UNIVERSITY OF CALIFORNIA
Santa Barbara

Exploiting Adaptation in a Java Virtual Machine to Enable Both Programmer Productivity and Performance for Heterogeneous Devices

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Lingli Zhang

Committee in Charge:

Professor Chandra Krintz, Chair

Professor John Gilbert

Professor Rich Wolski

March 2008

Exploiting Adaptation in a Java Virtual Machine to Enable Both Programmer Productivity and Performance for Heterogeneous Devices

To my parents, *Suzhuang* and *Jiqing*,

my husband, *Ye*,

and my son, *Yifan*.

Acknowledgements

I would like to thank all people who have contributed in some ways to the development of this dissertation.

First and foremost, I want to thank my advisor, Dr. Chandra Krintz, for her guidance, encouragement, and support. It has been an honor to be one of her first Ph.D. students. Thank you for helping me develop independent thinking and research skills, and preparing me for the future challenges. Thank you for having confidence in me even when I was wondering in the Ph.D. pursuit and providing me numerous research opportunities to exploit. Thank you for being so supportive during the difficult times. For all of your help, Chandra, I thank you, sincerely.

I would also like to thank the rest of my thesis committee members: Dr. John Gilbert, and especially, Dr. Rich Wolski for their time, valuable input and insightful discussions.

My special thanks go to Vinod Grover, my mentor during my internships in Microsoft. He is the one who introduced me the “future” programming construct, and got me interested in the parallel programming area in general. Without his inspiration, this thesis work would not be possible.

I would like to express my sincere gratitude for the friendship of all of the members in the RaceLab and MayhemLab, who have made my graduate student life enjoyable and full of fun. In particular, I would like to thank Sunil and Priya, for their fruitful discussions and collaborations.

I thank all the faculty members, staffs and students in the Computer Science department at UC Santa Barbara, whose presences and spirits make the department a warm, and fantastic place to learn, to think, and to have fun. I am especially grateful to Professor Tevfik Bultan, and the secretaries, MaryJane, Amanda, Greta, Sandy, and many more, for their continual support.

Finally, and most importantly, I would like to thank my family for all their love and encouragement. My deepest thanks go to my beloved parents, Suzhuang and Jiqing, who have always been there when I needed them most and have supported me through all these years. Most especially, I would like to thank my husband and best friend, Ye, for his love and tremendous support.

Curriculum Vitæ

Lingli Zhang

Education

2007	Doctor of Philosophy in Computer Science, University of California, Santa Barbara.
2007	Master of Science in Computer Science, University of California, Santa Barbara.
2000	Master of Science in Computer Science, Zhejiang University, China.
1997	Bachelor of Science in Computer Science, Zhejiang University, China.

Experience

2006	Summer Internship, Microsoft Research.
2005	Summer Internship, Microsoft Research.
2004	Summer Session Lecturer, University of California, Santa Barbara.
2002 – 2007	Research Assistant, University of California, Santa Barbara.

Publications

Lingli Zhang, Chandra Krintz and Priya Nagpurkar. *Supporting Exception Handling for Futures in Java*. In Proceedings of the ACM International Conference on the Principles and Practice of Programming in Java (PPPJ), pages 175–184, September, 2007

Lingli Zhang, Chandra Krintz and Priya Nagpurkar. *Language and Virtual Machine Support for Efficient Fine-Grained Futures in Java*. In Proceedings of the ACM Conference on Parallel Architecture and Compilation Techniques (PACT), pages 130–139, September, 2007

Lingli Zhang, Chandra Krintz, and Sunil Soman. *Efficient Support of Fine-grained Futures in Java*. International Conference on Parallel and Distributed Computing Systems (PDCS), November, 2006

Lingli Zhang and Chandra Krintz. *The Design, Implementation, and Evaluation of Adaptive Code Unloading for Resource-Constrained Devices*. ACM Transactions on Architecture and Code Optimization (TACO), Vol. 2, Number 2, pages 131–164, June, 2005

Lingli Zhang and Chandra Krintz. *Adaptive Code Unloading for Resource-Constrained JVMs*. ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), pages 155–164, June, 2004

Lingli Zhang and Chandra Krintz. *Profile-driven Code Unloading for Resource-Constrained JVMs*. ACM International Conference on the Principles and Practice of Programming in Java (PPPJ), pages 83–90, June, 2004

Abstract

Exploiting Adaptation in a Java Virtual Machine to Enable Both Programmer Productivity and Performance for Heterogeneous Devices

Lingli Zhang

Computer systems with which we interact on a daily basis consist of a vast diversity of devices. At the low end, battery-powered, handheld devices with limited resources, e.g., personal digital assistants (PDAs), smartphones, etc., are common and extremely popular. At the high end, powerful, multi- and many-core processor systems are increasingly ubiquitous in the laptops, deskside computers, workstations, and clusters that we use. The pervasiveness of heterogeneous systems requires that there be programming languages for application development, that are easy to learn and use, portable across diverse systems, and that facilitate extraction of high performance from the underlying, available device technology.

The Java programming language offers many of these characteristics. Its architecture-independent program transfer format and extant virtual machine technology for a wide variety of devices, provides programmers with a write once and run anywhere (WORA) model. Moreover, its object-orientation, robust library support, and high-level syntax, make Java easy to learn and develop applications with. Finally, modern virtual machine (VM) technology offers powerful adaptive services that combine program profiling and optimization, to extract high performance from applications with frequently executed code regions (hot spots). However, the Java language and its VM technology to date have not targeted device-specific opportunities that have the potential for enabling both programmer productivity and scalable high-performance.

In this dissertation, we investigate potential opportunities for devices at the two ends of the performance spectrum: battery-powered, handheld PDAs and parallel processing systems. Our thesis question asks whether we can employ adaptive JVM technologies to improve the performance of programs in resource-constrained devices and to facilitate efficient and scalable parallel programming in multi-core systems. We address each question with a specific and novel solution: (1) adaptive compiled code management for low end devices, and (2) adaptive code parallelization for high end, multiprocessing systems. The goal of our work is to extract high performance from the underlying device technology through novel JVM extensions while maintaining the ease-of-use of the Java programming language. We describe each of our foci in detail and present empirical evidence of its efficacy and potential.

Professor Chandra Krintz
Dissertation Committee Chair

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern computing devices are increasingly heterogeneous, pervasive, ubiquitous, and important in our everyday lives. For example, in 2006, world-wide, combined smart-phone and personal digital assistant (PDA) shipments have totalled over 90 million units. The authors in [54] predict that this number will reach 114.1 million and that the number of cell phones will exceed 1 billion, in 2007 [53]. The number of desktop computers is similarly immense: The 228.6 million systems in 2006 [55] has grown 10.9% in the first quarter of 2007 [56].

These vast compute resources are very diverse in their architecture and capabilities and continue to grow in complexity. For example, mobile handheld devices have limited compute and storage resources and the lifetime of batteries is very short, due to their constrained form factor. A typical PDA device features a low-power RISC processor (like ARM) clocked at 200-600MHz and 32-128MB memory. Reducing the software resource footprint and power usage is a major design challenge for these devices. On the other hand, laptops, desktops, workstations, and high performance servers are becoming increasingly computationally powerful by adopting multi- and many-core technology. Major processor manufacturers, such as Intel and AMD, have released dual-core processors since 2005. Intel launched its quad-core processors in December 2006. AMD will make its quad-core processors available in 2007. Intel has even developed an 80-core research processor [77] to demonstrate the power of future parallel processor technology. Exploiting parallelism to achieve maximal performance is a key challenge for high-end, multi-core systems.

The pervasiveness of computing devices significantly increases the demand for software for these devices. However, the diversity of device architecture and capabilities makes software development very challenging. To build efficient software, programmers must have expertise on wide variety of device platforms. For example, resource constrained handheld devices require that programmers carefully design, implement, and tune their programs to meet the constraints on CPU speed, memory size, and battery life. For high-end multi-core systems, programmers must effectively parallelize their programs to take advantage of the multiple processors in the underlying hardware (without introducing synchronization errors).

The complexity and diversity of modern systems make it increasingly difficult for programmers to extract performance and capability from a wide range of devices, via the programs that they write. As a result, there is an increasing need for programming languages that are easy to learn and use by most programmers (including novices), portable across diverse platforms, and that facilitate the extraction of both performance and capability regardless of the underlying device technology. One such language with the potential for enabling programmer productivity and performance for a wide range of devices is Java [17].

1.1 The Java Programming Language

Java is an object-oriented programming language. It features many modern high level language constructs, such as exception handling, templates, annotations, among others. The Java language support provides an extensive library for automatic use of most common data structures, algorithms, and system facilities. Java's execution model is based on a virtual machine [102] (JVMs). Developers compile a Java source program into an architecture-independent, stack-machine-based, intermediate and object-oriented format (called *bytecode*). At execution time, Java virtual machines consume bytecode programs incrementally, a module (i.e. *class*) at a time and dynamically translates the methods that the program invokes on-demand to native code. This system enables and ensures both type safety of the program as well as automatic memory management (garbage collection).

The virtual machine execution model, the high-level abstractions available in the Java language, and the type and memory safety the language system guarantees, makes Java very easy to learn and write programs in – for a vast diversity of devices and platforms. Moreover, this model enables portability of programs (the Write Once Run Anywhere (WORA) model). On any machine for which there is a Java Virtual Machine (JVM), a Java program will run (with some minor library-support caveats). For these reasons, Java has emerged as one of the most widely-adopted programming languages, for application and system development on a broad range of computing devices, from handheld devices to high performance servers. Based on TIOBE Programming Community Index [145], in May 2007, Java is the number one programming language in terms of popularity among programmers. Statistics from JavaOne 2007 show that there are total 5 billion Java-enabled devices to date (May 2007) including desktops, mobile phones, Java cards, set-top boxes, toys, navigation systems and robots, etc. Among them, 1.83 billion are mobile devices [83].

1.2 Performance Adaptation in Java Virtual Machines

The JVM execution model enables many programmer productivity benefits. However, this benefit does not come for free. The extra layer of abstraction that a JVM introduces can impose significant performance overhead, since Java programs must be interpreted or compiled on-the-fly (consuming both memory and time) *during* execution. Such translation however, enables portability and automates the burden of developers porting to a vast diversity of devices and systems. Interpretation or fast, non-optimized compilation produces very poor code quality and performance. Optimized compilation can produce high-quality, specialized code and high performance but is very expensive to apply, i.e., its runtime overhead is difficult to amortize via the improved execution time. To address this problem, modern Java virtual machines integrate adaptive compilation support in which the system monitors the application in a very lightweight way (based on counters [75, 32] or analytic models [7, 8, 9]), and then identifies the regions of programs that will benefit the most from (and amortize the cost of) optimized execution.

In particular, current adaptive optimization systems (e.g. [8, 32, 75]) dynamically identify code regions that consume significant portions of execution time. These code regions are referred to as *hotspots*. Hotspots commonly consist of a series of methods or basic blocks within methods. By optimizing hotspots, virtual machines attempt to balance compilation overhead and execution speed, and as a result, have been shown to enable significant performance gains [20, 8, 142, 32, 121]. Adaptive JVMs are inherently dynamic, have extensive runtime information, and are able to *make decisions* about and *adapt* to, program and system behavior while the program runs, to improve performance.

1.3 The Thesis Question

Despite its popularity and potential, extant Java technology fails to address many device-specific problems that challenge today's programmers. We focus on two particular limitations as part of this dissertation: (i) dynamic adaptation to improve performance in resource-constrained mobile devices, and (ii) easy yet efficient and scalable parallel programming. In the first case, programmers developing code for mobile devices must choose a slow but compact interpreter-based JVM, or use a compiler-based JVM and then manually identify opportunities for code-size optimization since native code is significantly larger than bytecode. The latter enables performance and effective use of limited resources but imposes a significant challenge on programmers thus reducing productivity.

The second problem is an increasingly important one as multi-core systems become ubiquitous. Concurrency in program design is an expert concept and support for concurrency in Java is non-intuitive to use, significantly different from the sequential, semantic equivalent, and challenging to debug. One primary reason that concurrent program development is extremely difficult, especially for non-experts, is that the average programmers trained to use the "average" programming languages typically are most comfortable with a *serial* approach to most programming problems. As a result, novel and effective support for concurrent programming in Java is needed that enables the average programmer to easily transition from a serial approach to the concurrent one.

Both of these problems are challenging because their solutions depend on the way the programs execute at runtime – and the dynamically changing conditions of the underlying resources and the behavior of the programs. Programmers write static programs and have a difficult time understanding and accurately estimating all of the intricacies of the behavior of the executing program and the resources it will consume. Extant JVMs provide little support to aid developers in these processes.

An alternative way to solve these problems, that has yet to be explored, is to provide the JVM with support that facilitates and automates these processes for developers. To investigate solutions to these problems, we observe that the characteristics of the problems are similar to those of the problem of trading off compilation overhead for performance, described above. In particular, we observe that an adaptive system within a JVM is able to

- Access dynamic runtime services, including dynamic loading, dynamic compilation, garbage collection, thread scheduling, etc.;

- Obtain accurate information on the dynamic behavior of both application execution and underlying resource availability, at low cost; and,

- Exploit and control low-level runtime constructs such as the heap, internal thread representation, thread execution stacks, etc.

As a result, we believe that we can apply similar adaptive technology to address the problems of mobile program code management and easy-to-use and efficient multi-threading in high-end, multi-core systems.

Hence, with this dissertation, we investigate the following question:

> *Can we exploit adaptation in the Java virtual machine in novel ways to enable both high programmer productivity and high performance of Java applications for diverse computing systems?*

In particular, we investigate

- Efficient memory management of executable code in resource-constrained JVMs; and
- Easy parallel programming with efficient and scalable future support in Java

The goal of our work is to extract automatically high performance from the underlying device technology while maintaining and improving the ease-of-use of the Java programming language. This dissertation, as a result, consists of two parts that build on the same internal adaptive JVM technology: Adaptive Code Unloading for Resource Constrained Devices, and Easy and Efficient Future Functionality for Multi-Processing Devices. We overview each in the following subsections.

1.3.1 Adaptive Code Unloading for Resource Constrained Devices

Compilation-based JVMs typically significantly outperform interpretation-based JVMs. This is because compilation-based JVMs are able to reuse very high-quality code. The benefits from code quality and reuse easily amortize the cost of compilation on high-end systems. However, for battery-powered, resource-constrained devices, interpretation is employed by most JVMs. This is because interpretation imposes minimal memory overhead and no compilation cost. The memory overhead of compilation in such systems can oftentimes even preclude some programs from executing (e.g. on small-memory systems). Unfortunately, the quality of interpreted code is very poor and code that is executed repeatedly is re-interpreted, which imposes pure overhead, however, without consuming memory for code storage. If we are to employ compilation, we must either exert a significant burden on programmers to shrink their code size or identify solutions that enable the best of both worlds: very limited memory (as for interpretation) and efficient execution (as for compilation), without programmer intervention or participation.

To enable easy efficiency for resource-constrained devices, we present a dynamic and adaptive code unloading framework. Our system dynamically identifies and unloads dead or infrequently used, compiled code to reduce the memory footprint of the JVM. The system exploits the runtime services of JVMs (such as dynamic loading, adaptive compilation, garbage collection, etc.) and information of both program behavior and underlying resource availability collected via a lightweight online sampling framework. We investigate a number of implementation alternatives that employ dynamic feedback from the program and execution environment to identify unloading candidates and to trigger unloading efficiently, transparently, and adaptively.

We implement and empirically evaluate our framework and unloading strategies using a popular, open-source JVM and a set of community benchmarks. Our empirical evaluation shows that our code unloading framework significantly reduces the memory requirements of a compiler-only JVM, while maintaining the performance benefits enabled by compilation. In particular, for the best-performing unloading configuration and the community programs that we studied, our system reduces code size by 36-62% and enables execution time benefits of 23% on average when memory is highly constrained.

1.3.2 Easy and Efficient Future for Multi-Processing Devices

The second part of this dissertation investigates the use of adaptation to enable both programmer productivity (ease of use) and high performance (efficiency as well as scalability) for emerging multi-core systems. To enable easy efficiency in this domain, we focus on the *future* parallel programming construct [70, 126], and its implementation in the Java programming language [86], for support of fine-grained parallelism. The future is a simple and elegant construct that programmers can use to identify potentially asynchronous computation. The current Java implementation of futures is a library level implementation with an interface-based programming model. Users employ a set of APIs to encapsulate potentially asynchronous computation and to define their own future execution engines.

This model unfortunately is non-intuitive and produces source that is significantly different from its sequential semantic equivalent (with which most programmers are most comfortable). Moreover, the library-based approach is unable to exploit information about the executing program and underlying resources to make its scheduling decisions. This approach also introduces significant memory overhead in JVM systems due to the multiple levels of indirection and encapsulation that the interface-based model can impose. Finally, Java futures require that users identify the regions of their program that *should* execute concurrently to improve performance (not simply what parts of their program *can* be executed concurrently). This imposition is significant since it requires that programmers have expert knowledge of not only their program and its behavior for all possible inputs, but also of the performance characteristics of the machines on which their programs ultimately execute. This latter requirement conflicts with Java's write-once-run-anywhere model, since it requires that programmers develop different schedulers for their concurrent tasks for different systems and workloads to extract high-performance from their programs.

Thus, we employ extant JVM adaptation mechanisms to enable future support for Java, that is easy to use, efficient, and scalable for applications with fine-grained parallelism. We present *directive-based*, *lazy*, futures (DBLFutures) with support of *as-if-serial* exception handling and side-effect semantics for Java. Using this model, programmers use a future directive, denoted as a new Java annotation, `@future`, to specify potentially asynchronous computations within a serial program. Moreover, programmers are not responsible for deciding when or how to execute these computations at runtime. The DBLFuture-aware JVM recognizes the future directive and makes effective scheduling decision automatically and adaptively by exploiting its runtime services (recompilation, scheduling, allocation, performance monitoring) as well as detailed, low-level, knowledge of system and program behavior.

The *as-if-serial* exception handling mechanism that we present, delivers exceptions to the same point at which they are delivered when the program is executed sequentially. By simply removing our future annotations, the concurrent version is the same as the serial version. This model enables programmers to develop and reason about serial programs first and then introduce parallelism gradually and intuitively, which significantly simplifies the process of parallel programming so that more applications programmers, not just expert programmers, are better able to take advantage of the current and next generation of systems with multiple processing cores.

We enhance further the programmer productivity of future programming in Java by supporting *as-if-serial* semantics for the execution of side effects. That is, programmers are no longer required to reason about the shared memory accesses of future executions and to ensure correctness. Instead, programmers can focus on identifying potential parallelism. The safe DBLFuture-aware JVM guarantees to deliver the correct *as-if-serial* semantics employing concurrent execution when possible.

The as-if-serial side-effect semantics for futures has been investigated in the Safe Future project [153]. However, we find that it has similar programmer productivity and performance disadvantages as the current library-based, Java Future implementation does. In addition, the system does not support nested futures, which makes it impractical since such support is essential for a wide range of applications, in particular, the divide-and-conquer style of applications (that implement fine-grained parallelism).

We extend our DBLFuture system with techniques of this safe future system. In addition, we add efficient support for nested futures, and investigate ways to improving its performance via exploiting the laziness of our DBLFutures system, as well as the adaptation mechanisms in the JVM.

We implement our system in a popular, open-source JVM and then empirically evaluate its efficacy using a number of Java programs that implement fine-grained parallelism. Our results show that the lazy future scheduling system produces comparable performance to the hand-tuned schedulers. In addition, our directive-based model enables speedups of 2-11 times over the interface-based approach implement in modern Java systems. Therefore, DBLFutures are significantly more scalable, and impose very low overhead. Our as-if-serial exception handling implementation introduces negligible overhead for applications without exceptions, and guarantees serial semantics of exception handling and side effects for applications that throw exceptions. Finally, our as-if-serial side-effect implementation imposes acceptable overhead for tracking shared data accesses, and negligible overhead for managing execution contexts.

In summary, in this thesis work, we investigate novel ways to exploit the adaptation of a Java virtual machine to enable both programmer productivity and efficiency. We do so for devices at both ends of the performance spectrum: battery-powered, resource-constrained devices, and multiprocessing systems. We propose techniques that use feedback within the JVM about the program execution behavior and the underlying resource availability, to guide code unloading in low end devices and to enable easy and scalable future task execution in high end devices. We implement our approaches in real JVMs and evaluate their efficacy for a wide range of Java programs. We show that by employing adaptive JVM technology in novel ways, we are able to facilitate programmer productivity and application efficiency concurrently to a much larger degree than that is possible with extant technology.

1.4 Outline

The outline of the remaining of this dissertation is as follows:

In Chapter 2 we give an overview of state-of-the-art adaptive techniques for JVM code optimizations as the background of our work. Many of the concepts and methodologies used in these techniques are important building blocks in our work.

The rest of the dissertation is organized into two parts. The first part discusses our work on automatic code management for resource constrained JVMs to enable compilation (and thus high-performance) for memory constrained devices without programmer intervention. In Chapter 3, we describe the problem of compiled native code management in resource constrained JVMs. We present analysis data to show the potentials of solving the problem. In Chapter 4, we discuss how we perform automatic unloading of compiled native code in a popular research JVM using adaptive technique and framework. We compare different unloading heuristics using empirical performance data.

The second part of this dissertation discusses our work on an adaptive implementation of futures in JVMs to enable easy and efficient parallel programming in Java. In Chapter 5, we introduce the future construct. We overview the current future support in Java 5 and discuss the advantages and limitations of its approach. In Chapter 6, we focus on an adaptive and lazy scheduling technique for fine-grained futures, which relieves programmers from managing future scheduling manually and explicitly, and enables faster execution speed. In Chapter 7, we further improve the easiness of parallel programming using futures in Java by allowing directive-based futures. We also show that directive-based futures have less overhead by creating less future objects than that of Java 5 futures. In Chapter 8, we discuss the support of as-if-serial exception handling for our directive-based lazy futures to enable smooth migration of serial programs to parallel versions. We show that our as-if-serial exception handling support introduces negligible overhead. In Chapter 9, we complete our support of easy and efficient futures in Java by enabling as-if-serial side-effect of futures. This allows not only "safe" future execution, but also easy serial-to-parallel program conversion coupled with as-if-serial exception handling.

We conclude our work and discuss future work in Chapter 10.

Chapter 2

Background on Adaptive Optimization in JVMs

In this chapter, we overview the current adaptive optimization techniques. Adaptive optimization is the process of optimizing only a small subset of program code in attempt to trade off the overhead of optimization for improved code quality (and thus performance). Adaptive optimization systems use dynamic information about the execution of a program to decide (i) when to apply optimization, (ii) which optimizations to apply, and (iii) whether there are aggressive specializations of the code that are possible and cost effective. The dynamic information typically identifies frequently executed code regions – adaptive JVMs predict that such regions are likely to continue executing and thus warrant the overhead required for the application of optimization. We refer to frequently executed code regions as “hotspots”.

The representative systems of adaptive optimization in Java include smart JIT [121], Sun HotSpot^{TM} [74, 117], Intel JUDO [32], IBM Product JIT [141, 142], and IBM

JikesRVM [20, 8]. In the following sections, we will discuss some common issues in adaptive optimization systems and then detail IBM JikesRVM as an example.

2.1 Execution Models

A decision that an adaptive optimization system must make is how to execute the program before any hot spots are identified, i.e., what is the baseline execution engine? One solution is to use an interpreter as that in smart JIT [121], Sun HotSpot^{TM} [74, 117], and IBM Product JIT [141, 142]. The other is to use a fast compiler which performs very few or even no optimization. The latter is adopted in Intel JUDO [32] and IBM JikesRVM [20, 8].

Both solutions have their own advantages and disadvantages. The mixed model (an interpreter and compiler) leads to good responsiveness of the system at startup time, which is very important for interactive applications. Also this model imposes less memory burden on the system because only a small portion of the program is compiled and stored in memory. However, the mixed execution of interpreted code and compiled code dramatically complicates the runtime system (to facilitate control transfer between interpreted and compiled code) as a result of the very different execution conventions. Moreover, interpretation is slow due to poor code quality and re-interpretation of previously executed code.

In contrast to the mixed model, the *compiler-only* model (a fast compiler and a full compiler) enables seamless transition between unoptimized and optimized compiled code. All methods are compiled by the fast compiler and stored for later reuse. This enables fast execution, but at same time results in a much larger memory footprint than that of interpretation. In addition, the compiler-only model suffers from slow startup time (and thus user perception of program performance).

Moreover, two levels of translation may not be sufficient to achieve the best trade-off between compilation cost and performance. For example, if we specify a lower threshold for "hotness", some optimization opportunities may be identified earlier. However, too many methods may be identified as "hot", which will introduce larger compilation cost and longer startup time. On the other hand, if we set a higher threshold for triggering optimization, we may miss some optimization opportunities. Also, a long learning time is required to identify hot methods.

Multi-level compilers address this limitation [142, 20]. These systems decompose the compilation process into multiple levels, each level with its own threshold and associated optimizations. The higher the level, the more complex optimizations the system performs. Such systems also have several drawbacks. For example, the "hottest" methods must be compiled several times before they reach the highest code quality, which results in longer learning time and more code expansion and compilation overhead. In short, there is no perfect execution model: one must pick one based on the tradeoff of memory footprint, responsiveness, and performance for a particular application suite.

2.2 Profiling: Identify Hot Spots and Collect Application Behavior

A precondition of any adaptive optimization is the availability of information about the runtime behavior of an application. Moreover, the runtime must be able to extract this information automatically and accurately without programmer intervention. Online profiling is the most commonly used method for identifying hot spots and collecting application runtime information. However, online profiling faces the same challenge as that of dynamic compilation: the more accurate the profile is, the more overhead the profiling introduces since the time for online profiling is also part of total execution time. As such, a runtime system must carefully trade off profile accuracy and profiling overhead.

There are primarily two types of online profilers: instrumentation-based and sample-based. Instrumentation-based profilers guide compiler insertion of extra instructions into the original program to collect data. For example, the Intel JUDO system [32] inserts counters to the entry and loop back edges of a method in the first level compiled code to catch its execution frequency. More complex and dedicated instrumentation can be inserted by the runtime into the original program to gather information such as branch taken frequency, invoked call sites, and runtime invariants. Instrumentation-based profiling provides accurate profile data, however, it can also introduce significant overhead. In addition, such systems require recompilation to remove the instrumentation. To reduce instrumentation overhead, some systems like Intel JUDO do not instrument the optimized code. But, this disables any additional or multi-level optimization opportunities.

Sample-based profilers do not insert code into the original program. Instead, a background thread periodically records snapshots of interesting parts of the runtime system. For example, the IBM JikesRVM system [8] records the top two frames on the stack of each thread per 10 ms and uses this information to approximately identify hot spots. Sample-based profiling has lower runtime overhead and can be disabled without any recompilation. Also, results in [154, 94] show that sample-based profiling is accurate enough in most cases to enable significant performance improvements.

Simple sample-based profiling, however, cannot collect all desired information. For example, a coarse-grained sampler can not gather the frequency of basic blocks and taken branches. Some researchers have proposed mechanisms that combine instrumentation-based and sample-based profiling techniques. For example, [11, 10] present a low overhead instrumentation framework and its application in online feedback-directed optimization for Java. The basic idea is to have duplicate copies of interesting pieces of code: one that contains the original code and is used for normal execution; the other that contains instrumented code and is used for gathering profile data. The instrumented copy is only invoked periodically to gather sample data. The sample frequency is configurable. This instrumentation sampling framework gathers necessary profile data with low overhead. However, this results in a larger memory footprint due to code duplication.

[142] proposes another technique to combine instrumentation and sampling. The instrumenting profiler in this system dynamically inserts instrumentation into the target method using an instrumentation planner. Later, after the desired data is gathered, the instrumented code is automatically extracted using code patching. This mechanism does not cause the code space problem, but may introduce architecture-specific complexities such as maintaining cache consistency.

Another way to reduce overhead of profiling is to gather profile data offline and use the information online via annotation [95]. Annotation-based approaches can significantly reduce the online profiling overhead and the learning time of optimization opportunities. However, they suffer the well known cross-input problem: offline-collected profiles may not reflect the runtime application behavior. Although techniques for coupling online and offline profile information are proposed [94], the cross-input problem still remains.

2.3 Decision Models

Once application runtime behavior is collected, the next step is to decide whether a method is hot enough and which subset of optimizations should be performed. A simple and commonly used way is threshold-triggered: a threshold is preset for each level of optimization; once the frequency exceeds the preset threshold, corresponding optimizations are performed. A threshold-triggered decision model is easy to implement, but is too coarse-grained: whether the benefit gained by the performed optimization can *actually* outweigh the overhead is not evaluated.

Instead of using threshold-triggered decision model, the IBM JikesRVM system [8, 7] employs a cost/benefit analytic model to determine if a hot method should be optimized and at which level it should be optimized. Their model is as following:

$$\begin{aligned} L_{opt} &= \{k | T_{k\Delta} > 0, T_{k\Delta} = max(T_{j\Delta}), i \leq j \leq N\}, \\ T_{j\Delta} &= C_j + T_j - T_i, \\ T_j &= T_i \star S_i / S_j \end{aligned}$$

Where L_{opt} is the result optimization level for the method m, i is the current compiled level of m (might be unoptimized level), N is the highest compilation level, C_j is the compilation cost for m at level j. T_i is the predicted future execution time of m if m keeps running at level i, T_j is the predicted future execution time of m if it is optimized at level j. T_j can be estimated based on T_i and the difference between speedups of level i (S_i) and level j(S_j).

This decision model is more accurate than threshold-triggered ones since it decides whether there is any benefit to optimization of a method ($T_\triangle$ must be positive) and tries to choose the most profitable optimization level for the method. In addition, it can promote a hot method directly to the most profitable optimization level rather than slowly updating it by one level at a time. However, to implement this decision model, some parameters must be provided: the cost of a compilation level, the speedup of the new optimization level over the previous one, and the future execution time of the method. All of these parameters are highly application and resource dependent and the precise prediction of them is still an open research problem.

2.4 Example System: JikesRVM

The JikesRVM is a research virtual machine developed at IBM T.J. Watson Research Center. This system is written almost entirely in Java and is one of the most advanced adaptive optimization systems currently. We have discussed its execution model, profiling techniques, and decision model in the previous sections. In this section, we show how different components of JikesRVM work together to deliver high performance. We do so since, it is this system that we employ as our base infrastructure and extend in this dissertation work.

The architecture of the adaptive optimization system in JikesRVM consists three subsystems: the *runtime measurement subsystem*, *the controller*, and the *recompilation subsystem*. In addition, the *AOS database* provides a repository of previous decisions for later query. The runtime measurement subsystem is responsible for gathering information about the application behavior (by samplers), summarizing the information (by organizers), and passing the summary to the controller via an event queue. A decay organizer periodically refreshes the sample data so that more recent behavior is emphasized.

Currently, two kinds of sample data are collected in JikesRVM: method invocations and call edges. The former is used to identify hotness and the latter is used to guide runtime inlining. The controller coordinates the activities of the other two subsystems based on the profile data. It takes information from the event queue and uses the cost/benefit analytic model to determine whether a higher level optimization should be performed and which level yields the best performance tradeoff. Then it puts the recompilation requests in the compilation queue, along with instrumentation plans that will provide desired profile data for further optimization. The recompilation subsystem takes compilation plans submitted by the controller and performs the requested compilation in the background. Finally, the previous code is replaced by the newly optimized code and execution continues with the optimized version.

All three subsystems are carefully engineered to impose very little overhead. For example, the total overhead introduced by the controller and organizers is only about 1%, which is negligible comparing to the significant performance improvements delivered by the adaptive optimization system. Performance improvements of 11% on average and up to 73% are reported for feedback directed inlining in [8].

Part I

Automatic Code Management for Resource Constrained JVMs

Chapter 3

Code management in Resource Constrained JVMs

Java virtual machines (JVMs) [102] have become increasingly popular for execution of mobile and embedded device applications. Statistics from JavaOne 2007 [83]. show that there are total 1.83 billion Java-enabled mobile devices to date (May 2007). This wide-spread use of Java for embedded systems has resulted from significant advances in device capability as well as from the ease of program development, security, and portability enabled by the Java programming language [17] and its execution environment (JVMs).

To execute a Java program, the JVM translates the code from an architecture-independent format (bytecode) into the native format of the underlying machine. In this chapter, we overview two models of code translation in JVMs, i.e., the interpretation model and the compilation model. We compare their advantages and disadvantages in the scope of resource constrained environments. In particular, we discuss why the compilation model is desirable even for resource constrained systems and the existing challenges to use the compilation model (e.g., big memory footprint, compilation overhead, memory management overhead, etc.). We then provide an empirical study of size and usage patterns of the compiled code in JVMs for a set of benchmarks, which reveals many opportunities of efficient, automatic code management techniques, and motivates our adaptive code unloading work in Chapter4.

3.1 Interpretation Versus Compilation

Most JVMs for embedded and mobile devices translate bytecode employ interpretation, i.e., instruction-by-instruction conversion of the bytecode [99, 26, 89]. The reason for this is that such translation is easy to implement and imposes no perceivable interruption in program execution. In addition, the native code that is executed is not stored; if code is re-executed, it is re-interpreted. The primary disadvantage of using interpretation is that an interpreted program can be orders of magnitude slower than compiled code due to poor code quality, lack of optimization, and re-interpretation of previously executed code. As a result, interpretation wastes significant resources of embedded devices, e.g., CPU, memory, battery, etc.

To overcome the disadvantages imposed by interpretation, some JVMs [32, 141, 5, 75] employ dynamic (Just-In-Time (JIT)) compilation. Programs that are compiled use device resources much more efficiently than if interpreted due to significantly higher code quality. Compilers translate multiple instructions concurrently which exposes optimization opportunities that can be exploited and enables more intelligent selection of efficient native code sequences. Moreover, compilation-based systems store code for future reuse obviating the redundant computation required for re-interpretation of the same code. According to studies of energy behavior of JVMs and Java applications in [149, 48], JVMs in the interpreter mode consume significantly more energy than in the JIT compiler mode. Thus, the JIT approach is a better alternative for embedded JVMs from both performance and energy perspectives.

Despite the execution speedup of compiled code over interpreted code and its power efficiency, dynamic compilation is still not widely used in JVMs for resource-constrained environments due to the perceived memory requirements. Dynamic compilation enlarges the JVM memory footprint in three primary ways: The extra code base introduced by the JIT engine, the intermediate data structures generated by the compiler during compilation, and the compiled code stored for reuse. Significant engineering effort and research [1, 92, 18] have been performed to address the first two problems by making the JIT compiler more lightweight while still enabling generation of high quality code.

Compiled native code is significantly larger than its bytecode equivalent. In resource-constrained environments, since the total available memory is limited, the memory consumed by these code blocks reduces the amount of memory available to the executing application. This increase in memory pressure can preclude some programs from executing on the device. Moreover, for systems that use garbage collection to manage compiled code, compiled code increases memory management costs, which can be significant when memory is severely constrained. If applications' code and data share the same heap and are managed by the same garbage collector, the impact of compiled code on memory management costs can be even more severe. As a result, dynamic compilation introduces memory overhead for compiled code not imposed by interpreter-only systems which can in turn negate any benefit enabled by code reuse and improved code quality. The goal of our work is to reduce the memory requirements introduced by compiled code to make the compilation model more feasible for the resource constrained JVMs.

3.2 Characteristics Study of Compiled Code in JVMs

To identify potential solutions for reducing the memory requirements introduced by compiled code for resource constrained JVMs, we have conducted an empirical study of the size and usage patterns of the compiled code in JVMs for a wide range of benchmarks. We have performed a series of experiments that measure various static and dynamic characteristics of native code, e.g., size, usage statistics, etc.

Bench-marks	Byte code (KB)	ARM Native Kaffe KB (/BC)	IA32 Native Kaffe KB (/BC)	IA32 Native Jikes KB (/BC)	Dead after startup KB (Pct.)
compres	12.4	210.8 (17.0x)	96.7 (7.8x)	98.0 (7.9x)	70.8 (72%)
db	14.5	242.2 (16.7x)	114.6 (7.9x)	105.9 (7.3x)	89.2 (85%)
jack	42.4	788.6 (18.6x)	318.0 (7.5x)	284.1 (6.7x)	72.5 (26%)
javac	78.3	1252.8 (16.0x)	555.9 (7.1x)	469.8 (6.0x)	75.9 (16%)
jess	32.9	559.3 (17.0x)	250.0 (7.6x)	223.7 (6.8x)	167.9 (75%)
mpeg	56.6	1386.7 (24.5x)	464.1 (8.2x)	452.8 (8.0x)	357.4 (79%)
mtrt	21.1	N/A	173.0 (8.2x)	160.4 (7.6x)	117. 6 (73%)

Table 3.1: Size and behavior of the compiled native code in JVMs.

Table 3.1 shows the size in kilobytes (KB) of the bytecode and native code for the SpecJVM benchmark suite [135]. Column 2 is bytecode size and columns 3–5 show the size of native code and the ratio (/BC) of native code size to bytecode size. We gathered this data using two platforms, ARM and IA32, and two JVMs, the JikesRVM [5], and the Kaffe embedded JVM [89] with jit3. Since JikesRVM does not have back-end for ARM, we show only data for IA32. This data shows that IA32 native code is 6-8 times of that of bytecode for both JikesRVM and Kaffe for these programs. ARM code is even larger (16-25 times that of bytecode) since its RISC-based instructions are simpler than the CISC IA32 instructions (which do more work per instruction). Even if the systems uses the compact instruction form, e.g., ARM/THUMB (potentially reducing native code size by half), the size of compiled native code is likely to be much larger than that of the corresponding bytecode.

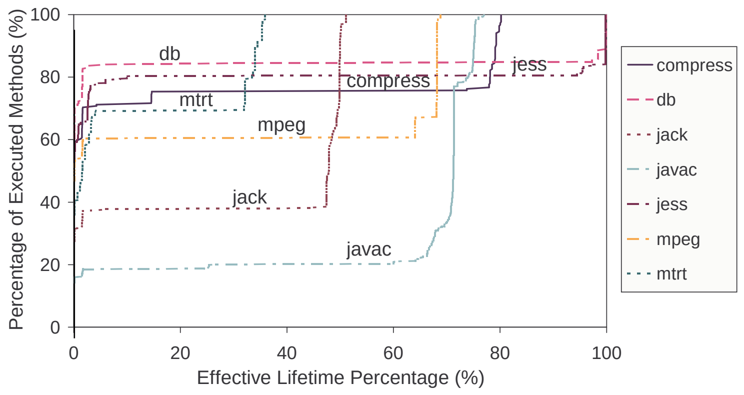


Figure 3.1: CDF of effective method lifetime as a percentage of total lifetime. The effective lifetime of a method is the time between its first and last invocations; the total lifetime of a method is the time from its first invocation to the end of the program. A point, (x, y), on a curve indicates that y% of that benchmark's executed methods have an effective lifetime of less than or equal to x% of its total lifetime.

The final column shows the amount of code that goes unused after program startup (we define startup as the initial 10% of the execution time). Interestingly, a large amount of executed code becomes dead after program startup; this portion of code remains in the systems and consumes precious system memory needlessly.

In addition, we found that a majority of the code that remains after startup in many benchmarks has short life spans. Figure 3.1 graphs the cumulative distribution functions of *effective lifetime percentage* of methods, i.e., the percentage of the effective lifetime (time between the first and last invocations of a method) over the total method lifetime (time from the method's first invocation to the end of the whole program). This metric is similar to the *trace lifetime* used in [69]. This figure shows that for most of the benchmarks (all but *javac* and *jack*), more than 60% of methods are effectively live for less than 5% of the total time they remain in the system.

We also found that methods with long effective lifetimes commonly are invoked infrequently. For example, method spec.benchmarks._213_javac.ClassPath.<init> has an effective lifetime of 75%, but is only invoked 4 times and executed for only 0.1% of its total effective lifetime.

In summary, the native code size in JVMs is much larger than that of bytecode. Since native code is stored by compilation-based JVMs for reuse, it consumes precious memory space on resource constrained devices. Moreover, a big portion of the compiled code blocks become dead after the startup phase. Finally, for those code blocks that are live after the startup phase, majority of them have very short lifetime. All of these invocation characteristics of the compiled code presents many opportunities for removing code blocks from the system temporarily or permanently.

Chapter 4

Adaptive Code Unloading

To exploit the performance enabled by a compilation-based approach to bytecode translation, and to reduce the memory requirements of such an approach, we propose a novel technique, called *adaptive code unloading*. Adaptive code unloading is an alternative to either not compiling code (as in the interpretation model) or keeping all compiled code (as in the current compilation model). A JVM with adaptive code unloading compiles all methods initially, then discards (*unloads*) the compiled code if unused or infrequently used, or when the system is severely memory constrained.

