculative Read* (SR) and *Speculative Write* (SW). For SR, the TLS mechanism needs to read the latest value which follows the original program order. For SW, it should provide temporary storage to store the written data and commit it to real memory later. The SW also needs to check if there are any load operations from the same address by a succeeding thread.

Only true memory dependences *Read After Write* (RAW) cause violations. The anti dependences *Write After Write* (WAW) and output dependences *Write After Read* (WAR) are properly handled by buffering in speculative storage and committing in sequential order. All of the speculative memory write operations have temporary storage and the stored values will be committed when a speculative session finishes. The commit will work sequentially in the order defined by the original program.

The speculative CMP employs data dependence tracking mechanisms which use hardware to detect data dependence violations, keeps uncertain data in speculative storage, rolls back incorrect executions, and commits data (the speculative status stored in speculative storage) to main memory only when speculative threads succeed. Thus, the speculative CMP provides the same programming interface as a normal CMP while supporting the safe, simultaneous execution of potentially dependent threads.

By using hardware to enforce dependence, the speculative CMP allows a compiler

to focus on improving performance without needing to fully prove data independence between the parallelized regions. A sequential program is decomposed into threads for a speculative CMP to execute in parallel; dependent threads cause performance degradation (because a thread whose memory operations violate the data dependency should rollback the memory write operation and restart execution) but do not affect correctness. The CMP executes a number of threads in parallel while enforcing correctness, the program's output is consistent with its sequential execution.

6.1.2 Where to Speculate?

There are two basic targets for speculative parallelization: method-level (or block-level) speculation and loop-level speculation. Method-level speculation will run a called method non-speculatively and then speculatively run the code following the method call. This approach can be used for other control-flow blocks, such as speculatively executing the code following an if-then-else statement. Loop-level speculation speculates that loop iterations are independent. Loop-level speculation has the following advantages:

- Implementation: method-level speculation may need all operation running speculatively in hardware or complex context analysis to decompose the sequential programs and determine where speculative operations are necessary. For loop-level speculation, the compiler needs to simply analyze the loop body. So, loop-level speculation is more easier to implement in the compiler if hardware support is limited.
- Load Balance: distributing loop iterations among processor evenly can achieve better load balance than performing method-level parallelization which depends on the granularity of each parallelized method.
- Parallelism: loops are the major target of parallelization for most current applications [72].

By considering the issues listed above and the LPC which has been already, we choose the loop-level speculation in this work. More method-level speculation

related adaptive optimizations will be implemented and evaluated in future work see Section 8.2.2.

6.1.3 Basic Terminolgies

Here are defined several basic terms that are used in TLS operations:

- *Epoch ID*: an epoch ID is used to maintain the sequential order of speculative parallelization. The speculative thread creator will assign each speculative thread an epoch ID in sequential order to maintain the correctness of a sequential execution. The epoch ID is an integer value incremented automatically every time a speculative thread is created (0, 1, 2 ...).

- *Speculative Session*: the execution of a speculatively parallelized code segment. The speculative session is composed of speculative threads which are executed in parallel.

- *Speculative Thread*: a lightweight thread (same as JaVM lightweight thread) that is used to exploit speculative parallelization within JaVM. Each parallel thread executes a parallelized code segment speculatively (i.e. the reads/writes on memory are speculative reads/writes) and has a globally unique epoch ID.

- *Speculative* and *Non-Speculative*: In speculative parallelization, all of the parallel threads should submit their modified data sequentially in order to maintain the sequential program's semantics. To achieve this aim, all of the parallel threads are defined as in one of the two states: non-speculative and speculative. There is only one non-speculative thread at any time, and only this non-speculative thread could commit its modified data from temporary storage to memory. As epoch ID is used to maintain sequential order, each parallel thread (speculative or non-speculative) has its own epoch ID in one speculative session. A parallel thread is the non-speculative thread only if it has the smallest epoch ID in the current speculative session.

- *Home Free Token* (HFT): A token which annotates which thread is the non-speculative thread. The token will be passed in the order of epoch ID,

so a committed non-speculative thread can pass the non-speculative state to its successor.

- *Speculative Session ID* (SID): is equal to an ID of the Java thread[1] which creates the speculative threads. As there may be more than one Java thread which can create parallel threads speculatively, this SID could be used to identify these different Java threads. A speculative thread will be identified by the combination of a SID (to identify which creator thread it belongs to) and an epoch ID (to maintain the sequential execution order).

6.2 JaTLS Architecture

JAMAICA TLS (JaTLS) is designed and implemented to support TLS on the JAMAICA CMP architecture. The aim of this design is to use a simple extension to support TLS efficiently at both hardware and software level.

6.2.1 Basic Structure

Basically, two issues need to be considered for TLS support:

1. Speculative Storage and State Management: including the speculative read-/write, the data dependence violation detection, and the speculative state transformation (i.e. state transformation from *speculative* to $non-speculative$).
2. Speculative Parallel Threads Management: including speculative thread creation, commit and restart (rollback).

Speculative Storage and State Management

JaTLS employs a centralized *Speculative Memory Table* (SMT) which contains several components used for managing speculative storage and tracing dependence violation (discussed in Section 6.3). The SMT is an extended module for the

[1]In the main thread which creates the branch threads, the SID equals the srcID in the context register. In the branch thread which is created by the main thread, the SID equals the value obtained by JAM_RCR -1 (see Chapter 4).

normal JAMAICA CMP (shown in Figure 6.1), it interacts with the L1 cache to load/store data. As the speculative states are maintained in SMT in the JaTLS solution, there is no modification to the normal JAMAICA CMP's cache coherence and consistency mechanism. JaTLS also extends a new context register for each processor context to maintain the speculative state information.

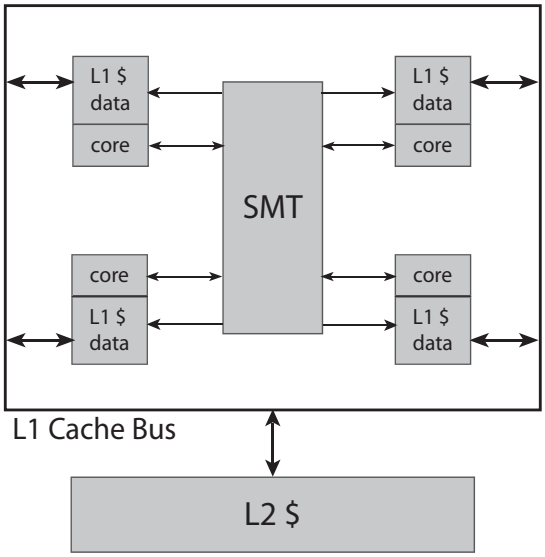


Figure 6.1: Speculative Memory Table in JAMAICA CMP.

Speculative Parallel Thread Management

The speculative thread management is performed by software. JAMAICA ISA is extended by adding 6 more speculation related instructions, thus the compiler can generate the code for controlling speculative thread creation, commit and restart (discussed in Section 6.4). This TLS related code generation is a functional extension of LPC (see Chapter 4), and the target of parallelization is still the loop in this solution.

By employing this simple and clear interface between hardware (i.e. centralized SMT) and software implementation (i.e. the extended ISA), JaTLS provides a flexible model for parallelization; the compiler can mix both speculative parallelization and normal parallelization together.

6.2.2 Execution Model

The execution model of JaTLS is composed of 6 basic speculative operations:

- *Speculative Thread Registration* (STR): enter speculative execution, the hardware will allocate corresponding resources to maintain the speculative state for this thread.
- *Speculative Execution* (SE): execute the code segment which has SR/SW operations, the hardware will check data dependence violation automatically and store the violation state into a thread local context.
- *Violation Check* (VC): get the current thread's violation state.
- *Rollback (Restart) Speculative Execution* (RSE): if there is a data dependence violation (detected by the *Check Violation* operation), the current thread needs to be collapsed and restarted.
- *Commit Speculative States* (CSS): if there is no data dependence violation, commit the speculative states (i.e. all of the speculative write data). If the HFT is not ready (i.e. the current thread is not non-speculative), the current speculative thread will be blocked and wait for the HFT.
- *Speculative Thread Deregistration* (STD): exit the speculative session, the hardware will release the resource allocated for the current session and pass HFT to the next speculative session.

These speculative operations are implemented in hardware (discussed in Section 6.3). The related instructions are provided, so a compiler can generate the code that controls the speculative threads (discussed in Section 6.4).

This execution model employs a *Lazy Collapse* model, which means that the thread collapse and restart can not happen until the collapsing thread has run to completion.

Figure 6.2 gives an example of JaTLS's execution model. A sequential program is decomposed into 4 speculative threads: $T0$, $T1$, $T2$, and $T3$. $T0$ and $T1$ finished at the same time but $T0$ holds HFT. $T1$ has to wait until $T0$ committed all of its speculative data. $T2$ also finished at the same time as $T0$, but $T0$ triggered a

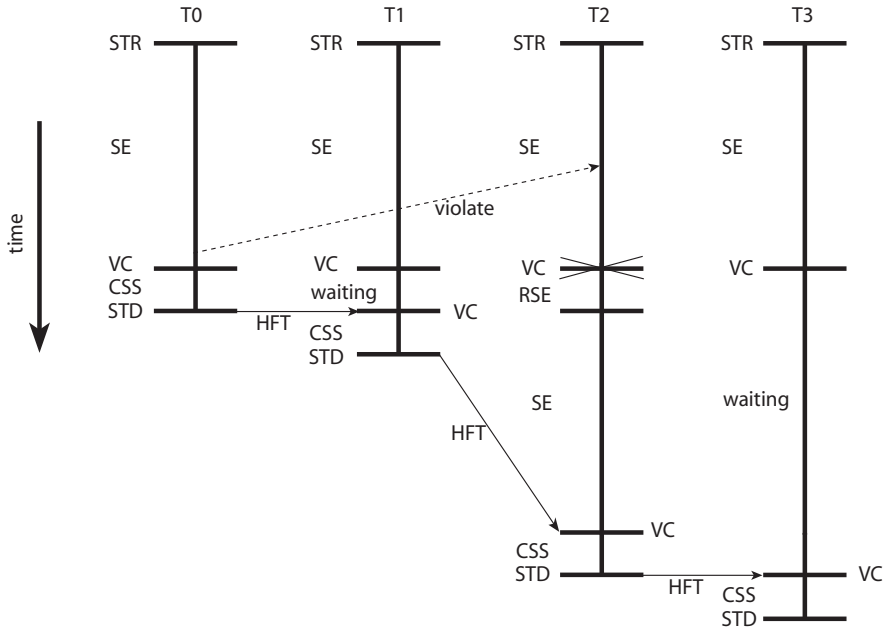


Figure 6.2: The Execution Model of JaTLS.

true dependence violation during $T2$'s execution, $T2$'s violation checking phase detected this violation and restarted execution. $T3$ keeps waiting until it gets the HFT from $T2$.

6.3 Hardware Components

To enable the hardware mechanism to detect data dependence violations and speculative non-speculative state transformations, JaTLS employs several hardware components. The following subsections will explain how these hardware components could cooperate together to implement the speculative operations.

6.3.1 Speculation Memory Table

The *Speculation Memory Table* (SMT) is the main functional module which is used to provide the speculative storage and maintain the speculative state transformation. It is composed of several *Memory Control Unit*s (MCU) (each processor context has one corresponding MCU), a *Multi-Version Cache* (MC), an *epoch Index Table* (EIT) and a series of *read/write log buffers* (LB) (one LB per processor context). Figure 6.3 (a) shows a SMT cooperating with a 4 processor CMP (one context per processor).

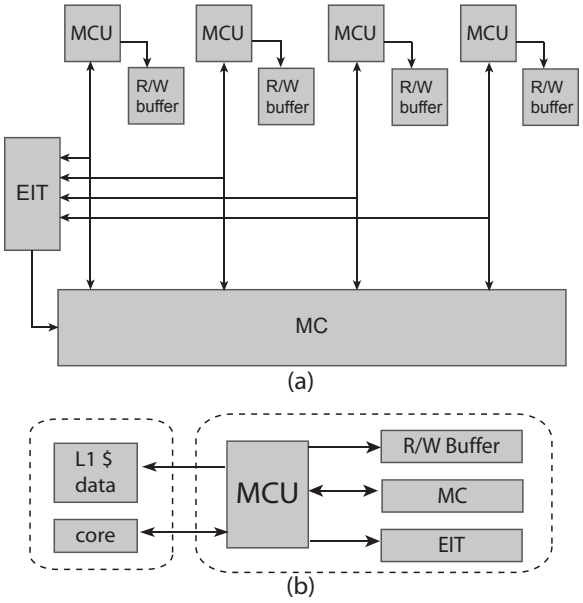


Figure 6.3: Speculative Memory Table and Memory Control Unit.

The MCU acts as an interactive interface between the processor contexts and SMT (shown in Figure 6.3 (b)). All of the speculative thread related instructions will be handled by the MCU. The number of MCUs depends on the number of processors. Figure 6.4(a) shows the input data to MCU, including SID, epoch, memory address and data (if the executing instruction is SW).

The MC provides temporary storage for all of the speculative threads, and also maintains a global SR/SW state word for each cache line. As the maximum number of speculative threads equals the number of processors in CMP, so each MC cache line has n data fields which correspond to the n processors. Figure 6.4 (b) shows a cache line structure in MC with a 4 processor CMP configuration:

one *Speculative Read State*(SRS) field, one *Speculative Write State* (SWS) field and 4 data fields.

The EIT is used to manage the HFT which performs speculative and non-speculative state transformations among the speculatively parallel threads (discussed in Section 6.3.2).