Our study of size and usage patterns of compiled code in JVMs in Section 3.2 reveals many opportunities for removing code blocks from the system temporarily or permanently to reduce the memory requirements. For example, code that is not used after startup can be unloaded after the system passes the startup stage. In addition, those code blocks that remains in the system after startup but has short life spans can also be considered candidates for unloading. Finally, when memory is highly constrained, we can even consider to unload methods with long life spans but are invoked infrequently to release memory pressure in the system temporarily.

In this chapter, we describe an extensible framework for adaptive code unloading that we developed to relieve memory pressure imposed by compiled code in resource-constrained JVMs. We identify the various components of the framework and explain how each component works and cooperates with the others to facilitate adaptive code unloading. We then use the framework to investigate a wide range of unloading strategies that lead to different tradeoffs between the JVM memory footprint and execution performance. Finally, we empirically compare the various strategies to identify the best-performing combination of design decisions, and evaluate the overall efficacy of the system.

4.1 Code Unloading Framework

Figure 4.1 depicts the extensible framework for adaptive code unloading that we developed to relieve memory pressure imposed by complied code in resource-constrained JVMs. The outer box is the boundary of a JVM. Inside this box, the left part is the control-flow of a JVM that employs dynamic and adaptive compilation, which we believe is crucial to achieve high performance with small memory footprint in resource-constrained environments. Adaptive compilation is the process

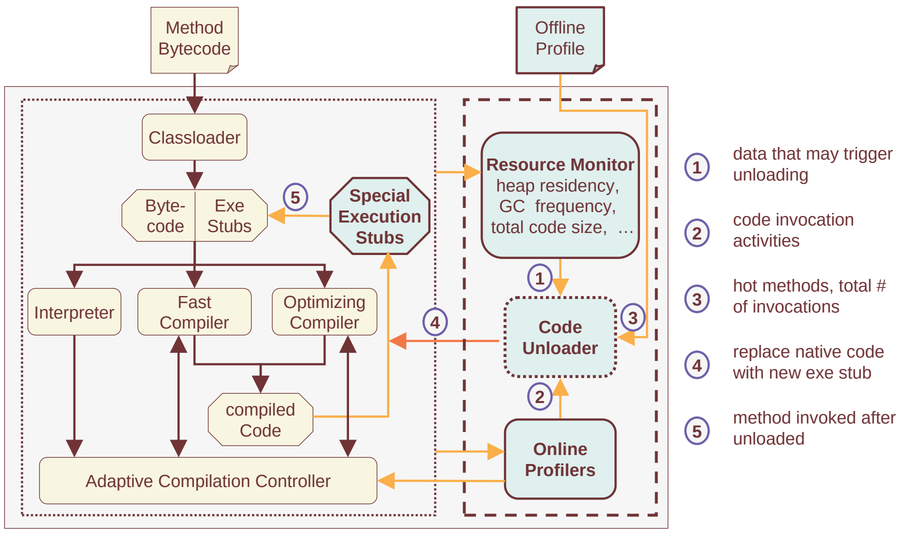

Method Bytecode
Offline Profile
Classloader
Byte-code
Exe Stubs
Special Execution Stubs
Resource Monitor heap residency, GC frequency, total code size, …
Interpreter
Fast Compiler
Optimizing Compiler
Code Unloader
compiled Code
Online Profilers
Adaptive Compilation Controller
1 data that may trigger unloading
2 code invocation activities
3 hot methods, total # of invocations
4 replace native code with new exe stub
5 method invoked after unloaded

Figure 4.1: Overview of the adaptive code unloading framework

of selectively compiling or recompiling code (guided by online performance measurements) that has been interpreted or compiled previously in an attempt to improve performance [8, 32, 75]. The right part of the figure shows our JVM extensions (darkened components) that enable adaptive code unloading.

While programs are executing, the *Resource Monitor* collects information about resource behavior, e.g. heap residency data, garbage collection (GC) invocation frequency, and native code size, etc. The online and offline *profilers* collect information about application behavior, such as hot methods and invocation activity of each method. The code unloading system can share the profilers with the adaptive compilation system if possible to reduce overhead. The *Code Unloader* takes information from these components, analyzes the cost and benefit of unloading, and decides when code unloading should commence and which methods to unload. As such, the framework *dynamically and automatically* identifies dead or infrequently used native code bodies and unloads them from the system to relieve memory pressure *whenever necessary*.

Once the unloader selects a method, it replaces its code entry in the dispatching table with a special stub, which we refer to as the *execution stub*. This stub is similar to the mechanism used by the JVMs to enable lazy, Just-In-Time compilation [5, 96]. However, we add additional information to specify how to execute the method, e.g., interpreting, fast compiling without optimization, or compiling at a particular optimization level, if it returns to the system after being unloaded.

The system reclaims the native code block of an unloaded method during the next garbage collection cycle since it is no longer reachable by the program. If the executing program invokes a method that has been unloaded, the execution stub will invoke the interpreter or an appropriate compiler to re-translate the method. If the method is compiled, its address (that of the stub) is replaced with that of the newly compiled method in the dispatching table. Future invocations of the method by the program execute the compiled method directly through the table entry.

Since the unloader and other framework components must operate *while* the program is executing, we designed the system to be very light-weight. Moreover, the framework implements a flexible and extensible foundation via a well-defined interface that we (and others) can use to investigate, implement, and empirically evaluate various code unloading strategies.

We implemented the framework as an extension to the open-source Jikes Research Virtual Machine (JikesRVM) [5]. JikesRVM is a compiler-only JVM, and thus, the special execution stub either fast compiles the method, or optimizes the method at certain level directly. No interpretation is performed. This implementation can be easily extended to handle interpretation in the execution stubs if an interpreter is included in the JVM. We then used the resulting system to investigate four strategies that identify unloading candidates and four strategies that trigger unloading. We detail each of these strategies in the following sections.

4.2 Unloading Strategies

There are four primary decisions that any dynamic code unloading system must make:

- When unloading should be triggered;
- How unloading candidates should be identified and selected;
- Whether optimized code should be handled differently from unoptimized code or not;

- How the system should record the profile information used in the decision process.

There are a number of possible answers to these questions; each of which leads to a different tradeoff between the JVM memory footprint and execution performance. We investigated a number of strategies that attempt to answer these questions in an effort to identify the best-performing combination of design decisions.

4.2.1 Triggering an Unloading Event

We first investigated various ways in which we can trigger unloading. That is, we implemented strategies that decide *when* to unload. We considered four different triggers: Maximum invocation count, a timer, GC frequency, and code cache size.

The first strategy is called *Maximum Call Times (MCT)* triggered. We use offline profiling to collect the total invocation times for each method. Then the code unloader uses this information to trigger unloading of methods following completion of the last invocation of each. The system records the invocation times upon each method return. This strategy is not adaptive to execution behavior but guarantees all methods are unloaded when the program is finished with them. The strategy introduces no additional compilation overhead since unloaded methods will not be reused again and therefore reloaded. This strategy is not realistic in the sense that it is unlikely that we will be able to have such accurate information (last call time) for all methods given non-deterministic execution and cross-input program behavior. This strategy does however, capture the potential of unloading dead methods and we use it as a limits study.

The second strategy is *TiMer (TM)* triggered. Our system unloads code at fixed, periodic intervals. To be simple and efficient, we use a thread-switch count to approximate the timer since thread switching occurs at approximately every 10 ms in our JVM. This JVM employs stop-the-world style of the garbage collection (GC) which halts all thread switching during GC. To compensate for these time periods, we extended the GC system to update the thread-switch count at the end of each GC by the amount corresponding to the time spent performing the GC.

Since *TM* is timer based, it is not adaptive, i.e., it does not exploit information about the execution characteristics of the VM, such as memory availability. Moreover, the length of the period is a difficult parameter to set. We found that different period lengths perform better for different programs and even for the same program across inputs. We detail the parameters we use in Section 4.3.

To address the limitations of TM, we investigated an adaptive *Garbage Collection (GC)* triggered strategy. The intuition behind this strategy is that code unloading frequency should adapt to the dynamically changing resource availability. Unloading frequency refers to how often unloading occurs, e.g., every 60 seconds, every 10 GC cycles, etc. When memory is highly constrained, code unloading should be triggered more frequently to relieve memory pressure. When unconstrained, the system should perform unloading less frequently to reduce the overhead of the unloading process itself and to reuse compiled code as much as possible. To this end, we initially investigated the use of *heap residency*, i.e., the ratio between the amount of memory occupied and the total heap size, to measure memory usage. If the system is short of memory, we will see high heap residency following garbage collection.

However, using heap residency alone to measure memory usage may raise many false alarms. For example, some programs may allocate most of its memory at the beginning of execution. The heap residency will remain high and cause repeated unloading even when no further allocations are made by the program. To avoid false alarms, we use heap residency indirectly by considering GC frequency. When the amount of available memory space is small and programs repeatedly allocate memory, GC will occur frequently. To capture this behavior, at the end of each GC cycle, the resource monitor forwards the percentage of execution time spent in garbage collection so far to the unloader so that the unloader can adjust the unloading frequency.

We specify unloading frequency using a dynamic "unloading window". Our system initiates unloading once per window. The size of an unloading window is defined by a specific number of garbage collection cycles. Users or system administrator can specify a minimal window size using command line options. The unloading system divides this minimal window size by the percentage of time spent in GC to determine the dynamic window size adaptively. The value is decremented upon each GC. When it reaches zero, the system performs unloading. Following each unloading session, the system resets the window size.

As we showed in Section 3.2, more than 70% of code is dead following the initial 10% of program execution time, i.e., the startup phase, for most benchmarks. To exploit this *phased* behavior, we investigated different unloading strategies that operate at different stages of program lifetime. We define the first 4 GC cycles (which we empirically determined and which can be changed via a command-line parameter) to be startup period. During the startup phase, the program uses heap residency alone to facilitate more aggressive unloading; following this period, the system uses the percentage of time spent in GC to determine when to unload.

The last strategy that we investigated is a *code Cache Size (CS)* trigger. This strategy is similar to "code pitching" in the Common Language Runtime (CLR) [16]. In this strategy, we store the compiled native code in a fix-size code cache. When the cache is exhausted, our system performs unloading. The advantage of this strategy is that the size of compiled code body is guaranteed to be below a specified maximum. However, we found that it is very difficult to find a general optimum cache size for all applications. An alternative is to use a small size cache initially and allow the cache to grow as necessary. However, to determine how often and at what increments to grow is equally difficult and application-specific. Regardless of the limitations, we are interested in understanding how this strategy impacts performance. We therefore parameterized this strategy for initial cache size, the growth increment, and the number of unloading sessions that triggers cache growth. We discuss how we selected the parameters for our empirical evaluation in Section 4.3.

4.2.2 Identifying Unloading Candidates

To identify which code should be unloaded, we developed strategies that identify methods that are *unlikely* to be invoked in the future. We hypothesize that the methods that have not been invoked recently are not likely to be invoked in the near future and can be unloaded. We present four techniques that use program profiling as well as snapshots of the runtime stack to identify unloading candidates.

The first strategy, called *Online eXhaustive profiling (OnX)*, uses exhaustive online profiling information to identify unloading candidates. To obtain method invocation counts, we modified the compiler to instrument methods. The instrumented code marks a bit each time a method is invoked. When unloading is triggered, the system unloads unmarked methods and resets the mark bits. In this way, both dead methods and infrequently invoked methods can be unloaded. This strategy guarantees that every method that has been invoked since the last unloading session will have its mark bit set. Therefore, only recently unused methods will be unloaded. However, since profiling is performed for every single method invocation, this strategy has the potential for introducing significant execution time overhead.

To overcome this limitation, we also investigated *Online Sample-based profiling (OnS)*. In this strategy, the JVM sets the mark bits of the top **two** methods on invocation stacks of application threads for every thread-switch (approximately every 10 ms). We determined this value (two) empirically in an attempt to balance the trade-off between introducing significant overhead of complete stack scan and incorrectly unloading used methods. Moreover, this sample-based approach can be turned off when sufficient memory is available to avoid *all* overhead. For exhaustive profiling (OnX), we do not turn off profiling since doing so requires recompilation and possibly on-stack replacement [49].

We also investigated the efficacy of using perfect knowledge of method lifetime – to facilitate our MCT (maximum call times) trigger. For this strategy, we gather the total invocation count for each method *offline*. Then we annotate this value in the class file as a method attribute for use by the JVM during program execution using an annotation system that we developed in prior work [95, 94]. At runtime, we use online profiling to identify a method's last invocation; at which time, we mark the method to be unloaded. Instead of the 1-bit counter used in *OnX*, this strategy requires that we increment an integer counter for each invocation. We assume that the same input is used for both offline profiling and online execution, that is, we use perfect information of method invocation counts. We refer to this strategy as *Offline exhaustive profiling (Off)*.

Our final strategy, called *NP* for “*No Profiling*”, simply unloads all methods that are not currently on the runtime stack when unloading is triggered. This strategy is the most aggressive one. The advantages include simplicity of implementation and avoidance of all profiling overhead. However, this strategy is not adaptive and may unload methods that will be invoked in near future, introducing significant recompilation overhead.

4.2.3 Unloading Optimized Code

Heretofore, we have not considered whether the method we are unloading is optimized or not. Unloading optimized code has the potential of increasing the performance penalty of unloading when a method is later reused. This is because optimizing code is much more expensive than compiling code without optimization. We refer to the latter as *fast compilation*. For JVM configurations in which only fast compilation is used, there is no optimized code to unload.

However, in addition to a JVM configured with only a fast compiler, we consider an *adaptive* configuration. In an adaptive optimization, a method is initially fast compiled. The JVM samples methods to identify those where the most time is spent, i.e., that are “hot”. The system then uses a counter-based model (as HotSpot [75] employs) or a cost/benefit analytic model (as JikesRVM [7] employs) to decide when and at which level to compile or recompile a method depending on its hotness. Higher level of optimization enables larger performance improvement, but also introduces additional compilation overhead.

For an adaptively optimizing JVM configuration equipped with code unloading, we must determine the level at which unloaded optimized code should be compiled if it is later reused. If we use fast compilation, the method will have to progress through the optimization levels again if it remains hot after unloading. Alternately, if we optimize the method at the level at which it was when it left the system, it may no longer be hot when it returns; this imposes unnecessary compilation overhead on the program.

We implemented three additional strategies to study the performance impact of unloading optimized code. First, we insert an optimization level hint to the recompilation stub. If an unloaded hot method is later invoked, our system re-compiles it at the optimization level that it was before unloaded. We call this strategy *RO*: *R*e-Optimize hot methods using an optimization level hint. Second, we avoid unloading all hot methods. We call this strategy *EO*: *E*xclude *O*ptimized methods from unloading. This strategy avoids the compilation overhead of optimization. However, some programs may have a significant percentage of hot methods (that should be unloaded). For example, in *javac* from the SpecJVM98 benchmark suite, there are 78 out of 876 methods that are hot. In comparison, *db* only has 3 out of 151 methods that are hot. Our third strategy accounts for cases like *javac*: The optimized (thus hot) methods will be unloaded. However, we delay unloading until the method is unused for two consecutive unloading sessions. If an optimized method is unloaded after giving second chance, it is fast compiled at next time it is invoked. We call this strategy *DO*: *D*elay unloading of *O*ptimized methods.

4.2.4 Recording Profile Information

Another implementation issue that we must address is how the system should record profile information. As we described above, for the implementation of the "what" strategies, we use a bit array to record information gathered either by instrumented code or by sampling. Every time a method is invoked or is on the top of stack when thread switching occurs, the system sets a bit in the array that corresponds to the method. When unloading occurs, the system unloads all unmarked methods and resets the array.

The benefits of using a bit array to record profile information, are that the implementation is simple and access to the array is very efficient. However, a bit array does not capture the temporal relationship between method invocations. That is, as long as two methods are invoked since last unloading session, they will be treated equally by the next unloading session no matter which one is the more recently invoked.

To evaluate whether such temporal information is important or not, we implemented an additional mechanism for recording profile information in which all methods are linked via a doubly linked list. A method is inserted at the end of the list when it is compiled. Whenever the method is invoked again (in the exhaustive profiling case) or sampled (in the sample-based case), the system moves it to the end of the list. As a result, all methods are always ordered by their last invocation (or sample) time. Such an implementation enables our use of the framework to investigate the efficacy of using the popular *Least Recently Used (LRU)* cache replacement policy for code unloading.

4.3 Experimental Methodology

We implemented and empirically evaluated the efficacy of our code unloading framework and various unloading strategies in the Jikes Research Virtual Machine (JikesRVM) [5] (x86 version 2.2.1) from IBM Research. Although JikesRVM was not originally designed for embedded systems, it has two different compilation configurations that we believe are very likely to be implemented in the next-generation JVMs (embedded or not): *Fast*, non-optimizing compilation, and *Adaptive* optimization (in which only methods that have the most performance impact are optimized).

We investigate both compiler configurations since it is unclear as to how much optimization should be used by JVMs for embedded devices.

One limitation of JikesRVM is that it does not implement an interpreter and thus, we are unable to use it directly to evaluate the impact of code unloading on selective compilation, i.e., a system that employs interpretation for cold methods and compilation (and increasing levels of optimization) for hot methods, e.g., as in HotSpot JVM from Sun Microsystems [75]. To our knowledge, no open source JVM implements an interpreter, a highly optimizing compiler, and adaptive optimization. As such, we use simulation to evaluate the impact of code unloading for selective compilation JVMs in Section 4.4.3. We describe our simulated setup and assumptions in that section.

We investigate maximum heap sizes of MIN and 32MB to represent memory resource constraints. MIN has the minimum heap size that is necessary for each benchmark to run completion (identified empirically) without an out of memory exception. We use MIN as an example of a scenario in which memory is highly constrained and 32MB as an example of unconstrained memory. Given this experimental methodology, we believe that our results lend insight into the potential benefits of adaptive code unloading on future, compilation-only and selective compilation JVMs for embedded systems.

In our experiments, we repeatedly ran the SpecJVM benchmarks (input 100), on a dedicated Toshiba Protege 2000 laptop (750 MHZ PIII Mobile) with Debian Linux

Benchs	Code (KB)	MIN (MB)	Memory Used(MB)	Exec Time (s)		GC Ratio(%)		CMP Ratio(%)	
				min	32MB	min	32MB	min	32MB
compress	98.4	20	122.2	66.8	61.2	9.99	3.10	0.04	0.04
db	105.3	22	83.7	78.8	50.6	38.96	7.41	0.03	0.05
jack	284.9	6	238.1	624.9	17.9	97.42	28.35	0.01	0.31
javac	468.5	24	232.3	128.9	46.8	77.61	41.89	0.10	0.26
jess	223.1	8	274.3	303.3	27.6	91.99	25.86	0.02	0.20
mpeg	455.4	9	14.3	56.1	54.4	1.69	0.00	0.11	0.11
mtrt	161.3	18	149.3	321.5	29.6	92.66	21.29	0.01	0.12

Table 4.1: Benchmark characteristics for Fast configuration

Benchs	Code (KB)	MIN (MB)	Memory Used(MB)	Exec Time (s)		GC Ratio(%)		CMP Ratio(%)	
				min	32MB	min	32MB	min	32MB
compress	143.8	22	130.3	26.3	21.5	23.59	8.07	0.80	1.04
db	157.8	23	95.3	115.6	45.1	64.78	12.42	0.21	0.51
jack	372.4	9	248.4	130.6	18.2	88.72	32.70	0.33	2.04
javac	582.8	26	247.2	152.3	55.5	80.94	49.42	0.44	1.08
jess	311.8	11	288.7	136.4	23.2	88.23	37.34	0.50	2.73
mpeg	541.4	12	45.9	29.7	20.3	30.27	3.27	2.56	4.45
mtrt	237.8	23	152.8	50.4	22.6	71.05	38.70	1.36	2.93

Table 4.2: Benchmark characteristics for Adaptive configuration

(kernel v2.4.20) using both the Fast and Adaptive JikesRVM compilation configurations. In both configurations, the commonly used VM code is compiled into the boot image. In addition, we employ the default JikesRVM garbage collector, a semispace copying collector. In all of our results, we refer to the reference (unmodified) system as *clean*.

The general benchmark statistics are shown in Table 4.1 (Fast configuration) and 4.2 (Adaptive configuration) for the clean system. In each table, the first column is native code size (including all compiled methods, applications and libraries) in kilobytes (KB). The second column is the empirically identified MIN value. The third column is the total size of memory allocated during the execution. The last six columns show the total execution time (in seconds), the percentage of time spent in GC and the percentage of time spent in compilation. Note that JikesRVM is equipped with a facility to track the time spent by each thread. For IA32, it uses system call ”gettimeofday” to get the current time. CPU time accumulation of one thread is stopped when the thread is switched out and resumed once its execution resumes. Based on this timing facility, JikesRVM is able to measure time spent in GC and time spent in compilation accurately.

To compare the different strategies, a set of parameters is required. We empirically evaluated a wide range of parameters for each strategy and only report results using best-performing values (on average) across all benchmarks. We set 10 GC cycles as the unloading window size for GC (garbage collection triggered), 10 seconds as the interval for TM (timer triggered). For CS (code size triggered), initial cache size is 64KB and grows by 32KB for every 10 unloading sessions (triggered by a full cache).

4.4 Performance Evaluation

In the subsections that follow, we evaluate the efficacy of our code unloading strategies for memory footprint reduction. We then present the impact of this reduc-

What Strategies	MIN		32MB	
	Fast	Adaptive	Fast	Adaptive
Off-TM	38.3	N/A	31.9	N/A
NP-TM	55.1	43.8	51.0	40.3
OnS-TM	53.1	42.8	50.5	38.7
OnX-TM	45.6	34.4	43.7	27.0

When Strategies	MIN		32MB	
	Fast	Adaptive	Fast	Adaptive
Off-MCT	42.7	N/A	50.9	N/A
OnS-CS	52.0	40.3	56.7	41.6
OnS-GC	61.8	46.9	46.3	36.0
OnS-TM	46.7	42.8	30.6	38.7

Table 4.3: Average code size reduction (%) of different "what" and "when" strategies

tion in memory pressure on performance. We evaluated all possible permutations of "what" and "when" strategies. However, we only present a subset of results in this section for conciseness and clear illustration. Finally, we explore the potential impact of coupling code unloading and selective compilation, i.e., a JVM configuration that employs interpretation of cold methods and compilation of hot methods using increasing levels of optimization.

4.4.1 Memory Footprint Reduction

We first compare the average code size reduction over a clean version of the system, in Table 4.3. The clean system is a compile-only JikesRVM system with no code unloading extensions. The left half of the table is for MIN memory configuration and the right half is for 32MB (again, MIN is the minimum heap size in which the program will run and 32MB represents a system with less memory pressure). For each heap size, we also present the results for different compilation configurations (Fast or Adaptive). We show two sets of data in this table: one for "what" strategies and the other for "when" strategies.

The upper part of Table 4.3 shows the four "what" strategies as described in Section 4.2.2: Off (Offline profiling), NP (No Profiling), OnS (Online Sample-based profiling), and OnX (Online eXhaustive profiling). We choose *Timer-triggered (TM)* to be the common "when" strategy since its periodicity enables us to focus only on the impact of different "what" strategies. We omit the results of Off-TM strategy for Adaptive configuration since the adaptive optimization system in JikesRVM is non-deterministic: It uses timing information to decide when and how to optimize and hence, we are unable to obtain deterministic offline profile of method invocation counts for the fast compiled version and the optimized version separately, which are required by Off-TM strategy to trigger unloading correctly.

For both memory configurations, *NP-TM* performs the best, followed by *OnS-TM*, *OnX-TM* and *Off-TM*. Strategy *Off-TM* does not unload a method until it is dead. Thus, it is the least aggressive. *NP-TM* always discards all compiled methods except those on the runtime stack during an unloading session resulting in the largest reduction in average code size. Online exhaustive profiling is more accurate than sample-based profiling in capturing recently invoked methods. Thus, *OnX-TM* unloads fewer methods than *OnS-TM*.

The code size reduction in 32MB setting is less than that in MIN setting in most cases. The reason for this is that programs execute and complete faster when more memory is available. Hence, fewer unloading sessions were triggered. Similarly, the reduction in Adaptive configuration is less than that in Fast configuration.

The bottom part of Table 4.3 compares four “when” strategies as described in Section 4.2.1: MCT (Maximum Call Time triggered), CS (Code Size triggered), GC (Garbage Collection triggered), and TM (TiMer triggered). We used the best-performing “what” strategy – OnS (online, sample-based profile) – for these experiments. The MCT “when” strategy requires an accurate, offline exhaustive profile of methods’ maximum invocation numbers (thus, we prefix the name with “Off-”). Again, we omit Off-MCT for Adaptive configuration due to the non-determinism.

Similarly to the results of “what” strategies, all “when” strategies have a significant code size reduction for all configurations. *OnS-GC* adapts the best to the memory availability: the more the memory is limited, the more native code is unloaded. *OnS-TM* is also sensitive to memory pressure since our implementation of this strategy updates the timer period for the time spent in garbage collection. OnS-TM is less adaptive than OnS-GC, however, since it does not account for changes in phases of

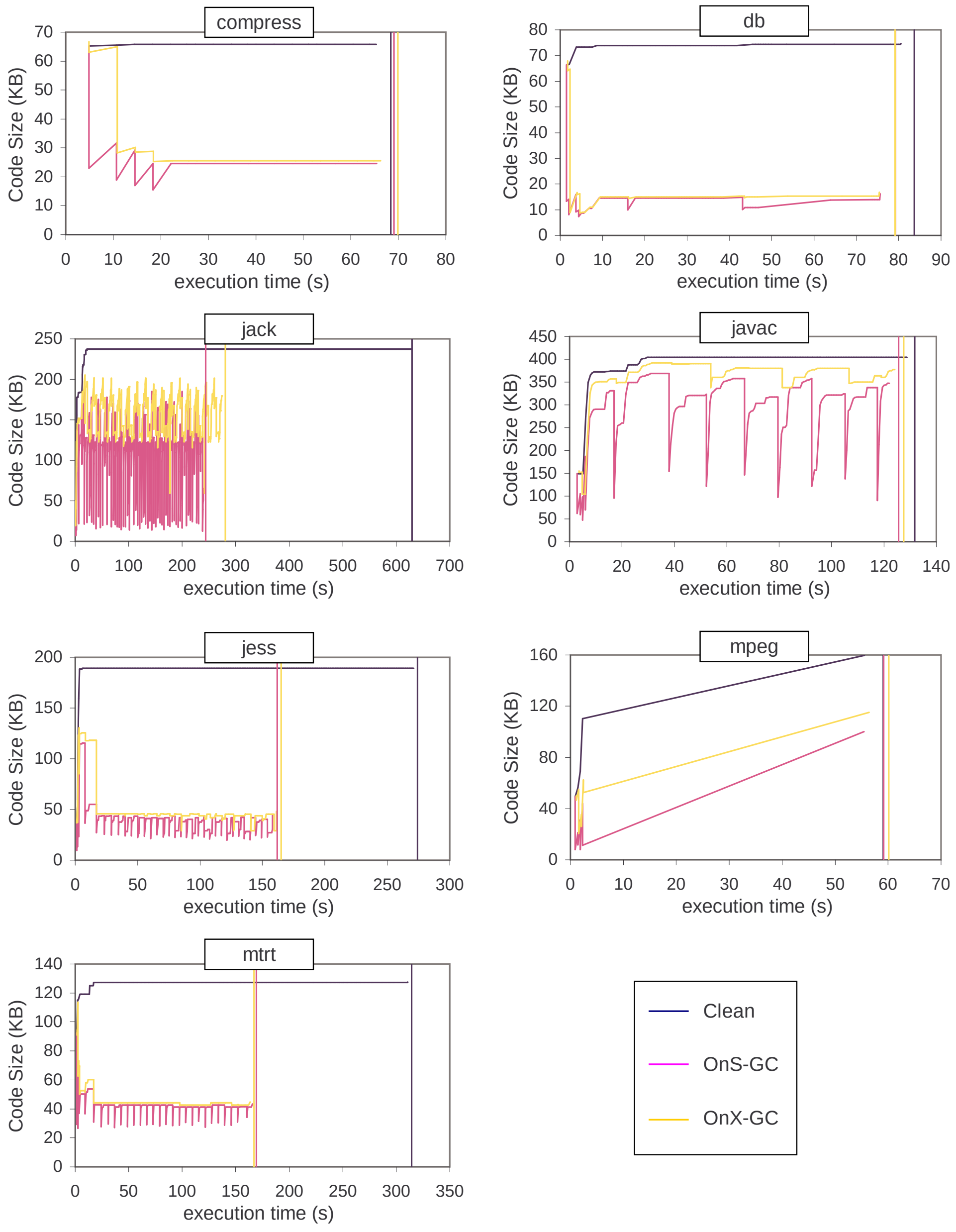


Figure 4.2: Size of code residing in the system during program execution of Clean, OnX (Online eXhaustive profiling) and OnS (Online Sample-based profiling) strategies of Fast Configuration

program execution. In contrast, *Off-MCT* and *OnS-CS* are not sensitive to memory availability at all.

Next, we show how code size changes over time with and without unloading. We only focus on the best-performing combinations of *What* and *When* strategies: online, sample-based profiling triggered by GC invocation count (OnS-GC). Figure 4.2 shows code size over the lifetime of each benchmark using MIN and the Fast compilation configuration. The x-axis is the elapsed execution time in seconds and the y-axis is the native code size in kilobytes. We record code size following each GC and at the end of execution. If unloading occurs during a GC, we record the code size both before and after unloading. We show the results for the clean, OnS-GC, and OnX-GC systems. By comparing OnS-GC with OnX-GC, we can better understand the impact of the more aggressive unloading performed by the OnS strategy. The vertical line for each graph indicates the time at which the program ends.

The graphs in this figure illustrate the impact of code unloading on heap residency. Code size in the clean system becomes stable after a very short startup period and remains at a high level until the application ends. In contrast, both OnX-GC and OnS-GC quickly reduce the code size significantly. OnS-GC is more aggressive than OnX-GC since it unloads any methods that it believes (inaccurately) are not used recently. In addition, both strategies exploit phase behavior by unloading code more aggressively in the early stages of execution, thus, they reduce code size significantly for many applications (such as compress, db, etc.) that have a large amount of dead code following program startup.

The above results show code size reduction enabled by code unloading. Our system requires very little memory for the implementation of code unloading in the form of internal data structures and code. As we discussed in Section 4.2.4, we use a bit array to record profiling information, one bit per compiled method. This implementation requires less than 50 bytes of memory on average. In addition, we add approximately 100 lines of Java code to the code base to perform profiling and code unloading.

4.4.2 Impact on Execution Performance

Code size reduction is not our only concern. If it were, never caching any code would be the best choice. Our ultimate goal is to achieve the best execution performance while maintaining a reasonably small memory footprint.

The performance of our JVM enhanced by adaptive code unloading is influenced by the overhead of recompilation, profiling, and memory management. Memory management overhead refers to the processing cost of stored native code. No matter how native code is stored in a JVM, when memory size is limited, the more of the heap that is allocated for native code, the less that is available for the application, and the more management overhead that is imposed by the stored code. Thus, unloading code when memory is highly constrained can reduce management overhead. The significance of such reduction, however, depends upon how native code is managed in a JVM.

In general, there are three ways to manage native code in a JVM: (1) using a dedicated memory area not managed by the garbage collector (GC); (2) using a GC-managed heap area separated from application heap; and, (3) using a GC-managed heap area shared by application. Storing native code in a GC-collectible memory area eases the memory management because garbage collector can manage the code memory and application memory uniformly. However, this introduces the extra GC overhead. Storing code in a dedicated memory area removes the interference between the native code and the applications. However, it also introduces extra overhead for maintaining multiple heaps and preventing code memory from being used by applications. It is an open question as to which of the three approaches is the best.

In this work, we evaluated the third option, i.e., storing code in the same GC-managed heap shared by the application. In subsections that follow, we compare the performance impact of different "what" and "when" strategies. We then evaluate the different ways in which we can handle optimized code and gather profiling information. Finally, we summarize two best strategies and show how our strategies adapt to available heap size.

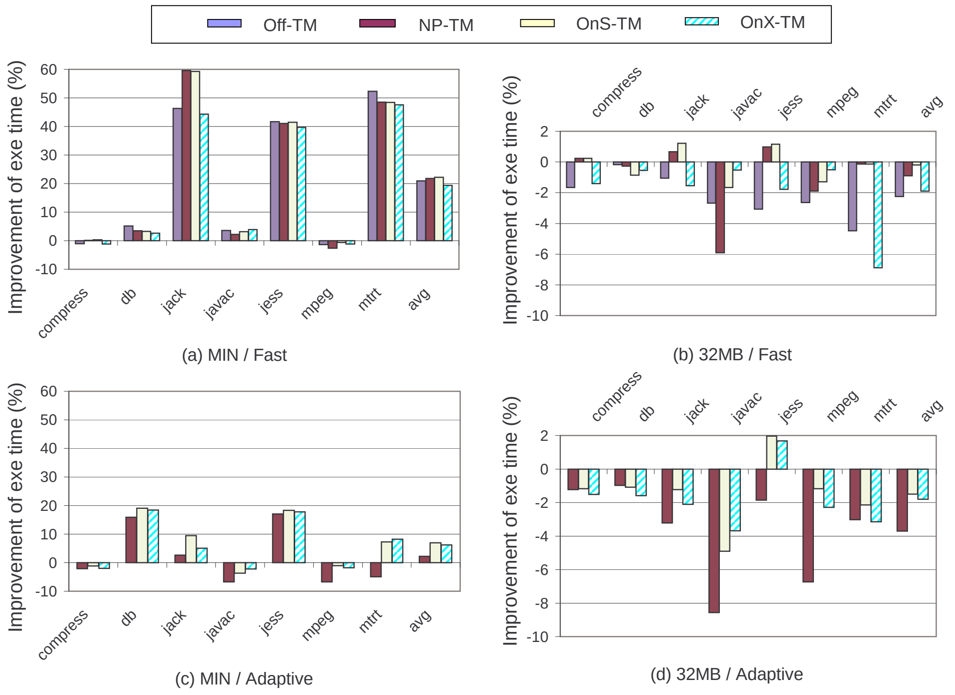


Figure 4.3: Comparison of performance impact of the four "what" code unloading strategies. Each graph shows results for one of the four memory and compilation combinations. MIN/32MB indicates the memory configuration used, and Fast/Adaptive indicates the compilation configuration. Strategies investigated are Off (Offline profiling), NP (No Profiling), OnS (Online Sample-based profiling), and OnX (Online eXhaustive profiling), with same "when" strategy, i.e., TM (TiMer triggered).

Comparison of *What* Strategies

Figure 4.3 shows the performance impact of the various strategies that decide *What* methods to unload: offline profiling (*Off*), no profiling (*NP*), online sample-base profiling (*OnS*), and online exhaustive profiling (*OnX*). For each strategy, we use the timer-triggered (TM) *When* strategy (with a 10 s period).

The y-axis in all graphs shows the percent improvement (or degradation) over the clean system. Graphs (a),(b),(c), and (d) show four different combinations of memory and compilation configurations: (a) and (b) show performance results when no optimizations are performed (Fast); (c) and (d) are results with adaptive compilation; (a) and (c) show the results when memory highly constrained (MIN); and (b) and (d) show the results when memory is unconstrained (32MB).

In general, when memory is critical, unloading some code bodies significantly relieves memory pressure. As stated above, compiled code is stored in a heap that is shared by the applications and is managed by the garbage collection system, for these results. The results indicate that, for such systems, reducing the amount of native code in the system significantly improves performance when memory is highly constrained since less time is spent in GC.

With the fast compiler (Figure 4.3(a)), Jack, jess, and mtrt show execution time reductions of over 40%. This is due to the continuous memory allocation requirements (and hence, GC activity) for these applications. Under high memory pressure, majority of the execution time is spent on thrashing between allocation and garbage collection. Therefore, a small amount of memory freed up by code unloading results in significant performance improvement. Compress and mpegaudio show performance degradation. This is because both benchmarks have relatively small memory requirements (18 and 3 GC cycles, respectively). As such, the benefit from unloading turns out to be less than the overhead introduced by unloading. The impact of code unloading also depends on unloading opportunities provided by the application, such as the percent of short-lived methods. For example, approximately 80% of javac's methods have long lifetimes (see Figure 3.1), which substantially limits the potential of unloading.

Using a fast-compilation JVM configuration, the re-compilation caused by the unloading imposes little overhead since compilation is very fast: the average total compilation time (in seconds) when memory is constrained is 0.1 (374 methods) for clean and Off-TM, 0.5 (3822 methods) for NP-TM, 0.4 (3330 methods) for OnS-TM, and 0.2 (1062 methods) for OnX-TM. Thus, aggressive unloading policies commonly perform well. One example of this is NP, the second-best-performing strategy overall. It performs well on average since it imposes no overhead for profiling and the recompilation cost is small.

Off-TM, the offline profile-base strategy, performs best in one benchmark (mtrt) since it is able to only unload dead methods with no recompilation overhead. However, it is unable to capture infrequently used methods, which turns out to be an important opportunity for unloading when memory is highly constrained. As such, Off-TM does not do as well as the other strategies for most benchmarks. Similarly, OnX-TM does not perform as well as OnS-TM since it is less aggressive than OnS-TM. The average performance improvement, when memory is highly constrained and the fast compiler is used, for each of the strategies, is 20.9% for Off-TM, 21.8% for NP-TM, 22.2% for OnS-TM, and 19.4% for OnX-TM.

With adaptive compilation, the performance improvements gained by code unloading is not as significant as for the Fast configuration. This is because the minimal heap sizes we used for the MIN configuration are the minimal PEAK memory requirements that programs need to run. Due to optimizations, the Adaptive configuration requires larger minimal heap sizes (see Table 4.2). However, optimizations do not happen all the time, and as such, a larger minimal heap sizes actually reduces memory pressure, which reduces GC overheads and improvements enabled by code unloading.

The tradeoff between overheads is slightly different with adaptive compilation (Figure 4.3(c)). Now recompilation overheads are much larger (since optimization is used). Blindly discarding all compiled code, as is done in NP-TM, does not enable the performance levels of the other strategies that use profile information. However, OnS-TM is still better than OnX-TM, which indicates that the profile overhead saved by sampling and the GC overhead reduction enabled by more aggressive unloading of OnS is still more important than the recompilation overhead introduced by inaccurate sampling information when memory is highly constrained. The average performance improvement in this case is 2.2% for NP-TM, 6.9% for OnS-TM, and 6.2% for OnX-TM.

The overhead of unloading (for profiling and recompilation) is more apparent when memory is unconstrained since unloading is unnecessary and thus, pure overhead. The average performance improvement (negative for degradation), when memory is unconstrained and the fast compiler is used (Figure 4.3 (b)), is -2.3% for Off-TM, -0.9% for NP-TM, -0.2% for OnS-TM, and -1.9% for OnX-TM. Off-TM and OnX-TM both perform poorly when memory is unconstrained since both impose overhead for exhaustive method profiling. For the adaptive configuration (Figure 4.3 (d)), the average performance improvement is -3.7% for NP-TM, -1.5% for OnS-TM, and -1.8% for OnX-TM. We can see that the negative impact of NP strategy is more apparent in this configuration due to higher recompilation overhead and less memory pressure.

In summary, the results indicate that a less aggressive and inexact unloading policy with low online measurement overhead (OnS) enables significant performance improvements when memory is critical. In addition, such a strategy imposes little or no overhead when there is ample memory available. That is, OnS provides an adequate estimate of infrequently used methods so that GC overhead can be reduced.