The SRS and SWS are used to maintain the SR/SW information of the MC cache line. In each SRS and SWS state, the number of data bits equals the number of processor contexts. There are n bits corresponding to n processors. The SR and SW can use this information maintained by SRS and SWS to find the latest data modification and check true dependence violations (as will be discussed in Section 6.3.4).

The MC cache line is identified by the memory address field. The index of data field is identified by SID and epoch ID. Figure 6.4(c) shows how to fetch a cache line in MC. The epoch ID is used to figure out which state bit needs to be set and which data field needs to be filled (epoch ID *mod* number of data fields).

The log buffer is used to store the memory address and data for each SR/SW operation, and the stored SW data will be written back to L1 cache when the speculative thread finishes its job and changes to a non-speculative thread. Each processor context has its corresponding log buffer.

6.3.2 EIT Structure

The EIT maintains the speculative threads information which helps the SR operations and SW operations figure out the latest modification and check true dependence violation. Figure 6.5(a) shows a row in EIT. There are 4 fields:

- *epoch ID*: stores the epoch value which the current data slot corresponds to.
- *State and Data Field Index*: indicates state bit which needs to be set in the state word (SRS and SWS), also indicates which data field needs to be read or written.

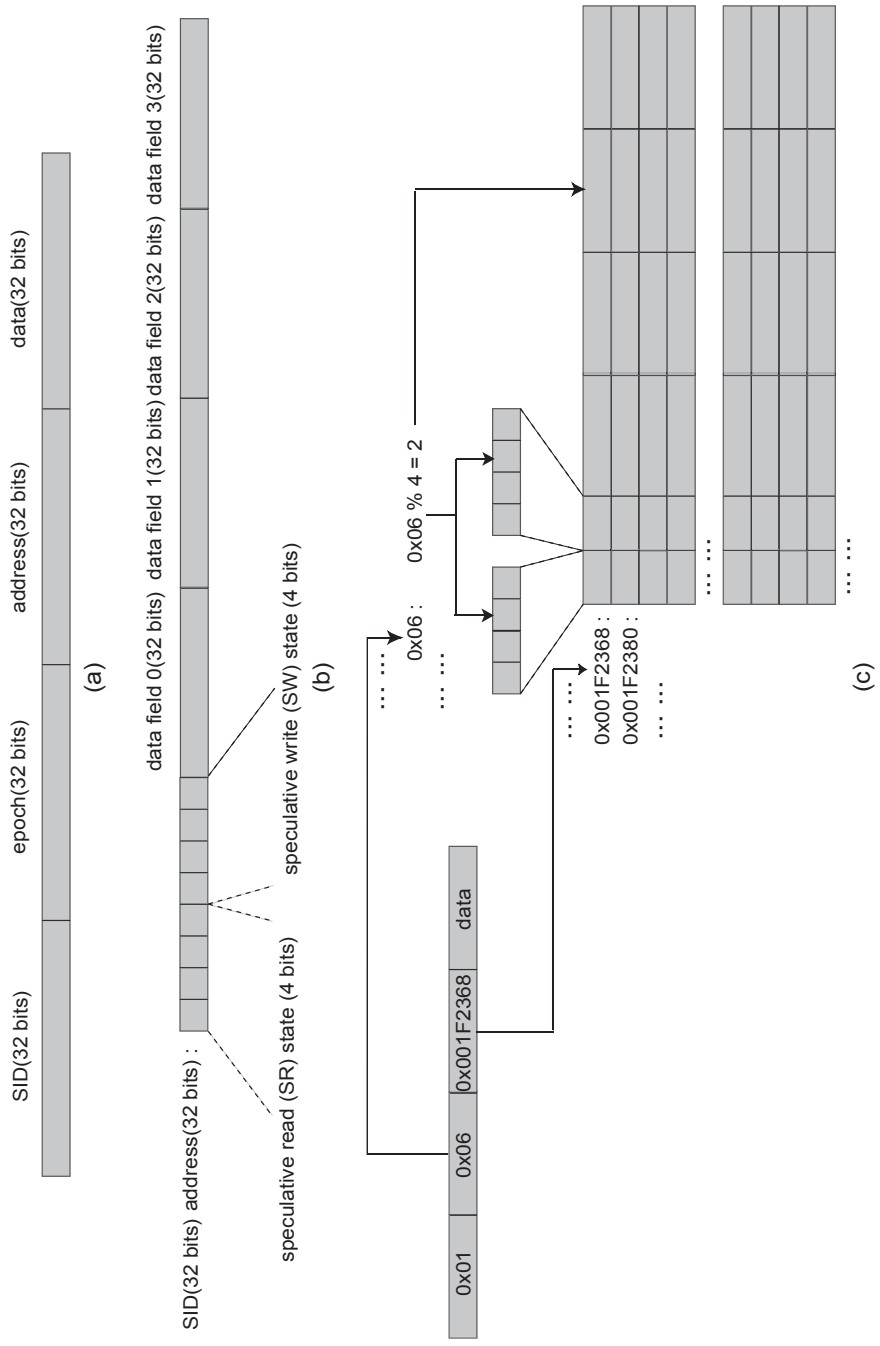


Figure 6.4: SMT Related Data Structure for 4-processors CMP.

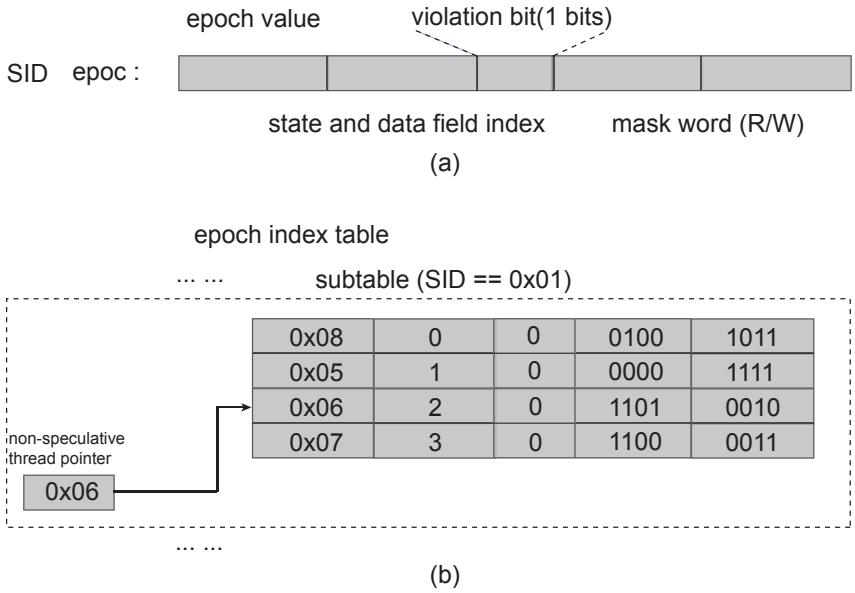


Figure 6.5: Epoch Index Table.

- *Violation State Bit*: indicates whether the current speculative thread has true dependence violation.
- *Mask Word*: performs data mask on SRS and SWS words; the result will be used to check data dependence violation. The mask word has two parts: one read mask and one write mask which correspond to SRS and SWS. The write mask is the inverted version of the read mask.

For a CMP system which has n processor contexts, the corresponding EIT should have n sub-tables and n data slots per sub-table. An example which has a 4 processor CMP configuration is shown in Figure 6.5 (b). The EIT's sub-table corresponds to speculative threads, so there is a map mechanism between the SID and the sub-table.

To implement the function of HFT, an additional data structure in the EIT sub-table is employed: Non-Speculative Thread Pointer (NSTP) which is used to point to the epoch ID which corresponds to the non-speculative thread (shown in Figure 6.5 (b)) in current SID. The NSTP performs the management of the speculative/non-speculative state transformation. Every time the NSTP is changed to point to a new epoch ID, it is equivalent to passing the HFT to the next speculative session.

6.3.3 State Transformation

As introduced in Section 6.1, JaTLS uses epoch ID to maintain the state transformation between non-speculative thread and speculative thread. EIT uses the NSTP that points to the data slot which corresponds to the non-speculative thread's epoch in the current speculative session. If a speculative thread wants to check if its state has been transferred to non-speculative, it just needs to query EIT and check if the NSTP is pointing to its epoch. The NSTP will be set in two scenarios:

- The speculative session's first speculatively parallel thread registers in EIT: as the first thread's epoch ID in a speculative session must be assigned 0 and epoch 0 must be a non-speculative thread, the NSTP should point to epoch 0's slot (shown in Figure 6.6 (a)).
- The current non-speculative thread deregisters from EIT: this means that the current non-speculative thread has finished its task and committed data, then its first succeeding speculative thread should be the new non-speculative thread. So the NSTP just moves itself to next slot in the EIT sub-table, and increments the epoch value (stored in NSTP) by 1 (shown in figure 6.6 (d)).

6.3.4 Mask Word and State Word

The state word (SRS and SWS) is used to annotate the current speculative read and speculative write states for its corresponding address in MC. The SR operation sets a state bit in SRS and SW operation sets a state bit in SWS every time they access a memory address. To check the state bits efficiently, we employ the mask word.

An SR operation needs to know if its preceding threads or current thread itself did store data in the same address, then it can read the latest value from MC. So the *write mask* word (WM) needs to get all of the SWS's state bits whose corresponding epochs are not larger than the current thread's.

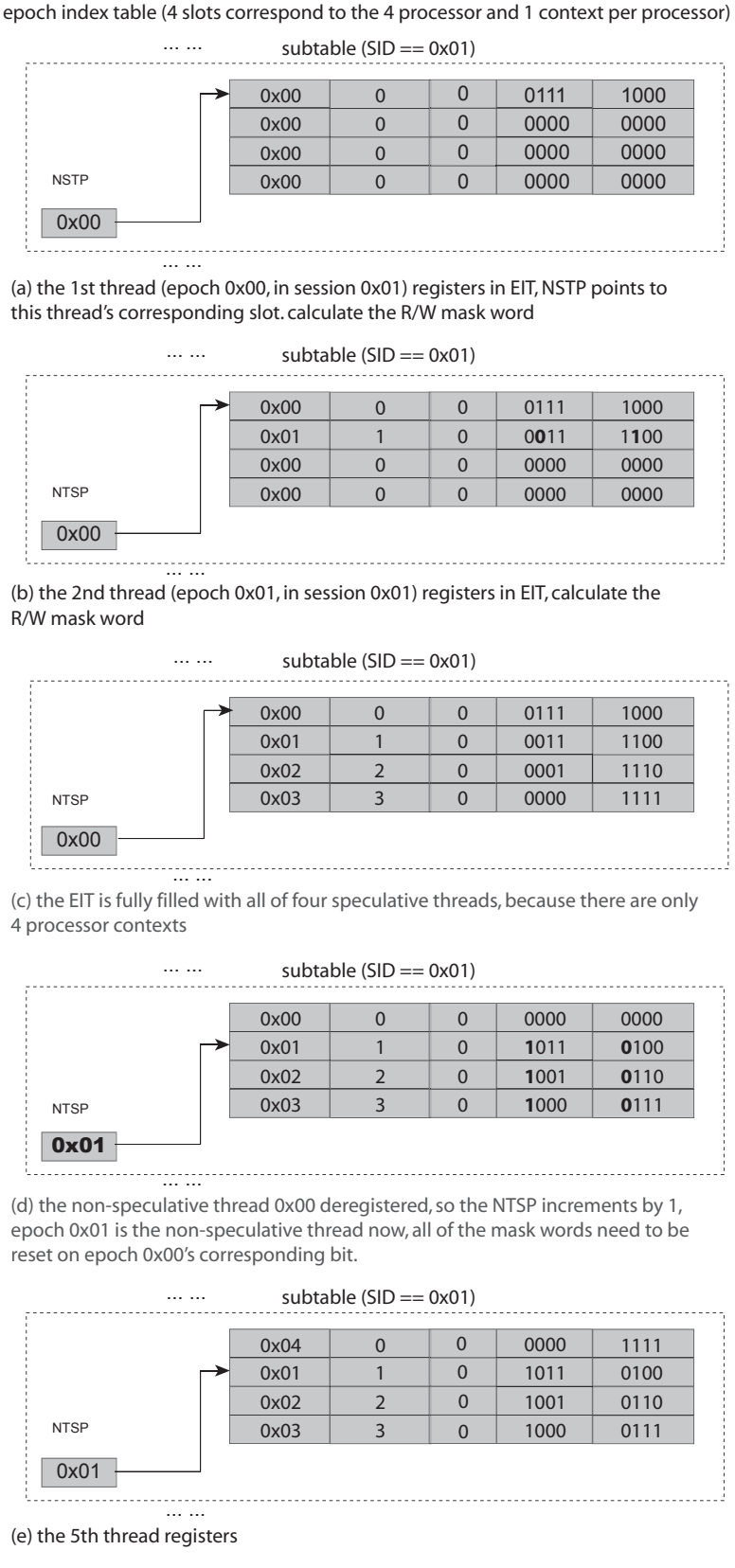


Figure 6.6: State Transformation in EIT.

For SW operation, it is necessary to know whether any succeeding thread did load data from this address. So the *read mask* word (RM) needs to get all of the SRS's state bits whose corresponding epochs are larger than current thread's. And the RM is an inverted value of WM.

Here is an example for mask word generation (see Figure 6.6 (c)). Given a 4 bit mask word (for a 4-processors CMP), the non-speculative thread epoch's offset is 0, and current speculative thread epoch's offset is 2. So the WM word should be *1110*, because the current speculative thread epoch's preceding epochs' offset is 0 and 1. The RM word should be *0001*, because the succeeding epoch's offset is 3. Doing an *AND* operation between write mask and SWS (WM & SWS) can check whether any preceding thread or current thread did write data into the current address, and doing an *AND* operation between read mask and SRS (RM & SRS) can check whether any succeeding threads did read data from the current address (to check true dependence violations).