Comparison of *When* Strategies

We next consider the impact of the various strategies that determine *when* unloading should be performed. For all of these results, we use the best-performing "what"

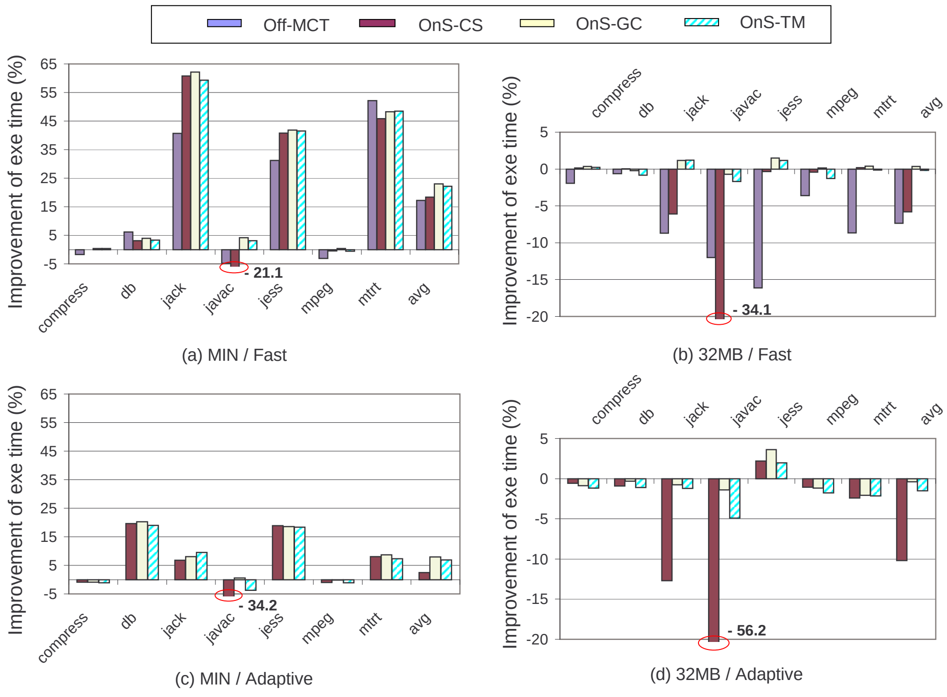


Figure 4.4: Comparison of performance impact of the four "when" code unloading strategies. Each graph shows results for one of the four memory and compilation combinations. MIN/32MB indicates the memory configuration used, and Fast/Adaptive indicates the compilation configuration. Strategies investigated are MCT (Maximum Call Times triggered), CS (code Cache Size triggered), GC (Garbage Collection triggered), and TM (TiMer triggered), with the same "what" strategy, i.e., OnS (Online Sample-based profiling), except MCT, with requires offline profiling (Off) to produce an exact count of maximum invocations.

strategy, OnS (for all strategies except Off-MCT – which requires offline profiling to produce an exact count of maximum invocations). The four *when* strategies that we investigated are MCT (trigger: max invocation count using perfect-profile information), CS (trigger: size of cached code), GC (trigger: GC count), and TM (trigger: timer alarm).

Figure 4.4 shows the performance results due to the various “when” unloading strategies. The format of the figure is the same as those presented previously. The y-axis in both graphs is the percent improvement (or degradation) over the clean system. With the fast compiler, the average performance improvement achieved by our “when” strategies is 17.2% for Off-MCT, 18.4% for OnS-CS, 23.0% for OnS-GC, and 22.2% for OnS-TM when memory is highly constrained. When memory is not critical (32MB), recompilation and profiling overhead introduced by code unload outweighs the GC benefits gained. The average improvement is -7.4% for Off-MCT, -5.8% for OnS-CS, 0.4% for OnS-GC, and -0.2% for OnS-TM.

In general, the best-performing strategy is GC which uses the frequency of garbage collections to trigger unloading. MCT imposes large profiling overhead. It also requires an accurate, input-specific, offline profile, which may not be realistic for mobile programs. CS works well when method working set size is similar to the code cache size. However, it is impossible to accurately predict code cache size. An incorrect prediction may cause significant performance degradation since it results in unnecessary unloading sessions, thus, introducing recompilation overhead. For example, the code cache size of javac in Fast configuration grows to 360 KB at the end of execution; this is much larger than the initial code cache size (64KB).

The performance impact for the Adaptive configuration is similar, except that: first, the GC benefits due to code unloading is smaller because of the larger MIN sizes as discussed previously; and second, the recompilation penalty of aggressive unloading is larger because of expensive optimizations. In summary, the average performance improvement is 2.5% for OnS-CS, 7.9% for OnS-GC, and 6.9% for OnS-TM when memory is highly constrained. When memory is not critical, the improvement (degradation if negative) is -10.2% , -0.4% , and -1.5% respectively.

Handling Optimized Code

To improve the performance of code unloading for the adaptive compiler configuration, we investigated three additional variants of OnS for unloading optimized code. They include delaying unloading of optimized code for an additional unloading session (*OnS-DO*), excluding optimized code when unloading (*OnS-EO*) and unloading optimized code and re-optimizing it at the same level if re-invoked (*OnS-RO*). The default OnS uses fast compilation when the unloaded method is re-invoked. All these strategies use the best “when” strategy, GC (garbage collection triggered). Figure 4.5 shows the results with both MIN (a) and 32MB (b) configurations.

The data in the figure shows that OnS-DO-GC (delay unloading) works best for most of benchmarks. The reason for this is that it gives the optimized code an extra chance to stay in the system. It also exploits the opportunities to unload outdated optimized code, unlike OnS-EO-GC. OnS-RO-GC saves the learning time to progressively re-compile a hot method when it is re-invoked. However, the result shows that

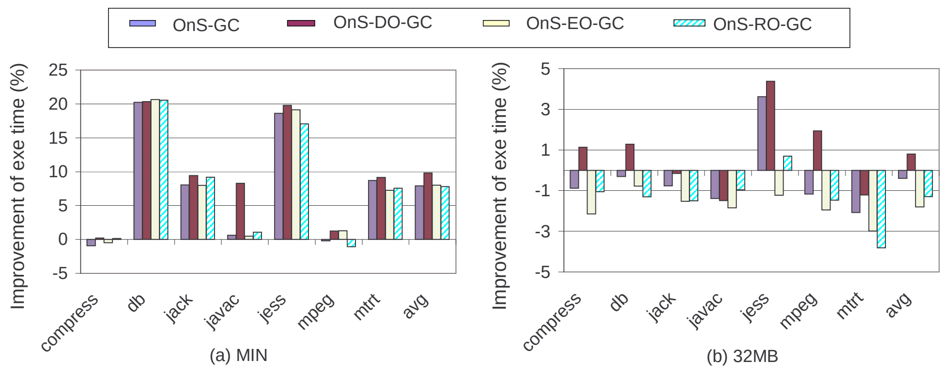


Figure 4.5: Comparison of four variants of online sample-based profile for Adaptive configuration. MIN/32MB indicates the memory configuration used. The strategies investigated are OnS-GC (Online Sample-based profiling, Garbage Collection triggered) with different ways to handle optimized code: OnS-GC treats optimized methods as same as other methods and recompiles them with the fast compiler; OnS-DO-GC gives the optimized methods a second chance to stay in the system, but recompiles them with the fast compiler if they do get unloaded and reinvoked later; OnS-EO-GC excludes optimized methods from unloading at all; OnS-RO-GC unloads a optimized method normally, but recompiles it at the optimizing level that it was compiled before unloaded.

in most cases, it does not work well because many of hot methods are no longer hot following unloading/reloading. For example, method *_201_compress.Input_Buffer.getbyte()* is hot (invoked more than 1.5 million times) before it is unloaded the first time. Due to the phase shift of the program, it is less hot during subsequent execution. However, the method is still invoked periodically (approximately 10 invocations between the two unloading sessions). Since its hotness is not enough to be recognized by the sampling profiler, it is unloaded during every unloading session and optimized upon returning. This introduces unnecessary yet significant overhead.

In summary, the average performance improvement (or degradation if negative), is 7.9% for OnS-GC, 9.8% for OnS-DO-GC, 8.0% for OnS-EO-GC, and 7.8% for OnS-RO-GC when memory is highly constrained (MIN). When resources are unconstrained (32MB), it is -0.4% for OnS-GC, 0.8% for OnS-DO-GC, -1.8% for OnS-EO-GC, and -1.3% for OnS-RO-GC.

Comparison With LRU

As we mentioned in Section 4.2.4, we used a bit array to record profile information with low overhead. One limitation of this implementation is that it does not capture temporal order of method invocations. One possible implementation alternative that can record temporal information is a LRU list: all methods are linked together in the order of their last invocations; whenever a method is invoked or sampled, it is moved to the end of the list. The overhead of maintaining this LRU list is much higher than that of a bit array since it requires several linked-list operations for each update. Moreover, the LRU list requires memory space for two reference fields per method. In this section, we investigate whether more accurate temporal order of method invocations will enable more efficient code unloading in spite of the memory overhead introduced.

We selected our best "what" and "when" strategy combinations so far (OnS-GC for the Fast configuration and OnS-DO-GC for the Adaptive configuration) and reimplemented their mechanism of recording profile information using a LRU linked list. Note that for the bit array, we unload all unmarked methods and reset all bit to 0 when unloading is triggered. Thus, the aggressiveness of unloading is controlled by the “when” strategy and no parameter is needed during code unloading. While with a LRU list, the profile information is not discrete (0 or 1), and hence, we need an extra parameter to decide how much we should unload from the LRU list when unloading is triggered. For now, we added a new command-line option, called *unloadFraction*, to control the portion in terms of code size of the LRU list that will be unloaded during each unloading session. We chose a fraction parameter instead of a absolute code size parameter to adapt to different workloads.

Similar to the scenarios of other parameterized strategies, there is no generally best value for this *unloadFraction* parameter across programs. We empirically evaluated a wide range of values for this parameter, and report results using best-performing parameter values (on average) across the benchmarks studied. They are: 40% for the MIN/Fast configuration, and 10% for the other configurations. Figure 4.6 shows our performance results: OnS-GC (OnS-DO-GC) labels the bit array implementation and SLRU-GC (SLRU-DO-GC) labels the LRU implementation. The memory and compilation configurations are: (a) MIN/Fast; (b) 32MB/Fast; (c) MIN/Adaptive; and (d) 32MB/Adaptive. This figure shows that with a fast compiler, more accurate temporal information provided by the LRU implementation does not enable enough benefits

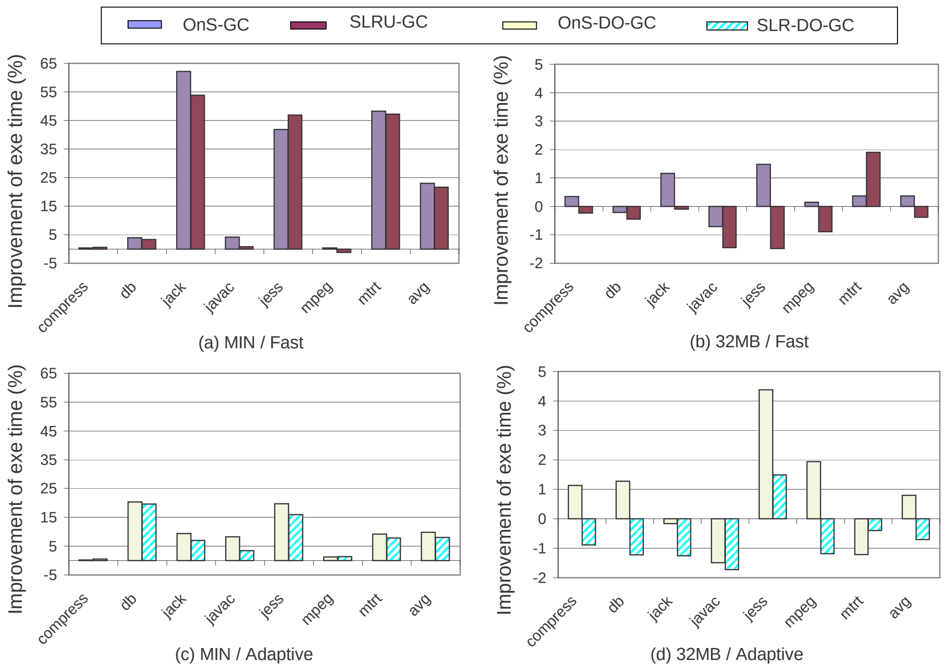


Figure 4.6: Comparison of performance impact of the two ways to record profile information: bit array (OnS-GC/OnS-DO-GC) and LRU list (SLRU-GC/SLRU-DO-GC). Each graph shows results for one of the four memory and compilation combinations. MIN/32MB indicates the memory configuration used, and Fast/Adaptive indicates the compilation configuration.

to amortize the additional overhead introduced. With an adaptive compilation configuration, the bit array still performs better than the LRU list in most cases. *mtrt* is an exceptional case, in which accurate temporal order of method invocations does enable more efficient code unloading. One possible reason is that *mtrt* is the only multi-threaded application in the benchmark suite. The interaction between threads makes its performance more sensitive to temporal order of method invocations. In such cases, maintaining a LRU list can improve performance since it enables more accurate code unloading. The average performance improvements across benchmarks are: 23.6% for OnS-GC and 21.6% for SLRU-GC in the MIN/Fast setting; 0.6% for OnS-GC and -0.3% for SLRU-GC in the 32MB/Fast setting; 9.8% for OnS-DO-GC and 8.0% for SLRU-DO-GC in the MIN/Adaptive setting; 0.8% for OnS-DO-GC and -0.7% for SLRU-DO-GC in the 32MB/Adaptive setting.

In summary, we conclude that for the cases we studied, maintaining a LRU list does not enable significantly more efficient code unloading; the bit array implementation is a better choice since it imposes lower overhead and effectively trades off the costs and the benefits of unloading.

Adaptation to heap sizes

Next, we summarize the improvements on code size and overall performance for a range of heap sizes. These results indicate the adaptability of our strategies. We present results of the best-performing combination of *What* and *When* strategies: online, sample-based profiling using GC invocation count triggered unloading (OnS-GC) for the Fast configuration, and OnS-DO-GC for the adaptive JVM configuration (Figures 4.7).

In all of the graphs in Figure 4.7, the x-axis is heap size. The y-axis in the left graphs is the average code size normalized to the clean version and the y-axis in right graphs is the execution time normalized to the clean version. Graph (a) displays

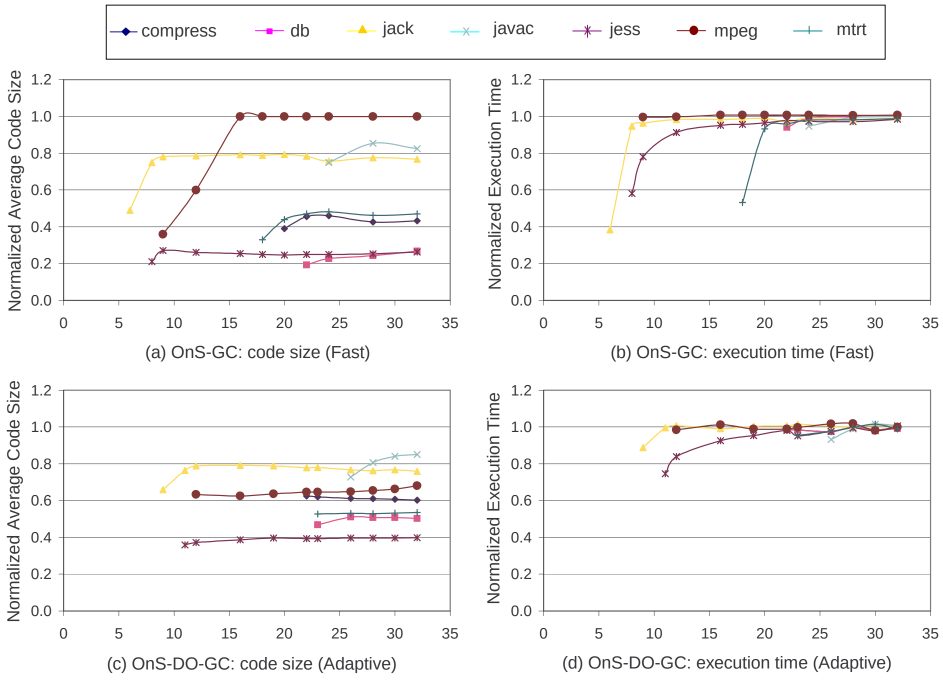


Figure 4.7: Summary of code side reduction and performance improvements enabled by our best strategies (OnS-GC for fast compilation and OnS-DO-GC for adaptive compilation) across different heap sizes.

the impact of OnS-GC on code size when the heap size grows from the minimum to 32MB. We can see that when memory is limited, OnS-GC unloads more aggressively, resulting in 61% code size reduction on average. When memory availability grows, the aggressiveness decreases quickly since fewer garbage collections are invoked. On the other hand, the startup strategy guarantees that even when memory is not critical, e.g., 32MB, the dead code in the startup phase will be unloaded. The code size reduction with a 32MB heap is 43% on average.

The reduction in code size enables execution time benefits while imposing very little overhead. Graph (b) shows the normalized execution time of OnS-GC for different heap sizes. We can see that with constrained memory space, unloading can not only reduce the size of cached code, but also improve execution speed by trading off GC time for compilation overhead. When memory size grows, the improvement decreases quickly since there is not much GC overhead to reduce. These results show that our framework and the OnS-GC strategy are able to adapt to dynamic memory availability using the Fast compiler configuration.

Similarly, Graph (c) summarizes the effect of OnS-DO-GC on code size and (d) shows its impact on execution time, with the Adaptive configuration. Since the adaptive configuration requires a larger minimal heap size than that of Fast configuration, the curves in (c) and (d) start from a larger initial heap size. On average, the code size reduction for OnS-DO-GC is 43% with minimal heap size and 38% with 32MB. The performance improvement is 10.3% and 0.1%, for MIN and 32MB, respectively.

4.4.3 Code Unloading for Selective Compilation Systems

The results in the previous section show that adaptive code unloading is able to monitor the system with low overhead, to make intelligent decisions about what methods to unload and when to trigger unloading, and to reduce code size dramatically without sacrificing performance. Moreover, if system memory is highly constrained, unloading code can enable significant performance improvement when code is stored with application on the garbage-collected heap since much less time is spent performing memory management.

The experimental methodology that we consider is a compile-only Java Virtual Machine, JikesRVM. An alternative to a compile-only system for embedded devices is one that employs *selective compilation*: methods are initially interpreted then compiled using increasingly higher levels of optimization as they become "hot". Even though interpretation of methods that are invoked multiple times has been shown to waste significant resources on embedded devices [149, 48], selective compilation JVMs produce less compiled code since they interpret many methods. In addition, for methods that are executed for a very small portion of total program execution time, the overhead required to compile them may not be amortized; selective compilation systems can interpret these methods.

In this section, we consider the impact of code unloading on selective compilation JVMs for resource-constrained systems. Code unloading has the potential to impact the performance of these systems in two ways. First, unloading will reduce the amount of native code stored in the system and possibly reduce memory management overhead by evicting a subset of compiled methods. Second, using code unloading, selective compilation systems can be more aggressive about compilation and optimization decisions. That is, since code unloading reduces the effective memory requirements for stored native code, more interpreted methods can be compiled – which has the potential for improving performance for methods that are invoked repeatedly.

Since our research platform, JikesRVM, is a compile-only system and there are no open-source, selective compilation, systems available (that implement an interpreter, a highly optimizing compiler, and an adaptive optimization system), we investigated the impact of code unloading for selective compilation JVMs using simulation. As we did previously, we consider the CISC, IA-32 architecture. Code unloading for a RISC system, e.g., one that uses the StrongARM processor, will produce even better results in terms of the amount of code unloaded since, as we articulated earlier, RISC native code is 16-25 times larger than x86 equivalent. As such, by considering x86, our results indicate a lower bound on the potential of code unloading for RISC systems.

Selective compilation systems decide which methods to compile and when to compile them using a number of system metrics in much the same way as the compile-only system. Such systems must consider the cost of applying compilation and optimization and the performance that will result if compilation is applied (or not applied). The latter metric requires an estimate of how long the method under consideration will execute in the future. If a method is hot and compiled too late, the compilation overhead may not be amortized and the resulting performance improvement may not impact overall program performance. A hot method thus, should be compiled (i.e., the selective compilation system must identify the method) as early as possible.

A selective compilation system can use a performance model to decide when to compile a method. For this discussion and our evaluation we assume that the model estimates total application execution time as the sum of execution times of every method invocation. We also assume that the speedup of compiled code over interpretation is a constant factor for all methods given the same compilation level. Given these assumptions, we can model the execution time of an application as:

$$T_{exe} = \sum_{i=1}^{n} \begin{cases} T_{jit_i} + T_{intrp_i} * I_i + \frac{T_{intrp_i}}{speedup} * (N_i - I_i) & \text{if } I_i < N_i \\ \\ T_{intrp_i} * N_i & \text{otherwise} \end{cases} \tag{4.1}$$

where $i = 1, ..., n$ denotes an invoked method, assuming there are n methods invoked. T_{exe} denotes the estimated overall execution time, T_{jit_i} denotes the time spent compiling the ith method, T_{intrp_i} denotes the execution time of one invocation of a method if it is interpreted, $speedup$ is the performance improvement achieved by executing stored compiled code, I_i denotes the invocation count of a method before it is compiled, and N_i denotes the total invocation count of a method. If method i is compiled at some point, the first case of the formula is used, otherwise, the second case is used.

Ideally, if the performance gain that results from compiling a method exceeds the compilation cost, the system should compile the method upon initial invocation of the method to enable optimal improvement, i.e., I_i should be zero for such a method. However, it takes time for a real system to identify (i.e. *learn about*) profitable methods, and thus, I_i will be greater than zero and is equivalent to N_i for those methods for which compilation overhead cannot be amortized.

The actual value of I_i depends on the mechanism that a JVM uses to define a "hot" method. One way a JVM can identify hot methods is by using counters. When the invocation count of a method exceeds a pre-defined threshold, the method is compiled. With this technique, I_i is same for all methods and can be replaced by the threshold in the model. Note that most systems also count back edges to catch methods with long loops; we ignore backedges in this portion of the study to simplify our analysis. In summary, model (4.1) indicates that the execution time of an application varies given different levels of JIT efficiency, the quality of the compiled code, and the mechanism the JVM uses to identify hot methods.

To understand the dynamics of selective compilation and code unloading, we conducted several experiments. First, we employed the profiling facilities of the Kaffe virtual machine [89] to gather execution time and compilation time for each method of the SpecJVM98 benchmark suite. Although Kaffe is not the only JVM with such profiling ability, we chose Kaffe since it is an open source project and we can easily extend the profiler to gather more information, e.g. bytecode size, compiled code size, etc. Moreover, Kaffe implements an interpreter and a JIT (but does not implement selective compilation, i.e., mixed-mode execution).

The average speedup that results from using JIT compilation over interpretation in Kaffe is over 20 times. This is because the interpreter is not well-tuned in any way (and not because the JIT applies aggressive optimization – only very simple optimizations are implemented in Kaffe). In a product JVM with selective compilation and a highly tuned interpreter, e.g., HotSpot, the difference between interpreted and JIT execution is much smaller, e.g., 3$\sim$15 times. Since HotSpot is not an open source system, we estimate the speedup enabled by selective compilation over interpretation using the speedups and compilation rates (bytecode in bytes per millisecond) that the JikesRVM compilers enable.

We obtain the compilation rates and speedups enabled by the JikesRVM compilers (and used by JikesRVM to make adaptive optimization decisions) by computing the geometric mean of each across a large set of applications. We assume that compilation with the minimal amount of optimization enables a speedup of 2 times over interpretation of a method; this value has been shown to be a reasonable and conservative estimate in other studies [1, 92, 141, 159]. We then use the JikesRVM compilation rates and speedups for higher levels of optimization (used when methods remain hot for a long period).

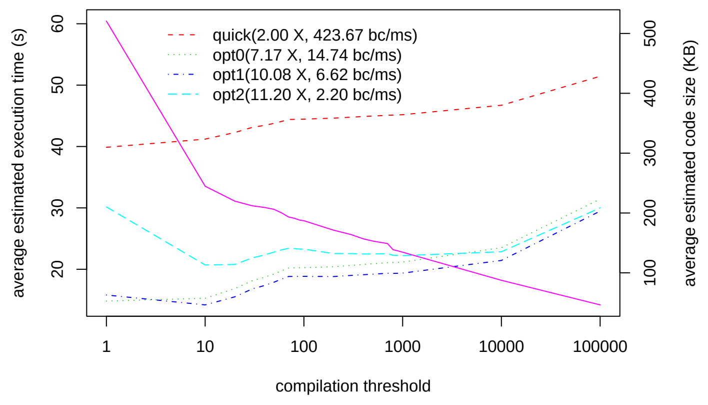


Figure 4.8: Average execution time and code size estimation for SpecJVM98 benchmarks using three parameters: compilation threshold, JIT overhead, and speedup of compiled code over interpreted code. The x-axis is the compilation threshold in log scale. The left y-axis denotes the estimated execution time in seconds, and the right y-axis is the estimated code size in kilobytes. The four dashed lines denote the estimated execution time at the four compilation levels, and the solid line represents the changes in code size for different thresholds.

We simulate the execution time and the size of compiled code using different "hot" thresholds (method invocation counts). In Figure 4.8 we show the average impact of this compilation threshold (x-axis using a log scale) across all of our benchmarks. The left y-axis denotes the estimated execution time in seconds, and the right y-axis is the estimated code size in kilobytes. The four dashed upward lines denote the estimated execution time at the four compilation levels. The solid downward line represents the changes in code size for different thresholds. We provide the estimated speedup and compilation rate (byte code in bytes per millisecond (bcb/ms)) of each compilation level in the legend. Since Kaffe does not have multiple compilation levels, we cannot accurately estimate the change in code size for each compilation level.

We measured the average size of native code produced by JikesRVM using different compilation levels. The size ratios of *opt0*, *opt1*, and *opt2* comparing to the quick compiler are around 0.64, 1.00, 1.11. Level *opt1* and *opt2* produce larger code than level *opt0* due to more aggressive inlining. We used these ratios to estimate roughly code size changes for different thresholds. Since all of the code size estimation lines are parallel, we only show the line for the quick compiler for clarity.

The figure indicates that a threshold of 10 achieves the best balance between code size and performance across all compilation overhead/speedup configurations. Code size drops dramatically when threshold moves from 0 to 10, which indicates that many methods are invoked fewer than 10 times. In addition, the performance improvements gains that result from compilation are negligible or negative if we compile these methods since the compilation overhead is not amortized. Once the threshold exceeds 10, the rate of decrease in code size slows while the performance gain becomes more apparent.

A smaller threshold results in more compiled code being stored by the system. At threshold 10, the size of generated code is about half of that of a compiler-only JVM and yet is still substantial (250KB) for embedded devices and can result in significant memory management overhead. If the memory is highly constrained, the JVM may not be able to store these code blocks to achieve the optimal performance. Our adaptive code unloading system can help in this situation. By monitoring the exe-

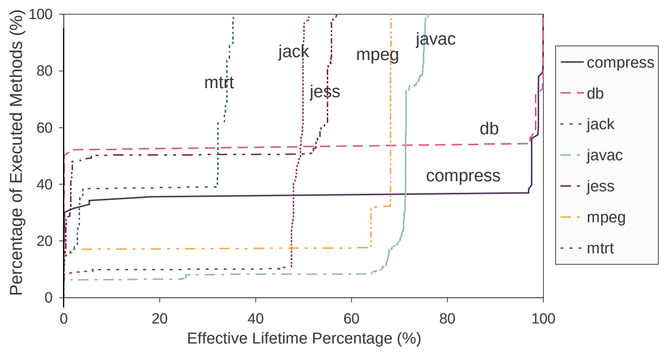


Figure 4.9: CDF of effective method lifetime as a percentage of total lifetime as shown in Figure 3.1. However, we only consider hot methods here. On average, 30% of these methods have effective lifetime percentage of less than 5%. The effective lifetime percentage for most of the other 70% of the "hot" methods is less than 60%.

cution behavior of the applications and resource availability with low overhead, the adaptive code unloading system enables the JVM to make better use of the precious memory by evicting less useful code blocks so that more aggressive compilation can be performed to carry out performance that is closer to optimal.

Another interesting question that we investigated is: How many methods continue to be hot after they are compiled and how long is their hot period? To investigate this question, we considered the effective lifetimes of methods (as we did previously in Section 3.2 in Figure 3.1). In Figure 4.9, we again plot effective lifetimes but omit those methods identified as cold (invoked fewer than 10 times) in the previous data set. The data shows that for hot methods, on average 30% of them have effective lifetime percentage of less than 5%. That is, the time between the first and last invocations of a method is less than 5% of the total time these methods are in the system.

Moreover, the effective lifetime percentage of most of the other 70% of all methods executed is less than 60%.

This data indicates that most methods compiled in a selective compilation system become useless very soon after they are compiled. Our adaptive code unloading system can remove these methods to avoid this memory waste, to reduce memory management overhead, and to enable aggressive compilation decisions in selective compilation JVMs as well as compile-only JVMs.

4.5 Related Work

This body of research is related to two primary areas of prior work: code management systems and code size reduction techniques.

Several code cache management techniques have been proposed in prior work. One such technique is *code pitching* which is used in Microsoft .NET Compact Framework [140]. The virtual machine for this framework uses a JIT compiler to translate intermediate code (CIL) into native code without optimization. When the total size of the code area exceeds a specified maximum, the system “pitches” (discards) the entire contents of the buffer [136, 140]. The VM expands the code cache when a newly compiled method cannot be accommodated even after code pitching or if the overhead of pitching is greater than 5% of the total execution time. Users can specify the initial code cache size and the upper bound of growing. The default value is 64MB for the initial size and the maximum integer value for the upper bound. The minimum initial size allowed specified is 64KB. Code pitching is easy to implement and imposes no profiling overhead. However, it needlessly unloads code when resources are not constrained. In addition, it discards all code (even hot methods) requiring recompilation of all methods that are invoked in the future.

Code cache management has also been used in binary translation systems. The Dynamo project [13] and its successor DELI [40] from HP, extract and optimize hot instruction traces from an executing program being translated. These systems store hot traces, called "fragments", in a fragment cache to be reused. When the cache fills, the systems "flush" the cache, discarding all fragments. Dynamo also performs a flush when it detects a dramatic increase in fragments over a short time. These systems employ this simple flushing strategy since many fragments are linked together in the fragment cache and selective unloading can introduce significant unlinking overhead. Our target is the Java virtual machine for which cached code is commonly method-based and unlinked. As such, selectively unloading code using lightweight profiling techniques like sample-based profiling are able to achieve good performance without unlinking overhead.

DynamoRIO [19] is another Dynamo extension that performs dynamic binary optimization. DynamoRIO uses an unbounded code cache by default. However, users can specify a size limit for the code cache. To manage a bounded code cache, DynamoRIO employs a circular buffer similar to that described in [68]. The granularity of such a circular buffer mechanism (FIFO) is investigated in [67]. Their results show that a medium-grained eviction policy results in better performance than both coarse and fine granularities.

The DAISY software emulation system from IBM [44] also employs code cache management. DAISY uses “tree-groups” to represent translated instructions, where control flow joins are disallowed. This causes a code space expansion problem due to tail duplication. The authors overview a simple, low-overhead, generational garbage collection technique to manage a large translation cache (100MB or more). However, we did not find any implementation details on this approach and thus were not able to compare it to our framework. [69] investigates a similar mechanism using DynamoRIO [19] and a generational cache simulator. Our strategies described in Section 4.2.3 handle the optimized code separately and can be considered as a simplified form of generational cache management.

The purpose of our work is to provide an flexible framework to empirically investigate the efficacy of different unloading strategies and implementation designs, and to help the JVM designers choose the best strategies. Both strategies used in the .NET compact framework and in Dynamo can be configured as *NP-CS* in our framework. NP-CS uses code cache size as the unloading trigger and throws away anything without any profile information when unloading is performed. Our results indicate however, that doing so does not work as well as using a sample-based, GC-triggered configuration.

Code size reduction for restricted resource environments is another research area that is related to our work. Sun's HotSpot technology [75, 35] limits the size of compiled code by only compiling the hottest methods and interpreting all other methods. Other work uses *profile-driven deferred* compilation or optimization [18, 155] to avoid generating code for cold spots in the programs. In contrast to their "never cache cold methods" strategy, which may impose large re-interpretation overheads, our framework enables a more flexible code caching strategy which can adapt to system resource status: whether and how long a method's code is cached is dynamically determined by the code unloader according to runtime information and system memory status. Moreover, our code unloading techniques can also be used to manage "hot" methods in these "never cache cold methods" systems,

Another mechanism for code size reduction that have been pursued by other researchers is compression. Compression is a compact encoding of data to reduce storage and transfer requirements. A number of different techniques for compressing compiled code are described in [47, 107, 39, 42]. These techniques, like those for deferred compilation, are complementary to our approach and can be used in combination with our code unloading framework to further reduce the memory overhead of compiled code.

4.6 Summary

In this chapter, we propose a framework for dynamic and adaptive unloading of compiled code so that more aggressive dynamic compilation can be performed. Our goal with this system is not only to reduce the size of compiled code. If it were, interpretation would be the better choice. Instead, our goal is to enable performance improvement via dynamic compilation *while* reducing the dynamic memory requirements of the JVM. That is, we seek to adaptively balance not storing any code (as in an interpreter-based JVM) and caching all generated code (as in a compile-only JVM), according to *dynamic memory availability*, i.e., the amount of memory available to the executing application for allocation of data (as opposed to code) over time.

Our code unloading system decides *what* code to unload and *when* unloading should commence. Each of these decisions can be made using a wide range of unloading strategies, each resulting in different tradeoffs between several sources of overhead and benefit. To study these tradeoffs, we used the framework to investigate a number of unloading strategies which employ dynamic feedback from the program and execution environment to identify unloading candidates and to trigger unloading efficiently and transparently.

We implemented and empirically evaluated our code unloading framework and unloading strategies using a high-performance, open source Java Virtual Machine from IBM T. J. Watson Research Center, the Jikes Research Virtual Machine [5] (JikesRVM). Our results indicate that by adaptively unloading compiled code, we are able to reduce code size by 36%-62% on average over the lifetime of the programs. Since the system is able to adapt to memory availability, it introduces no overhead when resources are unconstrained. When memory is highly constrained, reductions in code size translate into execution time improvements of 23% on average for the programs and JVM configurations that we studied.

Note that our code unloading system achieves the above code size reduction and performance improvement completely automatically, without requiring programmer intervention or participation. Our adaptive code unloading system allows programmers to develop applications without the concern for code size, yet facilitates efficient execution that results from doing so. Without such support, programmers must carefully reduce the code size by hand while developing applications so that they can fit into the constrained resources, which requires expert knowledge and significantly more programmer effort. Therefore, our system improves programmer productivity by providing automatic support of code unloading.

In summary, this work wakes the following contributions:

Opportunity Analysis. It provides an empirical analysis of code unloading opportunities.

Analysis Framework. It presents a novel code unloading framework that automatically unloads native code to reduce the overhead of performing garbage collection. This framework facilitates the implementation and empirical evaluation of unloading strategies.

Adaptive Algorithms. It describes a number of techniques that use dynamically changing program and system memory behavior to decide *what* code to unload and *when* to unload it.

Experimental Results. It presents an empirical comparison of our adaptive unloading techniques. We identify a set of strategies that, when resources are unconstrained, reduces code size by 47% while introducing zero overhead, on average. When memory is highly constrained, our system reduces code size by 62% and execution time by 23% on average for the programs and JVM configurations studied.

The text of chapter 3 and Chapter 4 is in part a reprint of the material as it appears ACM Transactions on Architecture and Code Optimization (TACO), Vol. 2, Number 2. The dissertation author was the primary researcher and author and the co-author listed on this publication ([160]) directed and supervised the research which forms the basis for these two chapters.

Part II

Easy and Efficient Parallel Programming Using Futures in Java

Chapter 5

Futures and its Support in Java

The second part of this dissertation, focuses on how to use the same adaptive framework as we did for code unloading for resource constrained devices, for high-end systems. In particular, we investigate doing so to facilitate easy efficiency of the *future* parallel programming construct in Java. In this chapter, we describe the future construct, its design rationale, its programming model, and the history of its use. We then overview the existing support of futures in Java. We detail the library support for this approach with simple examples, and then discuss the advantages and limitations of this approach.

5.1 The Future Construct

A future is a simple parallel programming language construct that lets programmers to specify computations that can be potentially executed in parallel. This construct was first proposed by Baker and Hewitt in the 70s [70], and made well-known by Halstead in MultiLisp [126]. Several years later, the same idea was reinvented as another parallel language construct, called Promise [103] by Liskov and Shrira. From then on, futures have appeared in many languages, such as Mul-T [93], Concurrent ML [125], C++ [28, 151], and more recently, Java 5.0 concurrent package [86], X10 [27], and Fortress [4].

By definition, a future is a value available in future. It is a placeholder of the value evaluated by an asynchronous computation. The future is immediately returned to the calling site as if the computation had finished and the value had been returned. The calling function continues execution until it accesses the future value, at which point, it is implicitly blocked until the value is made ready by the asynchronous computation.

The future construct is a simple and elegant way to introduce concurrency to serial program since it enables the decoupling of the parallel scheduling from the application logic. In addition, in this model, the synchronization is implicit and is delayed to the latest possible point (the future value usage point).

The original rationale behind futures is that "the programmer takes on the burden of identifying *what* can be computed safely in parallel, leaving the decision of exactly *how* the division will take place to the run-time system" [113]. The emphasis on "minimal programmer effort" of futures frees programmers from worrying about whether the overhead of spawning a computation in parallel can be paid off by its benefits, which usually is not an easy decision for programmers to make statically. As a result, programmers might specify a large number of small granularity computation as futures. It is, therefore, vital for performance that the runtime implementation of futures be efficient, and effectively make wise scheduling decisions, to avoid overhead, and to exploit concurrency. One of our goals for the second part of this dissertation is to investigate ways to enable such an efficient future scheduling system for Java by exploiting the adaptability of the Java virtual machine.

5.2 Support for Futures in Java

Version 5.0 of the Java programming language introduces the support of futures via a set of APIs in the `java.util.concurrent` package. The primary APIs include `Callable`, `Future`, and `Executor`. Figure 5.1 shows code snippets of these interfaces.

Using the Java 5.0 Future APIs, programmers encapsulate a potentially parallel computation in a `Callable` object and submit it to an `Executor` for execution. The Executor returns a `Future` object that the current thread can use to query the computed result later via its `get()` method. The current thread immediately executes the code right after the submitted computation (i.e., the continuation) until it invokes the `get()` method of the `Future` object, at which point it blocks un-

```
public interface Callable<T>{
  T call() throws Exception;
}

public interface Future<T>{
  ...
  T get() throws InterruptedException,
           ExecutionException;
}

public interface ExecutorService extends Executor{
  ...
  <T> Future<T> submit(Callable<T> task)
    throws RejectedExecutionException,
           NullPointerException;
}
```

Figure 5.1: The `java.util.concurrent` Futures APIs

```
public class Fib implements Callable<Integer>
{
  ExecutorService executor = ...;
  private int n;

  public Integer call() {
    if (n < 3) return n;
    Future<Integer> f = executor.submit(new Fib(n-1));
    int x = (new Fib(n-2)).call();
    return x + f.get();
  }
  ...
}
```

Figure 5.2: The Fibonacci program using Java 5.0 Futures API

til the submitted computation finishes and the result is ready. The Java 5.0 library provides several implementations of `Executor` with various scheduling strategies. Programmers can also implement their own customized Executors that meet their special scheduling requirements. Figure 5.2 shows a simplified program for computing the Fibonacci number (Fib) using the Java 5.0 Future interfaces.

The Java 5.0 Future programming model is simpler than a thread-based model since it decouples thread scheduling from application logic. However, there are several drawbacks of the current Java 5.0 Future model. First, given that the model is based on interfaces, it is non-trivial to convert serial versions of programs to parallel versions since programmers must reorganize the programs to match the provided interfaces, e.g., wrapping potentially asynchronous computations into objects.

Secondly, the multiple levels of encapsulation of this model results in significant, but unnecessary, memory consumption which can degrade performance significantly due to the extra memory management overhead.

Finally, to achieve high-performance and scalability, it is vital for a future implementation to make effective scheduling decisions, e.g., to spawn futures only when the overhead of parallel execution can be amortized by doing so. Such decisions must consider both the granularity of computation and the underlying resource availability. However, in the Java 5.0 Future model, the scheduling components (Executors) are implemented at the library level, i.e., outside and independent of the runtime. As a result, these components are unable to acquire accurate information about either computation granularity or underlying resource availability that is necessary to make good scheduling decisions. Poor scheduling decisions can severely degrade performance and scalability, especially for applications with fine-grained parallelism.

Users can create their own Executors and/or hard-code thresholds that attempt to identify when to spawn (and amortize the cost of spawning) or inline futures. However, this is against the original *minimal programmer effort* rationale of futures. Also, this requires expert knowledge about the dynamic behavior of the program and the characteristics (the spawn cost of futures, and the compilation systems, processor count and availability, etc.) of the platform on which the application ultimately executes. Moreover, regardless of the expertise with which the scheduling decisions are made, this model, since it is implemented outside and independent of the runtime, is unable to exploit the services (recompilation, scheduling, allocation, performance monitoring) and detailed knowledge of the system and program that the execution environment has access to.

All of these limitations motivate our work of the directive-based lazy futures with as-if-serial exception handling support, and as-if-serial side-effect guarantee, which we will discuss in details in the following chapters.

The text of this chapter is in part a reprint of the material as it appears in the proceedings of the Sixteenth International Conference on Parallel Architecture and Compilation Techniques (PACT'07) and the proceedings of the fifth international symposium on Principles and practice of programming in Java (PPPJ'07). The dissertation author was the primary researcher and author and the co-author listed on this publication ([161, 162]) directed and supervised the research which forms the basis for Chapter 5.

Chapter 6

Adaptive and Lazy Scheduling for Fine-grained Futures in Java

Given the *minimum programmer effort* design goal of futures, it is possible for a programmer to specify a large number of futures for programs that contain fine-grained, independent computations. For example, a simple Fibonacci program written using the Java Future APIs can easily produces more than millions of futures and most of them contained tiny-grained computations. It is, therefore, vital for performance that the runtime implementation of futures be efficient, and effectively amortize the cost of spawning a future in parallel, or execute the future sequentially (inline it into the current context). Naïve future implementations (e.g. one thread per future or with thread pool support) can result in significant overhead, and inefficient, even degraded, execution. Such future implementations in Java can quickly bring the system to a halt due to the multiple layers of abstraction and virtualization in the Java

Virtual Machine (JVM) for the support of system services, such as, threads, memory management, and compilation.

To limit the number of independent contexts that are spawned for fine-grained futures, programmers commonly specify thresholds to identify futures that will perform enough computation to warrant parallelization. This approach is time-consuming, and error prone. The thresholds are specific to, and different across applications, inputs for the same application, available underlying hardware resources, and execution environment, thereby, requiring significant effort and expertise by the programmer to identify optimal, or even efficient settings. Moreover, the requirement that users participate in deciding which futures to spawn or inline, is inconsistent with the original design goal of futures of placing a minimal burden on the programmer.

In this chapter, we investigate a runtime implementation that efficiently supports fine-grained futures without requiring programmer intervention with parallelization decisions. Prior work proposes several solutions for such support within functional languages or C++ [93, 113, 151]. Our focus is on supporting efficient fine-grained futures in Java. In contrast to the Java 5.0 library-level implementation of futures, we follow an runtime-based approach and extend the JVM runtime to effectively support futures. We do so since the JVM has access to low-level information about the executing program, and underlying resource availability.

Our approach, which we call *LazyFuture*, builds from, combines, and extends (i) lazy task creation [113] and (ii) a JVM program sampling infrastructure (common to many state-of-the-art JVM implementations) previously used solely for dynamic and adaptive compiler optimization. We couple these techniques with dynamic state information from the underlying, shared-memory, multiprocessor resources, to adaptively identify when to spawn or inline futures.

In the following sections, we first describe the design and implementation details of our LazyFuture system. We then empirically compare the various implementation alternatives and evaluate the overall efficacy of our system. Finally, we discuss the related work and then conclude.

6.1 Programming Model

LazyFuture is a futures implementation for Java that we propose to support efficient execution of fine-grained futures. Our goal is to eliminate the need for programmers to decide when, and how to spawn futures in parallel for applications with fine-grained futures. For such applications, programmers commonly specify a computational granularity that amortizes the cost of spawning a future in parallel. Figure 6.1 (a) is the Fibonacci program using the Java 5.0 Future APIs. This program uses a threshold to avoid spawning overhead for small computations.

Bench-marks	Inputs size	Total# of futures	CPU 1.60GHz proc#=2(base)	CPU 1.60GHz proc#=4(base)	CPU 2.40GHz proc#=2(base)	CPU 2.40GHz proc#=2(opt)
AdapInt	0-250000	5782389	7000000	8000000	17000000	19000000
FFT	2^{18}	262143	4096	32768	16384	65536
Fib	38	39088168	30	32	36	33
Knapsack	24	8466646	5	7	6	4
Quicksort	2^{24}	8384315	131072	131072	131072	524288
Raytracer	pics/balls.nff	265409	32	16	32	64

Table 6.1: Evidence that threshold values vary widely across configurations for the same program and input. We identified these thresholds empirically from a wide range of threshold values.

In practice, this threshold is difficult and tedious to identify, and can have a large impact on performance since the optimal values vary significantly across applications, inputs, available underlying hardware resources, and execution environments. To validate this claim, we empirically identified the thresholds for optimal performance for six benchmarks. We present these thresholds in Table 6.1. We gathered results on two machines: one with four 1.60GHZ processors, the other with two 2.40GHZ processors. On the 4-processor machine, we collected data with 2 as well as 4 processors. On the 2-processor machine, we used two different configurations of the same JVM. We provide specific details of our methodology in Section 6.3. This data confirms that the best thresholds vary across different configurations. LazyFuture frees the programmers from the task of threshold specification, and enable the system to decide when and how to spawn futures in parallel adaptively.

We define a new abstraction, called `LazyFutureTask`, which implemented the `Future` interface in Java 5.0 Future APIs. Users create a `LazyFutureTask` ob-

```
public class Fib
  implement Callable<Integer>
{
  private int n;
  public Fib(int n){this.n = n};

  ExecutorService executor = ...;

  public Integer call(){
    if(n < 3) return 1;

    if(n < THRESHOLD) {
      return (new Fib(n-1)).call()
        + (new Fib(n-2)).call();
    }else{
      Future<Integer> f =
        executor.submit(new Fib(n-1));
      int x = (new Fib(n-2)).call();
      return x + f.get();
    }
  }
}
```

(a) Fib using Java 5.0 Futures

```
public class Fib
  implements Callable<Integer>
{
  private int n;

  public Integer call() {
    if (n < 3) return n;
    LazyFutureTask<Integer> f =
      new LazyFutureTask(new Fib(n-1));
    f.run();
    int x = (new Fib(n-2)).call();
    return x + f.get();
  }
  ...
}
```

(b) Fib using LazyFutures

Figure 6.1: Comparing programming models of Java 5.0 Futures and LazyFutures

ject for each potentially asynchronous computation and invoke its `run()` method directly (in a way similar the traditional Java thread model). Figure 6.1 (b) shows the LazyFuture implementation of *Fib*. The LazyFuture-aware JVM recognizes this `run()` method (in each `LazyFutureTask`), and makes scheduling decisions automatically and adaptively based on the computation granularity and the underlying resource availability to achieve the best performance.

6.2 Implementation

Our implementation of LazyFutures is inspired by the technique proposed by Mohr et al. [113], called *lazy task creation* (LTC). LTC initially implements all futures as function calls. The system then maintains special data structures for the computation of future's parent, the caller (called a continuation), to be spawned. When there is an idle processor available, the idle processor steals continuations from the first processor and executes code in parallel with the future. Similar techniques are employed in many systems to support fine-grained parallelism [120, 51, 57, 144].

Our system, although similar, is different from these prior approaches in several ways. First, we combine information about computation granularity with resource availability. Prior work commonly considers only the latter, since estimating the computation granularity at runtime is complex, and can introduce significant overhead. Our implementation is, however, targeted at state-of-the-art JVMs, which implement a low-overhead runtime profiling system that the runtime uses to guide adaptive compilation and optimization [8, 117, 141, 85]. We leverage this mechanism to extract accurate and low-level program (e.g. long running methods) and system information (e.g. number of available processors) with low overhead.

The second unique aspect of our implementation is that we do not employ a worker-based, specialized runtime system for futures. Systems like LTC typically associate a worker with each physical processor, and this worker is responsible for executing the current task, stealing tasks from other workers, and managing the task queues. Such systems assume that futures (or special kind of tasks that the system supports) are the only kind of parallel activities in the system. In addition, these systems map runtime threads directly to operating system (OS) threads. Such a setup is not appropriate for a JVM since this would equate to mapping worker threads to Java threads (which are themselves mapped to OS threads), thereby, adding an additional level of indirection, and overhead to scheduling. Moreover, a JVM would need to accommodate varied types of parallel constructs specified in Java, other than futures.

In our system, we integrate future management with the existing thread scheduling mechanism in the JVM. When the system identifies a future to spawn on the runtime call stack of a thread, the system splits the thread into two – one that executes the future,and the other that performs the continuation. Both threads are considered Java threads by the thread scheduler. With this implementation, we take advantage of the highly-tuned JVM thread scheduler, synchronization, and load-balancing mechanisms, which significantly simplifies the implementation of futures, and makes our implementation compatible with Java threads and other parallel constructs.

Finally, as opposed to the commonly used work-stealing approach [113, 51], a thread in our LazyFuture system voluntarily splits its stack and spawns its continuation using a new thread. The system performs such splits at thread-switch points

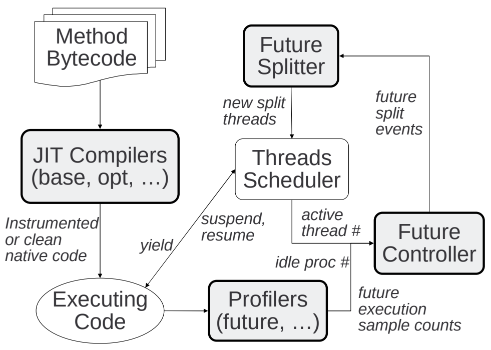


Figure 6.2: Overview of LazyFuture implementation.

(method entries and loop back-edges), when the monitoring system identifies an unspawned future call as long-running (“hot” in adaptive-optimization terms). With the volunteer stack splitting mechanism, we avoid the synchronization overhead incurred by work-stealing, which can be significant in a Java system [38].

6.2.1 Implementation Overview

Figure 6.2 overviews our system. All shaded components identify our extensions to the JVM. After a class is loaded by the class loader, the method bytecodes are translated to native code by the Just-in-time (JIT) compiler (non-optimizing, as well as optimizing). The compiler may insert instrumentation into the native code to collect profiling information from the program that the compiler can later use to perform optimizations.

We extend the JIT compilers to insert a small stub at the entry point and exit point of every future call. Initially, our system treats every future call as a function call, i.e., the system executes the code on the stack of the current thread. At the same time, we maintain a small side stack for each thread, called a *future stack* (See Figure 6.3). Every entry in the future stack has two words, one is the offset of a future frame on the current stack, the other is the sample count that holds an estimate of how long the future call has executed. The stubs push an entry onto the future stack at the beginning of a future call, and pop the entry when exiting the future call. We implemented these stubs carefully in the JIT compilers, and ensure that they are always inlined, to avoid unnecessary overhead.

To estimate the computation granularity of futures, we extend the existing JVM sampling system. In our prototype JVM, light-weight method sampling occurs at every thread switch (approximately every 10 ms), which increments sample counts of the top two methods on the current stack. Methods with sample counts exceeding a certain threshold will be identified as hot methods, and recompiled with higher levels of optimizations. We extend this mechanism by also incrementing the sample counts of executing futures. These sample counts provide our system with an estimate of how long the futures have executed. Our scheduling system spawns futures whose sample counts exceed a particular threshold. This process avoids spawning short-running futures – the overhead of which cannot be amortized by the benefits from parallel execution.

The system feeds the future sample counts into the *future controller*, which couples the sample counts with dynamic system resource information from the thread scheduler, e.g., the number of currently active threads and idle processors, to adaptively make decisions about splitting futures, in order to enable additional parallelism.

If the future controller decides that it is beneficial to split a future, it creates a *futureSplitEvent* that contains information about the future, such as the frame offset and sample counts. The controller forwards the event to the *future splitter*, which splits the current thread into a future thread and a continuation thread, and places both threads on the appropriate queue of the thread scheduler for further execution. Note that both the future controller and future splitter are services invoked by the current thread, when the thread yields to enable thread switching. Therefore, we require no additional synchronization since the system implements this process on a per-thread basis.

6.2.2 Future Splitting Triggers

Ideally, we should spawn a future when there is an idle processor. We refer to this approach *idleProc triggered*. In our system, future splitting is initiated by the running thread, and only occurs during thread switching. If a processor becomes idle during execution, there is a delay before a thread detects this and makes the splitting decision. There is also a small delay between when a future is spawned, and when it is scheduled to execute. Thus, the idleProc triggered policy may not utilize the system resource fully in some cases.

One alternative is to saturate the system with futures. To implement this policy, we maintain twice as many threads as processors for futures that the system selects. That is, if the sample count of a future call on stack exceeds the threshold, and the current number of active threads is less than twice the number of processors, the current thread will be split to make the future call a parallel call. We refer to this approach *sampleCount triggered*. This policy helps to pre-saturate the system if enough parallelism is available, but imposes a delay for "learning" that a future is long-running, i.e., the time it takes for the sample count to exceed the threshold.

Therefore, we consider a hybrid approach, which we call *sample+idle triggered*. Note that in all policies, since the system performs future splitting (spawning) only at thread switching, it automatically eliminates futures with granularity of less than 10 ms from spawning.

6.2.3 Future Splitter

Figure 6.3 overviews our process for splitting futures. In the figure, the current thread has three future calls on its stack. At some point, the future controller decides

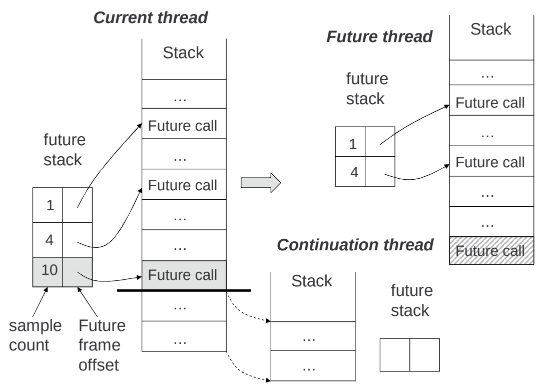


Figure 6.3: The future splitting process of LazyFutures.

that it is worthwhile to spawn the oldest future call with sample count 10 for parallel execution. The dark line identifies the split point on the stack. The future splitter then creates a new thread for the continuation of the spawned future call, copies the stack frames below the future frame, which corresponds to the continuation, restores the execution context from the stack frames, and resumes the continuation thread at the return address of the spawned future call. Note that we choose to create a new thread for the continuation instead of the spawned future, so that we do not need to setup the execution contexts for both threads. The spawned future call becomes the bottom frame of the current thread. The system deletes the future stack entry so that it is no longer treated as potential future call.

6.2.4 Optimizing Synchronizations

If the result of a future is used by its parent, the system will check whether or not the result is available. If it is not, the parent blocks until the future completes. This synchronization process can be avoided if the future is not spawned. In this case, the result of the future is ready at the time the future call returns to its parent, and thus, will always be ready at its usage points. To optimize this case, we add a *onStack* flag to each future object. We initialize the flag to true and set it to false if the future splitter spawns the future. When the result of a future is requested, if its onStack flag is true, the system returns the result directly, otherwise, we synchronize the process with its future execution.

6.3 Experimental Methodology

We implemented LazyFutures in the open source Jikes Research Virtual Machine (JikesRVM) [84] (x86 version 2.4.2) from IBM Research. To evaluate the efficacy of our approach, we also implemented two other alternatives to support futures in Java: one that spawns a thread for every future and another that uses a variable-length thread pool to execute futures. We refer to these implementations as *singleThread* (ST), *thread Pool* (TP), respectively.

To investigate the impact of LazyFutures on different application types, we developed two sets of benchmarks. The first set includes *Crypt*, *MonteCarlo*, *Series* and *SparseMatmult*, which is a subset of the multithreaded version of Java Grande Benchmark Suite [133]. These four benchmarks are chosen because there is no mutual dependency between spawned parallel tasks in these benchmarks, which makes them suitable to be expressed by futures. The structure of these benchmarks is similar: the main thread spawns several futures to compute subtasks, and then it waits for all futures to finish. The number of futures to spawn can be specified by users on the command-line option, and is usually set to the number of processors available. This kind of applications represents coarse-grained parallelism. The singleThread implementation is usually sufficient to handle such applications. We use this set of benchmarks to evaluate the overhead introduced by our LazyFuture implementation.

The second set of benchmarks includes *AdapInt, FFT, Fib, Knapsack, QuickSort, Raytracer*. All of the programs employ a divide and conquer model. We adopt them from the examples provided by the Satin system [148]. The recursive nature of these benchmarks results in excessive number of futures with very different granularities. We use this set of benchmarks to evaluate whether our LazyFuture implementation can make effective future splitting decisions automatically and adaptively.

We conduct our experiments on a dedicated 4-processor box (Intel Pentium 3 (Xeon) xSeries 1.6GHz, 8GB RAM, Linux 2.6.9) with hyper-threading enabled. Thus, we report results for up to 8 processors. We execute all benchmarks repeatedly and present the minimum. For each set of experiments, we report results for two JVM configurations respectively: one with the non-optimizing (baseline) compiler and the other with the highly-optimizing (opt) compiler. For the optimizing configuration, we use the adaptive setting [8] which optimizes frequently executed methods only. To eliminate non-determinism, we use the pseudo-adaptive configuration [14], which mimics the adaptive compiler in a deterministic manner by applying the optimizing compiler to code according to an advice file that we generate offline. We include results for both JVM configurations to show how well our future implementation identifies long-running futures. Unoptimized futures will execute for a longer duration than the optimized versions, and consequently, our system will automatically adapt to the code performance and execution environment, and make different spawning decisions.

Finally, we use a sample count of 5 as the splitting threshold for the sampleCount policy in our results. We selected this value empirically from a wide range of values that we experimented with. We find that this value, across benchmarks, imposes only a small "learning" delay, and effectively identifies futures for which the overhead of spawning is amortized by parallel execution.

6.4 Performance Evaluation

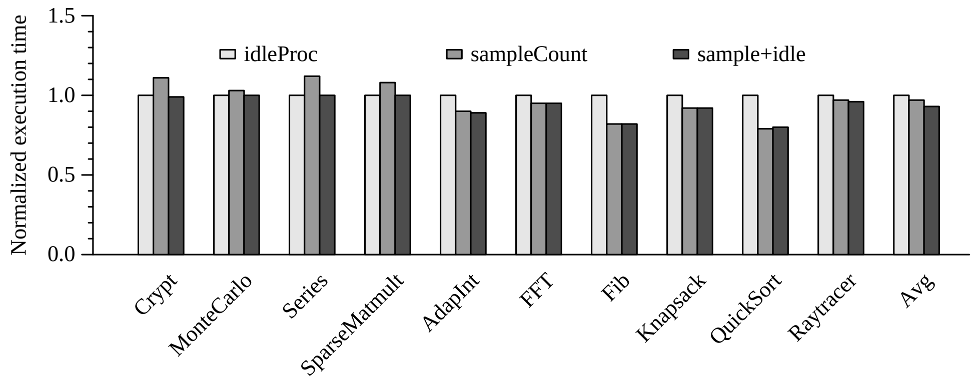
In this section, we first evaluate the efficacy of different future splitting triggers. Then we analyze the performance impact of our lazy future implementation in detail for all benchmark sets.

6.4.1 Comparison of Splitting Triggers

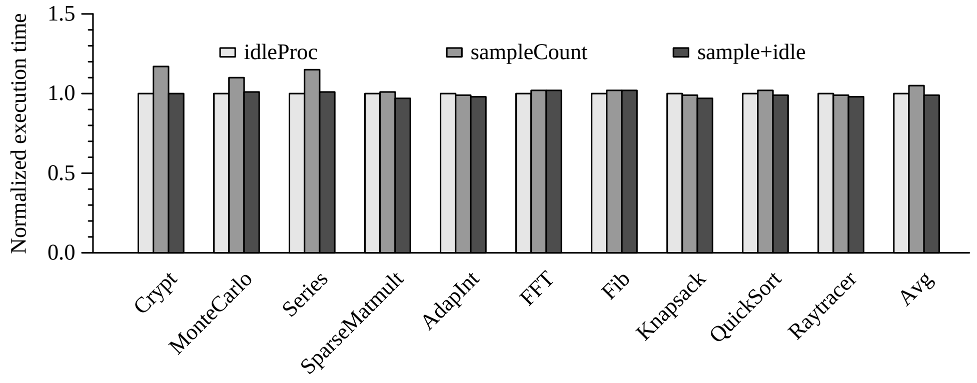
As we discussed in section 6.2.2, in our system, future splitting can be triggered by either available idle processors or high future sample count, or both. In this section, we compare performance of all three triggers.

Figure 6.4 shows the execution time for all benchmarks with different splitting triggers. We normalize the data relative to *idleProc* for comparison. The first four benchmarks are from the JavaGrande suite, and the rest are from our divide and conquer suite. Graph (a) shows the results when we use the baseline compiler and graph (b) shows results with the pseudo adaptive optimization setup.

The data indicates that for applications with few coarse-grained futures (the first four benchmarks), the *sampleCount* triggered policy is less effective than the *idleProc* policy. This is due to the delay required to “learn” whether a future will be short or long running by the *sampleCount* policy – when there are several idle processors available.



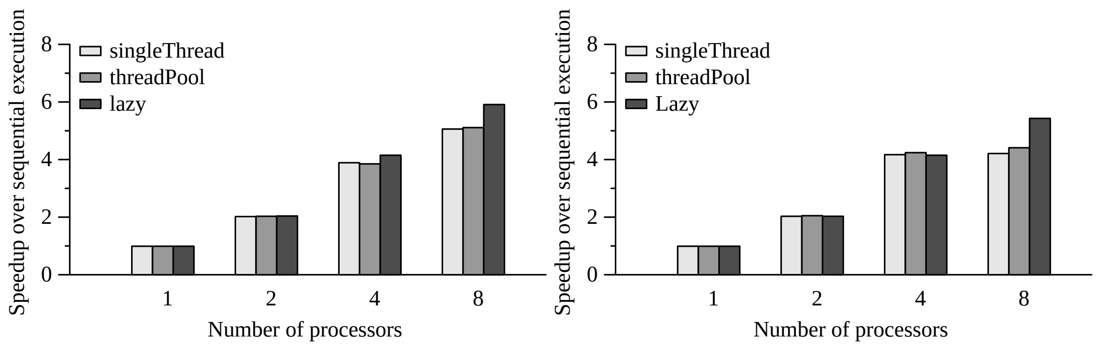
(a) With baseline compiler



(b) With optimizing compiler

Figure 6.4: Performance comparison of future splitting triggers.

For applications with a large number of fine-grained futures (the remaining benchmarks), the *sampleCount* trigger outperforms the *idleProc* trigger in most cases since it helps saturate the system with qualified futures to utilize the system better. This trend is more apparent when the baseline compiler is used. This is because the baseline compiler produces unoptimized code for both the system and the application, which makes the process of detecting idle processors, splitting and scheduling futures



(a) with baseline compiler (b) with optimizing compiler

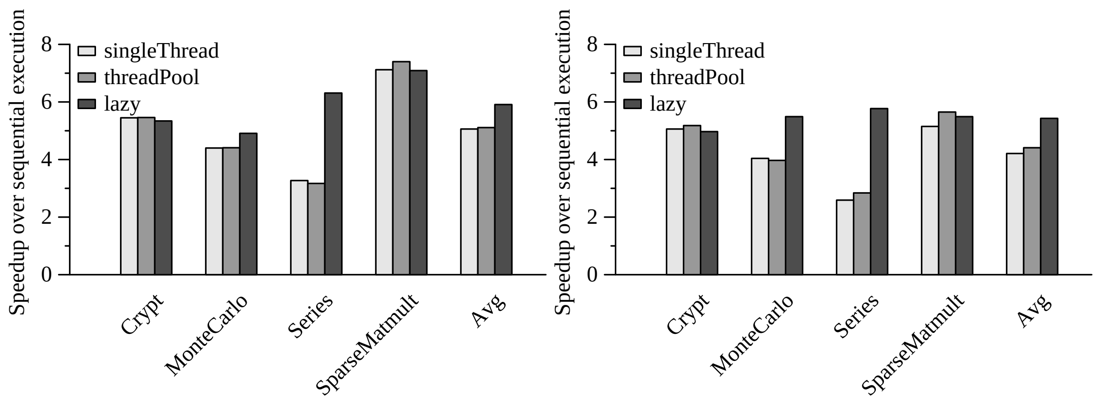
Figure 6.5: Average speedups of LazyFutures for JavaGrande benchmarks.

take longer. Thus, pre-saturating the system using the sampleCount trigger makes a bigger difference. In summary, by combining both triggers, the hybrid *sample+idle* policy achieves the best performance among all triggers. All results in further sections use the hybrid trigger.

6.4.2 JavaGrande Performance

In this section, we evaluate the performance impact of our LazyFuture implementation on the four JavaGrande benchmarks. This set of benchmarks represents applications with a small number of coarse-grained futures.

Figure 6.5 shows the average speedup over the sequential version of each benchmark. The x-axis is the number of processors used. Note that the 8-processor case is actually the 4-processor case with hyper-threading. We set the number of futures



(a) with baseline compiler (b) with optimizing compiler

Figure 6.6: Individual JavaGrande benchmark speedups of LazyFutures with 8 processors.

in the applications to the number of processors used. We present three implementation alternatives: one thread per future (*singleThread*), variable-length thread pool (*threadPool*), and our LazyFuture implementation (*lazy*). Graph (a) and (b) are results for the baseline compiler and the optimizing compiler, respectively.

The data shows that with baseline compiler, all three implementations produce similar average performance: 1% overhead with one processor and around 2x speedup with two processors. When there are more processors available and more futures created, the threadPool implementation starts show a small improvement over the singleThread implementation. Our LazyFuture implementation is competitive with the other alternatives, and outperforms them on average as the processor count increases.

Note that LazyFutures require a “learning time” of at least 10ms (time for one thread switching) for each future spawned to decide if the computation time warrants parallelization. The other two alternatives do not require a learning time.

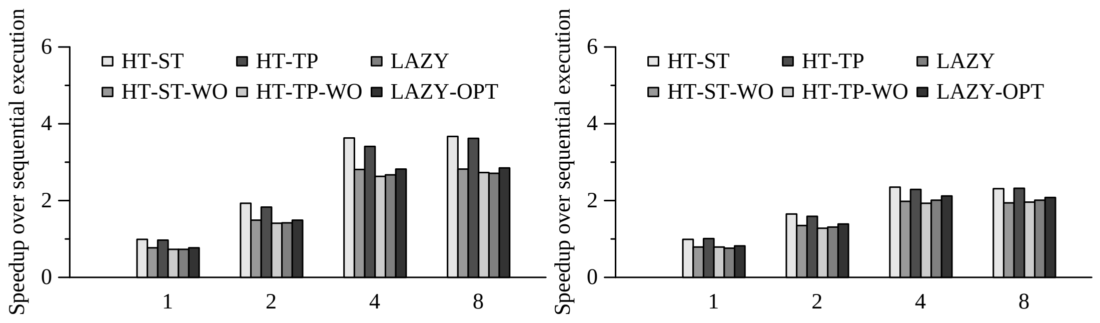
To investigate these results in greater detail, we present speedup of the individual benchmark in Figure 6.6 for the 8 processor data. Graph (a) and (b) are the results with the baseline compiler and the optimizing compiler respectively. This figure shows that the LazyFuture implementation does introduce some overhead ($< 2\%$) for two benchmarks (*Crypt*, *SparseMatmult*) due to the learning delay. However, for the other two benchmarks, especially *Series*, this slight splitting delay actually improves performance significantly. We believe that in this case, the slight slowdown of future creations of our system reduces the contentions of system resources, such as cache conflicts, comparing to the other alternatives.

The average speedup with 8 processors is 5.0x for *singleThread*, 5.1x for *threadPool*, 5.9x for *lazy* when the baseline compiler is used. With the optimizing compiler, the average speedup is 4.2x for *singleThread*, 4.4x for *threadPool*, and 5.4x for *lazy*. In both configurations, our LazyFuture system outperforms the other two on average.

6.4.3 Divide and Conquer Performance

We next evaluate the performance impact of our LazyFuture implementation for the divide and conquer benchmark suite. We compare our approach to the singleThread and threadPool alternatives above using a hand-tuned granularity threshold. We identify the best performing thresholds experimentally for our various configurations and benchmarks. These two alternatives represent the case where the programmer specifies the threshold for spawning given perfect knowledge of the underlying system. This in practice is not feasible for all inputs, operating and runtime systems, and processor configurations, and it introduces a tremendous burden on the programmer. Our LazyFuture system requires only that the programmer specify which code regions can execute in parallel. The comparison between our LazyFuture system and the singleThread and threadPool configurations with the best, hand-tuned thresholds indicates the degree to which our system makes the appropriate spawning decisions.

We consider an additional configuration in our result set for these benchmarks. Using the current Java Concurrency Utilities [86], the system will create a future object for each future regardless of whether it is executed inlined or in parallel. In the hand-tuned alternatives, we do not create future objects if the future computational granularity is below the threshold. To investigate and report the overhead of this object allocation and to show the overhead inherent in doing so, we also include configurations of the hand-tuned alternatives that create future objects for *all* future instances even those that are below the threshold; however, we only spawn those above the threshold.

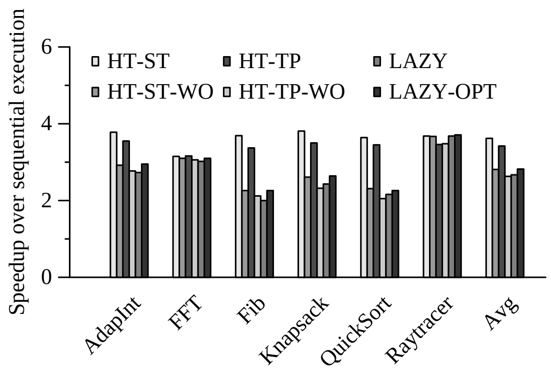


(a) with baseline compiler (b) with optimizing compiler

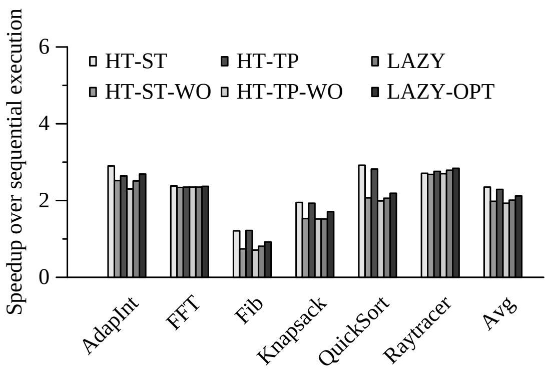
Figure 6.7: Average speedups of LazyFutures for Divide and conquer benchmarks.

Figure 6.7 shows the average speedup over the sequential version for our divide and conquer benchmarks (fine-grain parallelism). The x-axis is the number of processors that we used for each experiment. The first two bars are results for the singleThread (ST) implementation with hand-tuned (HT-) thresholds, the middle two bars are results for the threadPool (TP) implementation with hand-tuned (HT-) thresholds. The last two bars are results for the lazy (LAZY) implementation, without and with optimizing synchronizations (-OPT) (Section 6.2.4). We use "-WO" to identify the configurations that we create wrapper objects for all future instances for the hand-tuned alternatives.

The data indicates that the overall speedup for this benchmark set is less than that of the JavaGrande benchmarks due to the fine-grained nature of these programs. Our LazyFuture implementation produces comparable, in some case better, performance



(a) with baseline compiler



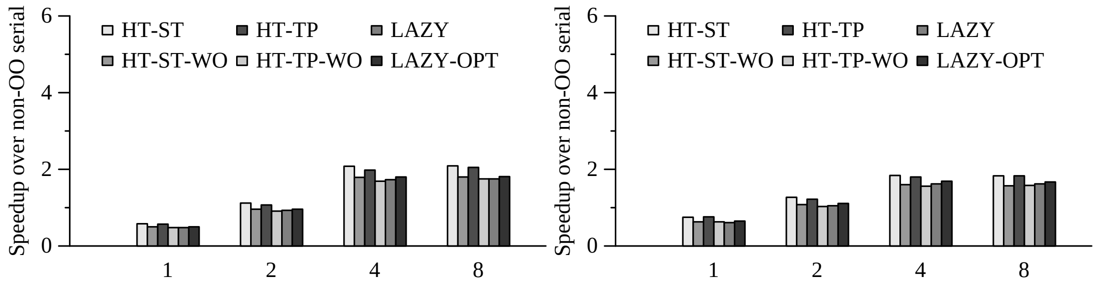
(b) with optimizing compiler

Figure 6.8: Individual divide and conquer benchmark speedups of LazyFutures with 4 processors.

than the hand-tuned thresholds – when we exclude the overhead of object allocation (HT-ST-WO and HT-TP-WO). The better performance is due to the fact that thresholds specified by programmers are static, and thus do not adapt to resource availability as our LazyFuture implementation does.

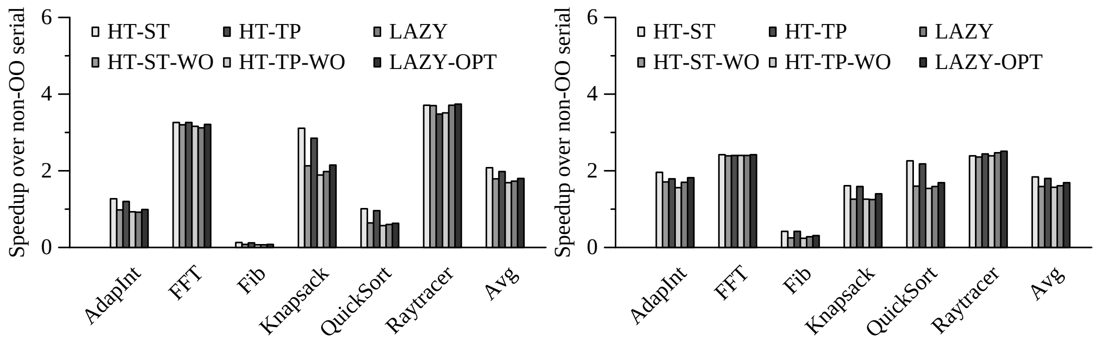
The differences between HT-ST-WO and HT-ST, or HT-TP-WO and HT-TP show that the extra unnecessary object allocation has significant performance impact on applications with fine-grained futures, although an optimizing compiler reduces the differences to some degrees (see Figure 6.7(b)).

Since these benchmarks almost always saturate the system with a large number of futures, hyper-threading does not help to improve the performance. Therefore, we show individual speedups with 4 processors for this set of benchmarks in Figure 6.8 to enable a more detailed analysis. This figure shows as more futures are created (more



(a) with baseline compiler (b) with optimizing compiler

Figure 6.9: Average speedups of LazyFutures for Divide and conquer benchmarks over non-OO serial version.



(a) with baseline compiler (b) with optimizing compiler

Figure 6.10: Individual divide and conquer benchmark speedups of LazyFutures over non-OO serial version with 4 processors.

than 5 million for four benchmarks, see the second column of Table 6.1), the larger is the difference between HT-ST and HT-ST-WO. The Fib benchmark represents the worst case by creating almost 40 million futures object. The optimizing compiler is able to reduce the overhead primarily by inlining object allocation and initialization; however, the overhead is still significant.

To investigate the performance impact of unnecessary object allocation further, we re-implemented the serial version of each benchmark in a non-object-oriented (non-OO) style. In the non-OO serial version, instead of creating a new object and invoking its virtual method for each computation, we invoke a static method without creating any object. We then generate speedup numbers over the non-OO serial versions, and present the results in Figure 6.9 for average speedups and Figure 6.10 for 4 processors. The speedup numbers in these two figures are much lower than those in Figure 6.7 and Figure 6.8.

In addition, the more futures are created, the larger is the difference. These differences, and the differences between HT-ST-WO and HT-ST (or HT-TP-WO and HT-TP), imply that there is a large potential performance gain for our LazyFuture implementation if the system is able to avoid creating unnecessary objects for future calls executed inlined. To achieve this, we believe that the language constructs ([126, 27]) as opposed to interface-based constructs (e.g., the Java Future API) will provide the JVM more flexibility and opportunities of optimizations, and thus, enable more efficient support of fine-grained futures. We will investigate this hypothesis in depth in next chapter.

Finally, to show the frequency of future spawning, we present Table 6.2. The table lists the number of Java threads created by each implementation alternatives with 4 processors. Since the "-WO" configurations have same thread number as its

Bench-marks	HT-ST		HT-TP		LAZY	
	base	opt	base	opt	base	opt
AdapInt	227	230	70	62	52	42
FFT	18	29	14	26	43	36
Fib	31	29	158	17	66	50
Knapsack	137	44	77	144	105	29
Quicksort	150	77	103	55	103	76
Raytracer	266	29	30	20	100	48

Table 6.2: Number of Java threads spawned.

corresponding non-WO version and LAZY and LAZY-OPT also have similar counts, we only show numbers for HT-ST, HT-TP, and LAZY. “base” stands for the baseline compiler, and “opt” stands for the optimizing compiler. Note that each configuration has different threshold, so the specific values are incomparable. Instead, the data shows the efficacy of our LazyFuture system by comparing the thread number created by the LAZY implementation to the number of futures created by these applications (see the second column of Table 6.1). In summary, our LazyFuture system is able to make intelligent future inlining/spawning decisions automatically and adaptively, based on dynamic information of system resource availability and future granularity.

6.5 Related Work

Load-based inlining [93] was the first approach proposed to address the fine-grained future problem. The idea is to make spawning decision at the creation time based on the system load. A future is computed in parallel if there is enough available resource. Otherwise, it is inlined. One major drawback of this approach is that the decision is not revocable: once a future is inlined, it cannot be parallelized anymore. Task starvation may occur due to imbalance work load and bursty task creation.

Lazy task creation (LTC) [113] is a more elaborate scheme to support fine-grained futures. In this approach, all futures are initially evaluated like a sequential call. But the system maintains minimum information to spawn the continuations of futures retroactively if a future is blocked, or a computation resource becomes available. This principle of sequential first, parallel retroactively if necessary, can be found in many systems that target fine-grained parallelism [120, 57, 51, 143], each with its own contexts and refinements. Our system follows the laziness principle as well. However, we believe that our system is the first effort to support fine-grained futures in a Java Virtual Machine. Our system is built upon the general thread scheduling system in the JVM and is incorporated with the sampling system which was previously used for dynamic compilation solely. This enables our system to exploit both system resource availability and futures' computation granularity while making inline decisions. While in the previous system, splitting is triggered only by a blocked task or an idle processor. The task granularity is not monitored and considered.

Another effort to support fine-grained futures is called *leapfrogging* [151]. Leapfrogging is a workcrew-style implementation. A task object is created for a future invocation and is put into a task pool. A worker takes a task from the pool and works on them one by one. When a worker is blocked due to some unfinished future, it steals a task that the current task is dependent on and starts to execute the stolen task on top of the current stack. Leapfrogging can be expressed in C's stack frame management mechanism, and thus, it is easier to implement and more portable comparing to LTC. Comparing to our approach, however, it does not consider the granularity of futures, and it has the queue management overhead introduced by its workcrew-style implementation.

There are several previous works related to our synchronization optimization. For example, in [51], there are two clones of each procedure: a fast clone used while the procedure is invoked locally and a slow clone that is used while the procedure is stolen by another processor. In the fast clone, all *sync* operations are translated to *noop* to avoid unnecessary synchronization. Our system is slightly different in that we do not keep two clones of a method. Instead, we use the *onStack* flag which is set dynamically by the future splitter to eliminate unnecessary synchronization. In [50], static analysis is used to eliminate redundant touch operations for futures, which is complementary to our dynamic approach.

Profiling has been used to choose the best parameters of parallel optimizations [41] or the optimal number of threads to use given available system resources [88], etc. In most of these systems, it is assumed that one computation will be invoked repeatedly and the execution will last for a long time. Therefore the system can use several initial runs for learning before making a decision. Our system, however, targets at fine-grained futures, most of which have very short execution time, and usually are not invoked repeatedly. Thus, we use sampling to monitor how long a future has been executed, and to make splitting decision for the current future, instead of its later invocation. We plan to investigate the possibility of exploiting profiling for repeatedly invoked computation as part of future work.

Safe futures proposed in [153] enforce the semantic transparency of futures automatically using object versioning and task revocation so that programmers are freed from reasoning about the side-effects of future executions to ensure correctness of programs. This is complementary to our system and we plan to investigate the performance impact of LazyFutures in combination with safe futures as part of future work.

The concept of futures is also employed in distributed environments to optimize task scheduling [81]. Data futures are created to refer to data products that have not yet been created. Their system is similar to our system in the sense of dynamic future scheduling based on cost/benefit estimation. But it is at a much more coarse-grained level with different cost/benefit tradeoffs.