If the non-speculative thread has finished and deregistered from EIT, other threads' mask words need to be recalculated, because the non-speculative pointer would be set to the next speculative thread (discussed in Section 6.3.3).

Here is an example for mask word recalculation (see Figure 6.6 (d)): for the speculative thread whose epoch's offset is 2, its WM is *1110*, the new WM should be *0110* (just clear the bit which corresponds to the deregistered thread's epoch), because the non-speculative thread's offset moves one step. To obtain the new SR, simply invert WM and get *0001*. So the new RM is *1101*.

6.3.5 Violation Counter

A violation counter register is employed to count the number of data dependence violations for each speculative session. Every time the violation bit is set in EIT's data slot, the violation counter register's value will be incremented automatically. And the value will also be incremented when speculative storage overflows happen (discussed in the next section). This value can be picked up by an extension of a speculative instruction, and will be used to drive adaptive optimization for speculative parallelization (discussed in the next chapter).

6.3.6 Speculative Storage Overflow

As the size of MC and LB are fixed, they may be overflowed when the speculatively parallel threads perform memory intensive operations. Here the hardware mechanism needs to handle speculative buffer overflow to allow the speculatively parallelized program to go on.

If the overflow happens on the log buffer, the SMT should obey the following constraints:

- If the overflow is in the non-speculative thread's log buffer, then that thread performs a commit operation automatically to free the space, and continues its execution. The violation counter register should be incremented by 1, because the overflow event can be treated as performance degradation and this information will be utilized by adaptive optimizations.
- If the overflow is in a speculative thread's log buffer, that speculative thread should be blocked on the SR/SW operation (because only the SR/SW need to insert a new element into the buffer), until it is transferred to non-speculative.

Because the LB is per-processor (i.e. each speculative thread has one LB), so the middle term commitment will not violate the correctness of the sequential program.

If the overflow happens on the MC, that means no more MC lines are available for any SR/SW, the SMT should obey the following constraints:

- All of the speculative threads should be blocked when they try to allocate a new cache line in MC.
- The non-speculative thread can continue its execution, but its behaviour of SR/SW operations will change. For SW, the non-speculative thread can write data to L1 cache directly if there is no corresponding cache line in the MC. For SR, there is no need to allocate a new MC cache line if there is no corresponding cache line in the MC. The violation counter register's value should be incremented.

- When the non-speculative thread finally commits and its succeeding speculative thread becomes the non-speculative thread, all of the speculative threads should be able to continue unless another overflow happens.

6.3.7 Speculative Operations

Speculative Session Registration and Deregistration

The speculative thread needs to register itself with EIT, so EIT can help any SR find the latest modification, and help any SW operation check true dependence violations. When a speculative thread finishes its job and its status is changed to non-speculative, it should commit data and deregister from EIT. Otherwise, a speculative thread has to wait until all of its preceding threads have finished and deregistered.

To register a speculative thread with SMT, the processor must provide the SID and epoch ID to EIT, then EIT can establish a map between the tuple (SID, epoch ID) and the other tuple (state bit and data field offset, violation state bit and mask word).

Both the registration and deregistration operations are atomic here. If two or more processors compete to register speculative threads, or some processor wants to register and another processor want to deregister, all of the requests should be queued and executed one by one, otherwise the EIT can not maintain the correct status.

For speculative thread registration (shown in Figure 6.6):

1. Use the SID to find the corresponding sub-table.
2. In the sub-table, find the epoch's corresponding slot (the offset of the slot is got by (epoch *mod* table size)). If this is the first slot being allocated (epoch equals 0), goto step 3, else goto step 4.
3. The current thread which sends this request should be a non-speculative thread (because the epoch starts from 0, so epoch 0 corresponds to a non-speculative thread definitely) and set the non-speculative thread pointer to this slot.

4. Set the state offset, and data field index. Both of these two values equal the current slot's offset in the sub-table. Set the violation bit as 0.

5. Generate the mask words.

The non-speculative thread deregistration operation should be the last step which is performed after a commit operation.

Speculative Write

1. Use the memory address to identify the cache line in MC (shown in Figure 6.3 (c)). If the corresponding cache line is not available, allocate one. Use epoch[2] to identify which state bit needs to be set and which data field needs to be modified.

2. Write data to the corresponding data field and set the corresponding state bit in SWS and load the state word (both of SRS and SWS) into MCU's temporary registers.

3. Use epoch to find the corresponding mask word in EIT, then check whether there are any SR operations whose epoch is larger than the current thread's epoch (explained in Section 6.3.4). This means that one or more succeeding threads have read data from the same memory address.

4. If there are such epochs, then annotate the epochs' corresponding speculative threads as violated by setting the violation bit in EIT.

5. Store the memory address and data value into the corresponding write buffer.

Note that step 1's cache line allocation operation and step 2 (setting state bit, writing data field and loading state word) are atomic operations. Because the n MCUs work simultaneously, two MCUs may try to allocate the same cache line. To avoid a duplicated allocation, the cache line allocation operation must be treated as an atomic operation.

[2]To find the corresponding state bit and data field,simply apply the module operation on epoch with the number of processors, e.g. epoch = 0x06, the index = 0x06 *mod* 4 = 2 (4 processors CMP).

Step 2 is an atomic operation between SR operations and SW operations on the same cache line. This means the state words (SRS and SWS) are locked when the SW operation is setting the state bit and write data on one cache line, any SR operations can not set the state bit on SRS and load the SWS on the same cache line.

Speculative Read

1. Use memory address to identify the cache line in MC (shown in Figure 6.3 (c)). If the corresponding cache line is not available, allocate one. Use epoch to identify which state bit needs to be set.

2. Set the corresponding state bit in SRS and load state word (SWS) in a temporary register.

3. Use epoch to find the corresponding mask word in EIT, then check whether there are any SW operations whose epoch is smaller than current thread's epoch (explained in Section 6.3.4). This means that one or more preceding threads have written data in same memory address in this speculative session.

4. If there is such epoch, then choose the data of the latest SW operation, otherwise load data from L1 cache[3].

5. Store the address value into the read buffer.

As in the SW operation, steps 1 and 2 are atomic operations too. Step 2 is an atomic operation between SR operations and SW operations on the same cache line. The SRS and SWS are locked and any SW operations can not set the state bit on SWS and load the SRS, when one SR operation is setting the state bit on the same cache line.

[3]This step might be optimized, we can send the data request to both SMT and the L1 cache at same time, if the SMT can provide data, then give up the L1 cache's data, otherwise wait for the L1 cache's data

Violation Check

The checking operation is simple. The following steps need to be done:

1. Use epoch to find the current speculative thread's corresponding violation bit in EIT and check whether it is 1. If so goto step 2, otherwise goto step 4.
2. Go through the read log buffer, clearing the read bit for each entry in MC cache line's SRS. Clear the read buffer finally.
3. Go through the write log buffer, clearing the write bit for each entry in MC cache line's SWS. Clear the write buffer finally.
4. Return the value of violation bit: 0 or 1.

Speculative Session Commit

To commit a thread, the following steps need to be done:

1. Check whether the thread has any true dependence violations (by checking the violation bit in EIT). If there is violation, goto step 2, otherwise, goto step 3.
2. Restore all of the initial states for the current thread and restart the current thread.
3. Check whether the current thread is the non-speculative thread (by checking the non-speculative pointer in EIT). If not, the process should be blocked here until current thread is annotated as a non-speculative thread in EIT.
4. Check the violation bit again to make sure there is no true dependence violation. If there is a violation, go back step 2.
5. Go through the read log buffer, clearing the read bit for each entry in MC cache line's SRS. Clear the read buffer finally.
6. Go through the write log buffer, and for each entry write back the data to the corresponding address in the L1 cache and clear the write bit for each entry in MC cache line's SWS. Clear the write buffer finally.

7. Deregister the non-speculative thread in EIT, do the same thing as *JAM_SEND* (mentioned in Section 6.3).

The operation for checking the violation bit is also implemented in the *Violation Check* operation, so the software can control violation checking flexibly.

6.3.8 Extended ISA

To enable compiler controlled speculative execution, the hardware provides 6 speculative execution related instructions that correspond to the 6 speculative operations introduced in the previous section:

- *JAM_SR*: speculative read (i.e. load data speculatively). This has the same format as *JAM_LDL* [4], but this load operation will seek data in SMT.
- *JAM_SW*: speculative write (i.e. write data speculatively). This has the same format as *JAM_STL* [5], but this store operation writes data to SMT.
- *JAM_SBEG*: speculative session registration. This registers the current thread as a speculative thread on SMT, then SMT can maintain the related information for the current thread's speculative execution.
- *JAM_SEND*: speculative session deregistration. This deregisters the current thread on SMT, so SMT can get rid of all related information for the current thread.
- *JAM_SCHECK*: violation check. This checks whether there is any data dependence violation in the current speculative thread.
- *JAM_SCOMMIT*: speculative session commit. This commits the current speculative thread's write data.

[4] *JAM_LDL* is the data load instruction in JAMAICA ISA.
[5] *JAM_STL* is the data store instruction in JAMAICA ISA.

6.4 Software Support

By adding the speculative instructions, the LPC utilizes them to generate code for speculative execution and supports TLS in a flexible way. The following subsections introduce the TLS code generation in LPC.

6.4.1 Speculative Thread Creation

Speculative thread creation still uses the normal JAMAICA's lightweight thread mechanism: the token ring is used to indicate the free processors and the main thread (thread creator) can create a parallel thread only if it can get a token from the token ring. The instruction *JAM_THB* and *JAM_THJ* are used to create parallel threads.

To enable speculative execution, the newly created thread needs to be assigned an epoch ID which is passed by the input window register and to set this epoch ID into the current processor context's context register. The JaTLS hardware system does not provide an epoch ID generation mechanism, so the speculative thread creator should control the epoch ID generation. As the epoch IDs must be generated in sequential order for maintaining the speculative states in sequential order, JaTLS's software uses a main thread creator to create all of the speculative threads. Using one centralized thread creator to control the epoch ID generation is easy to implement. Although the JAMAICA's lightweight thread mechanism allows any threads to create parallel threads and the thread creators can be distributed on several threads, to control multiple thread creators is more complex and this is considered in the future work in Section 8.2.2.

6.4.2 Speculative Thread Structure

Figure 6.7 shows the basic code layout of the speculative thread, (b), compared with the normal parallel thread generated by LPC, (a). The speculative thread has similar code layout to the normal LPC branch thread. In the speculative thread structure, one more input window register is needed to pass the epoch ID value. All of the loop constants are still stored in the main thread's stack memory segment. As the speculative thread might be restarted and reload all of

the input parameters, the input window registers' values are stored into thread local storage[6].

The speculative thread should write the epoch ID into the context register [7] first, then it can use *JAM_SBEG* to register the current speculative thread in SMT.

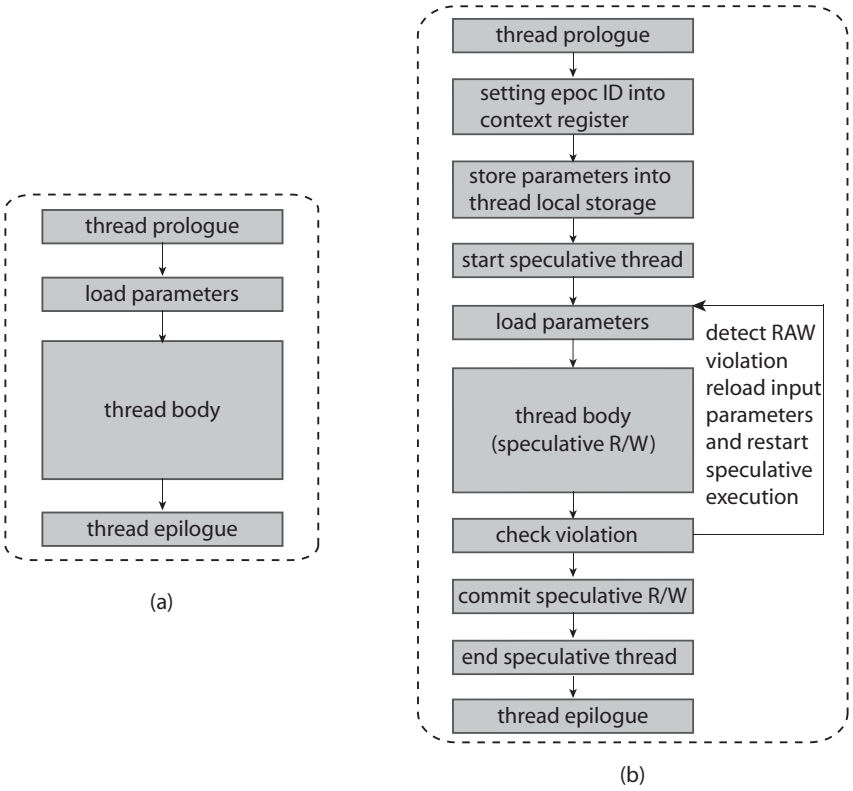


Figure 6.7: The Code Layout for TLS Thread.

A speculative thread rollback operation is implemented by *JAM_SCHECK* and a condition branch instruction. When the main thread body finishes execution, the *JAM_SCHECK* instruction is used to check violation. If the return value is a true (there is true dependence violation in this speculative thread), then the execution path is branched to the reloading segment and reloads all of the input parameters (input window registers' values) and loop constants and restarts the speculative execution. The speculative states in previous execution have been restored by *JAM_SCHECK* (see Section 6.3.7).