6.6 Summary

In this chapter, we introduce our first work towards easy and efficient future support in Java. We enable automatic future task creation and scheduling so that programmers do not need to manually and explicitly manage future execution in the code. We achieve this by providing runtime support in the JVM. Our method combines lazy task creation using stack split and adaptive task scheduling with sophisticated runtime program sampling. We empirically evaluated our LazyFuture system using a set of Java benchmarks with different implementation approaches and configurations. Our results show that our LazyFuture system not only makes future programming easier but also enables efficient future execution that is comparable with hand-tuned alternatives.

The text of this chapter is in part a reprint of the material as it appears in the proceedings of the Eighteenth International Conference on Parallel and Distributed Computing Systems (PDCS'06). The dissertation author was the primary researcher and author and the co-author listed on this publication ([163]) directed and supervised the research which forms the basis for Chapter 6.

Chapter 7

Directive-based Lazy Futures in Java

The LazyFuture model takes the burden of scheduling futures off programmers. However, its programming model follows an interface-based approach that is similar to (yet more efficient than) Java 5.0 Futures. As a result, it inherits similar programmer productivity and performance disadvantages. Using the interface-based approach, users must employ object encapsulation of futures, and thus, incur memory allocation and management overhead. In addition, the coding style using this methodology imposes an extra burden on the programmer and causes source code to be longer and less readable in order to specify and use the interface. To address these limitations, in this chapter, we propose a new implementation of futures in Java that we call *Directive-based Lazy Futures (DBLFutures)*.

DBLFutures are inspired by parallel programming models for other languages that employ keywords or directives to identify parallel computations [15, 101, 27, 3, 115]. Using the DBLFuture programming model in Java, users annotate the variable declarations of all variables that store the return value from a function that can be potentially executed concurrently with `@future` directives. Using DBLFutures, the parallel version of a program is the same as the serial version with annotations on a subset of variable declarations.

In this chapter, we present the design and implementation of DBLFutures, and then evaluate the performance impact of this directive-based programming model for a set of benchmarks with fine-grained futures.

7.1 Implementation

Our DBLFuture implementation builds upon and extends LazyFutures to improve the ease-of-use of future-based parallelism in Java as well as performance and scalability. DBLFutures exploit the Java language extension for annotations (JSR-175 [87]). Annotations are source code directives that convey program metadata to tools, libraries, and JVMs; they do not directly affect program semantics. In particular, we introduce a future annotation (denoted `@future` in the source code) for local variables. Users employ our future directive to annotate local variables that can be used as placeholders of results returned by function calls that can be potentially executed concurrently by the system. If a function call stores its return value to a annotated local variable, it is identified as a future function call. Note that in our system,

```
public class Fib
{
  public fib(int n) {
    if (n < 3) return n;
    @future int x = fib(n-1);
    int y = fib(n-2);
    return x + y;
  }
  ...
}
```

Figure 7.1: The Fibonacci program using DBLFutures

the scope of future annotations is within the method boundary. If the value of a local variable with the future annotation is returned by the method, the future annotation will not be returned with the return value. Figure 7.1 shows the implemented *Fib* program using this model.

Our DBLFuture model avoids creation (and thus, user specification) of `Callable`, `Future`, `LazyFutureTask`, or other objects when the future is inlined (executed sequentially) by the system. As such, we avoid the memory allocation, memory management, and extra source code required by previous approaches. With this model, users easily specify computations that can be safely executed in parallel with minimal rewriting of the serial programs. This programming methodology also provides the JVM with the flexibility to implement potentially concurrent code regions as efficiently as possible. Note that with our current implementation of DBLFutures, the JVM makes efficient spawning decisions automatically, but the users are still responsible to ensure the safety of concurrent execution.

Our DBLFuture-aware JVM recognizes the future directives in the source and implements the associated calls using a set of LazyFuture extensions and compiler techniques. First, the future directive in the source is saved as a method attribute in the bytecode. The class loader of our DBLFuture-aware JVM recognizes this attribute and builds a future local variable table for each method, which contains the name, index, and bytecode index range of each future local variable. Our Just-In-Time, dynamic compiler consults this table during compilation.

Initially, the JVM treats every future call as a function call, and executes the code on the runtime stack of the current thread. For each such call, the system also maintains a small stack that shadows the runtime stack for each thread, called the *future stack*. This future stack maintains entries for potential future calls only. Each entry contains metadata for the corresponding runtime stack frame of the future call that includes the location of the frame on the runtime stack and a sample count that estimates how long the future call has executed. The system uses this information to make splitting and spawning decisions.

Each DBLFuture shadow stack frame also contains the local variable index and the stack slot in the runtime stack of the caller of the future call that the compiler has allocated for this local variable. Our system employs this information to set up the future and continuation thread correctly upon a split and spawn.

For LazyFutures, the `LazyFutureTask.run()` method is the only marker of potential future calls in the program. In addition, the process of storing the return value of a future call and accessing the value later on is explicitly coded in the application via implementation of the `run()` and `get()` methods of the `LazyFutureTask` class. The `LazyFutureTask` object serves as the placeholder of the computation result, and is always created regardless of whether the computation is inlined or spawned.

The LazyFuture compiler implements a small, inlined, and efficient, stub in the prologue and epilogue of the `run()` method. This stub pushes an entry onto the future stack at beginning of a future call, and pops the entry off of the future stack when exiting the future call. In addition, the return type of the `run()` method is void, so the address of the first instruction of the continuation is the return address of the run method. Thus, upon future splitting, the system can extract the return address from the runtime stack frame for the `run()` method, and use it as the starting program counter (PC) of the new thread (that will execute the continuation). The system sets the original return address to a stub that terminates the current thread when the future call completes.

DBLFutures require a somewhat more complex compilation approach. We maintain the future stack for every marked future call as is done for LazyFutures. However, we want to allow any method call to be specified as a potential future call if it can be executed safely in parallel. We also want to allow the same method definition to be used in both a future and a non-future context. The extant compilation strategy requires that we produce two versions of compiled code for every method that may be used in the future context, and insert stubs into the prolog and epilog of all such methods. This is not desirable since it causes unnecessary code bloat and compilation overhead. Instead, we expand the future call cites and insert future stack maintenance stubs before and after the call site of the future.

The store of the return value after the future call completes requires special handling. If the call is not split, the return value must be stored into the specified local variable. If the future is split and spawned, the return value must be stored into a placeholder (i.e. a Future object) for access by the continuation thread. To enable this, we add one word to every runtime stack frame, for a *split flag*. This flag is a bitmap of spawned futures indexed by the future local variable index in the bytecode local variable array. For example, if the future call associated with a local variable at index 1 is spawned, the JVM sets the second lowest bit of the flag to 1. The JVM checks this bit at two points in the code: (i) at the store of the return value and (ii) at the first use of the return value. We currently support 32 futures (64 for 64-bit machines) per method given this use of a bitmap. However, we can extend this by using the last bit to indicate when there are more futures, and storing a reference to a full-fledged bit-vector if so.

Our compiler always allocates a slot on the runtime stack for every future-annotated local variable. This slot holds different variable types at different times: before splitting, its type is the declared type of the local variable; after splitting, it holds a reference to a `Future` object which is created and set by the splitting system in the JVM; after its first use, its type becomes the declared type again. To ensure correct garbage collection (GC), the compiler includes this slot in the GC maps and the garbage collector dynamically decides whether it holds a reference or not using the split flag.

We compile the return value storage point to a conditional branch. If the split flag is set, the code stores the return value directly in the local variable slot on the stack. Otherwise, the code extracts the reference to the `Future` object from the same stack slot, and stores the return value into the `Future` object.

We similarly expand instructions that use the return value. If the split flag is set, the codes uses the value in the local variable slot on stack directly; otherwise, the code executes the `get()` method on the `Future` object that it extracts from this same slot (which will block if the return value is not ready yet). In this latter case, when the system eventually returns a value from a method via the `get()` method, it also stores the value in the slot (an thus, the slot at this point holds the type of the original local variable). If there are multiple use points, our compiler only converts the first one (the one that dominates the others) since all uses thereafter are guaranteed to access the value with the original declared type. In addition, our compiler will insert a fake use of the future value before the method exit point if there is no usage of the future to prevent it escaping the method boundary. That is, a method will wait for all futures that it spawns to finish before it exits.

Finally, we must set the starting PC of the continuation thread correctly. Logically, if a future is split, the continuation thread should start at the point in the code *immediately after* the point at which the return value is stored. Note, though, that this is not the return address of the future call any longer (as is the case for LazyFutures). To provide this information to the JVM splitting mechanism, we insert a fake instruction after the return value store instruction which we pin throughout the compilation process. At the end of compilation we remove this instruction; but, we put its PC and the index of the associated local variable into a map which we store with the compiled code and query during future splitting.

By extending a JVM, our DBLFutures implementation avoids complicated source or bytecode rewriting or multiple code versions and yet easily enables migration from inlined to concurrent execution. In addition, our system is able to mix future calls with normal calls naturally since we have access to the Java operand stack and local method state. Non-JVM implementations cannot do this easily. For example, Cilk and JCilk [15, 101] do not allow non-Cilk method to call a Cilk method at all since a non-Cilk method is not compiled with parallel support (fast and slow clones) and is not migratable.

7.2 Experimental Methodology

We have implemented DBLFutures (as well as LazyFutures) in the popular, open-source Jikes Research Virtual Machine (JikesRVM) [84] (x86 version 2.4.6) from IBM Research. We have conducted our experiments on a dedicated 4-processor box (Intel Pentium 3(Xeon) xSeries 1.6GHz, 8GB RAM, Linux 2.6.9) with hyper-threading enabled. We report results for 1, 2, 4, and 8 processors – 8 enabled virtually via hyper-threading. We execute each experiment 10 times and present performance data for the best-performing.

For each set of experiments, we report results for two JVM configurations. The first uses a fast, non-optimizing compiler (BaseVM) and the second employs an adaptively optimizing compiler [8] (PAOptVM). With PAOptVM, we employ pseudo-adaptation (PA) [14], to reduce non-determinism in our experimentation. We include results for both JVM configurations to show the performance impact of DBLFutures for systems that dynamically produce very different code quality.

The benchmarks that we investigate are from the benchmark suite in the Satin system [148], including *AdapInt, FFT, Fib, Knapsack, QuickSort, Raytracer*. Each implements varying degrees of fine-grained parallelism. At one extreme is *Fib* which computes very little but creates a very large number of potentially concurrent methods. At the other extreme is *FFT* and *Raytracer* which implement few potentially concurrent methods, each with large computation granularity. We use this set of benchmarks to evaluate the performance impact of the directive-based programming model of our future implementation.

7.3 Performance Evaluation

Compared to the Java 5.0 Future model, our DBLFuture model provides programmers with two advantages: (1) programmers are free from the burden of scheduling futures; (2) programmers need not reorganize the serial program (e.g. wrapping computations in `Callable` objects, submitting to executors, etc.) to meet interface requirements.

Table 6.1 provides evidence that the task of scheduling futures by programmers is challenging, tedious, and typically requires expert knowledge of program and system behavior to achieve good performance since optimal scheduling decisions vary significantly across applications, inputs, available underlying hardware resources, and execution environments. Frequently, such information is not available to programmers statically.

	FFT	Raytracer	AdaptInt	Quicksort	Knapsack	Fib
J5Future	40	66	40	62	124	23
DBLFuture	27	32	27	38	85	11
Diff	13	24	13	24	39	12

Table 7.1: Source lines of code (SLOC) that is related to future implementation in both Java 5.0 Future version and DBLFuture version of each benchmark.

The metric we use to assess the ease with which our DBLFuture model can be used by programmers, is the number of *Source Lines Of Code* (SLOC), i.e., the number of *non-comment non-blank* lines in the source code of the program. Although this metric has its limitations, it is known to be a reliable predictor of programmer effort, and has been used in other research work [27]. Table 7.1 lists the SLOC for the code regions related to future implementation in both Java 5.0 Future version (the first row) and DBLFuture version (the second row) of each benchmark. The third row presents the difference between the two versions. The data shows that our DBLFuture model shortens the programs significantly, sometime even more than half (e.g. for Fib).

Another way in which DBLFutures eases programmer effort is that it enables concurrent versions of a program to be very similar to the equivalent serial version (semantically). Eliding the “@future” annotations, the DBLFuture version is the same as the serial version.

Bench-marks	Processor Numbers			
	1	2	4	8
FFT	1.11 x	1.13 x	1.12 x	1.03 x
Raytracer	1.01 x	1.02 x	1.01 x	1.00 x
AdapInt	2.90 x	2.97 x	3.02 x	4.61 x
QuickSort	5.53 x	5.29 x	5.22 x	5.65 x
Knapsack	1.57 x	1.58 x	1.64 x	1.64 x
Fib	42.55 x	44.63 x	46.67 x	51.09 x
Avg	9.11 x	9.44 x	9.78 x	10.84 x
Avg(w/o Fib)	2.42 x	2.40 x	2.40 x	2.79 x

(a) BaseVM

Bench-marks	Processor Numbers			
	1	2	4	8
FFT	1.08 x	1.12 x	1.01 x	1.00 x
Raytracer	1.01 x	1.01 x	1.00 x	1.01 x
AdapInt	1.23 x	1.18 x	1.26 x	1.47 x
QuickSort	1.87 x	2.10 x	2.27 x	2.72 x
Knapsack	1.31 x	1.57 x	1.76 x	1.86 x
Fib	4.46 x	6.64 x	12.42 x	18.17 x
Avg	1.83 x	2.27 x	3.29 x	4.37 x
Avg(w/o Fib)	1.30 x	1.40 x	1.46 x	1.61 x

(b) PAOptVM

Table 7.2: Speedup of DBLFutures over LazyFutures.

7.3.1 Directive-based versus Interface-based

We next compare the scalability of DBLFutures and LazyFutures. Table 7.2 shows the speedup of DBLFutures over LazyFutures for each benchmark, sorted by the rate of future generation. Columns 2-5 present results for increasing processor counts; Table (a) shows the results for BaseVM and (b) shows the results for

PAOptVM. DBLFutures enable significant performance gains over LazyFutures for all configurations and processor counts. On average, the DBLFuture implementation is 9.1 to 10.8 times faster than LazyFutures for all experiments for the BaseVM and 1.8 to 4.4 times faster for the PAOptVM case. Moreover, the performance gains increase with the number of futures (e.g. Fib versus Raytracer). Since Fib is an extreme case relative to the other benchmarks, we also show the average speedups across benchmarks not including Fib. This average is 2.4 to 2.8 times faster for the BaseVM and 1.3 to 1.6 times faster for the PAOptVM case.

The primary reason for the performance improvement is the programming model since these two future implementations share the same lazy and adaptive future scheduling system. For LazyFutures, the JVM has the flexibility to decide whether to inline or spawn a future, but must always create the `Callable` and `Future` object due to its interface-based model. The DBLFuture employs a function-call based model, which (1) avoids the creation of `Callable` objects completely; (2) grants the JVM the flexibility to create a `Future` object only when it decides to spawn a future based on underlying resource availability and dynamic program behavior. Our in depth analysis of the performance gains shows that the benefits that DBLFutures achieve is due primarily to the avoidance of memory allocation and management.

The improvements for PAOptVM are smaller than for BaseVM due to the efficient runtime services and dynamic code generation that PAOptVM performs (in-

Benchmarks	T_s/T_1	T_1/T_2	T_1/T_4	T_1/T_8
FFT	1.00 x	1.88 x	3.09 x	2.86 x
Raytracer	1.00 x	1.93 x	3.66 x	3.78 x
AdapInt	0.97 x	1.98 x	3.85 x	6.19 x
QuickSort	0.91 x	1.83 x	3.28 x	3.87 x
Knapsack	0.97 x	1.86 x	3.68 x	3.43 x
Fib	0.31 x	1.99 x	3.96 x	4.26 x
Avg	0.86 x	1.91 x	3.59 x	4.07 x
Avg(w/o Fib)	0.97 x	1.90 x	3.51 x	4.03 x

(a) BaseVM

Benchmarks	T_s/T_1	T_1/T_2	T_1/T_4	T_1/T_8
FFT	0.99 x	1.60 x	1.99 x	1.88 x
Raytracer	0.99 x	1.90 x	3.22 x	3.84 x
AdapInt	0.93 x	1.73 x	3.43 x	5.24 x
QuickSort	0.88 x	1.90 x	3.01 x	3.44 x
Knapsack	0.96 x	1.84 x	2.76 x	2.58 x
Fib	0.34 x	1.98 x	3.94 x	4.02 x
Avg	0.85 x	1.83 x	3.06 x	3.50 x
Avg(w/o Fib)	0.95 x	1.79 x	2.88 x	3.40 x

(b) PAOptVM

Table 7.3: Overhead and scalability of DBLFutures

cluding aggressive optimization of object allocation). In addition, the performance difference between BaseVM and PAOptVM speedups increase with the number of processors. This is because the more processors that are available, the more acute the competition for system resources and services. Thus, by eliminating most of the unnecessary object allocation, DBLFuture is able to reduce the conflicts in parallel memory management, which provides additional performance gains.

7.3.2 Overall Performance of DBLFutures

We then analyze the overhead and scalability of our DBLFuture system in Table 7.3. The table contains one section each for the BaseVM (a) and the PAOptVM (b) configurations. We use T_i to represent the execution time of programs written using DBLFuture with i processors, and T_s for the execution time of the corresponding serial version. Note that due to its function-call based coding style, this serial version is much faster than the serial version we used as the baseline for evaluation of Java 5.0 Futures and LazyFutures in Figure 6.7 and Figure 6.8. Therefore, we are setting a higher standard here to evaluate our DBLFuture system against.

Columns 2 shows the T_s/T_1 value, our overhead metric. Since there is only function call overhead for each potential future invocation in the serial version, the difference between T_1 (single processor) and T_s reveals three sources of overhead: (1) the bookkeeping employed to maintain the shadow future stack, (2) the activities of the future profiler, controller, and compiler, and (3) the conditional processing required by the DBLFuture version for the storing and first use of the value returned by a potential future call. The JVMs perform no splitting in either case. This data shows that our DBLFuture implementation is very efficient: only negligible overhead is introduced for most benchmarks. The worst case is *Fib*, which shows a 3x slowdown. This is because the Fib benchmark performs almost no computation for each future invocation (computing a Fibonacci value). The results for this benchmark represents an upper bound on the overhead of our system. The C implementation for a similar parallel system, called Cilk, introduce a similar overhead for this benchmark (3.63x slowdown [51]). Our system however, significantly outperforms the Java version of Cilk (JCilk) which imposes a 27.5x slowdown for this benchmark [38]).

The remaining columns for each JVM configuration show the speedups gained by DBLFuture when we introduce additional processors (which we compute as T_1/T_i as we increase i, the processor count). For the BaseVM case, the execution time on N processors scales almost to $1/N$ (average speedup is 1.91x, 3.59x, 4.07x for processor 2, 4, 8 respectively), that is, our system enables approximately linear speedup for most of the benchmarks that we investigate. Note that our hardware has 4 physical processors and uses hyperthreading to emulate 8 processors. Despite improvements in code quality enabled by the PAOptVM case, the DBLFuture version is able to extract average performance gains of 1.83x, 3.06x, 3.50x for 2, 4, and 8 processors, respectively. Again, we list the average data excluding Fib in the last row of the table to avoid Fib skewing the results. In summary, our DBLFuture implementation achieves scalable performance improvements with negligible overhead.

7.4 Related Work

There are many previous works that support parallel programming linguistically, either language-based, i.e., through the addition of new keywords in the language (e.g., Cilk [15], JCilk [101], X10 [27], Fortress [4]), or directive-based (e.g. OpenMP [115]). Many programming languages support the future construct to some extent, either via a library interface (e.g., Java [86], C++ [151]), or directly (e.g., Multilisp [126], C [21], X10 [27], Fortress [4]). Some concurrent logic programming languages (e.g., OZ [130]) generalize the concept of futures to rather extremes. In such languages, all logic variables are conceptually future variables: they can be bound by a separate thread and threads that access an unbound logic variable will be blocked until a value is bound to this variable. We follow the directive-based approach instead of language-based approach for easy implementation. The focus of our paper, however, is not the linguistic programming model itself, instead, we are interested in the performance impact of different future implementations for Java. We find that a linguistic approach provides the JVM and compiler with more flexibility to interpret future calls efficiently.

New extensions to the Java language can also be implemented by transforming the new constructs to calls to runtime libraries via either source-to-source transformation [38, 73] or bytecode rewriting [12, 90]. This approach has the advantage of portability and easy implementation since it does not require JVM modification. We show, in this work, however, that JVM support in a way that takes advantage of extant JVM services is important to achieve high performance and scalability. Also, our experiences show that by leveraging extant JVM design and implementations, and by eliminating extra abstraction layers, such JVM support to new language constructs can be feasible and sometime even simpler to implement comparing to higher-level alternatives.

7.5 Summary

In this chapter, we propose an improvement over our LazyFuture system by further liberating programmers from writing complicated future creation and management code in the Java programming language. We implement directive-based future programming support via a Java annotation. Our DBLFuture programming model enables programmers to identify potential parallelism opportunities in their programs using simple `@future` directives. DBLFutures make the migration from serial programming to parallel programming using futures much easier than does the conventional interface-based model. Based on our LazyFuture system, DBLFuture also eliminates unnecessary future object creations and provides better performance. We evaluate our DBLFuture empirically. The results show that our implementation enables significantly shorter programs, introduces negligible overhead, and is significantly more scalable than prior implementations.

The text of this chapter is in part a reprint of the material as it appears in the proceedings of the Sixteenth International Conference on Parallel Architecture and Compilation Techniques (PACT'07). The dissertation author was the primary researcher and author and the co-author listed on this publication ([161]) directed and supervised the research which forms the basis for Chapter 7.

Chapter 8

As-if-serial Exception Handling Support

An exception handling mechanism is a language control structure that allows programmers to specify the behavior of the program when an exceptional (unusual) event is caused by the program [36]. Exception handling is key for software fault tolerance and enables developers to produce reliable, robust software systems. Many languages support exception handling as an essential part of the language design, including CLU [104], Ada95 [78], C++ [139], Java [58], Eiffel [112], and many others.

As multi-processor computer systems become increasingly popular, many parallel programming languages or constructs (e.g. [27, 4, 86, 101]) have been proposed to enable programmers to express potential parallelism in programs easily so that the extra computation resources could be exploited. It is important to extend the exception handling mechanism to the concurrent context for fault tolerance and error recovery. However, exception handling semantics in a concurrent system are much more complex than for a serial environment. Their implementation requires careful design and must be implemented efficiently.

A key design goal of DBLFutures is to enable programmers to develop and reason about serial programs first and then introduce parallelism gradually and intuitively. We take this approach to simplify the process of parallel programming to improve programmer productivity so that more applications can take advantage of the current and next generation of systems with multiple processing cores. In a DBLFuture program, if we elide the future annotations, the program is in its serial form. As a result, programmers write their program as if it were serial and then identify code regions that can be safely executed in parallel and capture the return value from calls to these functions using an annotated local variable. Our goal with this chapter, thus, is to maintain these *as-if-serial* semantics and introduce a novel exception handling mechanism into DBLFutures.

In the following sections, we first review the exception handling mechanism for Java 5.0 Futures. We then present the design and implementation of our as-if-serial exception handling mechanism in the DBLFuture system. Finally, we evaluate the overhead of our approach and present related works.

8.1 Exception Handling in Java 5.0 Futures

A key feature of the Java programming language is its exception handling mechanism that enables robust and reliable program execution and control. Exception handling is also supported for the Java 5.0 Futures. Using the Java 5.0 Future APIs, the `get()` method of the `Future` interface can throw an exception with type `ExecutionException`. If an exception is thrown and not caught during the execution of the submitted future, the Executor intercepts the thrown exception, wraps the exception in an `ExecutionException` object, and saves it within the `Future` object. When the continuation queries the returned value of the submitted future via the `get()` method of the `Future` object, the method throws an exception with type `ExcecutionException`. The continuation can then inspect the actual exception using the `Throwable.getCause()` method. Note that the class `ExecutionException` is defined as a *checked exception* [58, Sec. 11.2] [86]. Therefore, the calls to `Future.get()` are required by the Java language specification to be enclosed by a a try-catch block (unless the caller throws this exception). Without this encapsulation, the compiler raises a compiler-time error at the point of the call. Figure 8.1 shows a simplified program for computing the Fibonacci number (Fib) using the Java 5.0 Future interfaces including the necessary try-catch block (line $10 \sim 14$).

```
public class Fib implements Callable<Integer>
{
  ExecutorService executor = ...;
  private int n;

  public Integer call() {
    if (n < 3) return n;
    Future<Integer> f = executor.submit(new Fib(n-1));
    int x = (new Fib(n-2)).call();
    try{
      return x + f.get();
    }catch (ExecutionException ex){
       ...
    }
  }
  ...
}
```

Figure 8.1: The Fibonacci program using Java 5.0 Futures with try-catch blocks

8.2 As-if-serial Exception Handling Design

One way to support exception handling for futures is to propagate exceptions to the *use point* of future return values, as is done in the Java 5.0 Future APIs. We can apply a similar approach to support exceptions in the DBLFuture system. For the future thread, in case of exceptions, instead of storing returned value into the `Future` object that the DBLFuture system creates during stack splitting, and then terminating, we can save the thrown and uncaught exception object in the `Future` object, and then terminate the thread. The continuation thread can then extract the saved the exception at the use points of the return value (the use of the annotated variable after the future call). That is, we can propagate exceptions from the future thread to the continuation thread via the `Future` object.

```java
public int f1() {
  @future int x;
  try{
    x = A();
  }catch (Exception e){
    x = default;
  }
  int y = B();
  return x + y;
}
```

(a)

```java
public int f1() {
  @future int x;
  x = A();
  int y = B();
  try {
    return x + y;
  }catch (Exception e){
    return default + y;
  }
}
```

(b)

Figure 8.2: Examples of two approaches to exception handling for DBLFutures

One problem with this approach is that it compromises one of the most important advantages of the DBLFuture model, i.e., that programmers code and reason about the logic and correctness of applications in the serial version first, and then introduce parallelism incrementally by adding future annotations. In particular, we are introducing inconsistencies with the serial semantics when we propagate exceptions to the use-point of the future return value. We believe that by violating the as-if-serial model, we make programming futures less intuitive.

For example, we can write a simple function `f1()` that returns the sum of return values of `A()` and `B()`. The invocation of `A()` may throw an exception, in which case, we use a default value for the function. In addition, `A()` and `B()` can execute concurrently. In Figure 8.2 (a), we show the corresponding serial version for this function, in which the try-catch clause wraps the point where the exception *may* be thrown. Using the aforementioned future exception-handling approach in which the exceptions are received at the point of the first use of the future return value,

```
public int f2() {
  @future int x;
  int w, y, z;
  try{
    w = A();
    x = B();     // a future function call
    y = C();
  }catch (Exception1 e){
    x = V1;
  }catch (Exception2 e){
    y = V2;
  }
  z = D();
  return w + x + y + z;
}
```

Figure 8.3: A simple DBLFuture program with exceptions

programmers must write the function as we show in Figure 8.2(b). In this case, the try-catch clause wraps the use point of return value of the future. If we elide the future annotation from this program (which produces a correct serial version using DBLFutures without exception handling support), the resulting version is not a correct serial version of the program due to the exception handling.

To address this limitation, we propose *as-if-serial* exception semantics for DBLFutures. That is, we propose to implement exception handling in the same way as is done for serial Java programs. In particular, we deliver any uncaught exception thrown by a future function call to its caller at the invocation point of the future call. Moreover, we continue program execution as if the future call has never executed in parallel to its continuation.

We use the example in Figure 8.3 to illustrate our approach. We assume that the computation granularity of `B()` is large enough to warrant its parallel execution with its continuation. There are a number of ways in which execution can progress:

case 1: `A()`, `B()`, `C()`, and `D()` all finish normally, and the return value of `f2()` is `A()+B()+C()+D()`.

case 2: `A()` and `D()` finish normally, but the execution of `B()` throws an exception of type `Exception1`. In this case, we propagate the uncaught exception to the invocation point of `B()` in `f2()` at line 6, and the execution continues in `f2()` as if `B()` is invoked locally, i.e., the effect of line 5 is preserved, the control is handed to the exception handler at line 8, and the execution of line 7 is ignored regardless whether `C()` finishes normally or abruptly. Finally the execution is resumed at line 13. The return value of `f2()` is `A()+V1+0+D()`.

case 3: `A()`, `B()`, and `D()` all finish normally, but the execution of `C()` throws an exception in type `Exception2`. In this case, the uncaught exception of `C()` will not be delivered to `f2()` until `B()` finishes its execution and the system stores its return value in `x`. Following this, the system hands control to the exception handler at line 10. Finally, the system resumes execution at line 13. The return value of `f2()` is `A()+B()+V2+D()`.

Note that in this chapter, we focus on the as-if-serial exception semantics in terms of the control flow of exception delivering, i.e., which and where exceptions should be handled. The complete as-if-serial exception handling semantics requires that the global side effects of parallel execution of a DBLFuture program is consistent with that of the serial execution. For example, in case 2 of the above example, any global side effects of `C()` must also be undone to restore the state to be the same as if `C()` is never executed (since semantically `C()'s` execution is ignored due to the exception thrown by `B()`). However, this side effect problem is orthogonal to the control problem of exception delivering that we address in this chapter. We will address the problem of preserving as-if-serial side-effect semantics and describe its integration with the as-if-serial exception handling semantics in the next chapter.

8.3 Implementation

To implement exception handling for DBLFutures, we extend the DBLFuture-aware Java Virtual Machine implementation described in Chapter 7. In this section, we detail this implementation.

8.3.1 Total Ordering of Threads

To enable as-if-serial exception handling semantics, we must track and maintain a total order on thread termination across threads that originate from the same context and execute concurrently. We define this total order as the order in which the threads would terminate if the program was executed serially. We detail how we make use of this ordering in Section 8.3.3.

To maintain this total order during execution, we add two new references, called `futurePrev` and `futureNext`, to the virtual machine thread representation with which we link related threads in an acyclic, doubly linked list. We establish thread order at future splitting points, since future-related threads are only generated at these points. Upon a split event, we set the future thread as the predecessor of the newly created, continuation, thread since this is how the the threads are executed in the serial execution. If the future thread already has a successor, we add the new continuation thread between the future thread and its successor in the linked list.

Figure 8.4 gives an example of this process. Stacks in this figure grow upwards. Originally, thread T1 is executing `f()`. The future function call `A()` is initially executed on the T1's stack according to the lazy spawning principle of our system. Later, the system decides to split T1's stack and spawns a new thread T2 to execute `A()`'s continuation in parallel to `A()`. At this point, we link T1 and T2 together. Then, after T2 executes the second future function call, `B()`, long enough to trigger splitting, the system again decides to split the execution. At this point, the system creates thread T3 to execute `B()`'s continuation, and links T3 to T2 (as T2's successor).

An interesting case is if there is a future function call in `A()` (`D()` in our example) that has a computation granularity that is large enough to trigger splitting again. In

```
public int f() {
  @future int x, y;
  int z;
  try{
    x = A(); //split point 1
    y = B(); //split point 2
  }catch(Exception1 e){
    ...
  }
  z = C();
  return x + y + z;
}
```

```
public int A() throws Exception1{
  @future int u;
  int v;
  u = D(); //split point 3
  v = E();
  return u + v;
}
```

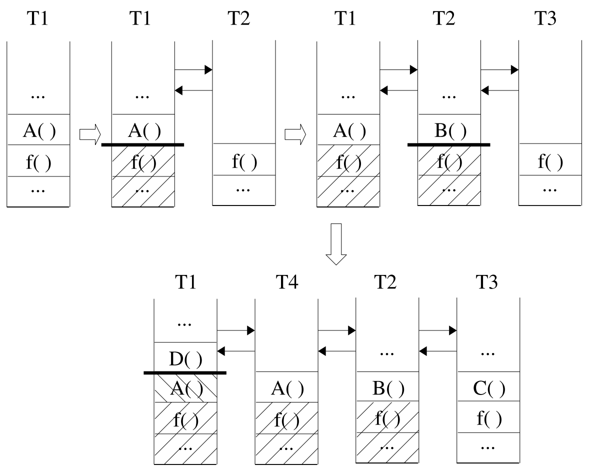


Figure 8.4: Example of establishing total ordering of threads.

this case, T1’s stack is split again, the system creates a new thread, T4, to execute `D()`’s continuation. Note that we must update T2’s predecessor to be T4 since, if executed sequentially, the rest of `A()` after the invocation point of `D()` is executed before `B()`.

The black lines in the figure denote the split points on the stack for each step. The shadowed area of the stack denotes the stack frames that are copied to the continuation thread. These frames are not reachable by the original future thread once the split occurs since the future thread terminates once it completes the future function call and saves the return value.

8.3.2 Choosing a Thread to Handle the Exception

One important implementation design decision is the choice of thread context in which we should handle the exception. For example, in Figure 8.4, if `A()` throws an exception with type `Exception1` after the first split event, we have the choice of handling the exception in T1 or T2.

Intuitively, we should choose T2 as the handling thread since it seems from the source code that after splitting, everything after the invocation point of `A()` is handed to T2 for execution, including the exception handler. T1 only has context up to the return point of `A()`, when it will store the future value and then terminate itself.

The problem is that the exception delivery mechanism in our JVM is synchronous, i.e., whenever an exception is thrown, the system searches for a handler on the current thread's stack based on the PC (program counter) of the throwing point. T2 does not have the throwing context, and will only synchronize with T1 when it uses the value of x. Thus, we must communicate the throwing context on T1 to T2 and inform T2 to pause its current execution at some point to execute the handler. This asynchronous exception delivering mechanism can be very complex to implement.

Fortunately, since our system operates on the Java stack directly and always executes the future function call on the current thread's stack, and spawns the continuation, we have a much simpler implementation option. Note that the shadowed area on T1's stack after the first split event is logically not reachable by T1. Physically, however, these frames are still on T1's stack. As a result, we can simply *undo* the splitting as if the splitting never happened via clearing the split flag of the first shadowed stack frame (the caller of A() before splitting), which makes the stack reachable by T1 again. Then, the exception can be handled on T1's context normally using the existing synchronous exception delivering mechanism of the JVM.

This observation significantly simplifies our implementation. Now, T2 and all threads that originate from T2 can be aborted as if they were never generated. If some of these threads have thrown an exception that is not caught within its own context, the thrown exception can also be ignored.

8.3.3 Enforcing Total Order on Thread Termination

In section 8.3.1, we discuss the way to establish a total order across related future threads. In this section, we describe how we use this ordering to preserve as-if-serial exception semantics for DBLFutures. Note that these related threads can execute concurrently, we simply require that their termination (commit) be ordered.

```
void futureStore(T value) {
  if (currentThread.futurePrev != null) {
    while (currentThread.commitStatus == UNNOTIFIED){
      wait;
    }
  } else {
    currentThread.commitStatus = READY;
  }
  Future f = getFutureObject();
  if (currentThread.commitStatus == ABORTED){
    currentThread.futureNext.commitStatus = ABORTED;
    f.notifyAbort();
    cleanup and terminate currentThread;
  } else {
    currentThread.futureNext.commitStatus = READY;
    f.setValue(value);
    f.notifyReady();
    terminate currentThread;
  }
}
```

Figure 8.5: Algorithm for the future value storing point

First, we add a field, called `commitStatus`, to the internal thread representation of the virtual machine. This field has three possible values: `UNNOTIFIED`, `READY`, `ABORTED`. `UNNOTIFIED` is the default and initial value of this field. A thread checks its `commitStatus` at three points: (i) the future return value store point, (ii) the first future return value use point, and (iii) the exception delivery point.

Figure 8.5 shows the pseudocode of the algorithm that we use at the future return value store point. The pre-condition of this function is that the continuation of the current future function call is spawned on another thread, and thus, a `Future` object is already created as the placeholder that both the future and continuation thread have access to.

```
T futureLoad() {
  Future f = getFutureObject();
  while (!f.isReady() && !currentThread.commitStatus == ABORTED){
    wait;
  }
  if (currentThread.commitStatus == ABORTED){
    if (currentThread.futureNext != null) {
      currentThread.futureNext.commitStatus = ABORTED;
    }
    cleanup and terminate currentThread;
  } else {
    return f.getValue();
  }
}
```

Figure 8.6: Algorithm for the future return value use point

This function is invoked by a future thread after it finishes the future function call normally, i.e., without any exceptions. First, if the current thread has a predecessor, it waits until its predecessor finishes either normally or abruptly, at which point, the commitStatus of the current thread is changed from UNNOTIFIED to either READY or ABORTED by its predecessor. If the commitStatus is ABORTED, the current thread notifies its successor to abort. In addition, the current thread notifies the thread that is waiting for the future value to abort. The current thread then performs any necessary cleanup and terminates itself. Note that a split future thread always has a successor. If the commitStatus of the current thread is set to READY, it stores the future value in the `Future` object, and wakes up any thread waiting for the value (which may or may not be its immediate successor), and then terminates itself.

The algorithm for the future return value use point (Figure 8.6) is similar. This function is invoked by a thread when it attempts to use the return value of a future function call that is executed in parallel. The current thread will wait until either the future value is ready or it is informed by the system to abort. In the former case, this function simply returns the available future value. In the latter case, the current thread first informs its successor (if there is any) to abort also, and then cleans up and terminates itself.

The algorithm for the exception delivering point is somewhat more complicated. Figure 8.7 shows the pseudocode of the existing exception delivering process in our JVM augmented with our support to as-if-serial semantics. We omit some unrelated details for clarity. The function is a large loop that searches for an appropriate handler block on each stack frame, from the newest (most recent) to the oldest. If no handler is found on the current frame, the stack is unwound by one frame. Finally, if the function finds no handler on the entire stack, it reports the exception to the system, and terminates the current thread.

To support as-if-serial exception semantics, we make two modifications to this process. First, at the beginning of each iteration (line $3 \sim 13$ in Figure 8.7), the current thread checks whether the current stack frame is for a spawned continuation that has a split future. If so, it checks whether the current thread has already been aborted by its predecessor. In this case, instead of delivering the exception, it notifies its successor (if there is any) to abort, cleans up, and then terminates itself. Note that the system only does this checking for a spawned continuation frame. If a handler

```
void deliverException(Exception e) {
  while (there are more frames on stack){
    if (the current frame has a split future) {
      while (currentThread.commitStatus == UNNOTIFIED){
        wait;
      }
      if (currentThread.commitStatus == ABORTED){
        if (currentThread.futureNext != null) {
          currentThread.futureNext.commitStatus = ABORTED;
        }
        cleanup and terminate currentThread;
      }
    }
    search for a handler for e in the compiled method
    on the current stack;
    if (found a handler) {
      jump to the handler and resume execution there;
      // not reachable
    }
    if (the current frame is for a future function call
          && its continuation has been spawned) {
      if (currentThread.futurePrev != null) {
        while (currentThread.commitStatus == UNNOTIFIED){
          wait;
        }
      } else {
        currentThread.commitStatus = READY;
      }
      currentThread.futureNext.commitStatus = ABORTED;
      Future f = getFutureObject();
      f.notifyAbort();
      if (currentThread.commitStatus == ABORTED){
        cleanup and terminate currentThread;
      }else{
        reset the caller frame to non-split status;
      }
    }
    unwind the stack frame;
  }
  // No appropriate catch block found
  report the exception and terminate;
}
```

Figure 8.7: Algorithm for the exception delivering point

is found before reaching such a spawned continuation frame, the exception will be delivered as usual since in that case, the exception is within the current thread's local context.

The second modification is prior stack unwinding (line $20 \sim 37$ in Figure 8.7). The current thread checks if the current frame belongs to a future function call that has a spawned continuation. In this case, we must rollback the splitting decision, and reset the caller frame of the current frame to be the next frame on the local stack. This enables the system to handle the exception on the current thread's context (where the exception is thrown) as if no splitting occurred. In addition, the thread notifies its successor and any thread that is waiting for the future value to abort since the future call finishes with an exception. The thread must still needs wait for the committing notification from its predecessor (if there is any). In case for which it is aborted, it cleans up and terminates, otherwise, it reverses splitting decision and unwinds the stack.

Note that our algorithm only enforces the total termination order when a thread finishes its computation and is about to terminate, or when a thread attempts to use a value that is asynchronously computed by another thread, at which point it will be blocked anyway if the value is not ready yet. Therefore, our algorithm does not prevent threads from executing in parallel in any order, and thus, does not sacrifice the parallelism in programs.

8.4 Performance Evaluation

Although the as-if-serial exception handling semantics is very attractive for programmer productivity since it significantly simplifies the task of writing and reasoning about DBLFuture programs with exceptions, it is important that it does not introduce significant overhead. In particular, it should not slow down applications for programs that throw no exceptions. If it does so, it compromises the original intention of the DBLFuture programming model which is to introduce parallelism easily, and to achieve better performance when there are available computational resources. In this section, we provide an empirical performance evaluation of our implementation to evaluate its overhead.

Our implementation is based on the previous DBLFuture system that is an extension to the popular, open-source Jikes Research Virtual Machine (JikesRVM) [84] (x86 version 2.4.6) from IBM Research. The test machine we use is a 4-processor box (Intel Pentium 3(Xeon) xSeries 1.6GHz, 8GB RAM, Linux 2.6.9). We only report data for the adaptively optimizing JVM configuration compiler [8] (with pseudo-adaptation (PA) [14] to reduce non-determinism) since results for the non-optimizing compiler are similar.

The benchmarks that we investigate are from the benchmark suite in the Satin system [148]. Each implements varying degrees of fine-grained parallelism. At one extreme is *Fib* which computes very little but creates a very large number of potentially concurrent methods. At the other extreme is *FFT* and *Raytracer* which implement few potentially concurrent methods, each with large computation granularity. Moreover, no future threads in these benchmarks finished exceptionally. We execute each experiment 20 times and present the average performance data in Table 8.1.

Table 8.1 has three subtable, each for results with 1, 2, and 4 processors, respectively. The second column of each subtable is the mean execution time (in seconds) for each benchmark in the DBLFuture system without exception handling support (denoted as *Base* in the table). We show the standard deviation across runs in the parentheses. The third column is the mean execution time (in seconds) and standard deviation (in parentheses) in the DBLFuture system with the as-if-serial exception handling support (denoted as *EH* in the table). The fourth column is the percent degradation (or improvement) of the DBLFuture system with exception handling support.

To ensure that these results are statistically meaningful, we conduct the independent t-test [52] on each set of data, and present the corresponding t values in the last column of each subtable. For experiments with sample size 20, the t value must larger than 2.093 or smaller than -2.093 to make the difference between Base and EH statistically significant with 95% confidence. We highlight those overhead numbers that are statistically significant in the table.

Benchs	Base	EH	Diff	T
AdapInt	29.36 (0.09)	27.96 (0.18)	-4.8%	-31.79
FFT	7.89 (0.03)	7.78 (0.03)	-1.5%	-11.49
Fib	16.47 (0.13)	17.04 (0.06)	3.5%	17.81
Knapsack	11.27 (0.04)	10.79 (0.03)	-4.3%	-41.78
QuickSort	8.11 (0.04)	8.01 (0.03)	-1.3%	-9.20
Raytracer	21.22 (0.09)	20.91 (0.07)	-1.4%	-12.12

(a) With 1 processor

Benchs	Base	EH	Diff	T
AdapInt	15.02 (0.25)	15.40 (0.81)	2.5%	1.97
FFT	4.92 (0.08)	5.03 (0.10)	2.2%	3.78
Fib	8.34 (0.09)	8.48 (0.06)	1.7%	5.94
Knapsack	6.36 (0.16)	6.35 (0.14)	-0.2%	-0.22
QuickSort	4.31 (0.08)	4.28 (0.04)	-0.5%	-1.07
Raytracer	11.18 (0.10)	11.28 (0.14)	0.9%	2.56

(b) With 2 processors

Benchs	Base	EH	Diff	T
AdapInt	8.47 (1.01)	8.67 (1.35)	2.4%	0.53
FFT	4.24 (0.09)	4.18 (0.10)	-1.6%	-2.33
Fib	4.26 (0.02)	4.33 (0.04)	1.6%	6.47
Knapsack	4.40 (0.19)	4.40 (0.15)	0.1%	0.07
QuickSort	2.52 (0.03)	2.54 (0.03)	0.9%	2.34
Raytracer	6.26 (0.07)	6.33 (0.07)	1.1%	3.27

(c) With 4 processors

Table 8.1: Overhead and scalability of the as-if-serial exception handling for DBL-Futures. The *Base* and *EH* column list the mean execution time (in seconds) and standard deviation (in parentheses) in the DBLFuture system without and with the as-if-serial exception handling support. The *Diff* column is the difference between *Base* and *EH* (in percent). The last column is the T statistic computed using data in the first three columns. Those difference numbers that are statistically significant with 95% confidence are highlighted.

This table shows that our implementation of the as-if-serial exception handling support for DBLFutures introduces only negligible overhead for some benchmarks. The maximum percent degradation is 3.5%, which occurs for `Fib` when one processor is used. Most of the overhead numbers are less than 2%.

These results may seem counter-intuitive since we enforce a total termination order across threads to support the as-if-serial exception semantics. However, our algorithm only does so (via synchronization of threads) at points at which a thread either operates on a future value (stores or uses) or delivers an exception. Thus, our algorithm delays termination of the thread, but does not prevent it executing its computation in parallel to other threads. For a thread that attempts to use a future value, if the value is not ready, this thread will be blocked anyway. Therefore, our requirement that threads check for an aborted flag comes for free.

Moreover, half of the performance results show that our EH extensions actually improve performance (all negative numbers). This phenomenon is common in the 1-processor case especially. It is difficult for us to pinpoint the reasons for the improved performance phenomenon due to the complexity of JVMs and the non-determinism inherent in multi-threaded applications. We suspect that our system slows down thread creation to track total ordering and by doing so, it reduces both thread switching frequency and the resource contention to improve performance.

In terms of scalability, our results do not show a relative increase in overhead when we introduce more processors. Although we only experiment with up to 4 processors, given the nature of our implementation, we believe that the overhead will continue to be low given additional processors.

In summary, our system guarantees the as-if-serial exception handling semantics for future-based applications that throw exceptions. Moreover, our implementation of these semantics introduce little overhead for applications without exceptions.

8.5 Related Work

Many early languages that support futures (e.g. [126, 21]) do not provide concurrent exception handling mechanisms among the tasks involved. This is because these languages do not have built-in exception handling mechanisms, even for the serial case. This is also the case for many other parallel languages that originate from serial languages without exception handling support, such as Fortran 90 [45], Split-C [97], Cilk [15], etc.

For concurrent programming languages that do support exception handling, most of them focus on the exception handling mechanism within thread boundaries, but have none or limited support for concurrent exception handling. For example, for normal Java [58] threads, exceptions that are not handled locally by a thread will not be automatically propagated to other threads, instead, they are silently dropped ”on-the-floor”. The C++ extension Mentat [59] does not address the exception handling problem at all. In OpenMP [115], a thrown exception inside a parallel region must be caught by the same thread that threw the exception and the execution must be resumed within the same parallel region.

Most of more recent languages that adopt futures (e.g. [86, 27, 4]) do provide concurrent exception handling for futures to some extent. For example, in Java, while future values are queried via invoking `Future.get()`, an `ExecutionException` is thrown to the caller if the future computation terminates abruptly[86]. Similar exception propagation strategy is used by the Java Fork/Join Framework [100], which supports the divide-and-conquer parallel programming style in Java. In Fortress [4], the `spawn` statement is conceptually a future construct. The parent thread queries the value returned by the spawned thread via invoking its `val()` method. When a spawned thread completes exceptionally, the exception is deferred. Any invocation of `val()` then throws the deferred exception. This is similar to the Java 5.0 Future model.

X10 [27] proposes a *rooted exception* model, that is, if activity A is the `root-of` activity B and A is suspended at a statement awaiting the termination of B, exceptions thrown in B are propagated to A at that statement while B terminates. Currently, only the `finish` statement marks code regions as a root activity. We expect that future versions of the language may soon introduce more such statements, including the `force()` method, which extracts the value of a future computation.

The primary difference between our as-if-serial exception handling model for futures and the above approaches is the point at which exceptions are propagated. In these languages, exceptions raised in the future computation that cannot be handled locally are propagated to the thread that spawns the computation when it attempts to synchronize with the spawned thread, such as using the returned value. While in our model, asynchronous exceptions are propagated to the invocation point of the future function call as if the call is executed locally. In this sense, the exception handling mechanism for the Java Remote Method Invocation model [82] is closer to our approach since the exception context where remote execution exceptions are propagated back to the caller thread is the invocation point of the remote method. However, an RMI is usually blocking while a future call is asynchronous.

JCilk [101, 38] is the one most related to our work. JCilk is a Java-based multithreaded language that enables a ”Cilk-like” parallel programming model in Java. It strives to provide a faithful extension of the semantics of Java’s serial exception mechanism, that is, if we elide JCilk primitives from a JCilk program, the result program is a working serial Java program. In JCilk, an exception thrown and uncaught in a spawned thread is propagated to the invocation context in the parent thread, which is same as our model.

However, there are several major differences between these two. First, JCilk does not enforce ordering among spawned threads before the same `sync` statement. If multiple spawned threads throw exceptions simultaneously, the runtime randomly picks one to handle, and aborts all other threads in the same context. In our model, even when there are several futures spawned in the same try-catch context, there is always a total ordering among them, and our system selects and handles exceptions in their serial order. In this sense, JCilk does not maintain serial semantics to the same degree as our model does. Secondly, JCilk requires a `spawn` statement surrounded by a special `cilk try` if exceptions are possible. In our DBLFuture model, normal Java `try` clause is sufficient. Finally, since JCilk is implemented at library level, it requires very complicated source level transformation, code generation, and runtime data structures to support concurrent exception correctly (e.g., `catchlet`, `finallet`, `try tree`, etc.), whereas our implementation is much simpler thanks to the direct access to Java call stacks and the stack splitting technique.

There are only a few concurrent object-oriented languages that have built-in concurrent exception handling support, e.g., DOOCE [76], Arche [80, 79], etc. DOOCE addresses the problem of handling multiple exceptions thrown concurrently in the same `try` block by extending the `catch` statement to take multiple parameters. Also, multiple `catch` blocks are allowed to associated with one `try` block. In case of exceptions, all `catch` blocks that match thrown exceptions, individually or partially, will be executed. In addition, DOOCE supports two kinds of model for the timing of acceptance and the action of exception handling: (1) waiting for all subtasks to complete, either normally or abruptly, before starting handling exceptions (using the normal `try` clause); (2) if any of the participated objects throws an exception, the exception is propagated to other objects immediately via a *notification message* (using the `try_noti` clause). In addition to the common termination model ([127], i.e., execution is resumed after the `try-catch` clause), DOOCE supports resumption via the `resume` or `retry` statement in the `catch` block, which resumes execution at the exception throwing point or the start of the `try` block.

Arche proposes a cooperation model for exception handling. In this model, there are two kinds of exceptions: *global* and *concerted*. If a process terminates exceptionally, it signals a global exception, which is propagated to other processes that communicate synchronously with it. For multiple concurrent exceptions, Arche allows programmers to define a customized *resolution function* that takes all exceptions as input parameters and returns a *concerted* exception that can be handled in the context of the calling object.

Other prior works (e.g. [123, 106, 22, 127, 158]) have focused on general models for exception handling in distributed systems. These models usually assume that processes participating in a parallel computation are organized coordinately in a structure, such as a *conversation* [123] or an *atomic action* [106]. Processes can enter such a structure asynchronously, but have to exit the structure synchronously. In case that one process throws an exception, all other processes will be informed and an appropriate handler is invoked for all participants. With regards to the problem of handling concurrently signaled exceptions, a technique, called *exception resolution* [22] is used. Multiple exceptions are resolved to a single one based on different resolution strategies, such as the exception resolution tree [22], the exception resolution graph [157], or user defined resolution functions [80].

Our exception handling mechanism for DBLFutures is different from other work in concurrent exception handling in that the intention of preserving serial semantics grants our model special properties that simplify the implementation significantly. For example, the exception resolution strategy of our model is very simple: pick the one that should occur first in the serial semantics. Also, although our model organizes involved threads in a structured way (a double linked list), one thread does not need to synchronize with all other threads in the group before exiting like the way conversation and atomic action work. Instead, threads in our system only communicate with their predecessors and successors, and exit according to a total order defined by the serial semantics of the program.

Safe Java futures are described in [153]. Their system uses object versioning and task revocation to enforce the semantic transparency of futures automatically so that programmers are freed from reasoning about the side-effects of future executions and ensuring correctness. This transaction style support is complementary to our as-if-serial exception handling model, and we plan to integrate it into our system as part of future work. Note that the authors of this work do mention that an uncaught exception thrown by the future call will be delivered to the caller at the point of invocation of the `run` method, which is similar to our as-if-serial model. However it is unclear as to how (or if) they implemented this since the authors provide no details on their design and implementation.

8.6 Summary

In this chapter, we propose an *as-if-serial* exception handling mechanism for the DBLFutures. The goal is to identify a design that is both compatible with the original language design and that preserves our as-if-serial program implementation methodology. Our as-if-serial exception handling mechanism delivers exceptions at the same point as they are delivered if the program is executed sequentially. In particular, an exception thrown and uncaught by a future thread will be delivered to the invocation point of the future call. In contrast, in the Java 5.0 implementation of futures exceptions of future execution are propagated to the point in the program at which future values are queried (used).

We show that the as-if-serial exception handling mechanism integrates easily into the DBLFuture system and preserves serial semantics so that programmers can intuitively understand the exception handling behavior and control in their parallel Java programs. With DBLFutures and as-if-serial exception handling, programmers can focus on the logic and correctness of a program in the serial version, including its exceptional behavior, and then introduce parallelism gradually and intuitively. We present the design and implementation of our exception handling mechanisms based on the DBLFuture framework in the Jikes Research Virtual Machine. Our results show that our implementation introduces negligible overhead for applications without exceptions, and guarantees serial semantics of exception handling for applications that throw exceptions.

The text of this chapter is in part a reprint of the material as it appears in the proceedings of the fifth international symposium on Principles and practice of programming in Java (PPPJ'07). The dissertation author was the primary researcher and author and the co-author listed on this publication ([162]) directed and supervised the research which forms the basis for Chapter 8.

Chapter 9

As-if-serial Side-effect Guarantee

The goal of our directive-based lazy futures (DBLFutures) with as-if-serial exception handling support is to enable programmers to write and reason about the logic and correctness of programs in a serial version first, and then to introduce potential parallelism gradually and intuitively. To do so, users specify asynchronous computations that can be executed safely in parallel using the “@future” annotation. This model simplifies parallel programming since programmers write in a way that is intuitive to them, i.e., according to serial semantics. In addition, this model facilitates migration of legacy serial programs to concurrent programs.

However, in this model, programmers still must reason about whether it is safe to execute the future and its continuation in parallel. The programmer must provide protection for shared data as necessary to avoid data races, which can require significant programmer effort. To simplify this process, we relieve this burden from programmers via support of *as-if-serial* side-effect semantics. With this semantics, regardless of how the program is executed, sequentially or in parallel, the virtual machine guarantees that side-effects occur in the same way. Note that the as-if-serial side-effect semantics is stronger than the extant serializability semantics that many data race prevention techniques attempt to achieve, such as lock-based synchronization or transactional memory techniques: in addition to serializability, as-if-serial semantics enforces *the order* of side-effects according to its serial semantics. Although this may seem too strong for some cases (and may limit scalability and concurrency), it provides significant programmer productivity benefit: the concurrent version is guaranteed to be correct once the programmer completes a working serial version, without requiring that the programmer debug a concurrent version.

In this chapter, we will first evaluate the prior work on this subject. We then investigate ways to exploit the adaptation of the JVM to guarantee correct concurrent execution in DBLFutures.

9.1 Background: the Safe Future System

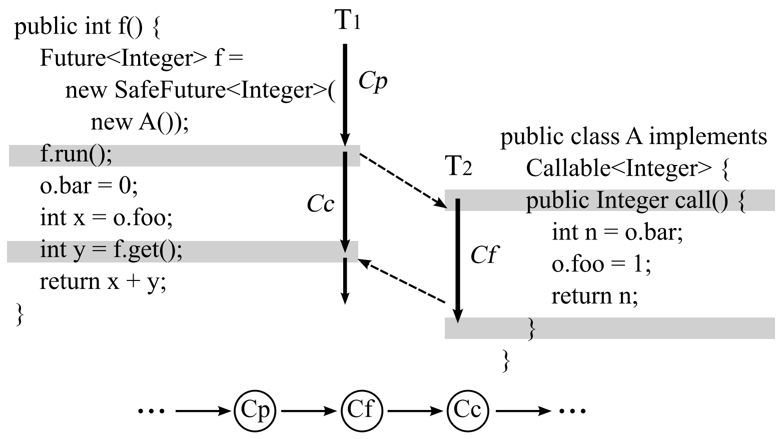
As-if-serial side-effect semantics for futures has been investigated in the Safe Future project [153]. In this section, we overview the programming model and implementation of the Safe Future system, and discuss the limitations of the implementation on the support of easy-to-use and efficient parallel programming using futures in Java.

9.1.1 Programming Model

The programming model of Safe Futures is similar to that of Java 5 Future APIs. The system provides a `SafeFuture` class that implements the Java 5 `Future` interface. To spawn a computation as a future, programmers first wrap the computation in a class that implements the `Callable` interface. At the spawning point, programmers create a `SafeFuture` object that takes the `Callable` object as a parameter, and then call the `frun()` method of the `SafeFuture` object. Upon the invocation of the `frun()` method, the system spawns a new thread to evaluate the computation enclosed by the `SafeFuture` object. At the same time, the current thread immediately continues to execute the code right after the call site of the `frun()` method (i.e., the continuation), until it attempts to use the value computed by the future, when the `get()` method is invoked. The current thread is blocked until the value is ready. Figure 9.1 shows a simple example that uses the `SafeFuture` API.

9.1.2 Execution Contexts

To preserve the as-if-serial side-effect semantics, the Safe Future system divides the entire program execution into a sequence of *execution contexts*. Each context



```
public int f() {
   Future<Integer> f =
      new SafeFuture<Integer>(
         new A());
   f.run();
   o.bar = 0;
   int x = o.foo;
   int y = f.get();
   return x + y;
}
```

```
public class A implements
   Callable<Integer> {
   public Integer call() {
      int n = o.bar;
      o.foo = 1;
      return n;
   }
}
```

Figure 9.1: Example of execution context creation for Safe Futures in Java

encapsulates a fragment of computation that is executed by a single thread. These execution contexts are totally ordered based on the logical serial execution order, which the system implements via a linked list of contexts.

The program execution starts with a primordial context. Upon a future invocation, the system pauses the current context. The system creates a new thread and a future context as well as a new continuation context. The system assigns the current thread to the continuation context.

In the linked list, the current context is the predecessor of the future context and the future context is the predecessor of the continuation context. The future context ends once it returns from the future computation. The continuation context ends at the invocation point of the `get()` method which retrieves the result of the future computation. This process is depicted in Figure 9.1 for a simple example, where C_p,

C_f, and C_c represent the primordial context, the future context, and the continuation context, respectively. The grayed boxes indicate the start and end of each context. The system resumes the primordial context, C_p, once the future and continuation contexts complete successfully.

9.1.3 Preserving As-if-serial Side-effect Semantics

The Safe Future system defines two types of data dependency violations:

- Forward dependency violation: C_c does not observe the effect of an operation performed by C_f;
- Backward dependency violation: C_f does observe the effect of an operation performed by C_c

For the example in Figure 9.1, if T_1 read `o.foo` before T_2 writes to it, it has the forward dependency violation. Alternatively, if T_1 writes to `o.bar` before T_2 reads it, it has the backward dependency violation. If there is no violation, the program execution is defined as safe, i.e., the as-if-serial side-effect semantics is preserved.

Every read or write to the shared data is guarded by a compiler-inserted barrier, which tracks shared data accesses by each context. The barriers prevent dependency violations to preserve as-if-serial semantics.

To prevent a backward dependency violation, each context keeps a private copy of all shared data that it has written to. Also, each item of shared data maintains a list of all private copies created for it, which is sorted under logical context order. Upon a write, the execution context creates a private version of the shared data, and put the new copy tagged with the context ID (via an extra word in the object header) into the version list. It also replaces any reference to the data on stack with the new version so that all subsequent reads get the correct version. Upon a read, the context searches the version that tagged by itself or the version created by its most recent predecessor, i.e., a context will never see a version that is created by its logical future contexts, which prevents backward dependency violation.

To prevent the forward dependency violation, the system maintains two bit-maps for each execution context to record reads and writes to shared data of the associated computation fragment. Upon committing, the system detects conflicts by checking the read bit-map of the execution context against the write bit-map of all the execution contexts in its logical past. If there is any overlap, a conflict is detected, and the context is revoked, i.e., all of its side-effects are discarded, and its associated computation is re-executed.

9.1.4 Committing and Revoking Execution Contexts

There are three outcomes of a context commit: success, failed, and aborted. Success means that the side-effects are safe (i.e., they preserve as-if-serial semantics) to make permanent and thus, seen by other contexts. Failed means that all contexts in the logical past of this context have successfully committed, but the current context has conflicts with at least one of its predecessors and must be revoked. Finally, aborted means some context in the logical past of this context has been revoked, and the current context should be discarded without re-execution, since the current context will be re-executed within a new context via re-execution of the revoked context. Any revocation of a predecessor context results in abortion of all contexts thereafter.

Different kinds of execution contexts have different committing triggers and revocation algorithms. For the future context, the system attempts to commit its side-effects at the end of the future computation. The commit of a future context triggers the commit of its primordial context, which recursively triggers the commit of all contexts in the logical past. The commit of the continuation context is triggered by calling the `get()` method. The continuation context first waits for its corresponding future context to finish. If the future context aborts, the continuation also aborts. Otherwise, if there is a conflict detected, the continuation context is revoked.

Since the computation of a future context is wrapped in a `Callable` object, its revocation implementation is straightforward: the computation is enclosed by a loop with a successful commit as the exit condition. The revocation of a continuation context is more complex. The Safe Future system uses bytecode rewriting to insert code at the beginning of a continuation, which saves the state of all local variables and stack locations at that point, into the future object. The system also inserts extra bytecode to restore the saved states from the future object, and records the start of the code segment as the point of revocation for the continuation context. The bytecode rewriter then generates new exception handling code, which handles the internal *re-voke* exceptions that are thrown by the system when a continuation context is revoked. The exception handler extracts the starting point code segment from the future object encapsulated in the exception, and jumps to that point to begin re-execution.

9.1.5 Limitations of Safe Futures

The Safe Future system is an interface-based approach that is similar to Java 5 Futures. As we have shown in previous chapters, this approach has programmer productivity and performance disadvantages that we avoid with our directive-based programming model. Programmers must manually identify the "right" computation granularity for spawning a future to amortize the overhead of thread and context creation. Also, the Safe Future system requires significant and unnecessary object wrapping, which requires non-trivial rewriting to futurize the serial program and can result in significant memory management overhead for fine-grained futures.

The second limitation of this system is its assumption of the linear future creation pattern, i.e., it assumes futures are created one after another in the same function. In other words, it only allows a continuation context to create a future, but does not allow a future to create a future. This assumption simplifies implementation significantly. For example, the system can generate context identifiers simply by incrementing a global counter and organize contexts using a single linked list. However, this also prevents future composition: what if the future computation calls some functions in a third-party library, which might also be futurized? Supporting nesting is the key to improve the composability of a program [65]. It is also an essential requirement for some types of applications to use futures, e.g., the divide-and-conquer style of applications with fine-grained parallelism.

Another limitation of the Safe Future system is that there is no information aggregation in the system. Every context must check its read map against the write maps of ALL of its predecessors. As this list grows, so does the overhead of conflict detection.

Finally, the context management of this system is implemented at the bytecode level. The bytecode rewriter inserts code to save and restore the local states of the program and to correctly handle revocations using exception handling. Such rewriting imposes significant overhead, does not exploit the functionality of the compiler in the JVM, and requires extra memory space to perform local state bookkeeping.

In summary, the Safe Future system is unnecessarily complex, difficult to use, is only able to support a limited number of concurrent program types, and does not exploit the rich information available in the virtual machine to improve performance. We seek an alternative approach that preserves the as-if-serial side-effect semantics of futures more effectively and more efficiently to make the safe future programming model practical.

9.2 Supporting Nested Futures Safely

To support as-if-serial side-effect semantics, we extend our directive-based lazy future implementation to produce a system called Safe DBLFuture (SDBLFuture). We employ many of the Safe Future technologies including dependency violation tracking and prevention, execution contexts, data-access barriers, read/write bit-maps, and version list maintenance.

However, due to the differences between our DBLFuture approach and the Safe Future library-level, interface-based, approach, SDBLFuture is significantly different from the Safe Future system. SDBLFuture system inherits all programmer productivity and performance advantages enabled by DBLFutures. For example, instead of being forced to carefully hardcode the spawning granularity in the program, programmers annotate futures of any granularity. SDBLFuture automatically and adaptively spawns futures only when doing so will improve performance. Our approach also makes serial programs very easy to futurize through our use of the "@future" annotation for all potentially asynchronous computations. SDBLFuture only creates future objects when it spawns a future (via stack splitting) – it generates no other unnecessary wrapper objects. We have presented empirical results that show the benefits of this approach in prior chapters.

SDBLFuture extends Safe Futures in multiple ways. First, we support as-if-serial side-effect semantics for any level of future nesting, thereby supporting a much broader range of concurrent programs, including divide-and-conquer programs with fine-grained, function-level parallelism. Our use of a virtual machine implementation also significantly simplifies the implementation of as-if-serial side-effect semantics. We require no bytecode rewriting by associating execution context creation with stack splitting. We avoid redundant local state saving and restoring by accessing context state directly from Java thread stacks. Finally, we implement context revocation with a bit flip and thus avoid the overhead of revocation via expensive exception handling.

9.2.1 Layered Context ID

In the Safe Future system, each execution context has a unique context ID. This context ID represents the logical order among execution contexts: the earlier a context in the logical order, the smaller its context ID. An execution context tags versions of all objects it creates with this ID using an extra word in the object header. The system uses the tagged ID of object versions to identify the correct one for a context to read. The system maintains a global counter which it uses to assign the next ID when it creates an execution context. When spawning a future, the system creates the future context first, then the continuation context to guarantee that the future context has a smaller ID than the continuation context.

This ID-assignment scheme is simple but does not allow nesting. For example, using this ID scheme, the context C_p, C_f, and C_c in Figure 9.1 gets ID 0, 1, 2 respectively. If the future context C_f creates another future, the new future context will get an ID 4 using this ID scheme and the consistency between the logical order and context IDs is violated. The first step to enable support of nested futures is to design a new context ID scheme so that a new context can be created at an arbitrary nesting level, dynamically, while preserving the order of contexts.

We use a hybrid, layered approach for ID assignment. Figure 9.2 describes this scheme. The context ID can be either a pointer to an ID object or a simple ID; the last bit of the value indicates which (0 for ID, 1 for object). The last two bits of any address in our system are unused due to object alignment.

For a simple context ID, we divide the most significant 30 bits into 15 layers. Each layer has one of three binary values: 00, 01, and 10, which corresponds to the primordial context, the future context, and the continuation context of that nested

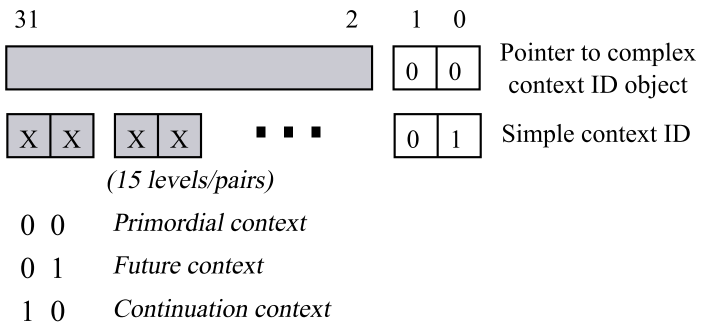


Figure 9.2: Layered execution context ID for supporting nested futures

layer, respectively. We set unused layer bits to 0 and use bit significance to indicate layer order. The higher the bit significance the lower the layer order.

For example, when the system spawns the first future, there are only three execution contexts in the system (C_p, C_f, and C_c), and their ID are $0\text{x}00000001$, $0\text{x}40000001$, and $0\text{x}80000001$ respectively. Upon spawning, the new context inherits the context ID of the current context, and then sets the next layer to 01 for a future context or 10 for a continuation context. Note that the current context has 00 at the next higher layer, which identifies it as the primordial context of the group.

Except for the initial primordial context whose ID is $0\text{x}00000001$, all contexts are either a future context or a continuation context relative to a spawning point in the program. The same future or continuation context can be the primordial context of the next layer if there is nesting. For example, if the future context $0\text{x}40000001$ spawns another future, the context ID of the new future context and continuation context is $0\text{x}50000001$ and $0\text{x}60000001$, respectively. Context $0\text{x}40000001$ becomes the primordial context for this layer. If the continuation context $0\text{x}80000001$ spawns a future, the new context IDs are $0\text{x}90000001$ and $0\text{x}A0000001$.

Using bits from high to low significance makes all sub-contexts spawned by a future context have context IDs that are smaller than that of all sub-contexts spawned by the continuation context. This property is preserved for the entire computation tree using this model. Therefore, the values of context IDs are consistent with the logical order of execution contexts, which facilitates simple and fast version control.

Our system only supports up to 15 layers for simple context ID. Usually 15 layers of nesting is sufficient for most applications since a system should not spawn so many layers of futures unless there is a very large number of processors available. Our lazy and adaptive scheduling system is very effective in making intelligent spawning decisions based on the computation granularity and the system resource availability. In case that more than 15 layers are necessary, we change the simple context ID to a reference to a ID object that implements bit vectors to support arbitrary levels of nesting.

9.2.2 Tree Structure of Execution Contexts

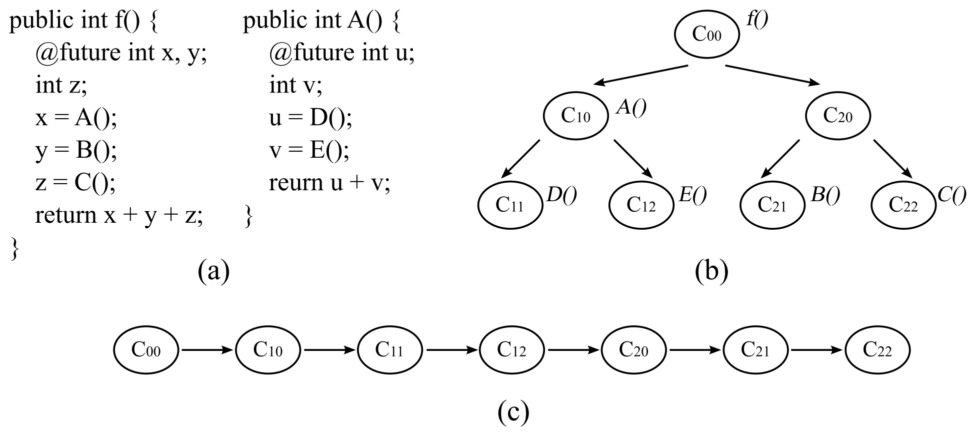
The Safe Future system organizes all execution contexts as a single linked list based on their logical order. With our hybrid and layered context ID scheme, the single linked list structure is also sufficient to support safe nested futures in SDBL-

Futures: upon spawning, we link the new future and continuation contexts together, and then insert them after the primordial context that creates them.

However, there are several disadvantages imposed by this linked-list structure. We use the example in Figure 9.3 in the following discussion. For clarity and concision, we use base-4 number presentation to represent the context IDs and omit unused layers in all of our examples unless specifically noted. Graph (a) in this figure is a simple DBLFuture example; Graph (c) is the linked list of all created contexts.

The linked list structure of contexts loses the parent-child hierarchy information of the computation in a program. By definition, both future and continuation contexts will never conflict with their primordial context since both contexts start after the primordial context and there is no concurrent data access between them and the primordial context. Similarly, for the nested futures, we want to avoid false conflicts between a context and all of its ancestors on the spawning path.

For example, in Figure 9.3, the conflicts between C_{11} (and C_{12}) and C_{10}, C_{00} are false. The condition that Safe Future uses to avoid such false conflicts is that two contexts share the same execution thread. This condition only works for the continuation context in their system. So if the future context reads something that is written by the primordial context, there is always a revocation which is not necessary. With the new layered context ID scheme, we are able to detect ancestor-descendant relationship among contexts using their context ID. However, using this implementation, our sys-



```
public int f() {                public int A() {
    @future int x, y;               @future int u;
    int z;                          int v;
    x = A();                        u = D();
    y = B();                        v = E();
    z = C();                        reurn u + v;
    return x + y + z;           }
}
```

Figure 9.3: Tree structure of execution contexts

tem is forced to traverse the entire list prior to a context, to detect such relationships and avoid false conflicts.

The list implementation also prevents data aggregation. In the Safe Future system, the read/write maps and generated versions of an execution context are kept in the context even after the context has committed. To detect a conflict, the system compares the read map of a context against the write maps of all of its predecessor contexts. For example, the system performs six map comparisons for conflict detection for context C_{22}. Thus, the cost of conflict detection depends on the number of contexts – the more contexts, the larger the overhead. Moreover, without complex lock management, list access imposes costly synchronization overhead. Our goal is to aggregate map information at primordial contexts and enable simple, low-overhead access.

To address these limitations, we replace the linked list implementation with a context tree. The Graph (b) in Figure 9.3 shows the structure for the contexts in the example. In this structure, the primordial context is the parent of the future and continuation contexts. The structure handles primordial context suspension in a straightforward way. The system suspends the context at the point of th future spawn and resumes it after the future and continuation complete. Thus, when the system commits a primordial context, then it has committed the entire subtree of computation below the context.

Upon a context commit, the system merges the shared data accessing information from a child context into the parent. Specifically, the system merges the read/write bitmaps of the child context with that of the parent (i.e. performs a bitwise "OR" operation on the bitmaps). In addition, for an object version created by the child context, if the parent context also has a private version for the same object, we replace that version with the child's version; otherwise, the child's version is tagged with parent's context ID and is recorded as an object version created by the parent context. Note that a continuation context initiates a commit only if its corresponding future context commits successfully. Therefore, our system requires no synchronization for merging.

With such information aggregation and layout, we only need to check contexts against a root (primordial context) of a subtree as opposed to all nodes in the subtree, when checking conflicts for contexts that occur logically after the root. For example, in Figure 9.3(b), C_{22} only needs to check conflicts against C_{21} and C_{10}, since C_{10} has aggregated information of C_{11} and C_{12}. We require no synchronization for any tree update since all are now thread-local. In summary, using a tree structure of execution contexts is more natural and efficient to support nested, safe futures.

9.2.3 Adaptive and Lazy Execution Context Creation

Our DBLFuture implementation initially treats a future call as any other method invocation. If the system detects that the future computation is computationally large enough to amortize the overhead of spawning, it performs stack splitting to spawn a new thread for execution of the continuation. The system avoids creation of future objects until a stack split occurs. Similarly, SDBLFuture system does not create new execution contexts for a future call unless the stack splitting occurs. This laziness of context creation avoids unnecessary context management overhead for fine-grained future computations.

Given that our system waits until it determines (i.e. *learns*) the granularity of an executed future method, the spawn point of a future is later in time than the function entry point of the future. Any shared data access (shared with the continuation) that occurs in the future prior to spawning is guaranteed to be safe since the system executes the future prior to spawning sequentially. Thus, our learning delay may avoid conflicts and better enable commits in a way not possible in the Safe Future system which trigger spawning eagerly of all futures (without delay).

9.2.4 Simple Context Revocation

The Safe Future system implements context revocation using complicated bytecode rewriting. The bytecode rewriter inserts extra bytecode that stores and restores the local states to and from the future object at the beginning of a continuation. It also inserts new exception handlers to catch a revocation exception, and to transfer control to the correct revocation point. In our system, we provide a much simpler implementation of context revocation since we have direct access to the runtime stack frames of Java threads.

Upon stack splitting, our system spawns a new thread to execute the continuation, and uses the current thread for the execution of the future call. The system sets a *split* flag on the future call's caller frame to indicate the splitting. When the future call returns, it checks this flag to decide whether it should return directly as a normal function call, or if it should store the computed value into a future object, prior to termination. Note that the local state of the continuation is kept on the future's stack even after splitting. Since we have access to both stacks, we need not perform duplicated work to save or restore these states. To revoke the continuation, we only need to reset the split flag of the caller frame of the future call. This causes the future call to return as a normal call, and the current thread continues to execute the code right immediately following the future call – which completes the revocation of the continuation context. This process is similar to the technique for supporting as-if-serial exception handling that we described in Chapter 8. To revoke a future context, we revoke the ancestor context that is (i) closest to the future context and (ii) that is a continuation context. This design may waste the work done by some contexts, but it simplifies the implementation of context revocation significantly: we perform all revocations by simply reseting the split flag.

9.2.5 Local Commit and Global Commit

Given the tree structure of execution contexts, there are two potential strategies for context committing. In the first strategy, the committing process only detects local conflicts. That is, the future context always commits successfully immediately after it finishes computation, and the continuation context only detects conflicts it has with its future context. After committing, we merge both contexts with the parent context, i.e., the primordial context of the group. The committing process continues recursively up to the root of the tree. We call this strategy as *local committing*.

In the second strategy, which we call *global committing*, a context waits for all the contexts in its logical past to finish, and then it detects conflicts against all the previous contexts. In case of conflicts, the system picks the first continuation context (including the current context) on the path up to the root in the context tree for revocation. In addition, since we know that all previous contexts have finished, we set the read maps of the revoked context to null to avoid further conflict detection.

Note that although logically the global committing strategy forces a context to wait for and test against all of its predecessors except for its ancestors, information aggregation within the tree structure enables us to only compare contexts that contain aggregated information for all of its predecessors. The best candidates for such aggregated contexts are the future contexts that are siblings of the continuation contexts that are ancestors of the current context since this set of contexts is able to cover all predecessors with minimal number of contexts. We define this set of contexts as the *test set* of a context.

For example in Figure 9.4, the test set of C_{211} includes C_{100}, while the test set for C_{222} includes C_{221}, C_{221}, C_{210}, and C_{100}. In addition, in the test set, a context need only wait for the context that is closest to itself since this context is the latest in the logical order. Committing of this context indicates that all contexts in the test set have committed. For example in Figure 9.4, C_{222} only waits for C_{221}, C_{221} waits for C_{210}, C_{210} waits for C_{100}, and so on.

The advantage of the local committing strategy is the parallelism it enables, especially when there are no or few conflicts in the program since there is no waiting between a future context and its predecessor contexts that are in other subtrees. In

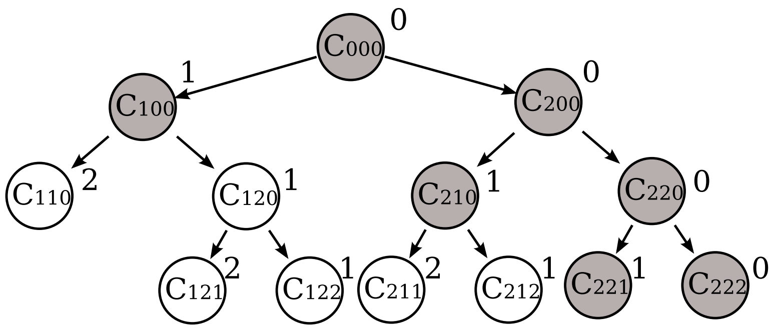


Figure 9.4: Local committing and global committing of execution contexts

particular, if the function that spawns the future is computationally intensive (i.e., spends time computing after both the future and continuation contexts finish), it is more efficient (enables more parallelism) to allow the contexts to commit without waiting for their predecessors. We refer to this pattern of computation as having a *long tail*; as opposed to *short tail* computations.

The disadvantage of the local committing strategy is that it delays conflict detection. For some execution patterns, such delay can potentially cause a large amount of work to be performed wastefully. We use the example context tree in Figure 9.4 to explain this. If there is a conflict between C_{110} and C_{222}, the system will detect this conflict when all contexts in the subtree of C_{200} have committed to their parent contexts, and when C_{200} is detecting conflicts against C_{100}. After the conflict is detected, the system will revoke the computation associated with the entire subtree of C_{200} since after meta data aggregation, the system cannot distinguish the point at which the conflict occurs (C_{222}). In contrast, using the global committing strategy, this conflict will be detected when C_{222} attempts to commit, and the system will revoke only the computation of C_{222}.

In summary, there are tradeoffs between the two committing strategies. The impact of these tradeoffs on performance varies depending on the frequency of conflicts, conflict patterns, computation patterns, and other program and system behaviors. To achieve the advantages of both strategies, we propose a hybrid committing strategy.