[6]A new spill area is allocated in the VM_Processor object for storing input window registers' values.

[7]Both of the SID and epoch ID are stored in context registers, so MCU can pick up these values when it handles SR/SW operations.

The final step is committing all of the speculative write data by *JAM_SCOMMIT* and using *JAM_SEND* to deregister the speculative thread on SMT.

A speculative thread's *Registration*, *Executing*, *Rollback/Restart*, *Commit*, and *Deregistration* are all controlled by the compiler. So not only a branch thread can work as a speculative thread, the main thread which creates the branch threads can also register itself as a speculative thread and execute a loop iteration speculatively. It is a flexible model, and the compiler can control the distribution of speculative threads.

6.4.3 Rollback Operation

As mentioned in the previous section, JaTLS uses a software mechanism to perform speculative thread rollback and restart operations. The compiler needs to handle the reinitialization process carefully to guarantee the speculative thread could restore all initial status when it was rolled back. In the current speculative structure, the speculative thread needs to load all of the live-ins from the memory segment.

In a speculative thread, all of the loop constants parameters are stored in the main thread's stack memory and input parameters should be stored in the thread local storage. If the branch thread needs to rollback, it should reload all of the loop constants and input parameters, so the initial environment is restored.

6.4.4 Java Language Issues

Exceptions

As discussed in Chapter 4, LPC has employed several approaches to avoid runtime exceptions. Implicit or explicit exceptions simply force speculation to stop. So the TLS solution still uses the same policy as the normal LPC implementation.

Garbage Collection

The SMT employs a simple mechanism to manage speculative storage, using memory addresses as the index of MC. GC operation can violate this mechanism, because the GC may move objects within the memory space and the object fields' physical addresses will be changed. So the GC is prohibited when a Java thread starts a speculative session. In the current implementation, all of the yield points are removed in the parallel loop body, thus the parallelized loop is a non-GC segment in the program (see Chapter 4).

Memory Allocation

Object allocation is permitted in speculative threads, but it is unnecessary to buffer access to an object allocated speculatively. The allocation can force the speculative thread to stop if an *OutOfMemoryError* would be triggered as a result. The restarted threads which triggered the data dependence violation will re-allocate the object and this would be potentially dangerous if the restart operation is triggered frequently.

6.5 Evaluation and Discussion

6.5.1 Experimental Setup

The experiments are performed on the extended JAMAICA simulator which supports the JaTLS. Different numbers of processor contexts are evaluated here. To demonstrate the speedup of parallelization, the one context per processor configuration is still used. The size of Multi-Version Cache is 8K and the size of Log Buffer is 1K.

To evaluate the efficiency of JaTLS, 6 benchmark programs are selected (shown in Table 6.1). $RayTracer$, $Molydn$ and FFT are selected from JavaGrand benchmark suite; $JEquake$ is the Java version of 168.equake test in SpecCPU2000; $JMgrid$ is selected from NPB benchmark suite [10]; $JDSMC$ is a Java version of $DSMC3D$ in HPF-2 benchmark suite [41].

Benchmark	Description	Type	Loop Count	No. of Parallelized Loops *
FFT	Fast Fourier transform	Float	5	3
RayTracer	Ray Tracer	Float	14	1
JDSMC	Direct Simulated Monte Carlo	Float	15	12
JEquake	Earth Quake simulation	Float	36	12
Moldyn	Molecular dynamics	Float	6	4
JMgrid	Multi-Grid simulation	Float	52	10

* the inner loops are not counted if the outer loops get parallelized

Table 6.1: The Benchmarks Selected for Evaluating TLS.

6.5.2 Performance Evaluation

Figure 6.8 (a) presents the performance of the JaVM running with a varying number of processors (p) (1 thread context per processor) on the JaTLS simulator. The distribution policy used here is thread per iteration[8]. The benchmarks were chosen as they contain complex loop code which is not statically analysable by LPC. The results show the benefit of the system is a 42% to 208% speed up, depending on the benchmark and the number of processors available. Figure 6.8 (b) shows the speedup got from the same run, but includes the cost of dynamic compilation. By counting the cost of compilation, the adaptive parallelization includes more runtime overhead.

FFT has a three level nested loop in its computation kernel. The outer-most loop has true dependence between each pair of iterations, so there is no benefit from parallelizing the outer loop speculatively. The inner-most loops are selected for parallelization.

RayTracer is a general purpose program. It has a similar problem in loop selection. There is a two level nested loop that is identified as in a hot method and parallelized. The inner loop is selected for parallelization. In the parallelized body, there is a method call which needs to create a new object as the return value. As memory allocation is synchronized in JaVM, the concurrent memory allocation decreases the runtime performance. This program also has a load imbalance problem. The method in the loop body has a recursive call graph (see Figure 6.9);

[8]This is a simple case of CHBD (introduced in Section 4.1.4) whose chunk size is 1.

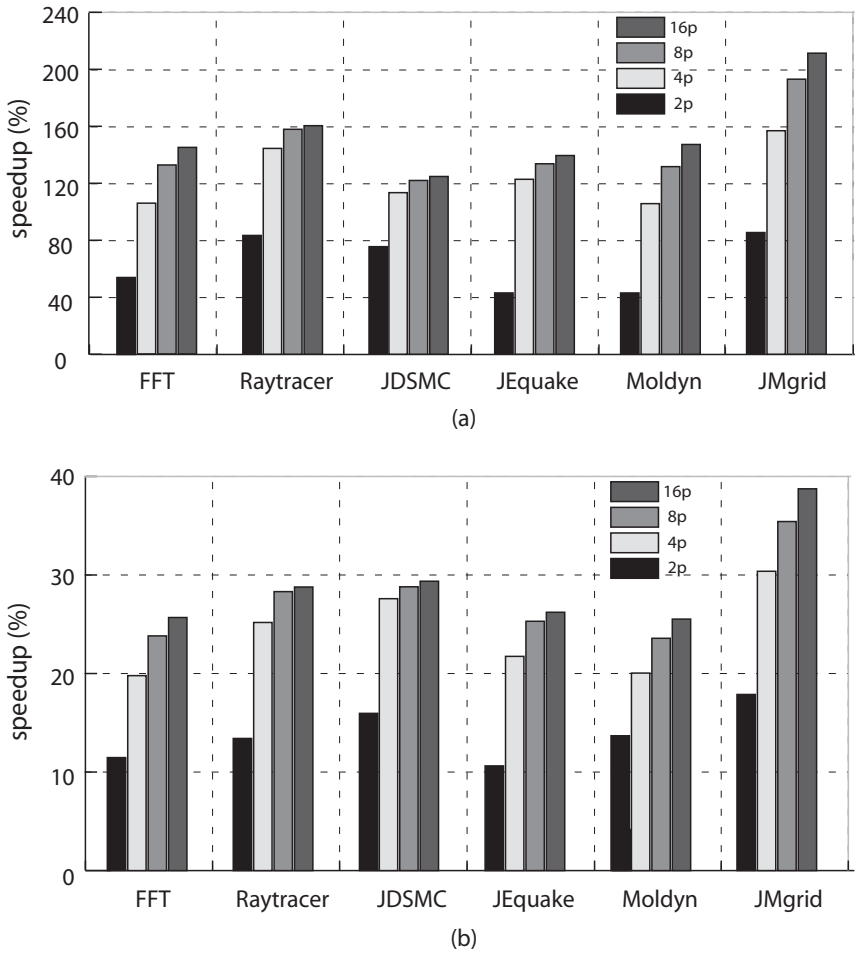


Figure 6.8: Performance of TLS parallelization on 6 Benchmarks Compared to Sequential Execution.

```
for (int i = 0; i < width; i ++) {
  ... ...
  ... = trace(...);
  ... ...
}

Vec trace(int level, double weight, Ray r) {
  ... ...
  boolean condition = ...
  ... ...
  if (condition) {
    ... ...
    return shade(level, weight, P, N, r.D, inter);
  }
  ... ...
}

Vec shade(int level, double weight, Vec P, Vec N, Vec I, Isect hit) {
  ... ...
  boolean condition = ...
  ... ...
  if (condition) {
    ... ...
    ... = trace(...);
    ... ...
  }
  ... ...
}
```

Figure 6.9: The Loop Code in RayTracer.

The nested level of this recursive call is irregular, so the loop iterations can not be distributed evenly among the parallel threads. As just mentioned, in JaVM, all of the concurrent memory allocation operations are synchronized. The method call uses a newly created object as the return value, so the parallel threads may be synchronized when they try to allocate memory at same time.

Molydn has one loop which has true dependence (the loops are only one level, there's no other choice). By increasing the number of parallel threads, the probability of dependence violation is also increased. That is why these programs can not improve greatly in 8 and 16 processor configurations, although they have some other *DoAll* parallelizable loops.

JEquake shares a similar loop selection problem with *FFT*, there are some two level nested loops whose outer loops have true dependence. In this evaluation, the inner loops are selected for parallelization. Most of the parallelized loops have no more than 4 iterations, which is why *JEquake* can not get significant speedup in 8 and 16 processors configurations.

The best speedup is obtained by *JMgrid*. This program is a typical scientific computing applications whose computation kernels are *DoAll* parallelizable loops and most of the loops are multiple level nested loops. The inner-most loops are selected in *JMgrid*, because the outer-most loops in *JMgrid* triggered speculative storage overflow frequently. Usually, the higher level nested loops are coarse grain which the LPC prefers to parallelize. But the higher level nested loops consume more speculative storage than the lower level nested loops. By applying a large data set which needs more loop iterations, the higher level nested loops more easily trigger speculative storage overflow which makes the parallel threads work sequentially and decreases the runtime performance.

For the result shown in the Figure 6.8, we applied one loop iteration per thread because of the various sizes of those loop bodies. There are a lot of small loops in most of these tests, and the one loop iteration per thread generated more runtime overhead for thread creation. By applying a larger chunk per thread (e.g. 5 iterations per thread), the performance can be improved.

From evaluating these benchmark programs, there are several issues which affect the performance in TLS:

- Data dependence violation and speculative storage overflow.
- Load Imbalance.
- Thread Creation/Completion Overhead.

To alleviate these effects, a program should be decomposed carefully by selecting suitable chunk sizes and loop levels. Although we can tune the compiler manually to get better decomposition (e.g. select the inner-most or outer-most loops), and this is what was done here, the runtime system itself should be enabled to search for the best decomposition. A runtime mechanism is proposed to decompose the program dynamically, driven by runtime feedback. This will be discussed in the next chapter.

6.6 Summary

This chapter introduced the basic TLS implementation in the JAMAICA CMP, including hardware and software support. An extended memory module was added to the JAMAICA CMP to maintain the speculative states. Six TLS related instructions were added to the JAMAICA ISA, so the compiler can generate speculative parallelized code flexibly. Finally, the experimental results based on standard Java benchmarks are given. The experimental results show that TLS does provide more opportunity for exploiting parallelization than normal compiler analysis, but the decomposition policy was an important factor to achieve good runtime performance.

Chapter 7

Adaptive Optimization for TLS

The previous chapter raised the question: how to decompose the program efficiently so that the speculatively parallelized version can get the best runtime performance? Again, the adaptive runtime optimization mechanism is employed to decompose the program.

This chapter introduces adaptive optimizations for TLS based parallelization. Section 7.1 introduces the basic motivation for dynamic decomposition. Section 7.2 introduces how the OTF adapts the optimization to decomposition policy selection. Subsection 7.2.5 gives three adaptive optimizations which can reduce the TLS's overhead. Section 7.3 shows the experimental results and discusses them. Section 7.4 summarizes this chapter.

7.1 Motivation

The motivations of adaptive optimization in TLS based parallelization are to find where and how much to speculate; refine speculation and reduce overhead. To achieve these aims, an efficient decomposition policy is needed. The following subsections introduce the basic overhead in TLS and the potential optimizations for finding suitable decomposition.

7.1.1 Overhead in TLS

As discussed in Section 6.5, speculatively parallelized programs have runtime overhead which will obviously effect the performance. All of these overhead can be divided into three major categories:

General Thread Overhead

Although the current JAMAICA architecture provides a light-weight thread mechanism, there remains a significant thread creation/completion overhead (i.e. the thread spawning and synchronization) for creating a large number of small threads, where the overhead accounts for a large fraction of thread execution time. Given a loop which has a small body (i.e. lower than 80 cycles), the overhead of thread creation/completion could discount the benefit of speculative parallelization (similar to the normal parallelization discussed in Chapter 4).

Data Dependence Violation and Speculative Storage Overflow

True dependences that cross speculative thread boundaries may lead to data dependence violations and cause speculative thread restart (rollback) as shown in Figure 7.1. The restarted speculative thread may cause more violations for its succeeding speculative threads, because the restarted execution may be later than the succeeding thread and incurs risk of true dependence violation. The restart process also incurs the overhead of restoring initial execution states.

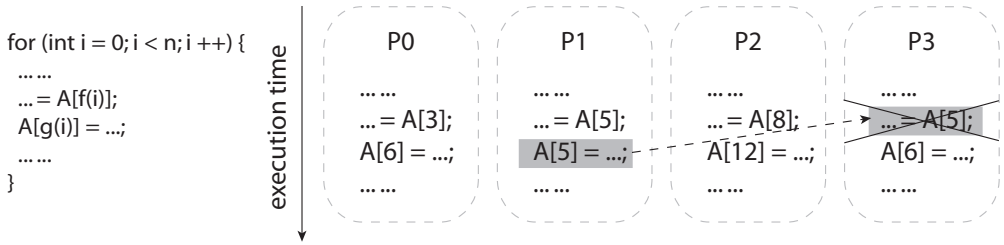


Figure 7.1: The Threads are Squashed by True Dependence Violation.