9.2.6 Hybrid Committing Strategy

The principle of our hybrid committing strategy is to allow as many contexts as possible to locally commit to exploit available parallelism, but identify contexts that impose a significant delay when conflicts occur so that they can globally commit.

The question is how to identify such contexts efficiently and effectively. First we note that the amount of wasted work due to local committing is related to distance between the two conflicted contexts. Since all contexts are logically total-ordered, and a context only checks for conflicts against contexts in its logical past, given a certain context in the tree, the later another conflicting context is in the logical order, the more delay penalty due to local committing is. For example, in the context tree in Figure 9.4, if C_{121} conflicts with C_{110}, using local committing, the system detects this conflict while committing C_{120} and revokes C_{120}, which wastes all work done by contexts in the subtree of C_{120}. In contrast, with global committing, the system detects the conflict while committing C_{121}, and as a result, revokes C_{120} immediately and aborts C_{122}. The difference between the two committing strategies for this case is just the partial work done by C_{122} (depends on how early the abortion happens) and the partial work done by C_{120} after both C_{121} and C_{122} commit, which is wasted using local committing, but not with global committing. For the case that C_{122} conflicts with C_{110}, the local committing still cause all work done by C_{120}, C_{121}, and C_{122} wasted. But the global committing is able to preserve the work done by C_{121} and C_{120}. The penalty difference of the two committing strategies becomes larger. For the case that C_{222} conflicts with C_{110}, the penalty difference is even larger: global committing is able to preserve work done by 6 contexts which is all wasted if we use local committing.

We know that for each spawning point, contexts in the future subtree are all earlier than contexts in the continuation subtree in the logical total order. So contexts in the continuation subtree might appreciate global committing relatively more than those in the future subtree as we can see from the above example. A simple heuristic we could use is to give contexts following the continuation path in the context tree higher priority to perform global committing than those follow the future path at the same layer.

Using this heuristic, our algorithms works as follows. We maintain a new property, called *spawning level*, for each execution context. The spawning level of the root primordial context is 0. For a future context, its spawning level is the spawning level of its parent context plus one. The spawning level of a continuation context is same as its parent context. The spawning levels of contexts in Figure 9.4 are labeled on the right side of the nodes. The deeper a context is in the future subtrees, the larger its spawning level is. The system uses the spawning level to indicate the priority with which the system performs global committing. The smaller the spawning level is, the higher the priority is.

We then define a parameter called the *global committing threshold*. The system decides whether to perform local or global committing for a future context based on its spawning level and the global committing threshold. A future context performs global committing only if its spawning level is equal to or less than the global committing threshold. A continuation context performs global committing if the corresponding future context globally commits since a continuation context always waits for its future context. The shadowed nodes in Figure 9.4 represent contexts that are globally committed when the global committing threshold is set to 1, while those not in the shadow are locally committed.

Different applications require different global committing thresholds to achieve the best tradeoff between parallelism and conflict detection delay penalty. The factors that play a role in this tradeoff include conflict frequency (no/light conflicts versus heavy conflicts), conflict patterns (who conflicts with whom), and computation patterns (long tail computation versus short tail computation), etc. The strategy we use is to set this threshold to 1 initially, i.e., the top two levels of contexts are globally committed, and all other contexts are locally committed. This strategy facilitates parallelism for most of contexts, and at the same time avoids unnecessary delay of conflict detection at the top levels, which usually causes computational waste from revoked contexts. When the system detects that a context whose spawning level is greater than the current threshold is revoked, it increases the threshold to that spawning level adaptively to avoid additional wasted work. Although this approach is not optimal, it is simple and enables good performance for all of the benchmarks that we investigate in our experimental evaluation.

9.2.7 History-based Learning

We also exploit the adaptation of the Java virtual machine in other ways in SDBL-Futures. In particular, we investigate two learning strategies that attempt to minimize the number of revocations and wasted work adaptively using the behavior of executing contexts. The first strategy we investigate is not to split a future again if its continuation has been revoked in the past. We call this *Not-To-Split* (NTS) learning strategy. The rationale behind this strategy is straightforward: the continuation will most likely be revoked, so it is better to execute sequentially to save computation resources.

```
1  public int f() {
2      @future int x, y, z;
3      int w;
4      x = A();
5      y = B();
6      z = C();
7      w = D();
8      return w + x + y + z;
9  }
```

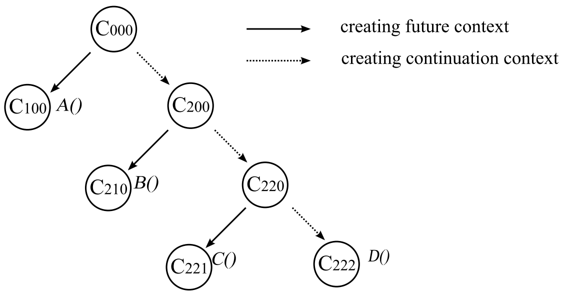


Figure 9.5: A simple program that spawns 4 futures

The NTS learning strategy works effectively for some conflict patterns, but not always. We use the example in Figure 9.5 to demonstrate. In this example, we create three futures in function $f()$. The corresponding context tree is shown in the right figure. The base-4 context IDs are labeled inside each context node, and the associated computation is tagged on the right.

- **case 1** Assume $C()$ and $D()$ have a conflict.

  The system revokes C_{222} after detecting this conflict. At the same time, the system records this information in a revocation history database. If $f()$ is called again, according to the NTS learning strategy, the system will not spawn the continuation for the future call $C()$ at line 7, and the revocation is avoided. This behavior does not change given different global committing thresholds since the conflict itself is local.

- **case 2** Assume $B()$ and $D()$ have a conflict.

- *case 2.1* The global committing threshold is 0, i.e., all contexts perform local committing. In this case, conflict detection is delayed until C_{220} attempts to commit. The system revokes C_{220} as a result. Next time, the system does not spawn the continuation of $B()$ at line 6. Although it will still spawn continuations for $A()$ and $C()$ since there is no revocation history in the database for these spawning points, the revocation is prevented since $B()$ now is executed by an ancestor of the context that executes $D()$.
- *case 2.2* The global committing threshold is 1, i.e., all contexts in this tree perform global committing. Now the conflict is detected when C_{222} attempts to commit since by global committing, the test set of C_{222} includes C_{100}, C_{210}, C_{221}. As a result, the system revokes C_{222} at line 7. We see that for the first round, the global committing strategy helps to avoid wasting work done by C_{221}. In the second round, the NTS learning strategy prevents the system from spawning the continuation of $C()$ at line 7. However, this does not prevent $B()$ from executing concurrently with $D()$ if the continuation at 6 is spawned. Instead, it takes longer for the system to detect the conflict, and one more round to prevent the revocation completely.

For *case 2.2*, the NTS learning strategy does not work effectively since it ignores one piece of important information: the conflict was detected after all previous contexts of C_{222} have successfully committed, otherwise, C_{222} would have been aborted, instead of revoked, based on the global committing algorithm. This information indicates an important temporal property: as long as we start C_{222} after all previous contexts finish, there will be no conflict.

Based on this observation, we introduce another learning strategy: for a revoked spawning point in the global committing zone, in the next round, instead of simply not splitting, the system performs the stack splitting as usual, but suspends the continuation context right after its creation. The system also deletes the read maps of the continuation context to avoid conflict detection across this context. Once the future context commits itself successfully, it will resume the suspended continuation context. We refer this strategy as *Split But Suspend* (SBS) learning strategy. This learning strategy does not work for local committing since there is no such temporal information can be learned from a revoked context that is locally committed. Therefore, we use the NTS strategy for the local committing zone and the SBS strategy for the global committing zone, which makes it a hybrid strategy. We refer this hybrid strategy as *NTS+SBS* learning strategy. Now for *case 2.2*, in the second round, the continuation context C_{222} is still spawned at line 7, but suspended until C_{221} commits successfully and notifies it. C_{222} will not perform any conflict detection since its read-map is null. The system effectively prevents revocations and wasted work completely.

9.2.8 Integration with As-if-serial Exception Handling

As-if-serial exception handling attempts to preserve the exception handling behavior of a concurrently executed future application as if it were executed sequentially. This simplifies the task of writing concurrent applications using futures further since programmers now can reason about the exception handling behavior of an application in the serial version, and then introduce "@future" annotations to improve parallelism without worrying about the exception handling behavior of the concurrent version. In Chapter 8, we describe and evaluate our implementation of as-if-serial exception handling support for DBLFutures.

Note that the true as-if-serial exception handling semantics defines which and where exceptions thrown by concurrently executed computations should be handled, which was the focus of Chapter 8. However, this semantics also requires that all side-effects caused by computations that are before the handled exception in the logical serial order be preserved, while those side-effects that are in the logical future of the handled exception be discarded. We cannot preserve this part of the as-if-serial exception handling semantics without the as-if-serial side-effect support. Now, with the as-if-serial side-effect guarantee provided by the SDBLFuture system, we can support the complete semantics of as-if-serial exception handling, which is our focus of this section.

In Chapter 8, we maintain a total ordering on thread termination across threads that originate from the same future spawning point and execute concurrently. In SDBLFutures, maintaining this total ordering of threads are not necessary since we can derive the required ordering information of threads from their associated execution contexts, which are totally ordered based on their logical serial execution order. Therefore, all extra data structures that we introduced in algorithms in Chapter 8 for this purpose, such as the new fields of the internal thread objects including `futurePrev`, `futureNext`, and `commitStatus`, are not necessary anymore.

In addition, in Chapter 8, we augment the algorithms of *futureStore* and *futureLoad* in the DBLFuture system with the logic that enables the current thread that is waiting for the previous thread in the total ordering to finish and cleanup itself if it is aborted before performing the real actions of both functions (see Figure 8.5 and Figure 8.6). This augmentation is necessary to preserve the total ordering on termination of threads in this case.

However, in the SDBLFuture system, similar logic is already part of the algorithms of *futureStore* and *futureLoad* to preserve the as-if-serial side-effect semantics, except that we use execution contexts instead of threads. This means that the extra work in *futureStore* and *futureLoad* that was required to support as-if-serial exception handling now comes for free in the SDBLFuture system. Moreover, the cleanup on abortion that is performed by the SDBLFuture system includes removing all private object versions that are created by the aborted context. Since the side-effects of an execution context are kept as private object versions of that context and will not be visible until it commits, such cleanup reverts all side-effects of the computation associated with the aborted context completely which cannot be done by algorithms in Chapter 8.

The only extra algorithm that is still necessary to support as-if-serial exception handling in the SDBLFuture system is the exception delivery algorithm, which has similar logic to that of the delivery algorithm in Figure 8.7, but with slightly different implementation details since now the total ordering is implemented via execution contexts instead of threads. The new exception delivery algorithm is shown in Figure 9.6.

Comparing to the normal exception delivery algorithm in the unmodified virtual machine, this exception delivery algorithm has two extra parts. The first part is executed before searching for a handler in the compiled method of the current frame (line $3 \sim 11$ in Figure 9.6). This part ensures that an exception thrown by a continuation context, but that is not handled within the continuation context before it unwinds to the splitting frame, will not be handled unless the current context commits successfully.

Successfully committing the current context indicates that all side-effects of the concurrent executed contexts up to this point are guaranteed to be same as if the

```
void deliverException(Exception e) {
  while (there are more frames on stack){
    if (the current frame has a split future) {
      // a frame for a continuation context
      currentContext = currentThread.executionContext;
      try to globally commit currentContext;
      if (currentContext is aborted || currentContext will be revoked){
        cleanup currentContext;
        terminate currentThread;
      }
    }
    search for a handler for e in the compiled method on the current stack;
    if (found a handler) {
      jump to the handler and resume execution there;
      // not reachable
    }
    if (the current frame is for a future function call
          && its continuation has been spawned) {
      currentContext = currentThread.executionContext;
      try to globally commit currentContext;
      if (currentContext is aborted) {
        cleanup currentContext;
        terminate currentThread;
      }else{
        abort continuationContext;
        reset the caller frame to non-split status;
      }
    }
    unwind the stack frame;
  }
  // No appropriate catch block found
  report the exception and terminate;
}
```

Figure 9.6: Algorithm for the exception delivering point in the SDBLFuture system

program is executed sequentially. Therefore, we can proceed to search for a handler in the current compiled method as in serial execution. But if the current context is aborted or revoked, which indicates that the current exception may not have existed if the program is executed sequentially, the current context is cleaned up and the current exception is ignored. Note that a continuation context usually ends and commits at the usage point of the future value, but in case of exceptions, it ends and commits at the exception throwing point.

The second extra part in this algorithm (line $17 \sim 28$ in Figure 9.6) occurs after a handler is searched but not found in the current frame, and before the stack is unwound to the next frame. Similar to the first part, this part ensures that an exception thrown by a future context, but that is not handled locally even when the stack is unwound to the stack frame of the future call, will not be handled unless the future context commits successfully. In case of abortion, the current context is cleaned up and the exception is ignored as in the first part. If the future context is indeed successfully committed, to handle the thrown exception on the current stack as if the future call is a normal call as we described in Section 8.3.2, the system resets the split flag of the caller frame to revert the stack splitting. The system also aborts the continuation context, which recursively aborts all contexts in the logical future of the current context, and reverts all side-effects caused by these contexts that should not exist if the program is executed sequentially. Finally, the stack is unwound, and the algorithm is repeated for the next stack frame.

In summary, supporting as-if-serial exception handling and preserving as-if-serial side-effect semantics have many common requirements and can share many common implementations. Therefore, integrating the support of as-if-serial exception handling support to the SDBLFuture system is simple and straightforward. Moreover, with the underlying support of preserving as-if-serial side-effects in the SDBLFuture system, the complete as-if-serial exception handling semantics, which also defines the side-effect behavior in the presence of exceptions, is now supported.

9.3 Performance Evaluation

We have implemented SDBLFutures over the DBLFuture system, which is implemented in IBM Jikes Research Virtual Machine (JikesRVM) [84] (x86 version 2.4.6). For comparison, we also port the Safe Future system to the same version of JikesRVM. Again, our test machine is a dedicated 4 processor box (Intel Pentium 3 (Xeon) xSeries 1.6GHz, 8GB RAM, Linux 2.6.9) that was used for all of our previous experiments. Since optimizations are essential for the SDBLFuture system to reduce the significant overhead caused by the large amount of read/write barriers, we only present experiment with the VM configuration that employs an adaptively optimizing compiler. We use pseudo-adaptation [14] to reduce non-determinism in our experiments.

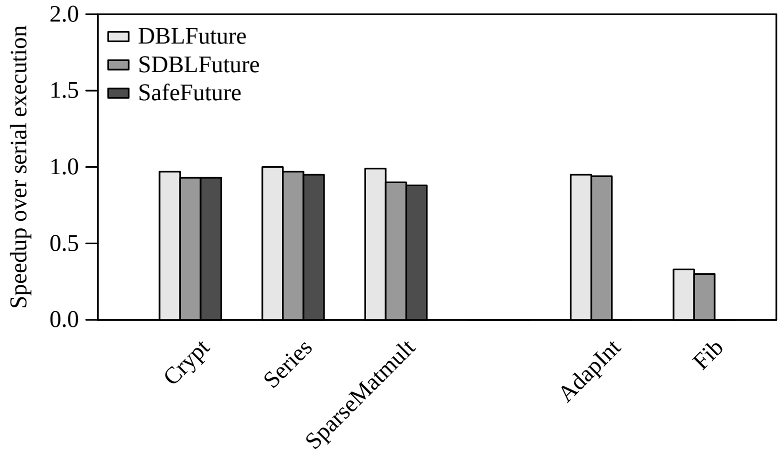
In the following sections, we evaluate the performance of our SDBLFuture system. First, we compare its performance with our DBLFuture system and the Safe Future system for a set of benchmarks that have no dependency violations. We then study the performance impact of our various strategies within the SDBLFuture system, such as local committing versus global committing, and different learning strategies for a wide rage of computation patterns and conflict patterns.

For all the experiments in this section, we measure the execution time (T) of each configuration, and present the speedup over the serial execution time (T_s), which is computed as T_s/T. That is, if the speedup is bigger than 1, there is performance improvement over serial execution. Otherwise, there is performance degradation over the serial version.

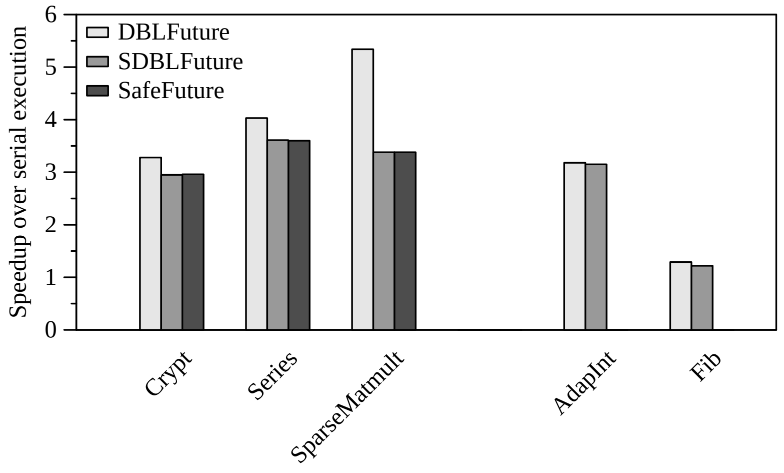
9.3.1 Performance of Benchmarks with No Dependency Violations

For experiments in this section, we use two sets of benchmarks. The first set includes three benchmarks (*Crypt*, *Series*, *SparseMatmult*) from the multithreaded version of Java Grande Benchmark Suite [133]. The second set includes two benchmarks (*Fib*, *AdaptInt*) from the divide-and-conquer style of benchmarks that we adopted from the Satin system [148]. There are no data races (conflicts) in these benchmarks, so there is no revocation overhead.

The number of created futures for the benchmarks in the first set is the number of processors used (up to 4 for a 4-processor machine). Therefore, the overhead of managing execution contexts for these three benchmarks is negligible. We use this set of benchmarks to show the overhead of tracking accesses to shared data.



(a) With 1 processor



(b) With 4 processors

Figure 9.7: Performance evaluation of the SDBLFuture system for benchmarks with no dependency violations comparing to the DBLFuture system (the first bar) and the Safe Future system (the third bar). The left three benchmarks represent applications with a small number of coarse grained futures and the right two represent applications with a large number of fine-grained futures. There is no data available for the Safe Future system for the second set of benchmarks since this system does not support nested futures.

In contrast, there are a large amount of futures created for the two benchmarks in the second set given their recursive nature, and there are few accesses to shared data. Thus, we use this set of benchmarks to evaluate the overhead of execution context management. Figure 9.7 shows the speedup (degradation if < 1) gained by the SDBLFuture system for each benchmark over its unmodified serial execution. For comparison, we also show the performance of the DBLFuture system which has no as-if-serial side-effect guarantee and the Safe Future system on the side. The Safe Future system does not support nested futures, so there is no data for the second set of benchmarks for this system.

Our results show that for the first set of benchmarks, the performance of our SDBLFuture system is about the same as that of the Safe Future system since they share the same implementation for tracking accesses to share data. In addition, the overhead is similar to DBLFuture system, which does not have any support of the as-if-serial side-effect semantics. On average, with 1 processor (Graph (a)), the speedup (degradation in this case) caused by the DBLFuture system, the SDBLFuture system, and the Safe Future system, are 0.98, 0.93, 0.92, respectively. With 4 processors (Graph (b)), all systems are able to achieve almost linear, even super-linear speedup (on average 4.22, 3.31, and 3.31 for the three systems in order) for these benchmarks. The super-linear speedups we believe, are due to the improved data locality of the multithreaded versions of the benchmarks over their serial versions.

The results for the second set of benchmarks, which have a large amount of potential future spawning points (5.78 million for AdaptInt, 102.33 million for Fib), show that the management of execution contexts in our SDBLFuture system introduce negligible overhead compared to the unsafe DBLFuture system. The average degradation of the DBLFuture system and the SDBLFuture system with 1 processor are 0.64 and 0.62, while with 4 processors, the speedups are 2.24 and 2.19 on average.

In summary, for benchmarks without data races, our SDBLFuture system introduces acceptable overhead for tracking accesses to shared data. In addition, the overhead of managing execution contexts, even with a large number of futures, is negligible comparing to the un-safe DBLFuture system. With more computing resources available, our SDBLFuture system is able to achieve good speedup for both set of benchmarks.

9.3.2 Parallelism of Local Commit versus Global Commit

In our SDBLFuture system, local committing is a committing strategy that allows an execution context to commit to its parent without waiting for other contexts in its logical past. The conflict detection algorithm for local committing only tests conflicts in the scope of the current context group. That is, the future context of the group always commits successfully to its parent, and the continuation context only fails when it has conflicts with its future context. The conflict detection with other predecessor

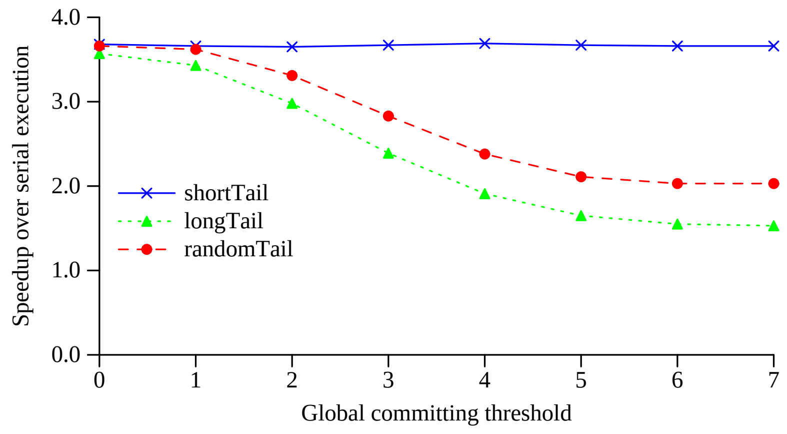


Figure 9.8: Performance impact of global committing threshold for various computation patterns. The longTail pattern refers to the case that a large amount of work is executed after the continuation context ends in a function. The shortTail pattern is the opposite case, and the amount of work distributed after the continuation context ends in the randomTail pattern is randomly generated and falls in between the other two cases.

contexts is delegated to the parent context. In contrast, the global committing strategy requires a context to wait for and to test against all contexts in its logical past. Our system employs a hybrid scheme that adaptively chooses the committing strategy for an execution context according to its spawning level and the global committing threshold, which is a dynamically changed parameter of our system. In this section, we investigate the parallelism of the committing strategies. In the next section, we will compare their ability to handle dependency violations given different conflict patterns.

The benchmark we use for this study is a synthetic program based on the Fibonacci computation (Fib). We modify Fib to make each recursive invocation of `fib()` do some extra amount of computation so that its execution is long enough to trigger the spawning testing. We also modify our system decision model to always split the stack when making the spawning decision. These modifications help us to force the system to generate large enough layers (7 in our experiments) of nested futures that we can use for further evaluation of our system. For the extra amount of computation in each invocation, we divide it into two parts: one part is executed before the future call, and the other part is executed after the continuation ends (the usage point of the value computed by the future). We change the ratio of these two parts to model three computation patterns: (1) All work is done before the future call (*shortTail*); (2) All work is done after the continuation ends (*longTail*); (3) The ratio between the two parts is random (*randomTail*). We then collect execution time for all three patterns and different global committing thresholds. Figure 9.8 shows the speedups over the serial execution for all configurations.

The results shows that with the shortTail pattern, the committing strategy does not impact performance. However, for the longTail pattern, the smaller the global committing threshold is, i.e., the more contexts are allowed to do local committing without waiting, the larger speedup the system is able to achieve. The curve of the randomTail pattern falls in between the two extreme patterns, but still shows the same trend as the longTail pattern does.

In summary, when there are no or few dependency violations, the local committing strategy enables more parallelism compared to global committing. This is the reason that we choose to set the initial global committing threshold to a small number so that more parallelism in the program could be exploited. In next section, we will explain why we set threshold to 1 instead of 0 although 0 achieves the best performance for the experiments in this section.

9.3.3 The OO7 Benchmark with Controlled Conflict Patterns

The OO7 benchmark suite [25, 24] is a well known benchmark in the objected-oriented database field. The OO7 benchmark operates on a hierarchical structure of data. On the top level, there are a certain number of modules, each consisting of several assemblies, which consist either of some composite parts (a *base* assembly) or several assemblies (a *complex* assembly). Each composite part consists a number of atomic parts that may connect to others via a bi-direction connection. The number of modules, assemblies per module, assemblies per assemblies, composite parts per assembly, atomic parts per composite part, and the layer of nested assemblies are all controllable via program parameters. For each iteration of the execution, the program performs a certain number of operations on this data hierarchy, each operation randomly follows paths in the hierarchy tree to pick a composite part, and then traverses the atomic parts of that composite part. For each visited atomic part, the program either changes some attributes of that part if it is a write, or it does nothing. Another nice parameter is the lock depth of data accessing. This parameter controls at which level, a lock is used to protect a data access. The deeper the lock depth is, the finer the lock's granularity is.

The authors of [153] choose this benchmark to evaluate the performance of the Safe Future system since it allows easy control over the amount of contention for access to shared data via flexible benchmark parameters such as database structures, ratios between private and shared reads/writes. In their work, they use $M+1$ modules for M futures. Each future has a private module for private reads/writes, and the extra module is used for shared reads/writes.

In this work, we also use the OO7 benchmark to evaluate the performance of our SDBLFuture system. The purpose of this set of experiments is to investigate the ability of our system to handle dependency violations with different committing strategies (local versus global), and different learning strategies (no learning, not-to-split (NTS), or split-but-suspend (SBS)). We find that although the fractions of private/shared reads/writes could control the contention level of shared data in the program, they do not directly reflect the conflict patterns among futures due to the randomness of data access distribution. To make the results more meaningful in our evaluation, we have modified the OO7 benchmark and set up the experiments as follows:

- Instead of generating $M+1$ modules for M futures, we only generate 1 module for all futures. That is, all data accesses are shared at the module level.
- At each hierarchy, we generate $M+1$ sub-components (assemblies or composite parts). The nth future picks the nth sub-component to operate on if it's a private operation. Otherwise, it operates on the last sub-component.
- Each future performs a set of operations, including zero or one shared read, zero or one shared write, and several private reads and writes.
- Whether a future performs shared read/write is controlled by a program parameter, called *conflict pattern*. The conflict pattern parameter is used as a bit-map, with higher bits representing earlier futures. A future will perform one shared read and one shared write if its corresponding bit in the conflict pattern is set. For example, with 4 futures, if the conflict pattern is 1100, then the first and second futures both perform the shared data access, which results in a dependency violation between them. We tried 12 conflict patterns for 4 futures, including 0000, 1100, 1010, 1001, 0110, 0101, 0011, 1110, 1101, 1011, 0111, and 1111.
- The timing of the shared write operation among all operations is controlled by a parameter called *writePosition*. The value of this parameter is 0 to 100. 0 means that the shared write operation should be done as the first operation,

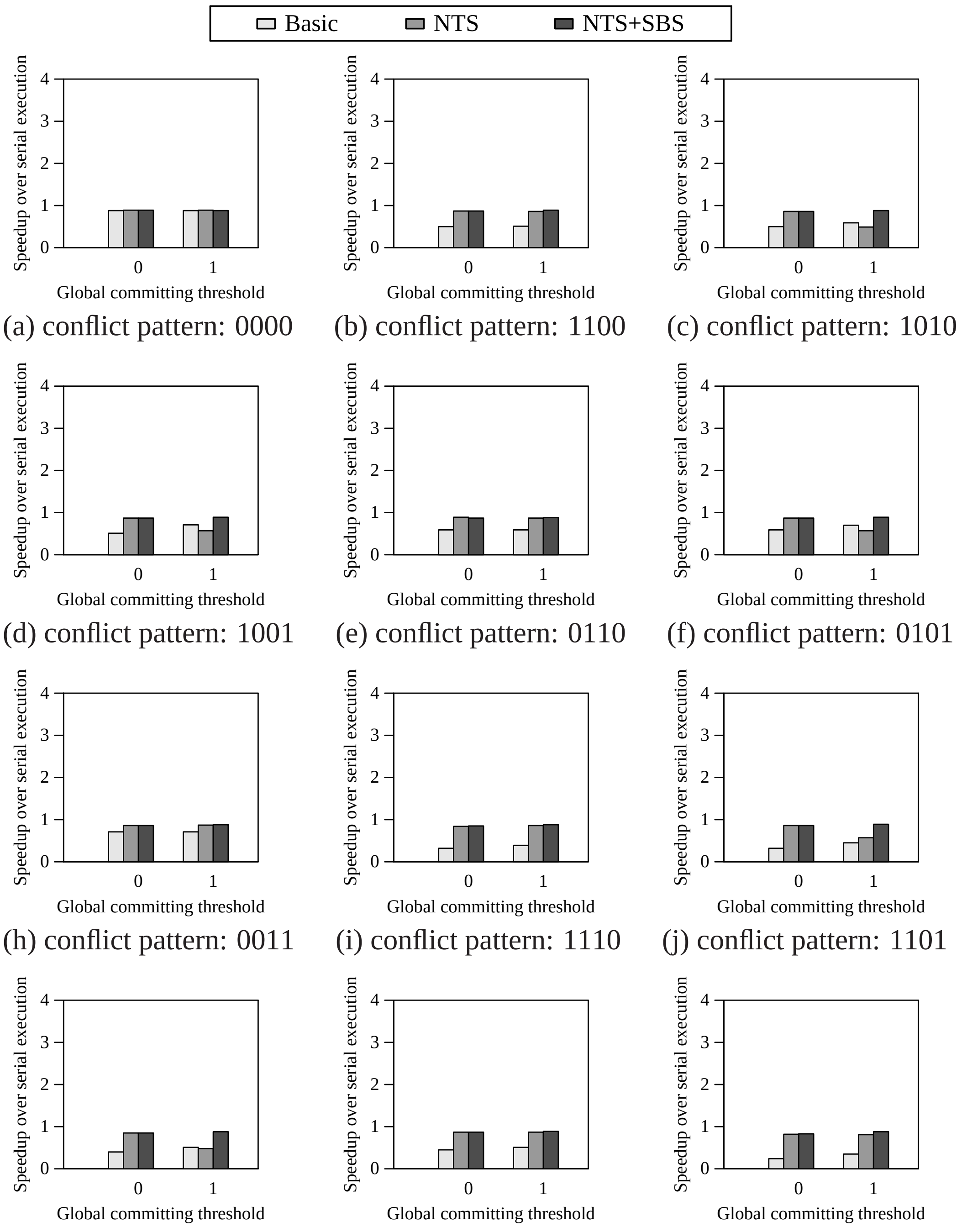
while 100 means it should be the last operation. A value x means the shared write operation should be performed after x% of operations have been done.

- For each execution of the OO7 benchmark, two identical iterations are executed. Each iteration spawns M futures, each works on N operations. For all results in this set, $M = 4$, $N = 10$.
- Other parameters of OO7 are set as same as in [153]: 7 assembly levels, 20 atomic parts per composite part, 3 connections per atomic part, and the document size is 2000 bytes, the manual size is 100000 bytes.

By removing the randomness in the program and designating one specific path for shared data accesses, we guarantee that if it is a shared write operation, it indeed operates on the same component, which will result in a dependency violation. This help us to understand the results better and to draw meaningful conclusions accordingly.

For each conflict pattern, we have investigated two global committing thresholds and three learning strategies. For the OO7 benchmark, all futures are created linearly at the same level. So setting global committing threshold to 0 make all contexts locally committed. If it's set to 1, all contexts are then globally committed. We tried both setting to study their performance across different conflict patterns. The three learning strategies we evaluated include no learning (*Basic*), not-to-split if revoked (*NTS*), and not-to-split if locally committed and split-but-suspend if globally committed (*NTS+SBS*). The results are shown in Figure 9.9 (for 1 processor) and Figure 9.10 (for 4 processors). Each graph in these two figures represents performance data for all strategy combinations for one particular conflict pattern. The x-axis is the global committing threshold used. The y-axis is the speedup of the second iteration over the serial execution. We choose to show the performance of the second iteration to evaluate the effectiveness of different learning strategies, which are represented by the three bars in the graphs. In addition, to help understand the results better, we list the total number of created futures and revocations for each configuration in Table 9.1. In each column of data, the first number is the total number of created futures for two iterations. The corresponding revocation count is listed inside the parentheses. Theses numbers are the same for 1 processor runs and 4 processor runs because they are only dependent on conflict patterns. Note that the number of futures created and the number of revocation do not always map to the execution time. For example, it is possible that the number of futures created by a faster execution is larger than a slow execution since there might be more futures created but quickly aborted in the first case. But in general, these two numbers reflect the amount of work, including wasted work, that one execution has done. Especially the revocations have a big impact on end performance.

Our first observation from these results is that when there is no learning involved, the global committing strategy works better than the local committing strategy in gen-



(a) conflict pattern: 0000 (b) conflict pattern: 1100 (c) conflict pattern: 1010

(d) conflict pattern: 1001 (e) conflict pattern: 0110 (f) conflict pattern: 0101

(h) conflict pattern: 0011 (i) conflict pattern: 1110 (j) conflict pattern: 1101

(k) conflict pattern: 1011 (l) conflict pattern: 0111 (m) conflict pattern: 1111

Figure 9.9: Performance impact of global committing threshold and learning strategies on the 12 controlled conflict patterns of the OO7 benchmark (1 processor).

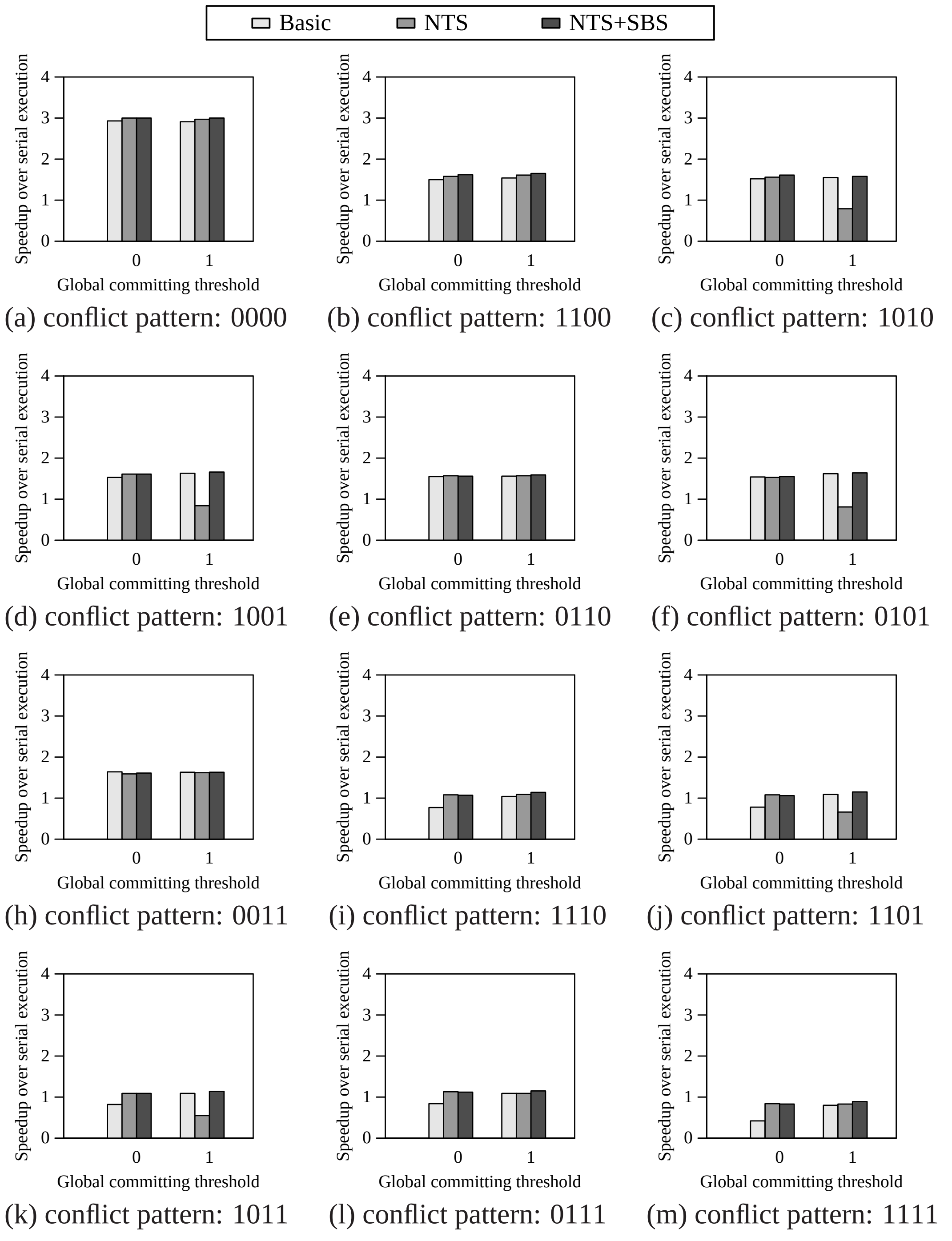


Figure 9.10: Performance impact of global committing threshold and learning strategies on the 12 controlled conflict patterns of the OO7 benchmark (4 processors).

Conflict	Local Committing			Global Committing		
patterns	Basic	NTS	NTS+SBS	Basic	NTS	NTS+SBS
0000	8 (0)	8 (0)	8 (0)	8 (0)	8 (0)	8 (0)
1100	14 (2)	10 (1)	10 (1)	14 (2)	10 (1)	10 (1)
1010	14 (2)	10 (1)	10 (1)	12 (2)	11 (2)	9 (1)
1001	14 (2)	10 (1)	10 (1)	10 (2)	9 (2)	8 (1)
0110	12 (2)	9 (1)	9 (1)	12 (2)	9 (1)	9 (1)
0101	12 (2)	9 (1)	9 (1)	10 (2)	9 (2)	8 (1)
0011	10 (2)	8 (1)	8 (1)	10 (2)	8 (1)	8 (1)
1110	22 (6)	10 (2)	10 (2)	18 (4)	11 (2)	11 (2)
1101	22 (6)	10 (2)	10 (2)	16 (4)	11 (3)	10 (2)
1011	18 (6)	9 (2)	9 (2)	14 (4)	10 (3)	9 (2)
0111	16 (6)	8 (2)	8 (2)	14 (4)	9 (2)	9 (2)
1111	30 (14)	8 (3)	8 (3)	20 (6)	11 (3)	11 (3)
Avg	16.0 (4.2)	9.1 (1.4)	9.1 (1.4)	13.2 (2.8)	9.7 (1.8)	9.2 (1.4)

Table 9.1: The number of created futures and revocations of the first two iterations for the 12 controlled conflict patterns of the OO7 benchmark using different global committing thresholds and learning strategies.

eral for this benchmark. For example, for the worst conflict pattern 1111 (see Graph (m) in Figure 9.9 and Figure 9.10), i.e., all futures conflict to each other, the speedup value (degradation) on 1 processor with local committing is 0.24, while the global committing strategy is able to achieves 0.35, although still very low due to the heavy conflicts in this pattern. This is because the global committing strategy enables earlier detection of dependency violations, and has finer grained revocation (revoking the problematic context instead of the whole subtree of some ancestor of the problematic context). From Table 9.1, we see that the local committing results in 30 futures and 14 revocations, which is a significant amount of wasted work. The global committing reduces the number to 20 futures and 6 revocations, which are much lower. On average, without learning (the first bar of all graphs), the local committing achieves speedup 0.50 with 1 processor, and 1.32 with 4 processors. While the speedups of global committing are 0.58 and 1.46 for 1 and 4 processors respectively. Based on the above observation, we set the initial global committing threshold of our SDBLFuture system to 1 instead of 0 to take advantage of the better ability of global committing to handle revocations for applications like OO7.

The second observation from these results is that the not-to-split (NTS) learning strategy works very effectively to reduce wasted work for local committed contexts. For example, for the worst pattern 1111, NTS is able to reduce the number of created futures from 30 to 8 with only 3 revocations, instead of 14. Note that for this conflict pattern, the NTS strategy not only helps to eliminate revocations completely in the second iteration since it learns that all spawning points are not safe, it even helps in the first iteration to avoid splitting the same spawning point repeatedly which reduces wasted work significantly . The average speedups gained by this strategy combination are 0.86 for 1 processor and 1.47 for 4 processors, which are much better comparing to the basic, non-learning configuration.

The third observation is that for globally committed contexts, the NTS strategy improves performance for some conflict patterns, but degrades performance for other patterns including 1010, 1001, 0101, 1011. We found that this is because the NTS strategy delays the conflict detection in the second iteration due to its not-to-split decision. For example, for the conflict pattern 1010, in the first iteration, the conflict is detected when the first future attempts to commit. As a result, the system revokes the whole continuation of the second future. Now in the second iteration, the NTS strategy decides not to spawn the continuation of the second future, which makes the second future perform double amount of work that was done by the second and the third futures in the first iteration. The conflict is detected when the second future attempts to commit, which is much later comparing to the first iteration. When there are 4 processors, the performance impact of this delay becomes more significant since the NTS strategy results in idleness of some processors, which could have been used to detect the conflict earlier. On average, the strategy combination (global committing + NTS) achieves 0.73 speedup with 1 processor, which is slightly better than the basic, non-learning configuration. With 4 processors, the speedup gained by this combination is only 1.20, which is worse than the basic, non-learning one.

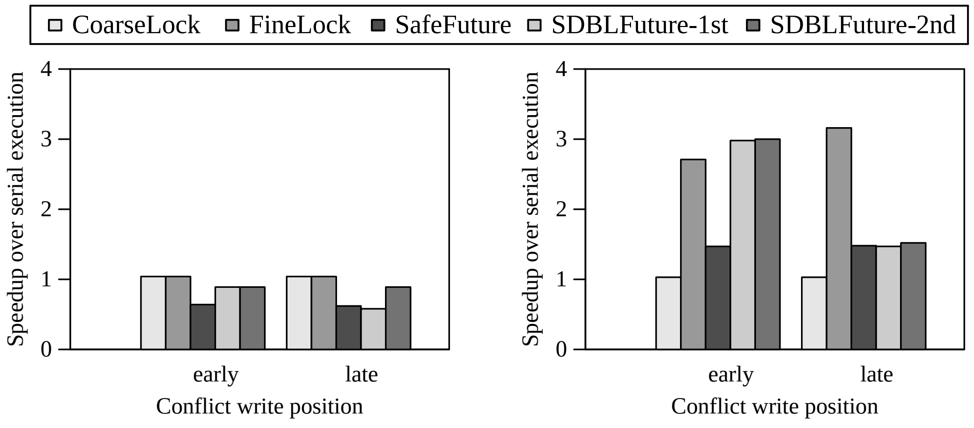
Fortunately, our hybrid NTS+SBS learning strategy works more intelligent than the NTS strategy does. For locally committed contexts, it works the exact same way as the NTS strategy. Therefore, their performance numbers are almost the same. For the globally committed contexts, this strategy exploits the temporal information available in a revocation, i.e., all contexts in the logical past of the current context have committed successfully at the point when the conflict is detected, thus, as long as the current context starts after its previous context commits, there will be no conflict.

So instead of simply not splitting, this strategy splits but suspends the problematic context in the second round, which eliminates revocations in the second iteration for all the patterns. The average speedups gained by this strategy combination (global committing + NTS+SBS) are 0.89 and 1.52 for 1 and 4 processors respectively, which is the best performing strategy combination among what we investigated.

Another interesting observation is that the performance penalty of not learning, which is significant with 1 processor, becomes much smaller, even negligible for some cases, when there are more processors available. This makes sense because with extra processors, it is OK to perform some wasted work if otherwise some processors are idle. In addition, in some cases, it actually helps to detect conflicts earlier comparing to other learning strategies. Of course, with limited computation resources, it's better to employ learning strategies to avoid wasted work as much as possible.

In summary, the combination of global committing and the hybrid learning strategy, i.e., NTS for locally committed contexts and SBS for globally committed contexts, is the most effective strategy among all strategy combinations across all conflict patterns to prevent wasted work given revocation history for the studied benchmark.

We next compare the OO7's performance using our SDBLFuture system (using the best strategy combination) with three other alternatives: coarse-grained lock implementation, fine-grained lock implementation, and Safe Futures. For both lock implementations of OO7, we use 4 threads instead of futures to perform same amount



(a) average with 1 processor (b) average with 4 processor

Figure 9.11: Average performance of lock-based, SafeFuture, and SDBLFuture implementations of the OO7 benchmark across 12 controlled conflict patterns. The conflict write position parameter specifies when a shared write is performed among all operations. “early” means the shared write is the first operation, while “late” means the shared write is the last operation. SDBLFuture-1st is the performance of the first iteration for the SDBLFuture version, and SDBLFuture-2nd is for the second iteration. For the other three versions, only the performance of the second iteration is shown since there is no difference between the two.

of assigned operations. For the coarse-grained lock version, we set the lockDepth parameter of OO7 to 1, which synchronizes at the module level. For the fine-grained lock version, we set the lockDepth to 9, which synchronizes at the composite part level. The Safe Future version is similar to our SDBLFuture version, but using the interface-based SafeFuture APIs. Another parameter we tested in this set of experiments are the timing of the shared write. We tried both extreme case: early write ($writePosition = 0$) and late write ($writePosition = 100$). Again, we collect the results for all conflict patterns. Figure 9.11 gives the average speedups across all conflict patterns for each implementation version and write position. For the SDBL-Future version, we show the results for both the first iteration (*SDBLFuture-1st*) and the second iteration (*SDBLFuture-2nd*) to demonstrate its learning ability. For the other three versions, only the data for the second iteration is used since there is no difference between the two iterations. The left graph contains the results for 1 processor, and the right graph is for 4 processors. The x-axes is the conflict write position used, and the y-axes are the speedups over the serial execution.

First of all, from these results, we can see that coarse-grained lock limits the parallelism of the program. The speedup of this version of OO7 are all about 1 for both conflict write positions with 1 or 4 processors, which means this coarse-grained lock implementation basically serializes the program. Of course, this implementation (lock at the module level) is kind of dumb and extreme, but it makes the point that coarse-grained lock is easy to program, but might hurt performance due to its limited parallelism. With more processors, the fine-grained lock version performs much better than the coarse-grained one since it enables much more parallel executions. In terms of conflict patterns, both lock-based versions are not as sensitive as the two future-based implementations, since there is no revocation, thus, no wasted work in the lock versions. But with 4 processors, the position of the shared write does makes a difference on the performance of the fine-grained lock implementation (average speedups are 2.71 for early, and 3.16 for late). This is because the later the conflicted write is, the later each thread has to synchronize with other threads, and as a result, the better parallelism of the execution is.

Second, we find that for the early conflict write pattern, our SDBLFuture system performs much better than the Safe Future version. The average speedup achieved by the Safe Future version with 4 processors is 1.47 in this case, while the SDBLFuture is able to achieve 3.00 speedup, which is even better than the fine-grained lock version, whose average speedup is 2.71. This is due to the laziness of our SDBLFuture system: a future is split only after the system has executed it for a while and predicts that it's beneficial to split it given its granularity and current system resource availability. With the early conflict write pattern, this laziness helps prevent all conflicts in the execution since the spawning happens after the shared write has been done by the current primordial context. The reason that the SDBLFuture version works better than the fine-grained lock version in this case is that there is no synchronization overhead in the SDBLFuture version provided its optimism nature in terms of data contention (i.e., assume no contention initially, but revoke if contentions occur).

For the late conflict write pattern, the SDBLFuture version works similarly to the Safe Future version for the first iteration, but it is able to reduce the penalty of revocations more effectively in the second iteration which cannot be done by the Safe Future implementation. Again, with more processors, this difference becomes smaller since the performance penalty of wasted work is not big anymore with more computation resources available. The average speedups with 1 processor is 0.62 versus 0.89 for the Safe Future version and the SDBLFuture version respectively. While with 4 processors, the results are 1.48 versus 1.52.

Finally, we find that with extra computation resources, both future implementations, which even have significant revocation overhead, still achieves better performance than the coarse-grained lock version. This is encouraging since the goal of many automatic memory protection techniques, including Safe Futures, our SDBLFutures, and all transactional memory work, is to achieve both the easiness of using coarse-grained locks to program, and the efficiency of fine-grained locks. Our results show some promising potentials of these systems in this direction.

In summary, for applications with no dependency violations, our SDBLFuture system introduces acceptable overhead for tracking shared data accesses and for maintaining meta data. In addition, the overhead of managing execution contexts is negligible even for a large amount of futures. For applications do have shared data contentions, our SDBLFuture system is able to achieve better performance than the Safe Future system, sometime even better than the fine-grained lock version, thanks to its laziness and the learning ability that are enabled by exploiting the rich, low level information and the adaptation of the Java virtual machine.

9.4 Related Work

Besides the SafeFuture system [153], which we have discussed in Section 9.1, the techniques we use to preserve as-if-serial side-effect semantics in our SDBLFuture system are related to two closely related active areas that both exploit optimistic concurrency: thread-level speculation (TLS) and transactional memory (TM).

Thread-level speculation (TLS) is a technique that attempts to automatically extract parallelism from sequential programs. It optimistically execute chunks of code in the sequential program in parallel threads although it is uncertain whether those code areas are actually independent. The system tracks memory access to detect any inter-thread data dependency violations according to the serial execution ordering. In case that any dependency violation does occur, the offending thread is squashed, and all side-effects of the offending thread is discarded. TLS techniques complement the traditional parallel compiler techniques and help to exploit extra parallelism from the applications whose data dependency information cannot be analyzed statically. There has been a rich body of research on TLS, which are either implemented completely in hardware (e.g. [134, 147, 146, 61, 138]), or completely in software (e.g. [124, 60, 91, 33, 119]), or hybrid(e.g. [137, 62, 118, 34]). Most of the work on TLS has targeted at the loop-level parallelism (e.g. [147, 34, 43, 30, 60, 124, 33]). Others exploit the speculative parallelism at the method-level parallelism (e.g. [29, 152, 116, 31, 105]), even at basic block level (e.g. [150]).

SDBLFutures share many common aspects with TLS techniques, but with one big difference: the SDLFuture system relies on programmers to identify the potential parallelization points using the “@future” annotations, while one of the main tasks of TLS systems is to automatically identify parallelization candidates via static analysis (e.g. [43, 31]) or profile informations (e.g. [30, 105, 111, 156]). The co-operative model between programmers and the system of SDBLFutures significantly simplifies the compiler and runtime implementation since programmers usually have better knowledge of the program structure and semantics. However, those future annotations are only hints to the system, and we still need to carefully select profitable spawning points that are able to amortize parallelization overhead, and that have less probability of dependency violations. Currently, we have exploited some profile-based techniques to refine the parallelization candidates, such as the sampling based adaptive and lazy future scheduling mechanism and the revocation history-based learning strategies. In future work, we could apply the static analysis and profile-based techniques that have been exploited in the TLS works to make wiser scheduling decisions.

Transactional memory (TM) is an optimistic synchronization technique that was proposed as an alternative to lock-based synchronization. With the transactional memory programming model, programmers enclose code section that should be executed atomically in a transaction, and the system guarantees the atomicy and serializability of transaction executions and at same time attempts to achieve as high concurrency as possible. The TM model is much simpler to use and helps address many problems of lock-based synchronization, such as dead-lock, priority inversion, non-composability, etc. It has been an active research area recently. Similar to TLS, these works can be categorized to hardware-based (e.g. [72, 63, 6, 114]), software-based (e.g. [131, 71, 64, 46, 66, 2, 110]), or hybrid (e.g. [122, 98, 37, 129, 23, 132]).

There are two main differences between SDBLFutures and TM techniques. First, TM techniques target at the synchronization problem, which is orthogonal to the parallelization problem. The TM techniques usually assume that concurrent execution has been introduced to the program via some parallelization model, such as threads. In contrast, our SDBLFuture is a parallel language construct which introduces parallelism to serial programs. To guarantee the as-if-serial side-effect semantics, our system maps the whole concurrent tasks (futures and continuations) as transactions, which is more aggressive than most of current TM work that only maps critical sections to transactions. Secondly, there is no ordering constraint in general TM system, while our SDBLFuture system enforces the as-if-serial ordering among all tasks. Nevertheless, SDBLFutures can be seen as one application of TM model with serial ordering constraints. Therefore, as part of future work, our system could exploit many techniques that attempt to reduce the overhead of TM systems, especially those run-time optimizations [66, 2] and adaptation among various implementation alternatives [128, 108, 109].

9.5 Summary

In summary, our SDBLFuture system inherits many programmer productivity and performance advantages from the DBLFuture system. SDBLFuture builds upon and extends extant work on safe future implementation, yet provides support for nested futures, improved efficiency, and a simpler implementation. By employing the rich, low-level information available in the Java virtual machine, and the JVM's ability to learn about and modify program behavior dynamically, we are able to construct a simple system that dynamically adapt the performance of a wide range of application and computation patterns. These features enables a straightforward and efficient programming model for parallel computing in Java that simplifies programmer effort significantly and advances the current state of the art in JVM-based parallelization.

Chapter 10

Conclusion

Providing an easy way to program applications for a diversity of computing devices for average developers is a real challenge in the era of pervasive computing. Specialized in their own application areas, computing devices differ in terms of capability and resource availability. A fair amount of expert knowledge is required to write programs on different devices to achieve efficiency. Java, as a universal programming language for a large spectrum of devices, is portable, versatile and easy to program. However, its potential to reconcile the differences among devices and to provide a uniform, efficient and powerful programming method by implanting device specific knowledge in its runtime system has not been exploited to its maximal extent. More specifically, the real power of Java resides in its runtime execution environment, i.e. the Java virtual machine (JVM). The JVM has accurate runtime information of both program execution and system resources, can access to dynamic runtime services and low level runtime constructs and is able to make adaptive decision based on the runtime information and apply control over the runtime services. It is this adaptability that enables this thesis work.

10.1 Contribution

In general, this thesis work contributes to the goal of providing easy and efficient Java programming for diverse devices by applying JVM's adaptation to the problem of automatic management of system resource and capability in efficiency. By enabling this management in JVM runtime system and simplifying it in programming interface, programmers do not have to make explicit efforts and thus can be more focused on the application logic. By utilizing the JVM runtime services and constructs, the system resources and capabilities can be managed in a more efficient way, which is not achievable at application level.

In particular, our work focuses on two problems: managing code memory on resource constrained devices, and providing easy and efficient programming interface for exploiting the parallel capability of multi-core systems.

Just-in-time (JIT) compiler enables high performance of JVMs. However, on resource constrained devices, e.g. smart phones, personal digital assistants (PDAs), etc., JIT-based JVMs have limited presence. One of the main reasons is that compiled native code occupies a large amount of memory. By analysis, we find that there is a great potential to remove “dead code” from the code memory and thus reduce the memory footprint of JIT compilers. However, it is not feasible to unload “dead code” at application level by programmers. We build an adaptive code unloading system based on modern JVMs’ runtime services that completely automating the process of code memory management. By monitoring the system resource availability in real time and making unloading decision according to a cost-benefit model, we are able to greatly reduce code memory size for a set of Java benchmarks. We also achieve better performance for most of these benchmarks due to reduced garbage collection overhead. This part of our work makes it more promising to apply JIT compilation technology to mobile devices to achieve faster execution speed of Java programs.

At the high end, multi-core processors have made their way into not only just high performance servers, but also daily-used desktops. The more and more widely available massive hardware parallelism demands an easy and efficient programming support to extract maximal performance from the hardware. One potential candidate is Java future. Future as a language construct aims to make parallel programming easy to do. However, the current future implementation in Java is not only cumbersome, but also inefficient. It is mostly because it is built at the library level and lacks runtime support. Programmers have to manually create and schedule parallel future tasks using their inaccurate hunches. We build an adaptive system to support better future programming in the following four aspects:

- A lazy future support that creates and schedule future automatically according to the computation granularity and the available hardware parallelism. We achieve optimal performance for a set of benchmarks that is comparable to hand-tuned alternatives with much less programmer effort.

- A directive-based future programming interface. With this language support, programmers can identify parallelism in their programs by simple annotations. It also has the performance benefit due to the reduced future object creation.

- As-if-serial exception handling support. This makes it easy to migrate serial programs to parallel environment. Programmers can simply develop and reason serially and switch to parallel version without worrying about changing the exception handling behavior. The empirical results show that we introduce negligible overhead.

- As-if-serial side effect guarantee. This enables "safe" futures. Programmers thus do not need to worry about access to shared objects among parallel future tasks. It is also easy to switch from serial program since the serial access order of shared objects maintains. Our support of "safe" futures enables nested futures, simplifies implementations and improves performance.

In summary, our safe, directive-based, lazy future (SDBLFuture) implementation enables easy and efficient parallel programming for devices with massive parallelism capability, which may accelerate the adoption of multi-core technology.

10.2 Future Work

This thesis work is our attempt to achieve the ultimate goal of enabling easy and efficient programming for diverse devices. It is far from complete. We believe it can be improved in many aspects. In particular, there are still many interesting open research problems associated with the SDBLFuture system.

First, our online future scheduling system takes both program and system behavior to make profitable spawning decisions. Currently, we have exploited the estimated execution time and revocation history of contexts collected via low overhead online profiling techniques to guide this decision. We plan to explore more static and dynamic program information to refine this decision model. For example, we could perform static analysis and dynamic profiling to estimate the memory accessing patterns of contexts, and use that to avoid spawning potentially conflicting contexts.

Second, there are various implementation alternatives for tracking shared data accesses, detecting conflicts, and managing execution contexts. We want to evaluate these alternative to achieve better performance. For example, there are two important methods to maintain speculative updates to shared data: write-buffering and undo-logging. With write-buffering, an execution context keeps a private copy of every shared data it has modified, and makes them visible to other contexts only after successfully committing. Undo-logging instead writes the data directly to the shared memory, but keeps logs for undoing its side-effects in case of revocation. It has been shown in the transactional memory (TM) community that undo-logging is more efficient in most situations [128]. However, given the total ordering constraint of execution contexts in our SDBLFuture system, it is unclear which is better. We want to make a contrast study as our future work.

Third, our current implementation associates execution contexts with the whole future and continuation computation, which might result in very long contexts. As we have demonstrated in our discussion on local commit and global commit, the granularity of contexts has a great impact on overall performance, because larger contexts lead to more wasted computation in case of revocations. Fortunately, the as-if-serial semantics does not require us to map contexts at the boundaries of future and context computation, although that is the simplest and most nature mapping. One optimization we could explore is to slice the contexts dynamically to reduce the computation size so that when conflict happens, we can revoke in finer granularity and save more “innocent” work. However, slicing means creating more context and thus larger overhead in maintaining them. We want to apply our adaptive infrastructure to achieve the balance and eventually better performance.

Our as-if-serial methodology provides great programmer productivity benefit, however, it also limits the potential parallelism due the strong total ordering constraint. It would be interesting to investigate how much parallelism is sacrificed in order to preserve the as-if-serial semantics, including exception handling behavior and side effects. Based on this study, we want to provide another kind of annotation, say “@ufuture”, to allow programmers informing the system to relax the ordering constraint to improve better performance.

In addition, complete software-level implementation of SDBLFutures suffers the hight overhead of managing shared data accessing. If a hardware transactional memory or thread-level speculation system is available (say the TCC system from Stanford [63]), it would be very interesting to investigate a hybrid system, where the language level future semantics is mapping to the low overhead hardware support to improve performance, while the software (virtual machine) system provides more flexible policy control and adaptation.

Moreover, we want to use our SDBLFutures to develop a wide range of real applications, such as web servers, game engines, to study the usability and limitations of the future programming model, and to gain insights on a better parallel programming model that is easier to use and able to provide the runtime system greater flexibility to improve performance.

Beyond our two foci, namely code management and easy and efficient futures, in this thesis work, we believe the adaptive infrastructure in JVM that we build is applicable in a broader area. Specifically, we want to explore the possibility of managing energy consumption adaptively in JVM. This is interesting since power has become a critical issue not just for battery powered low end devices, but also for high end servers and desktops (e.g. the global warming problem).

Finally, given our experiences in supporting adaptive services to solve programming problems, we want to study a better virtual machine design. For example, we want to provide modular programming interfaces to facilitate easy exploitation of the JVM's adaptation for other language designs, especially the easy and efficient parallel programming models.

Bibliography

[1] A. Adl-Tabatabai, M. Cierniak, G. Lueh, V. Parikh, and J. Stichnoth. Fast, Effective Code Generation in a Just-In-Time Java Compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 280–290. ACM Press, May 1998.

[2] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the 2006 Conference on Programming language design and implementation*, pages 26–37. Jun 2006.

[3] E. Allan, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The Fortress language specification version 0.785. Technical report, Sun Microsystems, 2005.

[4] E. Allan, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The Fortress language specification version 0.954. Technical report, Sun Microsystems, 2006.

[5] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–221, 2000.

[6] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Washington, DC, USA, 2005. IEEE Computer Society.

[7] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM: the controller's analytical mode. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, Dec. 2000.

[8] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the jalapeo jvm. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'00)*, pages 47–65, 2000.

[9] M. Arnold, M. Hind, and B. Ryder. An empirical study of selective optimization. In *13th International Workshop on Languages and Compilers for Parallel Computing (LCPC'00)*, August 2000.

[10] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of java. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'02)*, pages 111–129, 2002.

[11] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation (PLDI'01)*, pages 168–179, 2001.

[12] The AspectJ Project. http://www.eclipse.org/aspectj/.

[13] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12. ACM Press, 2000.

[14] S. M. Blackburn and A. L. Hosking. Barriers: friend or foe? In *Proceedings of the 4th international symposium on Memory management*, pages 143–151, 2004.

[15] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216, 1995.

[16] D. Box. *Essential .NET, Volume I: The Common Language Runtime*. Addison Wesley, Nov. 2002.

[17] G. Bracha, J. Gosling, B. Joy, and G. Steel. *The Java Language Specification*. Addison Wesley, second edition, June 2000.

[18] D. Bruening and E. Duesterwald. Exploring Optimal Compilation Unit Shapes for an Embedded Just-In-Time Compiler. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, Dec. 2000.

[19] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 265–275. IEEE Computer Society, Mar. 2003.

[20] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeo dynamic optimizing compiler for java. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 129–141, 1999.

[21] D. Callahan and B. Smith. A future-based parallel language for a general-purpose highly-parallel computer. In *Selected papers of the second workshop on Languages and compilers for parallel computing*, pages 95–113, 1990.

[22] R. H. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE Trans. Softw. Eng.*, 12(8):811–826, 1986.

[23] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. Jun 2007.

[24] M. J. Carey, D. J. DeWitt, C. Kant, and J. F. Naughton. A status report on the oo7 oodbms benchmarking effort. In *OOPSLA '94: Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, pages 414–426, New York, NY, USA, 1994. ACM.

[25] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The 007 benchmark. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 12–21, New York, NY, USA, 1993. ACM.

[26] ChaiVM. http://www.chai.hp.com.

[27] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 519–538, 2005.

[28] A. Chatterjee. Futures: a mechanism for concurrency among objects. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 562–567, 1989.

[29] M. K. Chen and K. Olukotun. Exploiting method-level parallelism in single-threaded java programs. In *PACT '98: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, page 176, Washington, DC, USA, 1998. IEEE Computer Society.

[30] M. K. Chen and K. Olukotun. The jrpm system for dynamically parallelizing java programs. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 434–446, New York, NY, USA, 2003. ACM.

[31] P.-S. Chen, M.-Y. Hung, Y.-S. Hwang, R. D.-C. Ju, and J. K. Lee. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 25–36, New York, NY, USA, 2003. ACM.

[32] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing judo: Java under dynamic optimizations. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation (PLDI'00)*, pages 13–26, 2000.

[33] M. Cintra and D. R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 13–24, New York, NY, USA, 2003. ACM.

[34] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 13–24, New York, NY, USA, 2000. ACM.

[35] The cldc hotspot(tm) implementation virtual machine. White Paper, 2003. http://web2.java.sun.com/products/cldc/wp/CLDC_HotSpot_WhitePaper.pdf.

[36] F. Cristian. Exception handling and software fault tolerance. *IEEE Transactions on Computers*, 31(6):531–540, 1982.

[37] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 336–346, New York, NY, USA, 2006. ACM.

[38] J. S. Danaher. The jcilk-1 runtime system. Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, June 2005.

[39] S. Debray and W. Evans. Profile-guided code compression. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 95–105. ACM Press, 2002.

[40] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. Deli: A new run-time control point. In *Proceedings of the 35th Annual International Symposium on Microarchitecture (MICRO)*, pages 257–268, Nov. 2002.

[41] P. C. Diniz and M. C. Rinard. Dynamic feedback: an effective technique for adaptive computing. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 71–84, 1997.

[42] M. Drinić, D. Kirovski, and H. Vo. Code optimization for code compression. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 315–324. IEEE Computer Society, 2003.

[43] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 71–81, New York, NY, USA, 2004. ACM.

[44] K. Ebcioglu, E. R. Altman, M. Gschwind, and S. W. Sathaye. Dynamic binary translation and optimization. *IEEE Transactions on Computers*, 50(6):529–548, 2001.

[45] Ellis, T. M. R., I. R. Phillips, and T. M. Lahey. *Fortran 90 Programming*. Addison Wesley, first edition, 1994.

[46] R. Ennals. Efficient software transactional memory. Technical Report IRC-TR-05-051, Intel Research Cambridge Tech Report, Jan 2005.

[47] J. Ernst, W. Evans, C. W. Fraser, T. A. Proebsting, and S. Lucco. Code compression. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 358–365. ACM Press, 1997.

[48] K. I. Farkas, J. Flinn, G. Back, D. Grunwald, and J. M. Anderson. Quantifying the energy consumption of a pocket computer and a Java virtual machine. In

Proceedings of ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, pages 252–263, June 2000.

[49] S. J. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 241–252. IEEE Computer Society, Mar. 2003.

[50] C. Flanagan and M. Felleisen. The semantics of future and its use in program optimization. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages,* pages 209–220, 1995.

[51] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 212–223, 1998.

[52] G. W. Hill. ACM Alg. 395: Student's T-Distribution. *Communications of the ACM*, 13(10):617–619, Oct. 1970.

[53] Gartner Inc., Market Share: Mobile Terminals, Worldwide, 2Q06, Aug. 2006. http://www.gartner.com/.

[54] Gartner Inc., Quarterly Statistics: PDA and Smartphone Shipment Forecast, Mar. 2007. http://www.gartner.com/.

[55] IDC's Worldwide Quarterly PC Tracker, Jan. 2007. http://www.idc.com/.

[56] IDC's Worldwide Quarterly PC Tracker, Apr. 2007. http://www.idc.com/.

[57] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy threads: implementing a fast parallel call. *J. Parallel Distrib. Comput.*, 37(1):5–20, 1996.

[58] J. Gosling, B. Joy, G. Steel, and G. Bracha. *The Java Language Specification Second Edition*. Addison Wesley, second edition, 2000.

[59] A. S. Grimshaw. Easy-to-use object-oriented parallel processing with mentat. *Computer*, 26(5):39–51, 1993.

[60] M. Gupta and R. Nim. Techniques for speculative run-time parallelization of loops. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–12, Washington, DC, USA, 1998. IEEE Computer Society.

[61] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun. The stanford hydra cmp. *IEEE Micro*, 20(2):71–84, 2000.

[62] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 58–69, New York, NY, USA, 1998. ACM.

[63] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 102, Washington, DC, USA, 2004. IEEE Computer Society.

[64] T. Harris and K. Fraser. Language support for lightweight transactions. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402. Oct 2003.

[65] T. Harris, M. Herlihy, S. Marlow, and S. Peyton-Jones. Composable memory transactions. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming, to appear*, Jun 2005.

[66] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *Proceedings of the 2006 Conference on Programming language design and implementation*, pages 14–25. ACM Press, Jun 2006.

[67] K. Hazelwood and J. E. Smith. Exploring code cache eviction granularities in dynamic optimization systems. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 89–99. IEEE Computer Society, 2004.

[68] K. Hazelwood and M. D. Smith. Code cache management schemes for dynamic optimizers. In *Proceedings of the Workshop on Interaction between Compilers and Computer Architecture (Interact-6)*, pages 92–100, Feb. 2002.

[69] K. Hazelwood and M. D. Smith. Generational cache management of code traces in dynamic optimization systems. In *Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO)*, pages 169–179, Dec. 2003.

[70] J. Henry C. Baker and C. Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 55–59, 1977.

[71] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. pages 92–101, Jul 2003.

[72] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM.

[73] B. Hindman and D. Grossman. Atomicity via source-to-source translation. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 82–91, 2006.

[74] The Java Hotspot performance engine architecture, Apr. 1999. http://java.sun.com/products/hotspot/whitepaper.html.

[75] The Java HotSpot Virtual Machine. White Paper, 2001. http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.ps.

[76] S. ichi Tazuneki and T. Yoshida. Concurrent exception handling in a distributed object-oriented computing environment. In *ICPADS '00: Proceedings of the Seventh International Conference on Parallel and Distributed Systems: Workshops*, page 75, Washington, DC, USA, 2000. IEEE Computer Society.

[77] Intel Research Advances 'Era Of Tera', Feb. 2007. http://www.intel.com/pressroom/archive/releases/20070204comp.htm.

[78] I. Intermetrics, editor. *Information Technology – Programming Languages – Ada*. ISO/IEC 8652:1995(E), 1995.

[79] V. Issarny. An exception handling model for parallel programming and its verification. In *SIGSOFT '91: Proceedings of the conference on Software for citical systems*, pages 92–100, 1991.

[80] V. Issarny. An exception handling mechanism for parallel object-oriented programming: Towards reusable, robust distributed software. *Journal of Object-Oriented Programming*, 6(6):29–39, 1993.

[81] H. A. James and K. A. Hawick. Data futures in discworld. In *HPCN Europe 2000: Proceedings of the 8th International Conference on High-Performance Computing and Networking*, pages 41–50, London, UK, 2000. Springer-Verlag.

[82] Java Remote Method Invocation Specification. http://java.sun.com/j2se/1.4.2/docs/guide/rmi/.

[83] JavaOne 2007 Press Kit, May 2007. http://www.sun.com/aboutsun/media/presskits/javaone2007/index.jsp.

[84] IBM Jikes Research Virtual Machine (RVM). http://www-124.ibm.com/developerworks/oss/jikesrvm.

[85] BEA JRockit, Java for the Enterprise. http://www.bea.com/content/news_events/white_papers/BEA_JRockit_ Ent-Java_business_wp.pdf.

[86] JSR166: Concurrency utilities. http://java.sun.com/j2se/1.5.0/docs/guide/concurrency.

[87] JSR 175: A Metadata Facility for the JavaTM Programming Language. http://jcp.org/en/jsr/detail?id=175.

[88] C. Jung, D. Lim, J. Lee, and S. Han. Adaptive execution techniques for smt multiprocessor architectures. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 236–246, 2005.

[89] Kaffe – An opensource Java virtual machine, 1998. http://www.kaffe.org.

[90] M. Karaorman and P. Abercrombie. jcontractor: Introducing design-by-contract to java using reflective bytecode instrumentation. *Form. Methods Syst. Des.*, 27(3):275–312, 2005.

[91] I. H. Kazi and D. J. Lilja. Javaspmt: A speculative thread pipelining parallelization model for java programs. In *IPDPS '00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, page 559, Washington, DC, USA, 2000. IEEE Computer Society.

[92] A. Krall. Efficient JavaVM just-in-time compilation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 205–212. North-Holland, 1998.

[93] D. A. Kranz, J. R. H. Halstead, and E. Mohr. Mul-T: a high-performance parallel Lisp. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 81–90, 1989.

[94] C. Krintz. Coupling On-Line and Off-Line Profile Information to Improve Program Performance. In *International Symposium on Code Generation and Optimization (CGO'03)*, Mar. 2003.

[95] C. Krintz and B. Calder. Using Annotation to Reduce Dynamic Optimization Time. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 156–167, June 2001.

[96] C. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the Overhead of Dynamic Compilation. *Software-Practice and Experience*, 31(8):717–738, 2001.

[97] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in split-c. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 262–273, 1993.

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

[99] Java(TM) 2 Platform Micro Edition(J2ME(TM)) Technology for Creating Mobile Devices. White Paper, May 2000. http://java.sun.com/products/cldc/wp/KVMwp.pdf.

[100] D. Lea. A java fork/join framework. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, 2000.

[101] I.-T. A. Lee. The JCilk multithreaded language. Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, Aug. 2005.

[102] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, second edition, Apr. 1999.

[103] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 260–267, 1988.

[104] B. Liskov and A. Snyder. Exception handling in CLU. *IEEE Transactions on Software Engineering*, SE-5(6):546–558, Nov. 1979.

[105] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. Posh: a tls compiler that exploits program structure. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 158–167, New York, NY, USA, 2006. ACM.

[106] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *Proceedings of an ACM conference on Language design for reliable software*, pages 128–137, 1977.

[107] S. Lucco. Split-stream dictionary program compression. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 27–34. ACM Press, 2000.

[108] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Design tradeoffs in modern software transactional memory systems. In *Proceedings of the 7th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Houston, TX, Oct 2004.

[109] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive software transactional memory. In *Proceedings of the 19th International Symposium on Distributed Computing*, Cracow, Poland, Sep 2005. Earlier but expanded version available as TR 868, University of Rochester Computer Science Dept., May2005.

[110] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of software transactional memory. Technical report, Jun 2006. Held in conjunction with PLDI 2006. Expanded version available as TR 893, Department of Computer Science, University of Rochester, March 2006.

[111] P. Marcuello and A. González. Thread-spawning schemes for speculative multithreading. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 55, Washington, DC, USA, 2002. IEEE Computer Society.

[112] B. Meyer. *Eiffel: The Language*. Prentice Hall, second edition, 1992.

[113] E. Mohr, D. A. Kranz, and J. R. H. Halstead. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Trans. Parallel Distrib. Syst.*, 2(3):264–280, 1991.

[114] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. Logtm: Log-based transactional memory. In *Proceedings of the 12th International*

Symposium on High-Performance Computer Architecture, pages 254–265. Feb 2006.

[115] OpenMP specifications. http://www.openmp.org/specs.

[116] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, page 303, Washington, DC, USA, 1999. IEEE Computer Society.

[117] M. Paleczny, C. Vick, and C. Click. The Java HotSpot(TM) Server Compiler. In *USENIX Java Virtual Machine Research and Technology Symposium*, Apr. 2001.

[118] I. Park, B. Falsafi, and T. N. Vijaykumar. Implicitly-multithreaded processors. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 39–51, New York, NY, USA, 2003. ACM.

[119] C. J. F. Pickett and C. Verbrugge. Sablespmt: a software framework for analysing speculative multithreading in java. *SIGSOFT Softw. Eng. Notes*, 31(1):59–66, 2006.

[120] J. Plevyak, V. Karamcheti, X. Zhang, and A. A. Chien. A hybrid execution model for fine-grained languages on distributed memory multicomputers. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 41, 1995.

[121] M. P. Plezbert and R. K. Cytron. Does "just in time" = "Better late than never". In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'97)*, pages 120–131, 1997.

[122] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 494–505, Washington, DC, USA, 2005. IEEE Computer Society.

[123] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, 1975.

[124] L. Rauchwerger and D. Padua. The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 218–232, New York, NY, USA, 1995. ACM.

[125] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[126] J. Robert H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[127] A. Romanovsky, J. Xu, and B. Randell. Exception handling and resolution in distributed object-oriented systems. In *ICDCS '96: Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96)*, page 545, Washington, DC, USA, 1996. IEEE Computer Society.

[128] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006. ACM Press.

[129] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 185–196, Washington, DC, USA, 2006. IEEE Computer Society.

[130] S.Haridi and N.Franz. Tutorial of Oz, Mozart documentations. http://www.mozart-oz.org/ documentation/tutorial/ index.html.

[131] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213. Aug 1995.

[132] A. Shriraman, M. F. Spear, H. Hossain, V. J. Marathe, S. Dwarkadas, and M. L. Scott. An integrated hardware-software approach to flexible transactional memory. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 104–115, New York, NY, USA, 2007. ACM.

[133] L. A. Smith and J. M. Bull. A multithreaded java grande benchmark suite. In *Proceedings of the Third Workshop on Java for High Performance Computing*, June 2001.

[134] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 414–425, New York, NY, USA, 1995. ACM.

[135] SpecJVM'98 Benchmarks. http://www.spec.org/osg/jvm98.

[136] Rotor - the shared source cli, 2002. http://research.microsoft.com/programs/europe/rotor/default.aspx.

[137] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The stampede approach to thread-level speculation. *ACM Trans. Comput. Syst.*, 23(3):253–300, 2005.

[138] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 1–12, New York, NY, USA, 2000. ACM.

[139] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, second edition, 1991.

[140] D. Stutz, T. Neward, and G. Dhilling. *Shared Source CLI Essentials*, page 251. O'Reilly Associates, Inc., Mar. 2003.

[141] T. Suganuma, T. Ogasawara, K. Kawachiya, M. Takeuchi, K. Ishizaki, A. Koseki, T. Inagaki, T. Yasue, M. Kawahito, T. Onodera, H. Komatsu, and T. Nakatani. Evolution of a java just-in-time compiler for ia-32 platforms. *IBM J. Res. Dev.*, 48(5/6):767–795, 2004.

[142] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a java just-in-time compiler. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA'01)*, pages 180–195, 2001.

[143] K. Taura, K. Tabata, and A. Yonezawa. Stackthreads/mp: integrating futures into calling standards. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 60–71, 1999.

[144] K. Taura and A. Yonezawa. Fine-grain multithreading with minimal compiler supporta cost effective approach to implementing efficient multithreading languages. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 320–333, 1997.

[145] TIOBE Programming Community Index for May 2007. http://www.tiobe.com/tpci.htm.

[146] M. Tremblay, J. Chan, S. Chaudhry, A. W. Conigliaro, and S. S. Tse. The majc architecture: A synthesis of parallelism and scalability. *IEEE Micro*, 20(6):12–25, 2000.

[147] J.-Y. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P.-C. Yew. The superthreaded processor architecture. *IEEE Trans. Comput.*, 48(9):881–902, 1999.

[148] R. V. van Nieuwpoort, J. Maassen, T. Kielmann, and H. E. Bal. Satin: Simple and efficient Java-based grid programming. *Scalable Computing: Practice and Experience*, 6(3):19–32, September 2005.

[149] N. Vijaykrishnan, M. Kandemir, S. Tomar, S. Kim, A. Sivasubramaniam, and M. J. Irwin. Energy Characterization of Java Applications from a Memory Perspective. In *Proceedings of the USENIX Java Virtual Machine Research and Technology Symposium*, Apr. 2001.

[150] T. N. Vijaykumar and G. S. Sohi. Task selection for a multiscalar processor. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 81–92, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[151] D. B. Wagner and B. G. Calder. Leapfrogging: a portable technique for implementing efficient futures. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 208–217, 1993.

[152] F. Warg and P. Stenström. Limits on speculative module-level parallelism in imperative and object-oriented programs on cmp platforms. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 221–230, Washington, DC, USA, 2001. IEEE Computer Society.

[153] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for java. In *OOPSLA '05: Proceedings of the twentieth ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 439–453, 2005.

[154] J. Whaley. A portable sampling-based profiler for java virtual machines. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 78–87, 2000.

[155] J. Whaley. Partial Method Compilation using Dynamic Profile Information. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 166–179. ACM Press, Oct. 2001.

[156] J. Whaley and C. Kozyrakis. Heuristics for profile-driven method-level speculative parallelization. In *ICPP '05: Proceedings of the 2005 International*

Conference on Parallel Processing, pages 147–156, Washington, DC, USA, 2005. IEEE Computer Society.

[157] J. Xu, A. Romanovsky, and B. Randell. Coordinated exception handling in distributed object systems: From model to system implementation. In *ICDCS '98: Proceedings of the The 18th International Conference on Distributed Computing Systems*, page 12, Washington, DC, USA, 1998. IEEE Computer Society.

[158] J. Xu, A. Romanovsky, and B. Randell. Concurrent exception handling and resolution in distributed object systems. *IEEE Trans. Parallel Distrib. Syst.*, 11(10):1019–1032, 2000.

[159] B. Yang, S. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcioglu, and E. Altman. LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 128–138. North-Holland, 1999.

[160] L. Zhang and C. Krintz. The design, implementation, and evaluation of adaptive code unloading for resource-constrained devices. *ACM Trans. Archit. Code Optim.*, 2(2):131–164, 2005.

[161] L. Zhang, C. Krintz, and P. Nagpurkar. Language and virtual machine support for efficient fine-grained futures in java. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 130–139, Washington, DC, USA, 2007. IEEE Computer Society.

[162] L. Zhang, C. Krintz, and P. Nagpurkar. Supporting exception handling for futures in java. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 175–184, New York, NY, USA, 2007. ACM.

[163] L. Zhang, C. Krintz, and S. Soman. Efficient Support of Fine-grained Futures in Java. In *International Conference on Parallel and Distributed Computing Systems (PDCS)*, 2006.