All the hardware-based TLS implementations provide speculative storage which is known as the speculative buffer. Speculative buffer overflow will happen when

it can not provide enough storage for the current speculative threads. The speculative threads will wait until the non-speculative thread finishes and passes the HFT to its successor.

In JaTLS, a centralized multi-version cache is employed as the major speculative storage. Multiple speculative threads compete for the storage at the same time. If the multi-version cache or speculative state buffer is full, only the non-speculative thread can resume its execution and all other speculative threads have to suspend. So the program is executed sequentially in this scenario.

Load Imbalance Overhead

Load imbalance is highly dependent on the control flow regularity across the parallel threads. For example, inner loops with many input-dependent conditional statements may result in a significant load imbalance across the processors. If the non-speculative thread is long, and its succeeding speculative threads are short, then the succeeding threads may need to wait for the non-speculative thread.

7.1.2 Program Decomposition for Speculative Parallelization

Program decomposition for speculative thread is finding the most efficient way to split a program into speculative threads. Current work on searching for decomposition can be categorized into three types:

- Static Analysis: the compiler performs compile-time analysis to estimate the runtime behaviour of the program and selects the decomposition [24, 30]. This approach is limited by the inaccuracy of the estimate.
- Profiling Based Offline Empirical Search: use the compiler to plant instrumentation code into the application; run the application with a training data set and collect runtime profiling data; perform offline search for suitable decomposition based on the profiling data [46, 62, 88]. Like most profiling based optimization, this approach needs a profiling run to help the compiler perform offline optimization.

- Dynamic Decomposition: the runtime system uses runtime feedback to drive the decomposition process. For example, the Jrpm system [21] employs a hardware profiler to help the JIT compiler identify suitable loops that will provide the most benefit due to speculative parallelization and recompile those loops [1]; [63] employs a fully hardware mechanism to decompose programs at runtime based on the runtime profiling.

As discussed in Chapters 4 and 5, runtime feedback can provide more precise information to drive optimization and static analysis is limited (e.g. the difficulty of analyzing object reference aliasing or control flow at compile-time). So JaTLS employs runtime feedback to drive adaptive optimization to search for efficient decomposition. As this thesis concentrates on loop-level parallelization, how to decompose loops efficiently is the major concern.

The decomposition mechanism implemented here is dynamic decomposition. The aim of the mechanism is using simple hardware support to achieve performance improvement at runtime. Compared with most of the dynamic decomposition mechanisms [21, 63] which need complex hardware support (e.g. a hardware profiling buffer), JaTLS simply provides one additional hardware support for software: the *Violation Counter* (introduced in Section 6.3.5). The JaTLS hardware introduces a special register to count the number of speculative thread re-executions [2]. The major optimization work is performed at software level. The LPC generates the runtime reconfigurable code which will be tuned at runtime to find the best decomposition.

7.2 Online Tuning for TLS

To search for the best decomposition, the adaptive system needs to perform runtime tuning with different decomposition policies. An extended OTF is employed to perform adaptive optimizations at runtime.

[1] The first version of compiled program used for profiling has the instrumenting code for all of the loops. When the runtime system finishes analysis, the code needs to be recompiled to get rid of the redundant instrumenting code and switch loops that are not suitable for speculative parallelization back to sequential execution.

[2] The register value is also incremented when speculative buffer overflow happens.

7.2.1 Feedback Information

The normal JAMAICA OTF (see Chapter 5) performs runtime empirical search to find the optimal configuration for a parallelized loop. The feedback information is execution cycle count. For TLS based optimization, the adaptive system will use a new type of feedback information: *Collapse Rate* (CR).

$CR \leftarrow \frac{r_n}{s_n}$

r_n is the number of re-executions (obtained from the *Violation Counter*). s_n is the total number of speculative threads created for a parallelized loop.

The adaptive optimizations concentrate on how to decompose a loop efficiently and an efficient decomposition can reduce the number of speculative thread restarts. As both of the two major overheads in speculative execution, thread restart and speculative storage overflow, can be identified by the CR value (both these events increment the value of violation counter register), the CR value is employed here as the major runtime feedback.

To achieve load balance, the empirical execution cycle count is still needed (as discussed in Section 7.2.5). So the OTF for TLS should utilize two types of feedback information in some optimizations.

7.2.2 Basic Structure

Figure 7.2 shows the basic structure of TLS based OTF. It could be treated as a simple extension of normal OTF. When the speculatively parallelized loop finishes execution, the inserted profiling code stub picks up the violation counter register's value and execution cycle count [3] and calls an AOS routine to perform runtime evaluation.

To enable the code version switch, the program (loops or nested loops) need to be split into sub-methods. All of these sub-methods can be composed into different combinations which will correspond to different decomposition policies. By using runtime feedback, the speculatively parallelized program can get the best decomposition.

[3]The execution cycle count is optional for some optimizations which need not consider the load balance.

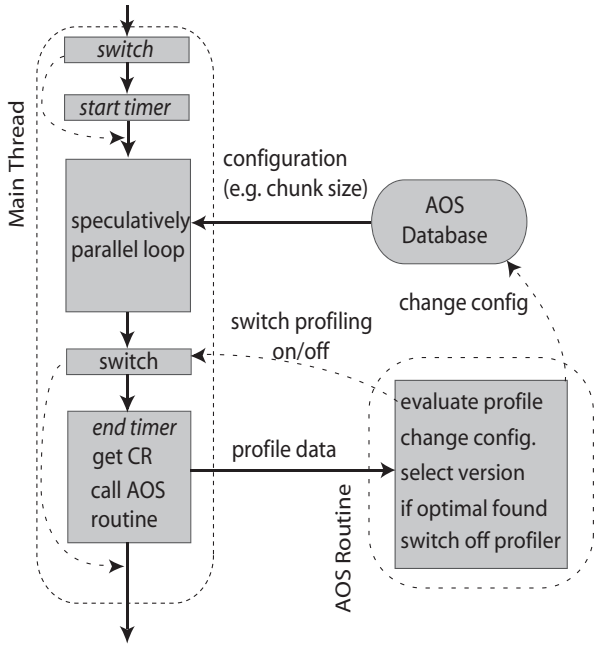


Figure 7.2: The Profiling Mechanism for TLS based OTF.

7.2.3 Tuning Mechanism

There are two basic tuning methodologies:

- Runtime Reconfiguration: reconfigure the loop parameters (e.g. the chunk size, thread number); this mechanism is the same as normal OTF.
- Runtime Switch: switch the code version between speculative and non-speculative. This mechanism is used to perform adaptive loop selection which needs to evaluate different loop levels for speculative parallelization.

The program listed in Figure 7.3 is from the main computation workload of the *FFT* benchmark. The transformed program is listed in Figure 7.4, four loops are split into four isolated *loop call methods* (LCM). Each LCM has two versions: a speculative version and a non-speculative version[4]. The LCMs' references are stored in the *Loop Call Method Table* (LCMT). In the original program, the loop code has been replaced with the corresponding call sites which will call the LCM in the LCMT's slot.

[4]The difference between the speculative and non-speculative loop call method is the code for the speculative thread prologue which registers and starts a speculative thread.

```
for (int bit = 0, dual = 1; bit < logn; bit ++, dual *= 2) {
  ... ...
  for (int b = 0; b < n; b += 2 * dual) {
    int i = fi1(b, dual);
    int j = fj1(b, dual);
    ... ...
    data[i] += f(data[j]);
    ... ...
  }
  ... ...
  for (int a = 1; a < dual; a ++) {
    for (int b = 0; b < n; b += 2 * dual) {
      int i = fi2(a, b, dual);
      int j = fj2(a, b, dual);
      ... ...
      data[i] += g(data[j]);
      ... ...
    }
  }
}
```

Figure 7.3: The Original Loops in FFT.

Figure 7.5 (a) shows the original program's call graph for these LCMs. The call graph is composed of call site nodes which point to LCMs stored in the LCMT. In Figure 7.5 (a), the outer-most loop (*loopCall0*) is executed speculatively. To switch the speculative execution to the two inner loops (*loopCall1* and *loopCall2*), the *loopCall0*'s corresponding call site node simply needs to be redirected to the non-speculative versions of code (shown in Figure 7.5 (b)).

7.2.4 Overhead

The overhead generated by the runtime optimization is very small, because the OTF mainly works on the CR value which can be maintained by one mathematical operation. The average cost of each evaluation step is less than 20 instruction cycles. For the optimizations which need to use the execution cycle count, the runtime overhead is same as the normal OTF. As mentioned above, the runtime reconfiguration and runtime switch are cheap (they simply need to set some table slots), so the whole dynamic decomposition mechanism is very efficient.

```
loopCall0(loopConstants, 0, logn, 1);

void loopCall0(int[] loopConstants, int initValue, int termValue,
               int strideValue) {
  ... ...
  // Initialize loop constants
  ... ...
  loopCall1(loop1Constants, 0, n, 2 * dual);
  ... ...
  // Initialize loop constants
  ... ...
  loopCall2(loop2Constants, 0, dual, 1);
}

void loopCall1(int[] loopConstants, int initValue, int termValue,
               int strideValue) {
    int i = fi1(b, dual);
    int j = fj1(b, dual);
    ... ...
    data[i] += f(data[j]);
    ... ...
}

void loopCall2(int[] loopConstants, int initValue, int termValue,
               int strideValue) {
  // Initialize loop constants
  ... ...
  loopCall3(loop3Constants, 0, n, 2 * dual);
}

void loopCall3((int[] loopConstants, int initValue, int termValue,
                int strideValue) {
  int i = fi2(a, b, dual);
  int j = fj2(a, b, dual);
  ... ...
  data[i] += g(data[j]);
  ... ...
}
```

Figure 7.4: The Loop Methods Call for FFT Code.

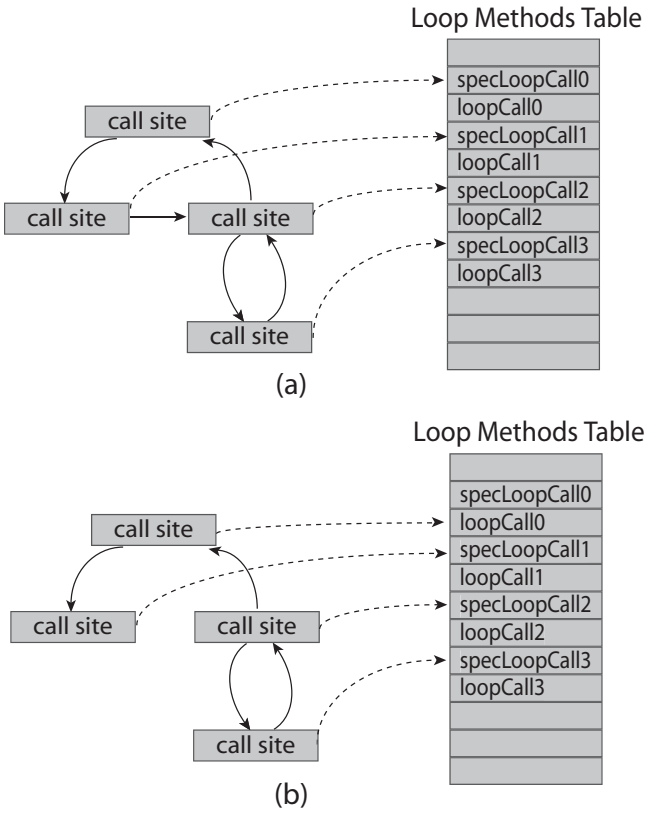


Figure 7.5: The Call Site Graph for FFT.

7.2.5 Adaptive Optimizations

This section discusses the basic adaptive optimizations for dynamic decomposition. There are four basic adaptive optimizations:

Adaptive Chunk Size Selection (ACSS)

This is the basic optimization for reducing the overhead for speculative thread execution (e.g. thread creation, initialization), especially for those loops whose loop body size is small. Increasing the chunk size or assigning more loop iterations to one speculative thread can reduce the overhead, but also increases the probability of data dependence violation and speculative data buffer overflow. So runtime feedback can help the search process find a suitable chunk size.

The search algorithm uses a simple hill-climbing based mechanism to search for the best chunk size (see Algorithm 15). The search process is driven by the CR value. It changes the chunk size incrementally and evaluates it on each loop execution until the CR value reaches a threshold T which is a configurable value (the current configuration is 1).

This optimization preserves the number of processors as the lower bound of the result. It can work properly when the number of processors (or the number of concurrent speculative threads) P_n is less than the data dependence distance d. Give a system which contains 8 processors and a parallel loop whose data dependence distance $d = 5$, it is able to execute 8 speculative threads concurrently: $t_0, t_1, t_2, ...t_7$. There are three pairs of threads that can generate a true dependence violation and collapse: (t_0, t_5), (t_1, t_6) and (t_2, t_7). In other words, there are only 5 threads that can really work in parallel, and increasing the chunk size just increases the probability of restart.

The chunk size obtained from ACSS is the best chunk size for the current problem size. By considering the load balance, the loop should select a suitable chunk size before executing in parallel. Algorithm 16 shows a pseudo-code of a prologue for selecting suitable chunk size. This prologue will be inserted before parallelized loops which will execute in TLS.

Input: threshold T, number of processor P_n, and initial chunk size cs_{init}
Output: optimal chunk size, whether overflow happened
Implementation:
Step 1: set initial chunk size: $i \leftarrow 0$, $cs_i \leftarrow cs_{init}$, $step \leftarrow 1$,
$isIncremental \leftarrow true$, $i \leftarrow 0$;
Step 2: execute the i_{th} run of the loop by applying TLS;
Step 3: get number of speculative threads: s_i, number of restarts r_i and the number of loop iterations $iter_i$;
$cr_i \leftarrow \frac{r_i}{s_i}$
if $cr_i \leq T$ **then**
  $cs_i \leftarrow cs_i + step$;
  if $\frac{iter_i}{P_n} \leq cs_i$ **then**
    $cs_i \leftarrow \frac{iter_i}{P_n}$
  end if
  if $isIncremental$ **then**
    $step \leftarrow step \times 4$;
  else
    $step \leftarrow \frac{step}{2}$;
  end if
else
  if $1 < step$ **then**
    $step \leftarrow \frac{step}{2}$;
    $cs_i \leftarrow cs_i - step$;
    $isIncremental \leftarrow false$;
  else
    return cs_i, $false$;
  end if
end if
Step 4:
if $s_i \leq P_n$ **then**
  return cs_i, $isIncremental$;
end if
$i \leftarrow i + 1$; goto step 2;

Algorithm 15: Adaptive Chunk Size Selection (ACSS).

Input: number of processors P_n, number of loop iterations N_{iter} and the current best chunk size cs_0
Output: optimal chunk size
Implementation:
$cr \leftarrow \lceil \frac{N\_iter}{P_n} \rceil$;
if $cr_0 < cr$ **then**
  $cr \leftarrow \lceil \frac{cr}{\lceil \frac{cr}{cs_0} \rceil} \rceil$;
else
  return cr;
end if

Algorithm 16: The Parallelized Loop's Prologue for Selecting Suitable Chunk Size.

Adaptive Speculative Thread Number Selection (ASTS)

As mentioned above, if the dependence distance d is less than the number of processors P_n but larger than 1, it still can be parallelized speculatively. To optimize this scenario, *Adaptive Speculative Thread Number Selection* (ASTS) is employed (see Algorithm 17).

Input: threshold T, the maximum number of concurrent speculative threads ST (equal to the number of processors)
Output: best number of current speculative threads
Implementation:
Step 1: set initial number of current speculative threads: $st_i \leftarrow ST$;
Step 2:
if $st_i = 1$ **then**
  switch to sequential execution model
end if
execute the loop by applying TLS, get number of speculative threads: s_i and number of restarts r_i;
$cr_i \leftarrow \frac{r_i}{s_i}$;
if $cr_i \leq T$ **then**
  return st_i;
else
  $st_i \leftarrow st_i - 1$;
  goto step 2;
end if

Algorithm 17: Adaptive Speculative Thread Number Selection (ASTS).

This algorithm tries different numbers of concurrent speculative threads from

the number of processors P_n to 1. Given a speculatively parallelized loop which has a set of dependence distances $d_0, d_1, ..., d_n$, the searching result st should equal to $MIN(d_0, d_1, ..., d_n)$. If st equals 1, then parallelization is impossible and the runtime system should switch this loop to the non-speculative version (i.e. sequential execution).

Adaptive Loop Level Selection (ALLS)

When loops are nested, only one loop nest level can be parallelized by the current compiler implementation and hardware support in JaTLS. Only parallelizing outer most loops or inner-most loops can not always achieve the desired performance. Therefore a judicious decision must be made to select the proper nest level to parallelize.

The search algorithm is simple (see Algorithm 18), and starts the evaluation from the outer-most loop to the inner-most loop. As any nested loops can be expressed as a tree structure, the search algorithm will automatically evaluate from the root node to the leaf nodes recursively.

When the algorithm gets a loop whose CR is less than the threshold, it will evaluate its inner sub-loops. Referring to Figure 7.5, if both of the outer-most and the two inner loops' CR values are less than the threshold (they are all suitable for speculative parallelization), the algorithm needs to select one loop level for TLS, but this may raise a problem for load balance. The search process can not guarantee which loop level could gain better performance. One solution is to select another threshold, and thus get rid of some redundant selection. But this may not be realistic. For example, given a two-level nested loop, if both the outer one and inner one have very low CR values, it is difficult to set the appropriate threshold. So a feasible solution chosen in current algorithm is to employ both the CR value and empirical execution cycle as runtime feedback information. When there are several candidate loop levels selected by CR, the AOS can use empirical execution cycle to evaluate them again and select the optimal one. If there are two loop levels that have the same empirical execution cycles, the AOS has to select the inner one.

Input: threshold T, the current root node of nested loop structure L
Output: boolean value
Implementation:
Step 1:
execute the loop by applying TLS, get the cycle count per loop iteration: E_0, the number of speculative threads: s and the number of restarts: r;
$cr_i \leftarrow \frac{r_i}{s_i}$;
Step 2:
if $cr_i > T$ **then**
  return $false$;
end if
$hasInnerLoop \leftarrow false$;
foreach *subnode of* L*:* l **do**
 if $alls(T, l)$ **then**
  $hasInnerLoop \leftarrow true$;
 end if
end
if $hasInnserLoop$ **then**
  execute the loop, get the cycle count per loop iteration: E_i;
  if $E_i \leq E_0$ **then**
    return $true$;
  end if
end if
set the current loop L for TLS;
return $true$;

Algorithm 18: Adaptive Loop Level Selection (ALLS).

Put All Together

To perform efficient runtime decomposition, an integrated optimization should be built based on the previous three optimizations. As the three optimizations all employ 1 dimensional linear search spaces:

- ACSS: searches from 1 to the maximum number of loop iterations that can be executed on one speculative thread.
- ASTS: searches from the number of processor P_n to 1.
- ALLS: searches from the root node to the leaf nodes on the nested loop tree.

The searching processes on these spaces can not interfere with each other, so they can be easily combined together (shown in Algorithm 19).

Input: threshold T, the root of nested loop structure L, number of processor P_n and the initial chunk size cs_{init}
Output: best decomposition policy
Implementation:
Step 1: $ALLS(T, L)$;
Step 2: $(cs, isIncremental) \leftarrow ACSS(T, P_n, cs_{init})$;
Step 3:
if $cs = 1$ and $Not\ isIncremental$ **then**
  $ASTS(T, P_n)$
end if

Algorithm 19: Integrated Adaptive Decomposition (IAD).

7.3 Evaluation and Discussion

7.3.1 Experimental Setup

The hardware and software systems used for evaluation are the same as in the previous chapter. The JaTLS simulation infrastructure acts as the hardware platform, and different numbers of processor cores are evaluated [5]. The size of

[5]For the same reason as in the previous chapter, we didn't evaluate the multi-context configuration which can gain benefit from reducing memory delay but is not good for TLS tasks.

the Multi-Version Cache used here is 8K and the size of Log Buffer is 1K. The threshold used for adaptive search algorithms is 0.5.

The 6 benchmark programs selected are shown in Table 6.1. *RayTracer*, *Molydn*, *FFT* are selected from JavaGrand benchmark suite; *JEquake* is the Java version of 168.equake test in SpecCPU2000; *JMgrid* is selected from the NPB benchmark suite; *JDSMC* is a Java implementation for *Direct Simulation of Monte Carlo*.

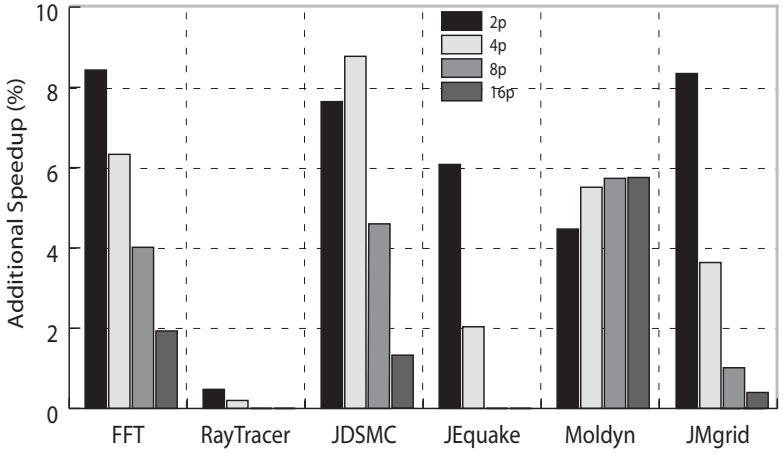


Figure 7.6: Performance of ACSS on 6 Benchmarks.

Benchmark	No. of Optimized Loops	No. of Threads Before Optimization	No. of Threads After Optimization			
			2p	4p	8p	16p
FFT	3	5630	1536	2048	2560	3072
RayTracer	1	800	200	400	800	800
JDSMC	10	5200	360	840	1580	3060
JEquake	11	5895	2025	3630	5422	5595
Moldyn	3	4096	1080	1280	1463	1878
JMgrid	5	56210	15480	29700	44734	49540

Table 7.1: The Optimal Thread Numbers Selected by ASTS.

7.3.2 Chunk Size Selection

ACSS is the simplest adaptive optimization for eliminating the thread creation overhead. Figure 7.6 shows the ACSS's speedup compared with the thread per

iteration policy[6]. Table 7.1 shows the number of created threads before and after optimization.

For those loops whose size of loop body is small but the number of iterations is large, ACSS can obviously benefit. The constraint for ACSS is speculative storage overflow. Increasing the thread size will consume more speculative storage which may result in overflow.

To avoid the speculative storage overflow, the inner-most loops are selected in this evaluation. Most of the benchmarks' inner loops have less than 8 or 16 iterations (e.g. most of *JEquake*'s inner-most loops have no more than 4 iterations, and has most of *JMgrid*'s inner-most loops have no more than 6 or 10 iterations), which is why most of the additional speedups drop down for 8 and 16 processor configurations. By increasing the number of processor contexts, more parallel threads are created (see Table 7.1), this will also increase the overhead for thread creation.

As mentioned in the previous chapter, *RayTracer* has a load imbalance in the selected loop caused by irregular iteration size. By applying the thread per iteration policy on 8 and 16 processor configurations, the workload can be distributed to the processors evenly. So there's no benefit for increasing the chunk size.

7.3.3 Thread Number Selection

The benefit of ASTS is mainly gained from reducing the overhead of thread restart. For those speculatively parallelized loops which have inter-iteration true dependency and can trigger violations frequently, shrinking the number of parallel threads can obtain better runtime performance.

Figure 7.7 shows ASTS's speedup compared with the thread per iteration policy. The three benchmarks selected here have implicit inter-iteration true dependencies. Table 7.2 shows the optimal thread numbers[7] selected by ASTS with different hardware configurations.

[6]Thread per iteration means that a speculative thread is created for each loop iterations.

[7]The optimal thread numbers shown in this table are just for the loops which have implicit inter-iteration true dependency. For example *JDSMC* has 12 parallelized loops, but only 2 of them have inter-iteration true dependencies.

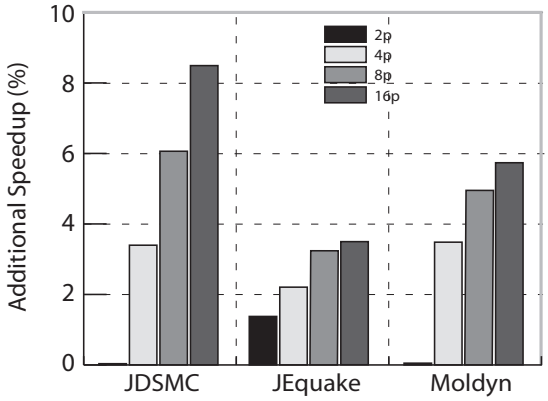


Figure 7.7: Performance of ASTS on 3 Benchmarks.

Benchmark	No. of Optimized Loops	2p	4p	8p	16p
JDSMC	2	2	3	3	3
JEquake	1	1	1	1	1
Moldyn	1	2	3	3	3

Table 7.2: The Optimal Thread Numbers Selected by ASTS.

For $JDSMC$, there are two loops whose dependence distances are irregular, both of them have an average dependence distance value of 3.6. There is no benefit in the 2-processor configuration, because the number of parallel threads is less than the dependence distance. By increasing the number of processor contexts (i.e. increasing the number of parallel threads), it can gain more speedup, when the AOS shrinks the number of speculative threads to 3.

Both $JEquake$ and $Moldyn$ have one loop whose dependence distance is 1. But $JEquake$'s loop iterations have a higher probability of triggering a true dependence violation even if there are only two parallel threads, so the optimal thread number is 1 and the loop was switched to sequential version.

The $Moldyn$ benchmark shares a similar behaviour with $JDSMC$. By increasing the number of processors, the number of speculative threads is increased, and the probability of violation is increased. So the 4-processor, 8-processor, 16-processor configurations can gain a benefit by shrinking the number of speculative threads down to 3.

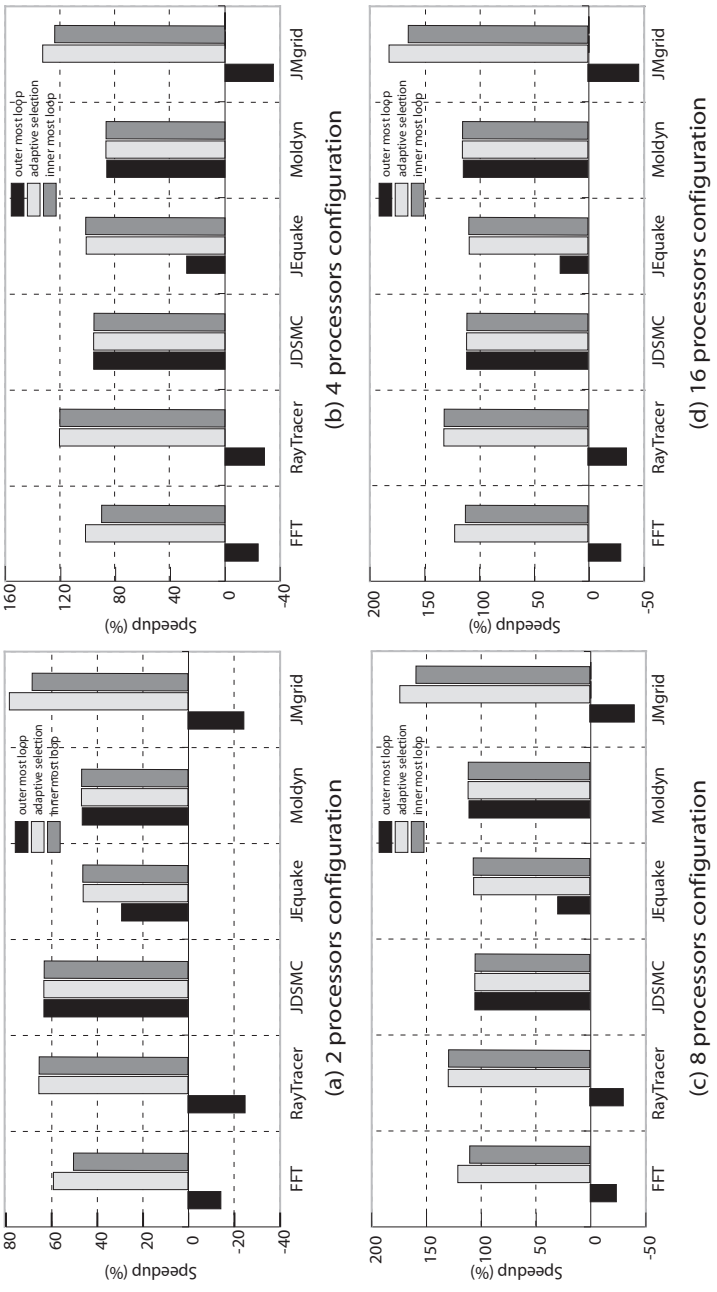


Figure 7.8: Performance of ALLS on 6 Benchmarks.

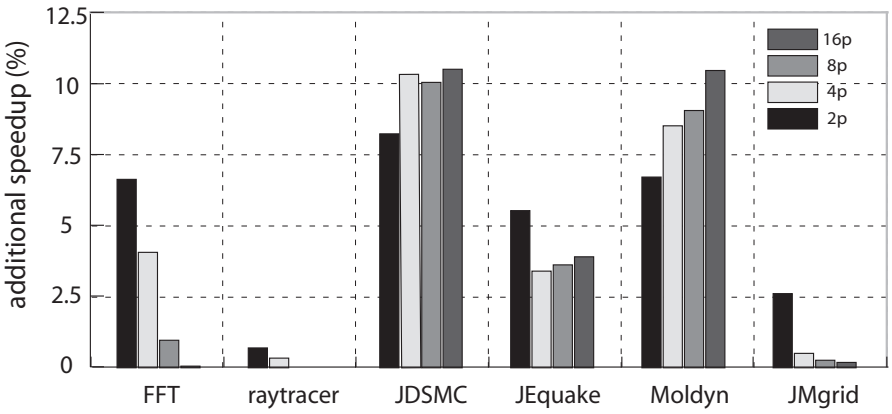


Figure 7.9: Additional Speedup By Integrating ACSS and ASTS.

7.3.4 Loop Selection

Loop selection has the biggest effect of all of these adaptive optimizations. Figure 7.8 shows the ALLS's speedup compared with simply parallelizing the inner-most loops or outer-most loops [8]. Four hardware configurations are evaluated: 2, 4, 8 and 16 processors.

FFT has a 3 level nested loop and the best loop level is the second level, so the ALLS beats the other two simple selections. The same is true of $JMgrid$, it has several 4 level nested loops, and the best level is the third level. $RayTracer$ and $JEquake$ just have simple 2 level nested loops, and most of the inner loops provide the best results.

Most of the loops in $Moldyn$ are single level loops, so there is no difference among these three selections. $JDSMC$ has a similar reason but the speedup is not obvious.

Figure 7.9 shows the speedup for the integrated optimization IAD compared with ALLS. Most of the benefit is obtained from the suitable chunk size by applying ACSS. $JEquake$ benefits from serializing the loop which collapses frequently. $JDSMC$ and $Moldyn$ benefit from optimal chunk size and optimal number of parallel threads. $JMgrid$ can not benefit much from the ACSS, because most of the loops' best chunk size is 1.

[8]Here the thread per iteration policy.

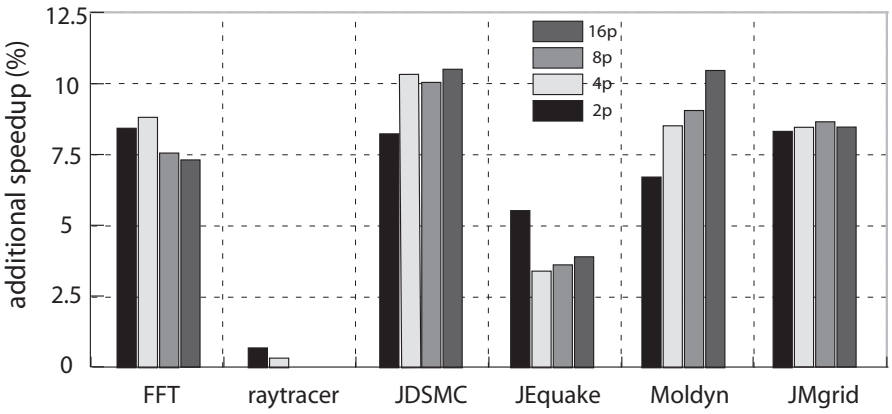


Figure 7.10: Performance of IAD on 6 Benchmarks.

7.3.5 Overall Performance

Figure 7.10 shows the speedup for the integrated optimization IAD compared with the result of plain automatic parallelization from Section 6.5 (see Figure 6.8(a) which is TLS based automatic parallelization using a thread per iteration distribution.).

By employing loop selection, FFT and $JMgrid$ get better performance than just applying ACCS on the inner most loops (see Figure 7.6). The other 4 benchmarks have the same results as shown in Figure 7.9.

7.4 Summary

This chapter introduced how runtime feedback could be used to search for a suitable decomposition policy for speculative parallelization. Section 7.2 introduced the modification OTF to adapt it to the TLS based program and gave three adaptive optimizations that can perform runtime searching for a suitable loop decomposition policy. The experimental results show that the runtime mechanism is flexible enough to make speculative parallelization more efficient.

Chapter 8

Conclusions and Future Work

Computer systems are becoming more and more complex, making the application of compilation techniques increasingly difficult. Developing accurate analytical models for program optimization to utilize the complex hardware system efficiently is challenging. The lack of target machine information and input data information is the major limitation for efficient compiler optimization. Porting applications to new computer systems is often simple and transparent currently, but optimizing applications at compile-time without target machine information and without knowing the input data patterns of the user can only further strain traditional static optimization paradigms.

Due to these trends, a range of research work and techniques have evaluated dynamic and adaptive optimization techniques: tuning applications at runtime when more complete target machine information and input data set information are available.

In this research, a Java Virtual Machine based fully-runtime optimization system was built and evaluated on a simulated CMP architecture. By employing runtime optimizations and the CMP architecture's fine-grain parallelism support, sequential Java programs can be parallelized and optimized efficiently. The simulated hardware infrastructure was introduced in Chapter 2. Chapters 3, 4 and 6 discussed the techniques that support runtime parallelism. In Chapter 5, the runtime adaptive optimizations for improving the automatically parallelized programs' load balance and data locality were evaluated. Chapter 7 evaluated the runtime optimization of searching for efficient decomposition for TLS execution.

8.1 Conclusions

8.1.1 Dynamic Compilation for Parallelism

From evaluating the LPC on the JAMAICA CMP, dynamic parallelization can benefit traditional numerical programs significantly. But for general purpose programs, the benefit is limited and depending on whether the program is parallelizable and statically analyzable (i.e. these programs usually had complex control flow graphs which contain implicit data dependency).

TLS simplifies the compiler's analysis and provides more opportunities for the runtime parallelization system to exploit more parallelism in sequential programs. The LPC enhanced by TLS support can parallelize more sequential programs which are difficult to analyze and parallelize by static analysis (e.g. $JMgrid$ and $Moldyn$ benchmarks mentioned in Chapter 7), but with a requirement for hardware TLS support.

8.1.2 Runtime Adaptive Optimization

The feedback directed runtime optimization can be helpful for improving the programs' runtime performance. By employing the online tuning mechanism, the parallelized programs can improve their load balance and data locality iteratively driven by execution cycle count.

For optimizing TLS, the OTF concentrates on how to find an efficient decomposition policy which can reduce the runtime overhead and wasted work [1]. The OTF uses *collapse rate* as the feedback information; the experimental results show that this feedback could drive the search well with very small overhead.

The major overhead for runtime adaptive optimization is the runtime code generation for evaluating multiple versions of code. By utilizing the extra processors, the dynamic compiler can leverage some of the overhead for runtime code generation.

[1] The wasted work includes data dependence violations which collapse and restart speculative threads and speculative storage overflows which suspend the speculative threads.

All of these runtime optimizations demonstrate good scalability and adaptation to different hardware configurations (i.e. different number of processor contexts, different sizes of L1/L2 caches) and different problem sizes. This demonstrates that the runtime mechanism is a suitable choice for applications' optimizations which can exploit increasingly complex hardware systems.

8.1.3 High Level Language Virtual Machine

All of the runtime optimizations are implemented in a Java Virtual Machine. The managed language system uses the virtual machine to perform compilation, memory management and runtime services. The optimizations are transparent to the application programmers and the virtual machine can employ more complex optimizations at runtime using feedback. Another advantage is that the application, compiler and optimizer work in the same process space, which reduces the communication overhead between the different runtime components. So a virtual machine based integrated runtime environment can act as an ideal adaptive layer between the user application and CMP hardware.

8.2 Future Work

8.2.1 Move to Practical System

CMP processors are becoming the main stream for computing. Applying current research work to the current commercial hardware systems (e.g. Sun Niagra [53]) can be the first step for future work.

The commercial systems provide various information via performance counter (e.g. execution cycle counter, cache miss counter) which can serve to guide runtime optimizations. This is analogous to how we use performance data from the simulated hardware in OTF currently. Compared with the simulated system, a practical system is more complex, which means there would be more challenges for tuning the performance metrics and exploiting a more complex search space. This will make the adaptive optimizations more applicable.

8.2.2 Enhance Optimizations on TLS

To exploit more optimizations on TLS, the method-level speculation and complex speculative thread creation mechanism (i.e. multiple speculative thread creators) need to be implemented. As load imbalance is a major problem for method-level speculation, tuning the decomposition for method-level speculation could be helpful to exploit speculative parallelism more efficiently.

8.2.3 Exploit Complex Optimizations

The complex optimizations include:

- Tuning and integrating more compiler optimizations.
- Developing a more complex search mechanism.

Tuning more optimizations could gain more benefit. Integrating optimizations will increase the complexity of the search space. So future research should prune the search space to make the tuning for complex optimizations more efficient.

Based on current work, the hill-climbing based search mechanism can quickly achieve the optimum, but the shortcoming is that it may only reach a local optimum which may not benefit the program very much. A more complex search space will introduce more local optima which increases the risk of not achieving a good result. Improving the search mechanism is another important thing for large scale runtime optimization.

8.2.4 Common Runtime Optimization

The code generator used in this research is based on the JikesRVM's IR code, which could be a common intermediate representation for any language. By providing a pre-compilation tool which translates the language to the common IR format (e.g. PearColator [73]), the OTF can be a common runtime optimization infrastructure for CMP architectures.

8.3 Concluding Remarks

This work investigated how to parallelize and tune applications adaptively at runtime, where the compilation and optimization system can benefit from being aware of the machine behaviour and the data set of a program, which are usually not available at compile-time. An adaptive parallelization and optimization system, OTF, is built on the JAMAICA CMP architecture which provides fine-grain parallelism support. By using the OTF, the sequential application can be parallelized at runtime and a range of high-level runtime optimizations can be effectively applied to these parallelized applications to improve the load balance and data locality.

Due to the limitation of compiler analysis, TLS support has been added into OTF to exploit more parallelism for those applications that are not easy to be analyzed statically. OTF applies a range of TLS related runtime optimizations to the parallelized applications and performs adaptive decomposition at runtime to find where and how much to speculate, to refine speculation and to reduce overhead.

With the current trends of multi-core system and compiler technologies, dynamic and runtime adaptive systems will be helpful for adapting applications to complex hardware and input data sets. The technologies developed and evaluated in this work are a contribution toward the development of a future runtime environment which can optimize programs transparently and efficiently.

Bibliography

[1] The java hotspot prformance engine architecture. 1999.

[2] OpenIMPACT compiler. `http://www.gelato.uiuc.edu/`, September 2004.

[3] The Jamaica project, May 2005.

[4] The jBYTEMark. `http://www.byte.com`, May 2006.

[5] MIPSPro compiler. `http://www.sgi.com/products/software/irix/tools/mipspro.html/`, May 2007.

[6] Haitham Akkary and Michael A. Driscoll. A dynamic multithreading processor. In *International Symposium on Microarchitecture (Micro'98)*, pages 226–236, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[7] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[8] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[9] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, pages 47–65, 2000.

[10] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber,

H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks: summary and preliminary results. In *Supercomputing 91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 158–165, New York, NY, USA, 1991. ACM.

[11] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[12] Utpal Banerjee. *Loop Transformations for Restructuring Compilers, The Foundations*. Kluwer Academic Publishers, Boston, 1994.

[13] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Advanced program restructuring for high-performance computers with polaris. In *IEEE Computer, 29(12)*, pages 78–82, 1996.

[14] Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*, pages 321–333, 2000.

[15] F. Bodin, T. Kisuk, P. M. W. Knijnenburg, M. F. P. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *The Workshop on Prole . 14 and Feedback-Directed Compilation, in Conjunction with the International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*.

[16] J. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. Benchmarking Java Grande applications. In *Second International Conference on The Practical Applications of Java*.

[17] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing compiler for Java. In *Proceedings ACM 1999 Java Grande Conference*, pages 129–141, San Francisco, CA, United States, June 1999. ACM.

[18] Steve Carr and Ken Kennedy. Compiler blockability of numerical algorithms. In *Supercomputing*, pages 114–124, 1992.

[19] Craig Chambers and David Ungar. Making pure object-oriented languages practical. In Norman Meyrowitz, editor, *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'91)*, volume 26, pages 1–15, New York, NY, 1991. ACM Press.

[20] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 519–538, 2005.

[21] Michael Chen and Kunle Olukotun. The Jrpm system for dynamically parallelizing sequential Java programs. In *International Symposium on Microarchitecture (Micro'03)*, pages 26–35, 2003.

[22] Michael K. Chen and Kunle Olukotun. Test: A tracer for extracting speculative thread. In *International Symposium on Code Generation and Optimization (CGO'03)*, pages 301–314, 2003.

[23] Shiyi Chen and Gary Doolen. Lattice boltzmann method for fluid flows. In *Annual Review of Fluid Mechanics*, number 30, pages 329–364, 1998.

[24] M. Cintra and J. Torrellas. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors, 2002.

[25] Marcelo Cintra, José F. Martínez, and Josep Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *International Symposium on Computer Architecture (ISCA'00)*, pages 13–24, Vancouver, Canada, June 2000.

[26] W. L. Cohagen. Vector optimization for the ASC. In *Proc. Seventh Annual Princeton Conf. Information Sciences and Systems*, page 169, Department of Electrical Engineering, Princeton University Princeton, N.J., 1973.

[27] Microsoft Corporation. *Microsoft C# Language Specifications.* Microsoft Press, Redmond, WA, USA, 2001.

[28] Pedro C. Diniz and Martin C. Rinard. Dynamic feedback: An effective technique for adaptive computing. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'97)*, pages 71–84, 1997.

[29] A. Dinn, I. Watson, K. Kirkham, and A. El-Mahdy. The Jamaica Virtual Machine: A Chip Multiprocessor Parallel Execution Environment. Technical report, University of Manchester, 2005.

[30] Jialin Dou and Marcelo Cintra. Compiler estimation of load imbalance overhead in speculative parallelization. In *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'04)*, pages 203–214, Washington, DC, USA, 2004. IEEE Computer Society.

[31] Stephen J. Fink and Feng Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *International Symposium on Code Generation and Optimization (CGO'03)*, pages 241–252. IEEE Computer Society, 2003.

[32] Grigori Fursin, Albert Cohen, Michael O'Boyle, and Oliver Temam. Quick and practical run-time evaluation of multiple program optimizations. *Transactions on High-Performance Embedded Architectures and Compilers*, 1(1):13–31, 2006.

[33] Grigori Fursin, Albert Cohen, Michael F. P. O'Boyle, and Olivier Temam. A practical method for quickly evaluating program optimizations. In *International Conference on High Performance Embedded Architectures & Compilers (HiPEAC'05)*, pages 29–46, 2005.

[34] Sridhar Gopal, T. N. Vijaykumar, James E. Smith, and Gurindar S. Sohi. Speculative versioning cache. In *International Symposium on High-Performance Computer Architecture (HPCA'98)*, pages 195–205, 1998.

[35] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. The benefits and costs of dyc's run-time optimizations. *ACM Trans. Program. Lang. Syst.*, 22(5):932–972, 2000.

[36] Lance Hammond, Benedict A. Hubbard, Michael Siu, Manohar K. Prabhu, Michael Chen, and Kunle Olukotun. The Stanford Hydra CMP. *IEEE Micro*, pages 71–84, March–April 2000.

[37] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael Chen, and Kunle Olukotun. The Stanford Hydra CMP. *IEEE MICRO Magazine*, pages 71–84, March-April 2000.

[38] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, pages 58–69, New York, NY, USA, 1998. ACM.

[39] John Hennessy, John L. Hennessy, David Goldberg, and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[40] John L. Henning. SPEC CPU2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, 2000.

[41] High Performance Fortran Forum. HPF-2 scope of work and motivating applications. Technical Report CRPC-TR 94492, Houston, TX, 1994.

[42] Urs Hölzle and David Ungar. A third-generation SELF implementation: Reconsiling responsiveness with performance. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'94)*, pages 229–243, 1994.

[43] Matthew J. Horsnell. *A chip multi-cluster architecture with locality aware task distribution.* PhD thesis, The University of Manchester, 2007.

[44] IBM. Jikes™Research Virtual Machine (RVM). `http://jikesrvm.org/`, December 2007. Last accessed.

[45] F. Irigoin and R. Triolet. Supernode partitioning. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88)*, pages 319–329, New York, NY, USA, 1988. ACM Press.

[46] Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Speculative thread decomposition through empirical optimization. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'07)*, pages 205–214, New York, NY, USA, 2007. ACM Press.

[47] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Ltd., Unite Kingdom, 1996.

[48] Ron Kalla, Balaram Sinharoy, and Joel M. Tendle. IBM Power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2), March/April 2004.

[49] Gerry Kane. *MIPS RISC architecture*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[50] Toru Kisuki, Peter M. W. Knijnenburg, and Michael F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'00)*, pages 237–248, 2000.

[51] Toru Kisuki, Peter M. W. Knijnenburg, Michael F. P. O'Boyle, Francois Bodin, and Harry A. G. Wijshoff. A feasibility study in iterative compilation. In *International Symposium on High Performance Computing (ISHPC'99)*, pages 121–132, 1999.

[52] Kathleen Knobe and Vivek Sarkar. Array SSA form and its use in parallelization. In *Symposium on Principles of Programming Languages (POPL'98)*, pages 107–120, 1998.

[53] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[54] Venkata Krishnan and Josep Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, 1999.

[55] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. *SIGARCH Comput. Archit. News*, 35(2):162–173, 2007.

[56] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'91)*, pages 63–74, 1991.

[57] Jeremy Lau, Matthew Arnold, Michael Hind, and Brad Calder. Online performance auditing: using hot optimizations without getting burned. In *PLDI*, pages 239–251, 2006.

[58] Daniel M. Lavery, Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen mei W. Hwu. The importance of prepass code scheduling for superscalar and superpipelined processors. *IEEE Transactions on Computers*, 44(3):353–370, 1995.

[59] Hung Q. Le, William J. Starke, J. Stephen Fields, Francis P. O'Connell, Dung Q. Nguyen, Bruce J. Ronchetti, Wolfram M. Sauer, Eric M. Schwarz, and Michael T. (Mike) Vaden. Ibm power6 microarchitecture. *IEEE Micro*, 24(2), March/April 2004.

[60] Hui Li, Sudarsan Tandri, Michael Stumm, and Kenneth C. Sevcik. Locality and loop scheduling on NUMA multiprocessors. In *Proceedings of the 1993 International Conference on Parallel Processing (ICPP'93)*, volume II - Software, pages II–140–II–147, Boca Raton, FL, 1993. CRC Press.

[61] Tim Lidholm and Frank Yellin. *The Java Virtual Machine Specification*. Sun Microsystems Inc., second edition, 1999.

[62] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: a TLS compiler that exploits program structure. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*, pages 158–167, New York, NY, USA, 2006. ACM Press.

[63] Pedro Marcuello and Antonio Gonzalez. Thread-spawning schemes for speculative multithreading. In *International Symposium on High-Performance Computer Architecture (HPCA'02)*, pages 55–64, 2002.

[64] John M. Mellor-Crummey, Vikram S. Adve, Bradley Broom, Daniel G. Chavarría-Miranda, Robert J. Fowler, Guohua Jin, Ken Kennedy, and Qing Yi. Advanced optimization strategies in the rice dhpf compiler. *Concurrency and Computation: Practice and Experience*, 14(8-9):741–767, 2002.

[65] Avi Mendelson, Julius Mandelblat, Simcha Gochman, Anat Shemer, Rajshree Chabukswar, Erik Niemeyer, and Arun Kumar. CMP implementation

in systems based on the Intel Core Duo processor. *Intel Technical Journal*, 10(2), May 2006.

[66] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT, 1996.

[67] José E. Moreira, Samuel P. Midkiff, Manish Gupta, Peng Wu, George S. Almasi, and Pedro V. Artigas. Ninja: Java for high performance numerical computing. *Scientific Programming*, 10(1):19–33, 2002.

[68] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[69] Steven S. Muchnick and Phillip B. Gibbons. Efficient instruction scheduling for a pipelined architecture. *ACM SIGPLAN Notices*, 39(4):167–174, 2004.

[70] Nagendra Nagarajayya. Improving application efficiency through chip multi-threading. *Sun Developers Forum*, 2005.

[71] Manohar K. Prabhu and Kunle Olukotun. Using thread-level speculation to simplify manual parallelization. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*, pages 1–12. ACM, 2003.

[72] Manohar K. Prabhu and Kunle Olukotun. Exposing speculative thread parallelism in spec2000. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*, pages 142–152, New York, NY, USA, 2005. ACM.

[73] Ian Rogers and Chris Kirkham. JikesNODE and PearColator: A Jikes RVM operating system and legacy code execution environment. In *2nd ECOOP Workshop on Programming Languages and Operating Systems (ECOOP-PLOS'05)*, 2005.

[74] Ian Rogers, Jisheng Zhao, Chris Kirkham, and Ian Watson. An automatic runtime doall loop parallelisation optimization for java. In *Parallel/High-Performance Object-Oriented Scientific Computing (POOSC'05)*, Glasgow, July 2005.

[75] Vivek Sarkar. The ptran parallel programming system. In *Parallel Functional Programming Languages and Compilers*, pages 309–391. ACM Press Frontier Series, 1991.

[76] T. Scholz and M. Schafers. An improved dynamic register array concept for high-performance RISC processors. In *HICSS '95: Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS'95)*, page 181, Washington, DC, USA, 1995. IEEE Computer Society.

[77] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, June 2005.

[78] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *International Symposium on Computer Architecture (ISCA'95)*, pages 414–425, New York, NY, USA, 1995. ACM.

[79] Y. N. Srikant and Priti Shankar, editors. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2002.

[80] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. The stampede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, 2005.

[81] T. Suganama, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Kakatani. Overview of the ibm java just-in-time compiler. *IBM System Journal,*, 39(1), 2000.

[82] T.C.Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–848, 1961.

[83] Marc Tremblay, Jeffrey Chan, Shailender Chaudhry, Andrew W. Conigliaro, and Shing Sheung Tse. The MAJC architecture: A synthesis of parallelism and scalability. *IEEE Micro*, 20(6):12–25, 2000.

[84] Theo Ungerer, Borut Robič, and Jurij Šilc. A survey of processors with explicit multithreading. *ACM Comput. Surv.*, 35(1):29–63, 2003.

[85] Michael Voss and Rudolf Eigenmann. ADAPT: Automated de-coupled adaptive program transformation. In *International Conference on Parallel Processing (ICPP'00)*, pages 163–, 2000.

[86] Michael Voss and Rudolf Eigenmann. High-level adaptive program optimization with adapt. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'01)*, pages 93–102, 2001.

[87] D. W. Wall. Register windows vs. register allocation. *WRL Technical Report 87/5*, 1987.

[88] Shengyue Wang, Xiaoru Dai, Kiran Yellajyosula, Antonia Zhai, and Pen-Chung Yew. Loop selection for thread-level speculation. In *Workshops on Languages and Compilers for Parallel Computing (LCPC'05)*, pages 289–303, 2005.

[89] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005.

[90] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[91] Robert Wilson, Robert French, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih Liao, Chau Tseng, Mary Hall, Monica Lam, and John Hennessy. The SUIF compiler system: a parallelizing and optimizing research compiler. Technical report, Stanford, CA, USA, 1994.

[92] Michael J. Wolfe. *High performance compilers for parallel computing*. Addison-Wesley, Redwood City, CA, 1996.

[93] Greg Wright. *A single-chip multiprocessor architecture with hardware thread support*. PhD thesis, The University of Manchester, 2001.

[94] Jisheng Zhao, Matthew Horsnell, Ian Rogers, Andrew Dinn, Chris C. Kirkham, and Ian Watson. Optimizing chip multiprocessor work distribution using dynamic compilation. In *Euro-Par*, pages 258–267, 2007.

[95] Jisheng Zhao, Chris Kirkham, and Ian Rogers. Lazy inter-procedural analysis for dynamic loop parallelization. In *Workshop on New Horizons in Compiler Analysis and Optimizations, Conjunction with International Conference on High Performance Computing (HiPC'06)*, December 2006.

[96] Jisheng Zhao, Dr. Ian Rogers, and Dr. Chris Kirkham. A system for runtime loop optimisation in the JikesRVM. In *Postgraduate Research in Electronics, Photonics and Communication (PREP'05)*, April 2005.

[97] Jisheng Zhao, Dr. Ian Rogers, Dr. Chris Kirkham, and Prof. Ian Watson. Loop parallelisation for the JikesRVM. In *Sixth International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'05)*, December 2005.